Otto-von-Guericke Universität Magdeburg
Fakultät für Informatik

Master´s Thesis

Entwicklung eines nativen Compilers für Feature-orientierte Programmierung

Verfasser:

Christian Becker

24. Mai, 2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
Institut für Technische und Betriebliche Informationssysteme

Dipl.-Wirt.-Inform. Christian Kästner
Institut für Technische und Betriebliche Informationssysteme

Becker, Christian:
Entwicklung eines nativen Compilers für Feature-orientierte Programmierung
Master´s Thesis, Otto-von-Guericke Universität Magdeburg, 2010.

Danksagung

An dieser Stelle möchte ich mich bei Christian Kästner für die Betreuung dieser Arbeit bedanken. Danke für die vielen nützlichen Ratschläge und Diskussionen, die mir bei der Erstellung dieser Arbeit sehr geholfen haben.

Mein Dank gilt Constanze Adler. Ohne sie wäre das Studium sicherlich nicht so spannend, lustig und erfolgreich gewesen. Danke für das Korrekturlesen der Arbeit und die Hilfe bei LaTeX.

Thomas Thüm danke ich für die LaTeX-Vorlage dieser Arbeit und für die Hilfe bei der Verwendung.

Danke auch an Jutta Becker, die mir beim Korrekturlesen der Arbeit sehr geholfen hat.

Bei meinen Schwiegereltern möchte ich für die Gastfreundschaft in Leitzkau während des Studiums bedanken.

Zuletzt möchte ich mich bei meiner Frau Dorothee Becker bedanken, die mir dieses Studium erst ermöglicht und mich die ganze Zeit liebevoll unterstützt hat.

Acknowledgements

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

AHEAD	Algebraic Hierarchical Equations for Application Design
AST	Abstract Syntax Tree
FOP	Feature-orientierte Programmierung
FST	Feature Structure Tree
IDE	Integrated Development Environment
JastAddJ	JastAdd Extensible Java Compiler
loc	lines of code
OOP	Objekt-orientierte Programmierung
SPL	Software-Produktlinie

1. Einleitung

Anwendungssoftware, Computerspiele oder andere Softwareprodukte werden von Anbietern auf den unterschiedlichsten Plattformen veröffentlicht. Dabei werden, je nach verwendeter Plattform, an die Software unterschiedliche Anforderungen gestellt. Bei einem PC steht reichlich Hauptspeicher zur Verfügung und es können hohe Auflösungen verwendet werden. Soll die gleiche Software für ein Smartphone veröffentlicht werden, stehen viel weniger Hardwareressourcen zur Verfügung. Für die Smartphone-Variante müssen eventuell Funktionalitäten entfernt werden, damit die Software vernünftig verwendet werden kann. Für eine andere Plattform können andere Funktionalitäten hinzugefügt werden, wenn beispielsweise andere Eingabegeräte, wie ein Touchpad, zur Verfügung stehen.

Dies führt dazu, dass ein Softwarehersteller für jede Plattform eine maßgeschneiderte Softwarelösung entwickeln muss. So eine maßgeschneiderte Software enthält nur die Funktionalitäten, die für die entsprechende Plattform benötigt wird. Aus betriebswirtschaftlichen Gründen soll der Aufwand für die Anpassung der Software an eine andere Plattform möglichst gering sein. Der Nachteil bei maßgeschneiderter Software ist, dass bestehender Quelltext bei der Portierung für andere Systeme nicht im vollen Umfang wieder verwendet werden kann. Dies liegt beispielsweise daran, dass eine Funktionalität oder ein Belang nicht modular in einer Softwareeinheit implementiert ist und somit nicht ohne weiteres durch eine andere Funktionalität ausgetauscht werden kann.

Eine Software-Produktlinie (SPL) ermöglicht verschiedene Varianten aus einer Quelltextbasis zu erzeugen [BCK05, PBL05]. Mit einer SPL kann ein Hersteller maßgeschneiderte Software mit unterschiedlichen Funktionalitäten anbieten und gleichzeitig bestehenden Quelltext effizient wiederverwenden. Für die Entwicklung einer SPL sind Implementierungstechniken notwendig, die es erlauben, Funktionalitäten optional zu gestalten.

Bei der Feature-orientierte Programmierung (FOP) handelt es sich um ein Programmierparadigma, das sich für die Implementierung von SPL eignet [Pre97, BSR03]. Der Quelltext wird anhand von Belangen (Features) modularisiert. Aus einer Menge

von ausgewählten Features kann dann eine Variante generiert werden. Querschneidende Featureimplementierungen können mittels Klassenverfeinerung realisiert werden. Die FOP setzt auf bestehende Programmiersprachen und erweitert diese um die Möglichkeit der Klassenverfeinerung.

AHEAD [BSR03] und FeatureHouse [AKL09] stellen zwei Umsetzungen der FOP dar. Diese erweitern unter anderem die Programmiersprache Java um die Möglichkeit der Klassenverfeinerung. Ein Kompositionsprogramm wandelt den feature-orientierten Quelltext in nativen Java-Quelltext um, der anschließend von einem Standardcompiler in Bytecode übersetzt wird.

Diese zweistufigen Ansätze, aus Kompositionsprogramm und Compiler, stellen heute die gebräuchlichste Form bei der FOP dar. Sie führen aber zu Nachteilen, die in dieser Arbeit näher untersucht werden sollen. Erzeugt der Java-Compiler einen Fehler, bezieht dieser sich auf die Zwischendarstellung. Der Entwickler muss den Fehler auf den feature-orientierten Quelltext abbilden um dort den Fehler zu beheben. Die Möglichkeit der Klassenverfeinerung und die Komposition von Features können zu zusätzlichen Fehlern führen, die erkannt werden müssen. Diese Fehler können nicht nur auf fehlerhafte Implementierungen einzelner Features deuten, sondern auf Fehler, die die SPL betreffen [TBKC07].

Ein nativer FOP-Compiler, der keine Zwischendarstellung, in Form von nativen Quelltext benötigt und der während der Komposition der Features auf Funktionen des Compilers (zum Beispiels das Typsystem) zugreifen kann, kann bessere Ergebnisse liefern, als die zweistufigen Ansätze. Dazu soll im Rahmen dieser Arbeit ein nativer FOP-Compiler prototypisch implementiert werden. Anhand dieses Prototypen wird untersucht, an welchen Stellen der Compiler bessere Ergebnisse liefern kann. Dieser Prototyp kann für zukünftige Forschungen im Bereich der FOP als Basis für Erweiterungen dienen.

Aufbau der Arbeit

Diese Arbeit gliedert sich wie folgt:

In dem Kapitel 2 werden zunächst SPL als effiziente Möglichkeit für die Wiederverwendung von Quelltext und zur schnellen Generierung von Varianten einer Produktfamilien, vorgestellt. Für die Implementierung von SPL bieten sich verschiedene Techniken an, die in dem Abschnitt 2.1.1 gezeigt werden. Die FOP eignet sich besonders für die Implementierung und wird in dem Abschnitt 2.2 gesondert vorgestellt. Mit AHEAD und FeatureHouse werden in diesem Abschnitt zwei prominente Umsetzungen der FOP präsentiert.

Des Weiteren werden im Kapitel 2 Konzepte zu Compilern und Compilerbau gezeigt. Mit JastAddJ wird in Abschnitt 2.4.1 ein Compiler-Framework vorgestellt, dass Änderungen an der Programmiersprache Java ermöglicht. Mit diesem Framework wird ein Prototyp eines nativen FOP-Compiler erstellt.

Im Kapitel 3 werden an Hand einiger FOP-spezifischen Fehler die Leistungsfähigkeit der zweistufigen Ansätze untersucht. Im Anschluss wird diskutiert, welche Vorteile durch Verwendung eines nativen FOP-Compiler entstehen.

In dem Kapitel 4 wird die Implementierung des Prototypen eines nativen FOP-Compilers mit Hilfe des JastAddJ Frameworks gezeigt.

Das Kapitel 5 beschäftigt sich mit der Evaluation des Prototypen eines nativen FOP-Compilers. Dazu wurden einige Programme ausgewählt, die nach dem feature-orientierten Paradigma mit AHEAD oder FeatureHouse programmiert wurden. Diese Programme wurden dann von dem nativen FOP-Compiler übersetzt und die Laufzeit mit den zweistufigen Ansätzen verglichen.

In dem Kapitel 6 werden die Ergebnisse dieser Arbeit nochmal zusammengefasst.

Das Kapitel 7 gibt einen Ausblick auf die Möglichkeiten und die Chancen, die durch die Verwendung eines nativen FOP-Compiler entstehen.

2. Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für die Implementierung und Evaluation eines nativen feature-orientierten Compiler benötigt werden. Zunächst werden im Abschnitt 2.1 Software-Produktlinien als effektives Mittel dargestellt, um bestehenden Quelltext wieder zu verwenden und schnell angepasste Varianten zu generieren. Neben weiteren Implementierungstechniken für Software-Produktlinien, die im Abschnitt 2.1.1 gezeigt werden, eignet sich besonders die FOP, die in Abschnitt 2.2 beschrieben wird. Im Abschnitt 2.3 werden einige Grundlagen über den Aufbau und die Funktionsweise eines Compiler vorgestellt. Diese werden für Techniken benötigt, die vom Compiler-Framework JastAdd verwendet werden, um neue Programmiersprachen zu erstellen oder bestehende Sprache zu erweitern. Das Framework wird in Abschnitt 2.4.1 vorgestellt.

2.1 Software-Produktlinien

Die Zeit, die für eine Produkteinführung benötigt wird (engl. Time-to-Market), spiegelt sich direkt in den Unternehmensgewinnen wieder [Sys06]. Aus diesem Grund möchten Hersteller schnell auf Veränderungen am Markt reagieren und ihren Kunden zeitnah angepasste Lösungen für ihr Problem bieten. Im Bereich der Softwareentwicklung können daher wiederholte Änderungen oder Anpassungen an bestehenden Programmen oder sogar die Neuprogrammierung nicht für jeden Kunden zielführend sein.

Softwareprodukte können über Konfigurationsdateien oder Menüs benutzerdefinierte Einstellungen und Funktionen bereitstellen. In vielen Fällen ist damit eine Anpassung an die Anforderungen des Kunden möglich. In diesem Fall befindet sich der volle Funktionsumfang in dem Programm. Ein Softwarehersteller möchte aber eventuell eine vereinfachte Variante der Software anbieten und zusätzliche Funktionen separat vermarkten. Hierbei sollten die zusätzlichen Funktionen in separaten Softwareeinheiten vorhanden sein. Aus Sicht des Anwenders der Software kann zusätzlich das Problem auftreten, dass das Softwareprodukt, durch zu viele Funktionen größer wird, und dem entsprechend auch mehr Ressourcen benötigt. Es sind viele Anwendungsszenarien denkbar, in

denen eine schlanke Software notwendig ist. Der Einsatz auf Netbooks, Smartphones und eingebetteten Systemen sind nur einige Beispiele, bei denen Hardwareressourcen auch heute noch begrenzt sind.

Eine *SPL* bietet eine effektive Möglichkeit bestehenden Quelltext wieder zu verwenden und unterschiedliche Varianten, mit der jeweils gewünschten Funktionalität einer Produktfamilie schnell zu generieren. Aus den oben genannten Gründen hat daher die Bedeutung von Software Produktlinien in den letzten Jahren stark zugenommen [BCK05, PBL05]. Die Entwicklung einer SPL gliedert sich hierbei in zwei Bereiche: die *Problemebene* und die *Lösungsebene* [CE00]. Die Problemebene beschäftigt sich mit der Analyse des Einsatzgebietes der SPL und der Umsetzung von Kundenanforderungen durch die Wahl von passenden Merkmalen. In der Lösungsebene findet die Implementierung und die Generierung der Varianten statt.

In der *Problemebene* wird zunächst der Einsatzbereich der SPL festgelegt und analysiert. Zur Analyse ist domänenspezifisches Wissen notwendig, um passende *Features* (dt. Merkmale) zu finden. Ein *Feature* ist hierbei eine Funktion oder Eigenschaft eines Softwaresystems, die für den Nutzer sichtbar ist. In der Literatur werden viele, teils unterschiedliche, Definitionen für den Begriff *Feature* verwendet. Beispielsweise liefern Kang et al. [KCH+90] folgende Definition:

> a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems
> (ein bedeutender oder kennzeichnender, für den Benutzer sichtbarer Aspekt oder charakteristische Eigenschaft eines Softwaresystems)

Eine Übersicht über weitere Definitionen des Begriffes Feature findet sich in [AK09]. In dieser Arbeit wird das gebräuchliche englische Wort *Feature* anstelle von Merkmal verwendet.

Eine SPL wird für einen Anwendungsbereich (eine Domäne) entwickelt. Für diese Domäne werden Features gesucht und die Abhängigkeit, in welcher sie zueinander stehen. Eine verbreitete Darstellungsform sind Feature-Modelle [KCH+90, CE00, Bat05]. Ein Feature-Modell ordnet die Features in einer Baumstruktur an. Je nach Verfasser unterscheiden sich teilweise die Darstellungen. Für diese Arbeit wird eine abgeänderte Darstellung von Batory [Bat05] verwendet, die beispielsweise auch in FeatureIDE [KTS+09] verwendet wird. Die Abbildung 2.1 zeigt ein Beispiel für ein Feature-Modell einer Chat-SPL. Das Feature *BasisChat* muss immer ausgewählt werden. Als Benutzerinterface kann entweder eine *GUI* oder eine *Konsole* gewählt werden. Die Textnachrichten können mittels des Features *Farbe* farbig gestaltet werden und zusätzlich kann die Nachricht mit zwei unterschiedlichen Algorithmen verschlüsselt werden. Beide Algorithmen können auch gleichzeitig verwendet werden. Das Feature *Verlauf* ist optional. Da es zusätzliche Abhängigkeiten gibt, die schlecht in der Baumform dargestellt werden können, besteht die Möglichkeit, diese durch zusätzliche boolesche Ausdrücke darstellen zu können. In diesem Beispiel wird für farbige Textnachrichten eine grafische Oberfläche voraus gesetzt (Farbe impliziert (=>) GUI). Hierbei handelt es sich um ein sehr kleines und konstruiertes Beispiel. Aber selbst aus dieser kleinen SPL mit 6 auswählbaren Features können 20 unterschiedliche Varianten erzeugt werden!

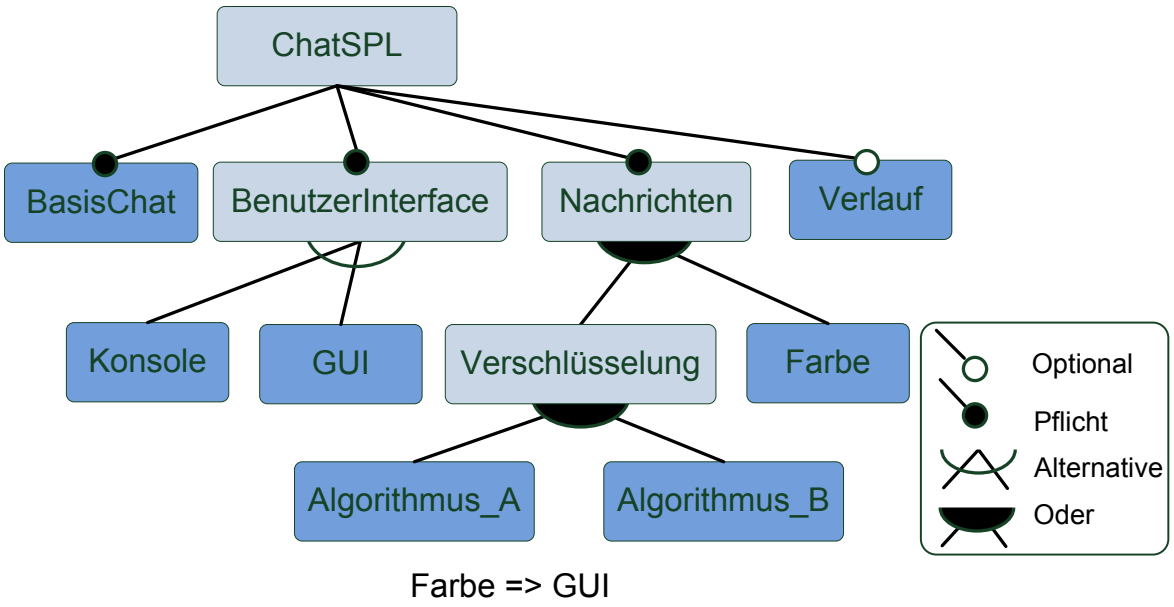

Abbildung 2.1: Feature-Modell einer Chat Software-Produktlinie

Neben den grafischen Darstellungsformen lassen sich Feature-Modelle in logische Formeln oder in eine Grammatik (z.B. die GUIDSL Grammatik [Bat05]) übersetzen, was die weitere Verarbeitung und eventuelle Optimierungen erleichtert.

Für die Erzeugung einer Variante muss eine Auswahl an Features getroffen werden. Grundlage hierfür bilden Feature-Modelle, die es beispielsweise dem Kunden ermöglichen passende Features zu wählen. Sollte keines der vorliegenden Features den Kundenwünschen entsprechen, können auch neue Features in das Feature-Modell eingefügt werden.

In der *Lösungsebene* müssen die einzelnen Features implementiert werden. Hierfür bieten sich unterschiedlichste Vorgehensweisen an, die im Abschnitt 2.1.1 näher erläutert werden.

Mit der Auswahl der Features aus der Problemebene und den implementierten Features soll nun möglichst ohne viel Aufwand ein lauffähiges Programm, beziehungsweise eine Variante der SPL erstellt werden. Im Anschluss daran muss die Variante getestet werden. Der ideale Fall wäre eine vollständige automatische Generierung und das automatisierte Testen der Variante.

In der Abbildung 2.2 werden die Entwicklungsschritte einer SPL nochmal grafisch zusammengefasst dargestellt. Eine SPL wird für eine Domäne entwickelt. Diese Domäne muss analysiert werden um passende Features zu finden. Als grafische Darstellung bieten sich die zuvor gezeigten Feature-Modelle an. Die einzelnen Features werden dann, mit denen im Abschnitt 2.1.1 vorgestellten Techniken, implementiert. Die Analyse der Domäne und die Implementierung wird unter dem Begriff Domänenentwicklung zusammengefasst. Soll eine Variante erzeugt werden, müssen zunächst die Anforderungen des

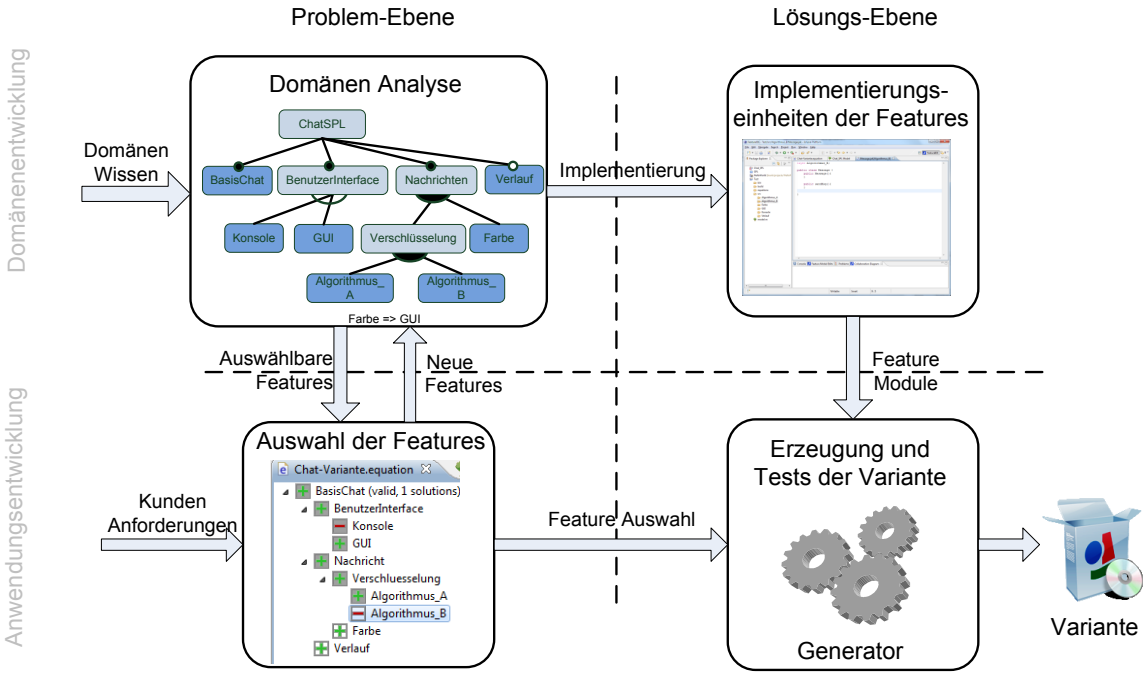

Abbildung 2.2: Entwicklungsschritte einer SPL, adaptiert von [Käs10]

Kunden analysiert werden. Aus dem Feature-Modell kann der Kunde eine Auswahl an Features treffen. Je nach Anforderung des Kunden, müssen zusätzliche neue Features in das Feature-Modell aufgenommen werden. Wurde eine Auswahl an Features getroffen, erzeugt ein Generator aus den zuvor implementierten Feature-Modulen eine Variante der SPL. Bevor die Variante ausgeliefert werden kann, muss sie getestet werden. Die Auswahl der Features, Erzeugung und Tests der Variante kann unter dem Begriff Anwendungsentwicklung zusammengefasst werden. Die Auswahl einer gültigen Kombination [AK09] und das Testen der Variante [NTJ06, CDS06, UGKB08] bilden weitere Forschungsgebiete. Auf diese Gebiete wird in dieser Arbeit nicht näher eingegangen und es sei auf die entsprechende Literatur verwiesen.

2.1.1 Implementierungstechniken für Software-Produktlinien

Eine grundsätzliche Vorgehensweise bei der Entwicklung von umfangreicher Software ist es, Probleme in kleinere handhabbare Einheiten zu zerteilen. Kleine modulare Einheiten erleichtern die Implementierung und die Wartbarkeit und erhöhen somit die Qualität des Quelltextes. In der Literatur wird hierfür der Begriff *Seperation of Concern* (dt. Aufteilung nach Belangen) verwendet, dieser geht auf die Arbeiten von Parnas [Par72] und Dijkstra [Dij82, Dij97] zurück.

Die *Objekt-orientierte Programmierung (OOP)* bietet mit Techniken wie Klassen und Vererbung Möglichkeiten modulare Software zu schreiben [LR09]. Nicht immer können alle Belange gleichzeitig modularisiert werden. In [KLM+97, TOHS99] werden die Grenzen des *Seperation of Concern* gezeigt. Belange, die eine Kernfunktionalität betreffen, können in vielen Fällen nicht modularisiert werden. Belange, die an vielen Stellen

im Quelltext auftreten, werden auch *Cross-Cutting Concern* [KLM+97] (dt. querschneidende Belange) genannt. Klassische Beispiele sind, Fehlerbehebung oder Logging. Solche Belange können ebenfalls selten modularisiert werden.

```
1  public class Nachricht{
2      String inhalt;
3      ArrayList verlauf = new ArrayList();
4      VerschlüsselungsAlgorithmusA algorithmusA;
5      //Viele weitere Zeilen
6      public void sendeNachricht(){
7          verlauf.add(inhalt);
8          inhalt = algorithmusA.verschluesselNachricht(inhalt);
9          sendeInhalt(inhalt);
10     }
11 }
```

Quelltext 2.1: Auszug aus der Klasse Nachricht der ChatSPL

Der Quelltext 2.1 zeigt die Klasse *Nachricht* mit einer Methode *sendeNachricht* aus der Chat-SPL. Die Methode *sendeNachricht* implementiert nicht nur die Basisfunktionalität des Chats, sondern auch die Features *Verlauf* und *Verschlüsselung*. Das eine Klasse vermischten Quelltext mehrerer Features enthält wird als *Code Tangling* [KLM+97] bezeichnet. Des Weiteren wird der Verschlüsselungsalgorithmus wahrscheinlich auch an anderen Stellen in der Chat-Software verwendet. Dies wird als verstreuter Quelltext (*Code Scattering*) bezeichnet. In diesem kleinen Beispiel, mag dies noch nicht als Problem auftreten, werden aber umfangreichere Programme betrachtet, kann eine Klasse sehr viele Features implementieren und der Quelltext eines Features kann über sehr viele Klassen verteilt sein (Beispielsweise das Feature Transaktion in einer Datenbank). Hier kann das Lokalisieren eines Quelltextes, der zu einem Feature gehört, zu einer sehr aufwändigen Arbeit werden.

Für die Implementierung von einer SPL sind Techniken notwendig, die es erlauben, Features optional zu gestalten. Dafür muss der Quelltext der zu einem Feature gehört identifizierbar sein. Das Problem des vermischten und verstreuten Quelltextes erschwert dies. Daher sind für die Implementierung von SPL Techniken von Vorteil, die diese Probleme minimieren.

In [KAK08] werden für die Implementierung von SPL zwei grundlegende Vorgehensweisen genannt: Annotations- und Kompositionsansätze. Annotationsansätze markieren Quelltext und entfernen diesen vor der eigentlichen Kompilierung. Kompositionsansätze kapseln den Featurequelltext in Einheiten und komponieren diese Einheiten zu einer Variante.

Annotationsansätze können beispielsweise mit einem Präprozessor realisiert werden. In der Programmiersprache C / C++ ist es möglich, mit Hilfe der Präprozessoranweisungen *#ifdef* *#endif* Teile des Programms bedingt zu kompilieren [KR90]. Der nachfolgende Quelltext 2.2 zeigt ein entsprechendes Beispiel. Hierbei handelt es sich um das Feature *Verlauf* aus der zuvor gezeigten Chat-SPL. Dieses Feature könnte beispielsweise so implementiert werden, dass eine Log-Methode aufgerufen wird, die den

Inhalt der Nachricht speichert, bevor diese versendet wird. Wird das Feature *Verlauf* nicht ausgewählt, entfernt der Präprozessor die entsprechenden Programmzeilen (Zeile 5 und Zeile 10) und somit auch das Feature.

```
1
2  public class Nachricht{
3      String inhalt;
4  #ifdef VERLAUF
5      ArrayList verlauf = new ArrayList();
6  #endif
7      //Viele weitere Zeilen
8      public void sendeNachricht(){
9  #ifdef VERLAUF
10         verlauf.add(inhalt);
11 #endif
12         sendeInhalt(inhalt);
13     }
14
15 }
```

Quelltext 2.2: Implementierung des Features Verlauf mit Annotationen

Dieser Ansatz setzt auf bekannte Techniken und findet daher unter Entwicklern eine hohe Akzeptanz. Einfache Präprozessoren sind für fast alle gängigen Programmiersprachen zusätzlich verfügbar. Ein weiterer Vorteil ist, dass ein bestehendes Programm durch Hinzufügen von Annotationen in eine SPL umgewandelt werden kann. In der Forschung wird der Einsatz von Präprozessoren eher als kritisch eingeschätzt [Spe92, KS94]. Schwierigkeiten treten bei quer schneidenden Features auf, da Annotationen für ein Feature in sehr vielen Klassen notwendig sind (Problem des vermischten und verteilten Quelltextes). Des Weiteren kann der Einsatz von Annotationen sehr Fehler anfällig sein. Viele *#ifdef*-Anweisungen machen den Quelltext schwerer lesbar, welches die Wartung der Software erschwert. Beispielsweise können Präprozessor-Anweisungen verschachtelt werden und fein Granulare *#ifdef*-Anweisungen erlauben es sogar einzelnen Wörter (Token) optional zu gestalten.

Abhilfe kann hier mittels so genannter disziplinierter Annotationen geschaffen werden [KA09, KAS10]. Diese schränken die Möglichkeiten der Annotationen ein, schwächen aber die Nachteile entscheidend ab.

Ein weiteres Beispiel für Annotative-Ansätze stellt *CIDE* [KAK08] dar. Hierbei handelt es sich um eine Entwicklungsumgebung, bei der Quelltext, der zu einem Feature gehört, mittels verschiedener Farben markiert werden. Die Farbmarkierungen verändern (verschmutzen) die eigentliche Quelltextbasis nicht. Mit *CIDE* können zusätzlich Feature-Modelle erstellt werden und über ein Auswahlmenü können Features gewählt und Varianten erzeugt werden. Abbildung 2.3 zeigt einen Bildschirmausschnit von CIDE und wie Quelltext mittels Farben markiert wurden.

Zu den Kompositionsansätzen zählen Ansätze, die Komponententechnologie [SGM02] verwenden. Hierbei wird versucht, dass jedes Feature in einer Komponente gekapselt

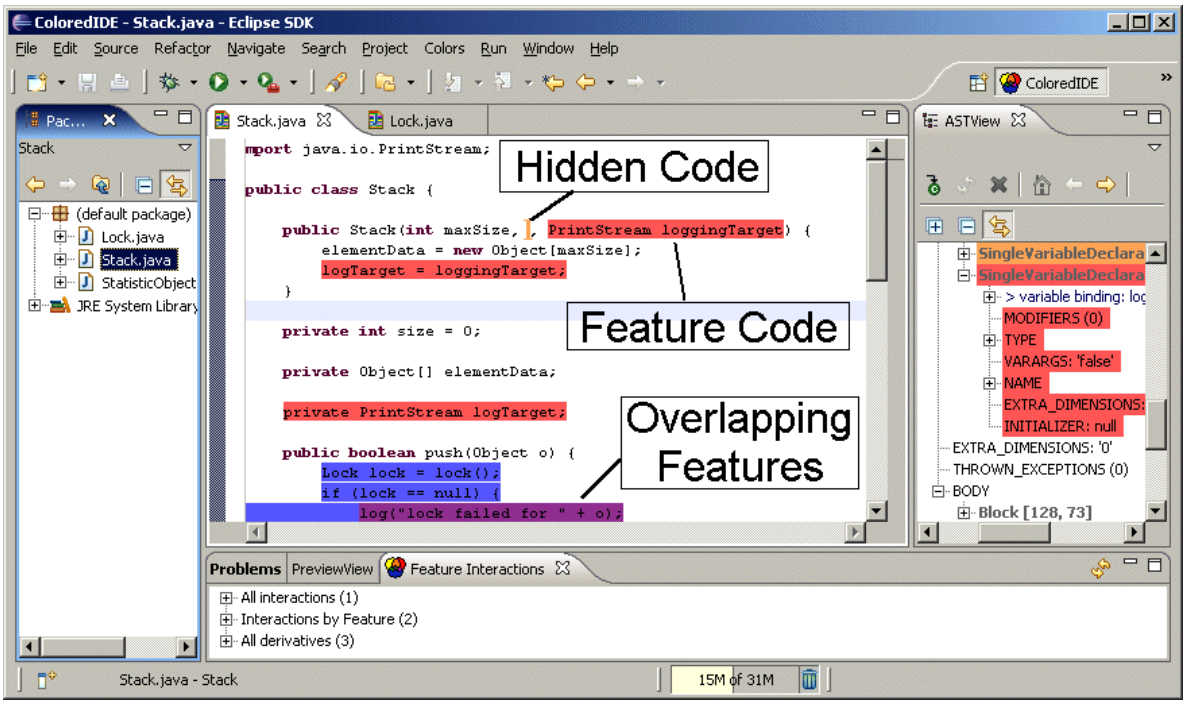

Abbildung 2.3: Entwicklungsumgebung CIDE [KAK08]

wird. Diese Komponenten werden dann zu einer Variante zusammengesetzt. Für das Zusammenspiel mehrerer Komponenten ist zusätzlicher Programmquelltext (Glue-Code) notwendig, der von einem Entwickler an die Auswahl der Features angepasst werden muss. Ein automatisches Generieren der Variante ist damit schwer umzusetzen.

Ebenfalls sind Frameworks [JF88] möglich, um eine SPL zu implementieren. Das Framework bietet Basisfunktionalitäten und wird mittels Plug-ins erweitert. Im Falle einer SPL müssen dann Features in Plug-ins gekapselt werden. Zu den Nachteilen der Frameworks gehört, dass bei der Entwicklung des Frameworks festgelegt wird, an welchen Stellen ein Plug-in Erweiterungen einführen kann (sogenannte Hot-Spots). Müssen neue Hot-Spots eingeführt werden, kann dies dazu führen, dass Änderungen an allen Plug-ins notwendig sind. Des Weiteren können Frameworks sehr umfangreich werden, was die Entwicklung von Plug-ins erschwert. Bei Komponenten und Frameworks besteht auch weiterhin das Problem, dass es unter Umständen nicht immer möglich ist, ein Feature in genau einer Komponente oder Plug-in zu modularisieren.

Die Aspekt-orientierte Programmierung [KLM+97, KHH+01] geht genau dieses Problem an und bietet sich daher für die Implementierung von SPL an. Hierbei beschreiben Aspekte an welcher Stelle im Quelltext zusätzlicher Programmcode ausgeführt werden soll. Aspekte sind sehr mächtig und bieten viele Sprachkonstrukte und Ausdrucksmöglichkeiten an. Dies wird teilweise aber auch als kritisch betrachtet, da dadurch die Syntax komplizierter wird, was in der Praxis zu einer erhöhten Fehleranfälligkeit führen kann [Ste06]. Für SPL kann ein Aspekt den Quelltext eines Features enthalten.

Dies kann im Umkehrschluss dazu führen, dass Aspekte sehr groß und unübersichtlich werden.

Ebenfalls zu den Kompositionsansätzen gehört die Feature-orientiere Programmierung. Aufgrund dessen, dass im Rahmen dieser Arbeit ein nativer feature-orientierter Compiler entwickelt werden soll, wird auf die FOP im nächsten Abschnitt gesondert eingegangen.

2.2 Feature-orientierte Programmierung

Die Feature-orientierte Programmierung (FOP) [Pre97, BSR03] ist ein Programmierparadigma, eignet sich für die Implementierung von SPL und zählt zu den Kompositionsansätzen. Bei der FOP wird der Quelltext, der zu einem Feature gehört, in einem Modul modularisiert. Dies führt dazu, dass die Abbildung von einem Feature auf die entsprechende Implementierungseinheiten eine 1:1 Abbildung ist, siehe Abbildung 2.4. Dies stellt für die Implementierung einen Vorteil dar. Quelltext, der zu einem Feature gehört, kann so leicht gefunden, modifiziert und gewartet werden. Damit wird das Problem des vermischten und des verstreuten Quelltextes minimiert.

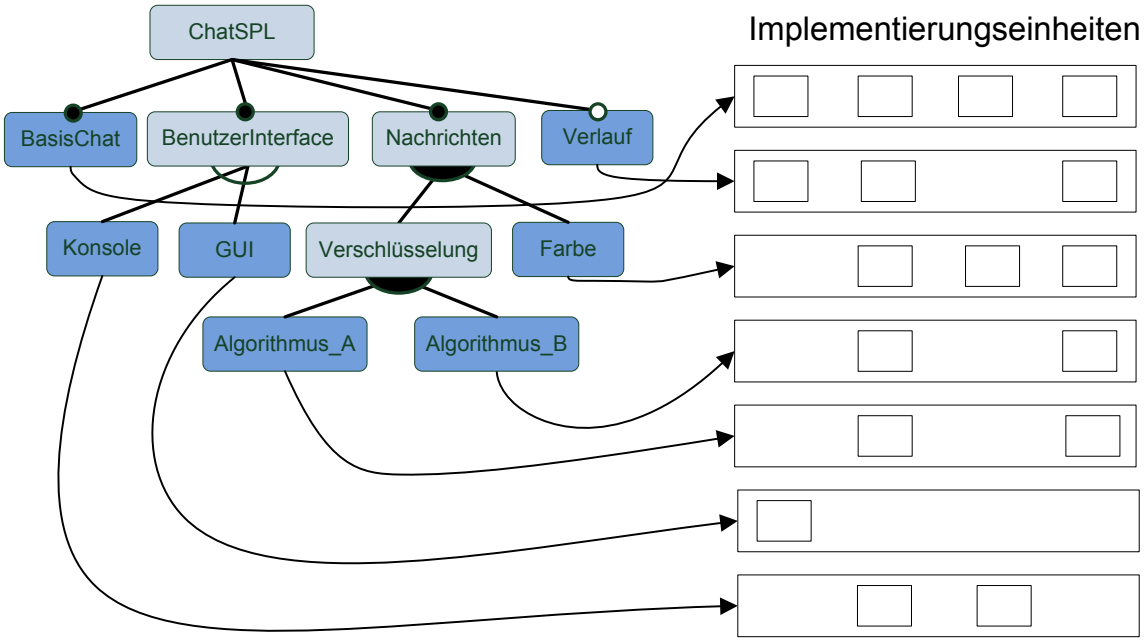

Abbildung 2.4: Abbildung der Features auf Implementierungseinheiten [KS09]

Bei der OOP und der Aufteilung in Klassen handelt es sich um ein bekanntes und etabliertes Konzept. FOP nutzt das objektorientierte Paradigma als Grundstruktur um auch hier eine höhere Akzeptanz zu schaffen und die Vorteile der OOP zu nutzen. Zuvor wurde aber gezeigt, dass mit OOP eine 1:1 Abbildung von Features auf Implementierungseinheiten nicht möglich ist, da Features häufig von mehr als einer Klasse implementiert werden, oder dass eine Klasse mehr als ein Feature implementiert. Aus diesem

Grund erweitert die FOP die OOP um die Möglichkeit Klassen aufzuteilen, damit eine 1:1 Abbildung möglich wird. Wird ein Feature von mehreren Klassen implementiert, nennt sich die Menge der Klassen *Kollaboration*. Der Teil einer Klasse, der ein Feature implementiert, nennt sich *Rolle*. Eine Klasse kann mehrere Rollen in unterschiedlichen Kollaborationen spielen. In Abbildung 2.5 wird dieser Zusammenhang grafisch dargestellt. Die Klasse *Server* spielt Rollen in den Features *BasisChat* und *Verlauf*. Das Features *GUI* besteht aus der Kollaboration der Klassen *Client* und *UserInterface*.

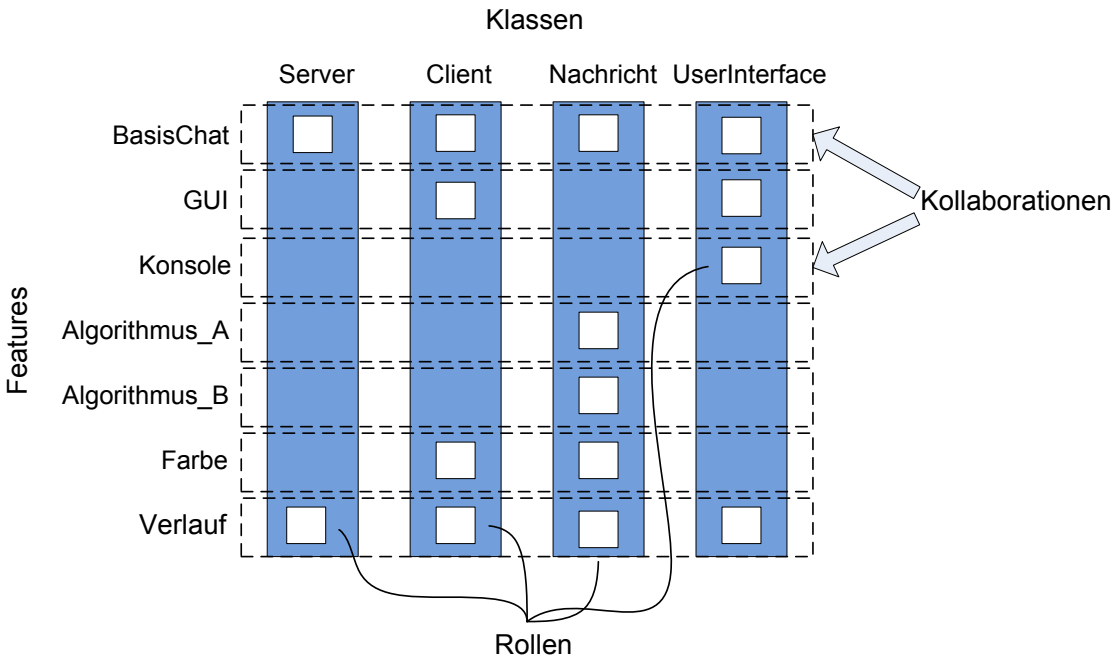

Abbildung 2.5: Kollaborationsdiagramm der Chat-SPL

In den nachfolgenden Abschnitten werden mit AHEAD und FeatureHouse zwei Vertreter der FOP vorgestellt. Diese beiden Vertreter sind recht weit verbreitet und es existieren einige Beispielprojekte. Als Teil dieser Arbeit soll ein nativer FOP-Compiler erstellt werden, der kompatibel zu AHEAD und FeatureHouse sein soll. Aus diesem Grund wird in den nächsten beiden Abschnitten recht detailliert auf die beiden Umsetzungen eingegangen. Daneben gibt es noch weitere Vertreter der FOP, auf die in dieser Arbeit nicht näher eingegangen wird. Für die Programmiersprache C++ gibt es beispielsweise FeatureC++ [ALRS05] und für XML existiert mit Xak [ADT07] eine entsprechende Umsetzung.

2.2.1 AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) wird von Batory et al. entwickelt. Dabei handelt es sich um ein Paket von Programmen für die FOP. AHEAD verwendet eine *schrittweise Verfeinerung (Step-wise Refinement)* [BSR03] zur Umsetzung der FOP. Eine Basis-Implementierung wird dabei schrittweise um weitere Funktionen erweitert. Übertragen auf SPL wird eine Basisimplementierung schrittweise

um weitere Features erweitert. AHEAD verwendet dabei modulare Blöcke, die jeweils die Quelltexte eines Features enthalten. Das Kompositionsprogramm (engl. Composer) von AHEAD erzeugt aus diesen Blöcken eine Variante der SPL. AHEAD kann als Basis-Sprache *Java* verwenden und erweitert diese um neue Schlüsselwörter und Sprachkonstrukte.

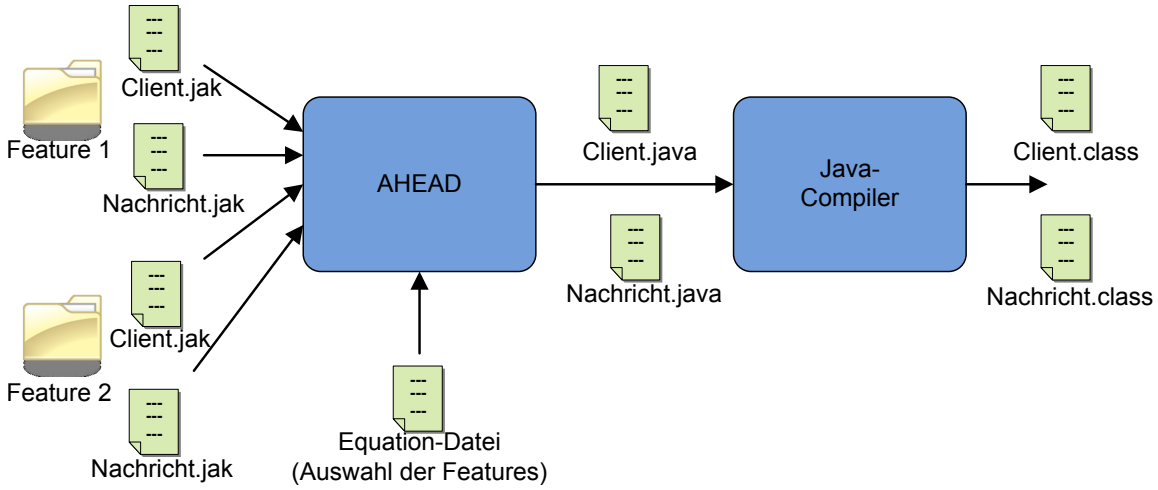

Abbildung 2.6: Umsetzung der FOP mit AHEAD

Abbildung 2.6 zeigt den grundsätzlichen Ablauf vom Feature-orientierten Quelltext zum Bytecode. AHEAD verwendet für jedes Feature einen Ordner. In diesem Ordner befinden sich die Quellen, die das Feature implementieren. Die Dateien verwenden als Endung *.jak*. In der *Equation-Datei* wird die Auswahl der Features und die Reihenfolge, in der die Features komponiert werden, festgelegt. Die Bedeutung der Reihenfolge der Featurekomposition wird in Abschnitt 2.2.3 erläutert. Nach der Komposition gibt AHEAD nativen Java-Quelltext aus. Dieser kann von einem Standard Java-Compiler übersetzt werden. Wie zuvor beschrieben, baut AHEAD auf Java auf und erweitert diese um neue Sprachkonstrukte und Schlüsselwörter. Diese erweiterte Java-Sprache heißt Jak, was die Kurzform von Jakarta ist. Die zusätzlichen Schlüsselwörter sind: *refines, layer und Super*. Die Verwendung der neuen Sprachkonstrukte wird anhand eines Beispiels im Quelltext 2.3, Quelltext 2.4 und Quelltext 2.5 gezeigt. Das Beispiel besteht aus zwei Features, die von AHEAD eingelesen und zu einer Klasse komponiert werden.

Die Komposition erfolgt mit Hilfe eines sogenannten Mixin-Ansatzes [SB02, KAL07]. Bei diesem Ansatz wird die schrittweise Verfeinerung mit Hilfe einer Klassenhierarchie umgesetzt. Das Ergebnis der Featurekomposition mit dem Mixin-Ansatzes befindet sich im Quelltext 2.5.

AHEAD unterstützt zusätzlich einen Jampack-Ansatz, der im Rahmen dieser Arbeit nicht näher betrachtet wird. Die Standardeinstellung bei AHEAD stellt der Mixin-Ansatz dar und Publikation zu AHEAD beziehen sich hauptsächlich auf die Umsetzung durch den Mixin-Ansatz. Des Weiteren wird mit FeatureHouse ein Vertreter vorgestellt, der einen Jampack-Ähnlichen Ansatz verwendet. In [KAL07] werden beide Ansätze verglichen.

```
1  layer Basis;
2
3  public class Nachricht{
4      Client client;
5
6      public void sendeNachricht(String inhalt){
7          //Einige Zeilen der Implementierung
8          sendeInhalt(inhalt);
9      }
10     //Viele weitere Zeilen
11 }
```

Quelltext 2.3: Basisimplementierung mit AHEAD

```
1  layer Verlauf;
2  import java.util.*;
3
4  public refines class Nachricht{
5      ArrayList verlauf = new ArrayList();
6
7      public void sendeNachricht(String inhalt){
8          verlauf.add(inhalt);
9          Super(String).sendeNachricht(inhalt);
10     }
11 }
```

Quelltext 2.4: Klassenverfeinerung der Klasse Nachricht mit AHEAD

Anhand dieses Beispiels können die neuen Sprachkonstrukte diskutiert werden. Das Schlüsselwort *layer* verwendet AHEAD, um eine Datei einem Feature zuordnen zu können und wird aus implementierungstechnischen Gründen benötigt.

Die Klassenverfeinerung erlaubt es, in bestehende Klassen neue Methoden und Felder einzufügen oder bestehende Methoden zu verändern. Mit dem Schlüsselwort *refines* wird eine solche Klassenverfeinerung beschrieben. Hinter dem Schlüsselwort steht dann der Name der Klasse, die verfeinert werden soll. Der Rest der Verfeinerung verhält sich dann wie eine normale Java-Klasse und kann entsprechend Felder, Methoden und Konstruktoren enthalten. Wie in dem Beispiel zuvor gezeigt wurde, wird bei der Komposition der Inhalt der originalen Klasse zu einer abstrakten Klasse umgewandelt. Der Inhalt der verfeinerten Klasse wird in eine Klasse verschoben, die von der abstrakten Klasse erbt. Gibt es nun in beiden Klassen eine Methode, die die gleiche Signatur hat, wird beim Aufruf der Methode jeweils die letzte Verfeinerung ausgeführt. Um nun auf die Methoden der original Klasse zugreifen zu können wird das Schlüsselwort *Super* verwendet. Hinter dem Schlüsselwort wird zunächst eine Liste der Typen der Parameter angegeben (hierbei handelt es sich ebenfalls um eine implementierungstechnische Notwendigkeit). Hinter dieser Liste steht dann die Methode, die aufgerufen werden soll. Aus dem Ergebnis wird deutlich, dass AHEAD das *Super* durch ein *super* ersetzt und sich somit die Klassenhierarchie zu nutzen macht.

```
1  package ChatSPL;
2  import java.util.*;
3
4  abstract class Nachricht$$Basis {
5      Client client;
6      public void sendeNachricht(String inhalt){
7          //Einige Zeilen der Implementierung
8          sendeInhalt(inhalt);
9      }
10 }
11
12 public class Nachricht extends  Nachricht$$Basis  {
13     ArrayList verlauf = new ArrayList();
14
15     public void sendeNachricht(String inhalt){
16         verlauf.add(inhalt);
17         super.sendeNachricht(inhalt);
18     }
19 }
```

Quelltext 2.5: Kompositions beider Klassen mit AHEAD

Neben Methodenverfeinerungen, bietet AHEAD zusätzlich noch eine Konstruktorverfeinerung an. Hierfür wird wieder das Schlüsselwort *refines* verwendet. Innerhalb einer Klassenverfeinerung kann ein `refines Kontruktor()` stehen. Der Inhalt dieser Konstruktorverfeinerung wird dann an das Ende des originalen Kontruktors gehangen.

Neben Klassen können auch Interfaces verfeinert werden. Die Verfeinerungen verhalten sich genauso wie Klassenverfeinerungen, natürlich mit den Einschränkungen, die ein Interface mit sich bringt.

Neben diesen Erweiterungen, die die FOP möglich machen, bringt die aktuelle Version der AHEAD Implementierung einige Einschränkungen mit sich.

- AHEAD unterstützt keine Packages. Besonders bei größeren Projekten fehlt somit ein wichtiges Werkzeug zur Modularisierung.
- AHEAD unterstützt nur Java 1.4. Somit sind neue Sprachkonstrukte, wie die neue For-Schleife und Generics nicht möglich.

Neben Java als objektorientierte Programmiersprache unterstützt AHEAD weitere Sprachen, wie zum Beispiel die Bali-Grammatik [Sar03] und XML-Dateien. Diese sind für die Arbeiten nicht weiter von Bedeutung und es wird auf die Internetseite von AHEAD [1] verwiesen.

2.2.2 FeatureHouse

FeatureHouse [AKL09] verwendet, im Gegensatz zu AHEAD, keine neuen Schlüsselwörter sondern setzt auf eine Formalisierung, die von Apel et al. [ALMK08] entwickelt

[1] http://userweb.cs.utexas.edu/~schwartz/ATS.html

wurde. Diese Formalisierung dient dazu, die Gemeinsamkeiten der unterschiedlichen Techniken zur Implementierung von Rollen zu analysieren und zu diskutieren. Die Formalisierung zeigt auch, dass das Feature-orientierte Paradigma nicht nur auf einzelne Sprachen angewandt, sondern für viele Softwareartefakte genutzt werden kann (Prinzip der Uniformität). Zu einer SPL können nicht nur Quelltexte gehören, sondern auch Dokumentationen, in Form von Text- oder HTML-Dateien und Grammatiken, die ebenfalls durch Features erweitert und verfeinert werden können.

Der erste Schritt der Formalisierung ist, dass ein Feature mit einem anderen Feature zu einem komplexeren Feature komponiert wird. Ein Programm besteht aus einer Menge von komponierten Features. Den nächsten Schritt bilden die sogenannten Feature Structure Trees (FSTs). Ein Feature besteht aus einem Quelltextartefakt, und kann in eine Baumstruktur überführt werden. Diese Bäume weisen die wesentliche Struktur des Artefaktes auf. Im Falle von Java können dies Pakete (Packages), Klassen, Felder und Methoden sein. Auf Informationen wie beispielsweise Initialwerte wird verzichtet.

In der Abbildung 2.7 wird die Klasse *Nachricht* als FSTs gezeigt. Jeder Knoten hat dabei einen Namen und einen Typ. Die Komposition stellt nun eine von der Wurzel beginnende Überlagerung (Superimpositions) von zwei FSTs dar (siehe Abbildung 2.7). Bis zu diesem Punkt ist dieser Ansatz komplett sprachunabhängig. Viele Programmiersprachen lassen sich in so eine Baumstruktur überführen und Bäume können rekursiv überlagert werden. FeatureHouse unterstützt eine Vielzahl von Sprachen, hierzu zählen: Java, C#, C, Haskell, JavaCC und XML. Dies zeigt, dass sich der FOP Ansatz bei FeatureHouse nicht nur auf Objekt-orientierte Sprache anwenden lässt, sondern auch auf imperative und funktionale Programmiersprachen und auf Auszeichnungsprachen wie XML.

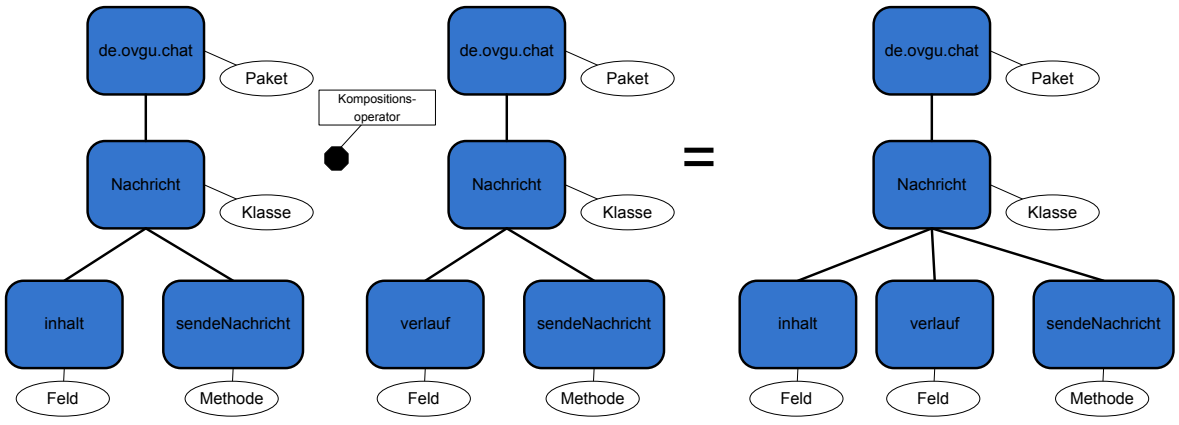

Abbildung 2.7: Überlagerung von Feature structured Tree

Haben zwei Knoten den gleichen Namen und Typ werden sie überlagert. Anschließend werden die Kinder rekursiv überlagert. Werden zwei Knoten auf Grund unterschiedlicher Typen oder Namen nicht überlagert so werden beide Knoten in den Zielbaum eingefügt. Für einige Knoten ist eine Überlagerung nicht trivial und es sind Kompositionsregeln notwendig. Dies betrifft zum Beispiel Knoten vom Typ Methoden oder

Felder. In der Formalisierung werden solche Knoten als *Terminalknoten* bezeichnet. In [AKL09] werden für Java beispielsweise einige Kompositionsregeln gezeigt:

- Zwei Methoden können komponiert werden, in dem die Methode aus dem ersten Baum von der Methode aus dem zweiten Baum überschrieben wird. Eine andere Möglichkeit ist die Verwendung eines `original()`-Aufrufes (ähnlich dem Super-Aufruf in AHEAD), der die original Implementierung der Methode ausführt. In der momentanen Version von FeatureHouse wird dies mittels Umbenennen und Aufrufen der alten Methode umgesetzt (siehe Quelltext 2.8). Der eigentliche Ansatz von FeatureHouse setzt auf sogenanntes Inlining [DA99], dabei wird der Inhalt der originalen Methode an die Stelle des `original()` eingefügt.
- Zwei Felder werden komponiert, wenn nur ein Feld einen Initialwert besitzt.
- Implementierte Interfaces werden übernommen und doppelte Einträge werden eliminiert.
- Eine Klasse, von der mittels *extends* geerbt wird, wird übernommen, sollte die andere Klasse von keiner anderen Klasse erben (keine Mehrfachvererbung)
- Die Liste der geworfenen Exception einer Methode wird übernommen und doppelte Einträge werden eliminiert.
- Die Liste der Import-Deklarationen wird übernommen und doppelte Einträge werden eliminiert.

Im Quelltext 2.6, Quelltext 2.7 und Quelltext 2.8 wird das gleiche Beispiel wie bei AHEAD gezeigt, diesmal aber mit dem Ergebnis, das FeatureHouse liefert. Anstelle einer Klassenhierarchie wird bei FeatureHouse ein Jampack-Ansatz [SB02, KAL07] verwendet. Das Ergebnis ist eine Java-Klasse in der alle Methoden und Felder untergebracht sind. Die Methode `sendeNachricht()` aus der ersten Klasse wird umbenannt in `sendeNachricht__wrappee__Basis()` und der `orignal()`-Aufruf wird in den Methodenaufruf `sendeNachricht__wrappee__Basis(inhalt);` umgewandelt. Die Ausgabe kann dann mit einem Java-Compiler übersetzt werden.

```
1 public class Nachricht {
2     String inhalt;
3     public void sendeNachricht(){
4         //Einige Zeilen der Implementierung
5         sendeInhalt(inhalt);
6     }
7 }
```

Quelltext 2.6: Basisimplementierung mit FeatureHouse

```
1 import java.util.*;
2 public class Nachricht {
3     ArrayList verlauf = new ArrayList();
4
5     public void sendeNachricht(){
6         verlauf.add(inhalt);
7         original(inhalt);
8     }
9 }
```

Quelltext 2.7: Klassenverfeinerung der Klasse Nachricht mit

```
1  import java.util.*;
2  public   class  Nachricht {
3      String inhalt;
4
5      public void sendeNachricht__wrappee__Basis(){
6          sendeInhalt(inhalt);
7      }
8
9      public void sendeNachricht(){
10         verlauf.add(inhalt);
11         sendeNachricht__wrappee__Basis(inhalt);
12     }
13
14     ArrayList verlauf = new ArrayList();
15 }
```

Quelltext 2.8: Komposition beider Klassen mit FeatureHouse

2.2.3 Reihenfolge der Featurekomposition

Ein weiterer Punkt, der bei der Komposition der Features berücksichtigt werden muss, ist die Reihenfolge in der die Features komponiert werden. Dies gilt für AHEAD und für FeatureHouse. Die Formalisierung, die im Abschnitt zuvor vorgestellt wurde, zeigt, dass die Komposition nicht kommutativ ist [ALMK08]. Ein Feature A komponiert mit dem Feature B, muss somit nicht das gleiche Ergebnis liefern, wie die Komposition von Feature B mit Feature A. Ein kleines Beispiel lässt diesen Zusammenhang leicht erkennen (vgl. Quelltext 2.9). Ein Feature Basis wird mit den beiden Features A und B komponiert. Je nach Reihenfolge der Features A und B erhält man als Ausgabe, „Rot Blau Text“ oder „Blau Rot Text“.

Damit eine eindeutige Reihenfolge festgelegt ist, verwendet AHEAD und FeatureHouse eine sogenannte *Expression*- oder *Equation*-Datei. In dieser Datei wird neben der Auswahl der Features auch die Reihenfolge festgelegt.

2.3 Compiler

In dem Abschnitt zuvor wurden die Grundlagen der FOP diskutiert und mit AHEAD und FeatureHouse wurden zwei mögliche Umsetzungen gezeigt. In diesem Abschnitt

```
1  public class Farbe { //Feature Base
2      public void male(){
3          System.out.println("Text");
4      }
5  }
6
7  public class Farbe { //Feature A
8      public void male(){
9          System.out.println("Blau ");
10         original();
11     }
12 }
13
14 public class Farbe { // Feature B
15     public void male(){
16         System.out.println("Rot ");
17         orignal();
18     }
19 }
```

Quelltext 2.9: Beispiel für die Reihenfolge der Features

werden Grundlagen über Compiler (dt. Übersetzer) und Compilerbau vorgestellt, die für die Implementierung eines nativen FOP-Compilers benötigt werden. Das Compiler-Framework JastAdd, das im Abschnitt 2.4.1 präsentiert wird, baut auf diesen Grundlagen auf. In Abbildung 2.6 wurde schon einmal der Begriff Compiler verwendet. Der Java-Compiler liest den Quelltext ein und wandelt ihn in eine andere Sprache um (in diesem Fall in Bytecode). Für die Entwicklung oder die Erweiterung eines Compilers ist nun ein detaillierter Blick auf die „Blackbox“ Compiler notwendig.

Intern wird der Compiler in zwei Bereiche eingeteilt, dem sogenannten *Front-End* und *Back-End* [SLA08]. Der interne Aufbau wird in Abbildung 2.8 dargestellt. Im Front-End findet die Analyse und eine Vorverarbeitung des eingelesenen Quelltextes statt. Dabei wird der Quelltext in eine andere Darstellungsform (Syntax-Bäume , Drei-Adress-Befehle, usw. [GE99]) transformiert, die für die weitere Verarbeitung besser geeignet ist. Im Back-End kann der Quelltext auf verschiedene Arten optimiert und am Ende in die Zielsprache transformiert werden. In Abbildung 2.8 wird ein typischer Aufbau eines Compilers dargestellt. Je nach Verfasser oder in konkreten Implementierungen können einige Blöcke zusammengefasst oder in dieser Form gar nicht vorhanden sein. Beispielsweise gibt es *scannerless parsing*, wobei dann ein Scanner und ein Parser nicht in separater Form vorhanden , sondern in einer Einheit zusammengefasst sind [Vis97].

Als erstes liest der Scanner (auch unter dem Begriff Lexer zu finden) den Quelltext ein. Die Hauptaufgabe des Scanners ist es, dass Programm in logisch zusammengehörige Einheiten (so genannte *Tokens*) zu zerteilen. Der Scanner erkennt hierbei Schlüsselwörter, Bezeichner, Operatoren und Konstanten, diese werden in die Symboltabelle eingetragen. Zusätzlich werden Leerzeichen und Kommentare entfernt.

Die nächsten Schritte werden von einem Parser erledigt. Der Parser arbeitet mit den vom Scanner erstellten Tokens. Aufgabe des Parsers ist, eine syntaktische Analyse des

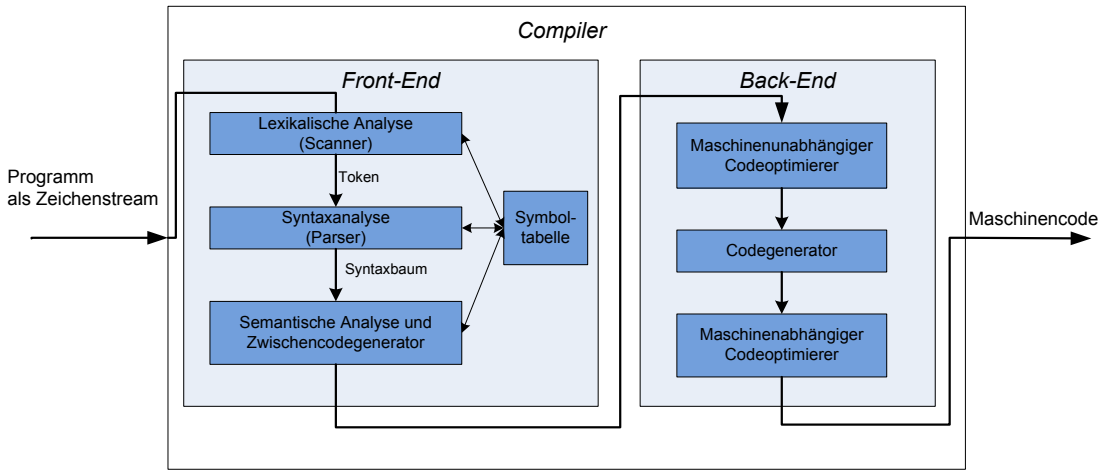

Abbildung 2.8: Aufbau eines Compilers

Programms und die Überführung in eine Darstellungsform, die zur weiteren Analyse besser geeignet ist. Grundlage hierfür bilden kontextfreie Grammatiken, die die Sprache beschreiben.

Im Quelltext 2.10 befindet sich eine Grammatik für eine Teilmenge von Java-Anweisungen. Die Pfeile werden als „kann die Form haben“ interpretiert. Eine solche Regel wird als *Produktion* bezeichnet. Schlüsselwörter wie *if*, *while* und die Klammern werden als Terminale bezeichnet. Die Variablen wie *stmt* oder *expr* stellen wiederum Sequenzen von Terminalen dar und werden als Nichtterminale bezeichnet.

Die Grammatik im Quelltext 2.10 beschreibt, dass eine Anweisung (statement) entweder die Form einer Zuweisung (Zeile 1), einer if-Anweisung (Zeile 2- 3), einer Schleife (do-while, und while) (Zeile 4-5) oder weiteren Anweisungen haben kann (Zeile 6). Die Zeilen 7 und 8 beschreiben, dass Anweisungen wiederum aus weiteren Anweisungen bestehen kann, oder keine Anweisungen enthalten (das *e* steht für den leeren String). Es existieren noch eine Vielzahl unterschiedlicher Grammatiken [GJ06], die für diese Arbeit aber nicht von Bedeutung sind.

```
1 stmt -> id = expression ;
2     |   if ( expression ) stmt
3     |   if ( expression ) stmt else stmt
4     |   while ( expression ) stmt
5     |   do stmt while ( expression ) ;
6     |   { stmts }
7 stmts ->    stmts stmt
8     |   e
```

Quelltext 2.10: Grammatik für eine Teilmenge von Java-Anweisungen

Wie zuvor beschrieben, eignet sich zur syntaktischen Analyse eine Zwischendarstellung besser, als der eigentliche Quelltext. Abstrakte Syntaxbäume stellen eine solche Zwischendarstellung dar. Für diese Arbeit wird der gebräuchliche englische Begriff *Abstract Syntax Tree (AST)* verwendet. Ein Ausdruck wird in einen AST überführt, dabei stellt

jeder innere Knoten einen Operator und die Kinder die Operanden dar. Als Beispiel kann der Ausdruck if (inhalt == 'Hallo') counter = counter +1; in die Baumstruktur überführt werden, die in Abbildung 2.9 dargestellt wird.

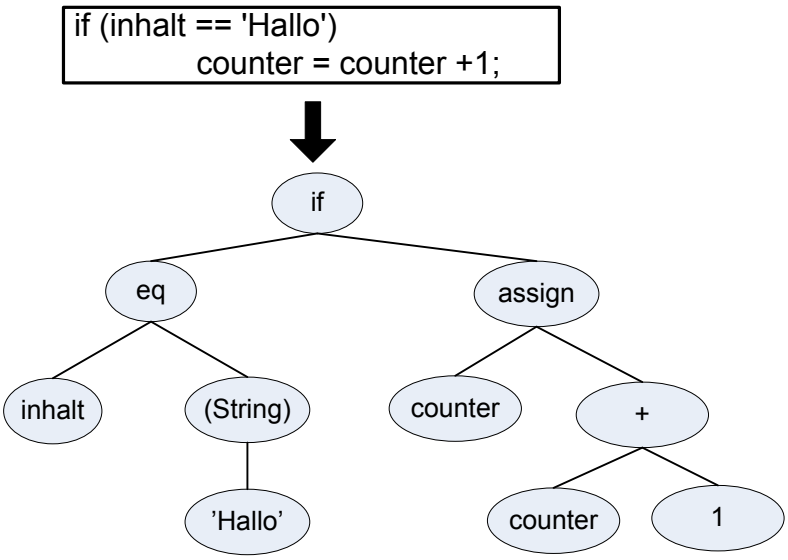

Abbildung 2.9: Transformation von Quelltext in eine Baumstruktur

Stand der Technik und der Wissenschaft sind heute Generatoren für Parser und Scanner. Diese werden mittels Konfigurationsdateien für die jeweilige Sprache konfiguriert und erzeugen leistungsfähige Scanner und Parser. Aus diesem Grund wird auf den internen Aufbau und die genaue Funktionsweise nicht näher eingegangen. Für die Implementierung des nativen FOP-Compilers werden solche Generatoren für den Scanner und Parser verwendet.

Im Anschluss wird eine semantische Analyse des eingelesenden Quelltextes durchgeführt. Hierbei kann beispielsweise überprüft werden, ob eine Variable deklariert wurde, bevor sie verwendet wurde. Zusätzlich kann bei einer Variablenzuweisung überprüft werden, ob der Wert vom Typ kompatibel zu der Variablen ist, hierfür werden Typ-Systeme verwendet [Pie02]. Bei der FOP kann beispielsweise überprüft werden, ob zu einer Klassenverfeinerung eine Klasse existiert, die verfeinert werden kann.

Zur Optimierung kann eine weitere Zwischendarstellung des Quelltextes besser geeignet sein. Hierfür kann der Quelltext beispielsweise in 3-Adress-Befehle umgewandelt werden [GE99].

Im Back-End findet zunächst eine maschinenunabhängige Codeoptimierung statt. Hierbei wird überprüft, ob Variablen eventuell nicht benötigt werden, oder ob Methodenaufrufe durch Inlining ersetzt werden können. Im Anschluss daran wird der eigentliche Maschinencode für die Zielmaschine erzeugt und in einem weiteren Schritt optimiert. Bei dieser Optimierung wird die Rechnerarchitektur des Zielsystems berücksichtigt, zum Beispiel den Einsatz der Register.

Damit ist ein möglicher Aufbau eines Compilers beschrieben. Soll ein bestehender Compiler erweitert werden, sind Änderungen an vielen Stellen des Compilers notwendig.

Compiler-Frameworks, die im nächsten Abschnitt vorgestellt werden, helfen dem Entwickler leichter Änderungen vorzunehmen. Da im Rahmen dieser Arbeit ein Compiler-Framework verwendet wird, welches das Back-End bereitstellt, wurde dieses Thema nur sehr oberflächlich behandelt.

2.4 Compiler-Frameworks

Ein Compiler besteht aus vielen komplexen Einheiten, die das eingelesende Programm analysieren, optimieren und in eine andere Sprache übersetzen. Soll ein bestehender Compiler erweitert werden, stellt dies keine triviale Aufgabe dar, da Änderungen an vielen Stellen im Compiler notwendig sind [HM03]. Des Weiteren können solche Änderungen leicht zu Fehlern führen. Aus diesem Grund sind Compiler-Frameworks entwickelt worden, die dem Entwickler bei der Neuerstellung und Erweiterung bestehender Compiler helfen sollen.

Im Rahmen dieser Arbeit soll, wie bereits erläutert, ein nativer FOP-Compiler entwickelt werden, der kompatibel zu AHEAD und FeatureHouse ist. Da es sich hierbei um eine erweiterte Java Sprache handelt, muss dem entsprechend auch ein Compiler-Framework für Java ausgewählt werden. Zu den bekannteren Vertretern gehören die folgenden Frameworks:

- JastAdd [Ekm06, EH07b]
- Polyglot [NCM03]
- Jaco [ZO01]

Jaco schied aus, weil unter anderem auf der Internetseite keine ausführliche Dokumentation vorhanden ist und das Framework nur Java 1.4 unterstützt und somit nicht sehr Zukunftsfähig ist.

Die Wahl fiel auf JastAdd, da die letzte Version von Polyglot von 14. August 2008 stammt und die Internetseite von JastAdd einen aktuellen Eindruck macht. Des Weiteren bietet JastAdd eine sehr ausführliche Dokumentation und Beispielprojekte an.

Mit den Möglichkeiten des JastAdd Frameworks wurde ein Java 1.4 Compiler auf die Java Version 1.5 erweitert. Dies zeigt eindrucksvoll die Erweiterungsmöglichkeiten. Das Framework erlaubt es neue AST-Knoten zu erstellen und diese mittels Aspekten zu erweitern.

Die Wahl auf JastAdd fiel in einem frühen Stadium dieser Arbeit, so dass die anderen Frameworks nicht weiter betrachtet wurden. Im nachfolgenden Abschnitt wird die Funktionsweise von JastAdd näher betrachtet.

2.4.1 JastAdd

Compiler-Frameworks sollen den Entwickler dabei unterstützen vorhandene Sprachen zu erweitern oder Compiler für neue Sprachen zu erstellen. Mit JastAdd steht ein auf Java-basiertes Framework zu Verfügung, das für diese Zwecke genutzt werden kann. Im Rahmen dieser Arbeit soll kein Compiler für eine neue Sprache entwickelt, sondern die Sprache Java erweitert werden. Aus diesem Grund wird nicht JastAdd in seiner Grundform verwendet, sondern der JastAdd Extensible Java Compiler (JastAddJ)-Compiler [EH07a]. Hierbei handelt es sich um einen erweiterbaren Java 1.5 Compiler, der mit den Möglichkeiten des Frameworks JastAdd erweitert werden kann. In diesem Abschnitt wird ein Überblick, über die Erweiterbarkeit des Frameworks gegeben. Eine detaillierte Vorstellung des Frameworks anhand der Sprache PicoJava findet sich in Anhang B.
Die Erweiterungsmöglichkeiten des JastAddJ-Compilers lassen sich in vier Gruppen einteilen.

- Verwendung von java-basierten Scanner- und Parsergeneratoren
- Hinzufügen neuer AST-Knoten
- Neue Methoden und Felder mittels Aspekt-Technologie in bestehende AST-Knoten einfügen
- Vorhandene Knoten durch eine *Rewritable Reference Attributed Grammar* [EH04] verändern

Durch die Verwendung von Java-basierten Scanner- und Parsergeneratoren setzt JastAdd auf bestehende Generatoren. Im JastAddJ-Compiler wird als Scannergenerator JFlex [2] verwendet. In einer Konfigurationsdatei werden beispielsweise Schlüsselwörter oder der Aufbau von Kommentaren der Programmiersprache Java beschrieben.

Als Parsergenerator wird Beaver [3] verwendet. In einer entsprechenden Datei wird die Java-Grammatik mit ihren Produktionsregeln beschrieben. Die Produktionsregeln haben einen ähnlichen Aufbau, wie die, die im Quelltext 2.10 gezeigten werden. Zusätzlich enthalten sie Anweisungen zum Erzeugen der AST-Knoten. Im Quelltext 2.11 wird ein Beispiel gezeigt, das die Produktionsregel für die leere Anweisung darstellt. Diese Produktionsregel beschreibt, dass die Anweisung nur aus einem Semikolon besteht. In den geschwungenen Klammern befindet sich der Konstruktoraufruf für den entsprechenden AST-Knoten.

```
1 EmptyStmt empty_statement =
2     SEMICOLON                                {: return new EmptyStmt(); :}  ;
```

Quelltext 2.11: Produktionsregel für eine leere Anweisung

Sollen neuen Sprachkonstrukte hinzugefügt werden, sind in vielen Fällen auch neue Knoten im AST notwendig. JastAdd verwendet hier eine abstrakte Grammatik, die die

[2] http://jflex.de/
[3] http://beaver.sourceforge.net/

Knoten beschreibt. Die Beschreibung eines Knotens enthält den Namen, von welchem anderen Knoten geerbt wird und welche Knoten als Kinder existieren. Aus dieser Beschreibung erzeugt das Framework eine Java-Klasse, die den AST-Knoten repräsentiert. Die Klasse enthält die passenden Konstruktoren und Felder. Methoden zum Erreichen der Felder oder Setzen neuer Kinder werden ebenfalls generiert. Die zuvor gezeigte leere Anweisung wird durch folgenden Ausdruck beschrieben: `EmptyStmt : Stmt;`. Der Doppelpunkt drückt aus, dass die leere Anweisung von dem Knoten *Stmt* erbt.

Das Framework JastAdd bietet zwei weitere Werkzeuge an, die es erlauben, neue Funktionalitäten in bestehende Knoten einzufügen. Diese Funktionalitäten können in Aspekten imperativ und deklarativ beschrieben werden.

Deklarative Aspekte beschreiben den Zielzustand einer Variablen. Des Weiteren lassen sich dadurch Werte im AST nach oben oder nach unten propagieren [Ekm06]. Ein gutes Beispiel stellt die PrettyPrint-Methode dar, die aus dem AST wieder den Quelltext als Zeichenkette erstellt. Der deklarative Aspekt beschreibt, dass es eine Zeichenkettenvariable gibt, und für die unterschiedlichen AST-Knoten wird angegeben, wie dieser Wert zu berechnen ist. Im Rahmen dieser Arbeit wurde von deklarativen Aspekten wenig Gebrauch gemacht und sie werden daher an dieser Stelle nicht tiefer behandelt. Für weitere Informationen sei auf die Internetseite von JastAdd [4] verwiesen.

Ein imperativer Aspekt fügt einem Knoten neue Methoden und Felder hinzu. Der Aspekt besteht aus normalen Java-Quelltext, der in die entsprechenden Klassen hinzugefügt wird. Zusätzlich besteht die Möglichkeit, Methoden aus anderen Aspekten zu überschreiben. Hierfür wird das Schlüsselwort *refine* verwendet. Die originale Methode kann durch einen speziellen Aufruf aufgerufen werden.

Die imperativen Aspekte wurden im Rahmen dieser Arbeit häufig verwendet und das Beispiel in Quelltext 2.12 soll die Funktionsweise darstellen. Der Aspekt *A* fügt der Klasse *Block* ein Feld *counter* und eine Methode *showDescription* hinzu. Aspekt *B* nutzt das Schlüsselwort *refine* um die bestehende Methode aus Aspekt *A* zu überschreiben. In Zeile 12 wird die originale Implementierung der Methode *showDescription* aufgerufen.

Mit der *Rewritable Reference Attributed Grammar* können bestehende Knoten durch andere ersetzt werden. Beispielsweise erzeugt der Parser einen Knoten für einen allgemeinen Methodenaufruf. Durch passende Regeln kann dann entschieden werden, ob es sich um einen statischen- oder virtuellen Methodenaufruf handelt. Das Framework ersetzt den allgemeinen Knoten durch einen spezialisierten Knoten.

Für die FOP werden neue Sprachkonstrukte in die Sprache Java integriert. Für das Verständnis, an welchen Stellen bei der Implementierung, die im Kapitel 4 gezeigt werden, neue Knoten eingefügt werden, ist ein schematischer Aufbau und Hierarchie des Java-ASTs im JastAddJ-Framework notwendig. Die komplette Java-Grammatik besteht aus fast tausend Zeilen mit Produktionen und dreihundert verschiedenen AST-Knoten. Ein vollständiger Überblick würde den Rahmen dieser Arbeit sprengen und auch nicht dem Verständnis dienen. Aus diesem Grund wird in Abbildung 2.10 eine stark vereinfachte Hierarchie gezeigt, die alle notwendigen Knoten enthält, die für diese Arbeit von Bedeutung sind.

[4] http://jastadd.org/jastadd-reference-manual/

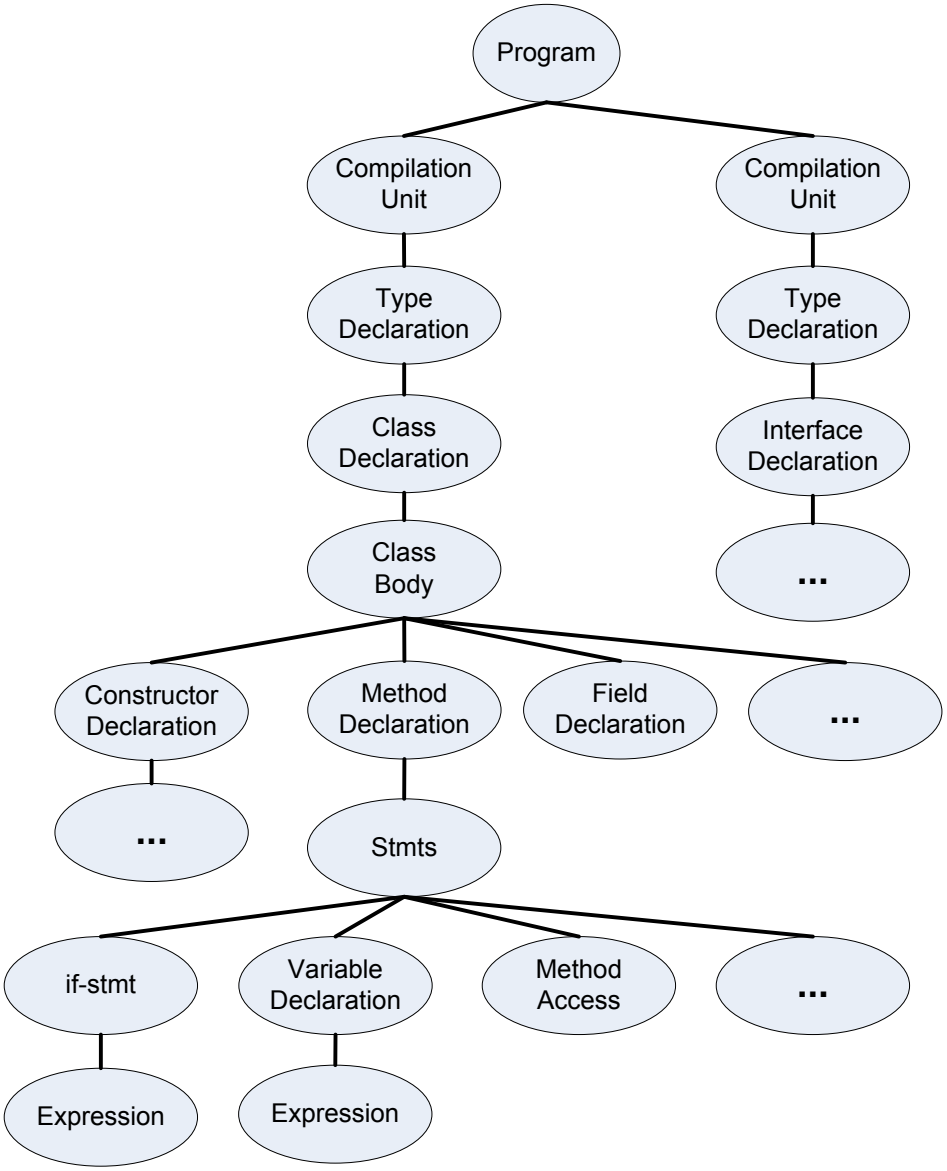

Abbildung 2.10: Vereinfachte Hierarchie des Java-ASTs

```
1  aspect A{
2      public int Block.counter;
3
4      public void Block.showDescription(){
5          System.out.println("Block");
6      }
7  }
8
9  aspect B {
10     refine A void Block.showDescription ( ) {
11         System.out.println("Ein verbesserter ");
12         A.Block.showDescription();
13     }
14 }
```

Quelltext 2.12: Imperativer Aspekt zum Einfügen neuer Methoden und Felder

Der oberste Knoten stellt die Klasse *Program* dar. Darunter befinden sich Knoten vom Typ *CompilationUnit*. Jeder CompilationUnit-Knoten repräsentiert eine eingelesende Quelltext-Datei, beziehungsweise eine kompilierbare Einheit. Diese kann wiederum Knoten enthalten, die als *TypeDeclaration* bezeichnet werden. Ein Knoten *TypeDeclaration* besteht wiederum aus Klassendefinitionen oder Interfaces. Klassen und Interfaces haben jeweils einen Klassen- oder Interfacekörper (Body), in dem dann Konstruktoren, Methoden oder Felder enthalten sind. Methoden bestehen aus Anweisungen (Statements), zum Beispiel if-Anweisungen, Schleifen oder Variablendeklarationen. Die nächst tiefere Ebene bilden Ausdrücke (Expressions). Hierzu gehören Vergleiche, Zuweisungen oder Boolsche Ausdrücke. Damit ist die Hierarchie der Knoten in sehr vereinfachter Form dargestellt. Im realen AST befinden sich noch viele weitere Knoten, die hier aus Gründen der Übersichtlichkeit weg gelassen wurden. Hierzu zählen beispielsweise Knoten vom Typ *Liste*, die aus implementierungstechnischen Gründen enthalten sind.

In diesem Abschnitt wurden die Möglichkeiten des Frameworks JastAddJ beschrieben. Mit Hilfe von Scanner- und Parsergeneratoren können neue Sprachkonstrukte eingefügt oder bestehende Sprachen erweitert werden. Der Parser erzeugt einen AST aus Klassen, die mit dem JastAdd Framework erstellt wurden. Die Knoten werden mittels einer abstrakten Grammatik beschrieben. Mit Hilfe von Aspekten können die Knoten verfeinert werden und mit der *Rewritable Reference Attributed Grammar* können bestehende Knoten durch andere oder spezialisierte Knoten ersetzt werden.

JastAdd zeichnet sich dadurch aus, dass die zuvor beschrieben Erweiterungsmöglichkeiten modular in einzelnen Dateien und diese in einem separaten Erweiterungsordner gespeichert werden. Alle Erweiterungen, die beispielsweise für die AHEAD-Sprachkonstrukte verwendet werden, können so modular implementiert werden. Eine Datei mit dem Namen *AHEAD.parser* beschreibt dann alle notwendigen Änderungen am Parser. Gleiches gilt für neue AST-Knoten oder Aspekte. Durch diese Vorgehensweise bleibt auch der erweiterte Java-Compiler weiterhin gut erweiterbar.

Damit sind die Grundlagen des Frameworks beschrieben, damit im Kapitel 4 die Implementierung eines nativen FOP-Compilers vorgestellt werden kann. Zunächst werden im

nächsten Abschnitt die Vorteile diskutiert, die durch den Einsatz eines solchen Compilers entstehen.

3. FOP Fehlererkennung und Erweiterungsmöglichkeiten

Die FOP kann bei der Programmiersprache Java Klassen- oder Interfaceverfeinerungen verwenden, um bestehende Klassen oder Interfaces durch neue Methoden oder Felder zu erweitern. Dazu verwendet AHEAD neue Sprachkonstrukte und FeatureHouse setzt auf eine Formalisierung und Überlagerung von Klassen oder Interfaces. Neben Programmierfehlern, die die Sprache Java betreffen, entstehen durch die Verfeinerungen neue potentielle Fehlerquellen, die erkannt werden müssen. In dem ersten Abschnitt dieses Kapitels werden einige spezifische Fehler gezeigt, die bei der FOP auftreten können. Diese Fehler dienen im darauf folgenden Abschnitt 3.2 dazu, zu belegen, wo es bei AHEAD und FeatureHouse Probleme und Verbesserungspotential gibt. Im Abschnitt 3.5 wird gezeigt, wie ein nativer FOP-Compiler diese Probleme lösen kann und welche weiteren Vorteile entstehen können.

3.1 FOP spezifische Fehler

Durch die Erweiterung bestehender Programmiersprachen um die Möglichkeit der Klassenverfeinerung, entstehen neue potentielle Fehlerquellen, die zusätzliche Überprüfungen benötigen. Dabei muss es sich nicht unbedingt um direkte Fehler handeln, sondern es kann auf ein problematisches oder fehleranfälliges Design hinweisen, das verbessert werden sollte. Thaker et al. beschreiben dies mit dem Ausdruck *design that „smells bad“* [TBKC07, FBB+99]. Die nachfolgenden Abschnitte zeigen einige solcher Designs und Fehler.

FOP Syntaxfehler

Verfeinerungen von Klassen können durch neue Sprachkonstrukte beschrieben werden. Dabei besteht die Möglichkeit, dass die neuen Sprachkonstrukte Syntaxfehler enthalten.

Es muss überprüft werden, ob eine Klassenverfeinerung richtig aufgebaut ist, beispielsweise `refines class Client` anstelle von `class refines Client`. Des Weiteren dürfen Methodenaufrufe zu verfeinerten Methoden (`Super()`-Aufrufe) nur innerhalb von Methoden verwendet werden, die in Klassenverfeinerungen stehen. Neue Schlüsselwörter dürfen nur an den vorgesehenen Stellen verwendet werden.

Leerlaufende Klassenverfeinerungen

Bei einer Klassenverfeinerung kann der Fall auftreten, dass die verfeinerte Klasse gar nicht existiert oder erst später, durch ein anderes Feature, eingeführt wird. Die Klassenverfeinerung läuft entsprechend ins Leere. Dies deutet darauf hin, dass entweder das Feature-Modell oder die Implementierung fehlerhaft ist.

Bei einem fehlerhaften Feature-Modell kann beispielsweise das Feature, das die Klasse einführen soll, nicht mit dem Feature der Klassenverfeinerung ausgewählt werden. Die Implementierung kann fehlerhaft sein wenn ein falscher Klassenname verwendet wird. In beiden Fällen sollte der Entwickler durch eine Fehlermeldung informiert werden [TBKC07].

In dem Quelltext 3.1 wird ein Beispiel zu den leerlaufenden Klassenverfeinerung gezeigt. Die Klassenverfeinerung in Feature A läuft ins Leere, da die Klasse *Nachricht* erst mit dem Feature B eingeführt wird.

```
1 layer A;
2 public refines class Nachricht{
3     /*...*/
4 }
5
6 layer B;
7 public class Nachricht{
8     /*...*/
9 }
```

Quelltext 3.1: Klassenverfeinerung ohne originale Klasse

Mehrfaches Einfügen von Klassen / Interfaces

Ein beliebiges Feature kann eine Klasse neu einführen. Wurde eine Klasse mit dem gleichen Namen zuvor durch ein anderes Feature eingeführt, wird je nach Kompositionsverfahren die alte Klasse überschrieben. Hierbei handelt es sich nicht direkt um einen Fehler und es kann vom Entwickler beabsichtigt sein. Thaker et al. beschreiben das mehrfache Einführen als mögliche Fehlerquelle oder als schlechtes Design [TBKC07]. Das Problem des mehrfachen Einführens von Klassen ist nicht nur FOP spezifisch sondern tritt beispielsweise auch bei der Aspekt-orientierten Programmierung [AGMO06] auf.

Zum Beispiel kann eine Klasse von mehreren Features verfeinert und dann anschließend von einem Feature durch eine komplett andere Klasse ersetzt werden. Im Quelltext 3.2

wird ein entsprechendes Beispiel gezeigt. Die Klasse *Nachricht* wird von dem Feature A eingeführt und von dem Feature B verfeinert. Feature C führt die Klasse *Nachricht* neu ein und überschreibt die vorher gegangenen Ergebnisse. Da stellt sich die Frage, ob die Features A und B überhaupt in Kombination mit Feature C ausgewählt werden dürfen. In so einem Fall ist eine Anpassung des Featuremodells sinnvoller [TBKC07].

```
1  layer A;
2
3  public class Nachricht{
4      int i;
5  }
6
7  layer B;
8
9  public refines class Nachricht{
10     String inhalt;
11 }
12 layer C;
13
14 public class Nachricht{
15     Client absender;
16 }
```

Quelltext 3.2: Mehrfaches Einfügen von Klassen

Komposition von Feldern

Wird ein Feld in eine bestehende Klasse eingefügt, in der ein solches Feld schon existiert, können, in Abhängigkeit der Initialwerte, Fehler bei der Featurekomposition entstehen. Je nach Kompositionsalgorithmus können eventuell Initialwerte übernommen werden. Besitzen beide Felder einen Initialwert, sollte ein Fehler erzeugt werden [AKL09].

In Quelltext 3.3 wird ein Beispiel für die Komposition von Feldern gezeigt, bei dem durch Feature A ein Feld i eingeführt und in der Methode *print* auf den Wert 23 gesetzt wird. In dem Feature B wird ebenfalls ein Feld i eingeführt und mit dem Wert 5 initialisiert. Zusätzlich wird die Methode *print* verfeinert, die die originale Methode aufruft und davor und danach die Variable i ausgibt.

Typfehler

Typfehler entstehen beispielsweise wenn einem Feld ein Wert oder Objekt zugewiesen wird, welches vom Datentyp nicht kompatibel ist.

Im Quelltext 3.4 wird ein Typfehler gezeigt, der durch die FOP entstehen kann. Das Feature A führt die Klasse *Nachricht* mit der Methode *empfangeNachricht* ein. Diese Methode hat als Rückgabewert eine Zeichenkette. Das Feature B verfeinert diese Methode und ruft die originale Methode mit einem `Super()`-Aufruf auf. Der Rückgabewert der originalen Methode wird in dem Integerfeld *status* gespeichert werden. Dies führt zu einem Typfehler, da der Rückgabewert vom Typ nicht kompatibel zu dem Integerfeld ist. Aufgabe des Typsystems ist es solche Fehler zu erkennen.

```
1  layer A;
2  public class Nachricht{
3      int i;
4
5      public void print(){
6          i =  23;
7      }
8  }
9
10 layer B;
11 public refines class Nachricht{
12     int i = 5;
13
14     public void print(){
15         System.out.println(i);
16         Super().print();
17         System.out.println(i);
18     }
19 }
```

Quelltext 3.3: Beispiel zur Komposition von zwei Feldern

```
1  layer A;
2  public class Nachricht{
3
4      public String empfangeNachricht(){
5       /*...*/
6       return "Foo";
7      }
8  }
9
10 layer B;
11 public refines class Nachricht{
12     int status;
13
14     public String empfangeNachricht(){
15         status = Super().empfangeNachricht();
16     }
17 }
```

Quelltext 3.4: Beispiel für einen Typfehler bei der FOP

Eine andere Art von Typfehler kann entstehen, wenn Methoden, Felder oder Objekte verwendet werden, die durch ein anderes Feature eingeführt werden. Wird das entsprechende Feature nicht ausgewählt, treten Fehler auf.

Quelltext 3.5 zeigt ein Beispiel, bei der in Feature C ein Feld und eine Methode in die Klasse *Nachricht* eingeführt werden. Feature B verwendet das Feld und die Methode. Es wird deutlich, dass Feature B nur verwendet werden kann, wenn Feature C ebenfalls ausgewählt ist. In diesem einfachen Beispiel ist dies noch sehr leicht zu erkennen. In Pro-

jekten, die aus vielen Features bestehen, kann ein Entwickler möglicherweise übersehen, dass die Implementierung der Features von einander Abhängig ist.

Diese Art von Fehlern lässt sich vermeiden, in dem das Prinzip der schrittweisen Verfeinerung beachtet wird und Referenzen sich nur auf zuvor gemachten Verfeinerungen beziehen [Wir71, LHBL06]. Für die Typsicherheit von Feature Featherweight Java wird ebenfalls vorgeschlagen, dass Felder und Methoden nur Referenzen zu Features haben dürfen, die zuvor eingeführt worden sind [AKL08].

In dem zuvor gezeigten Beispiel handelt es sich um einen Fehler, der das Prinzip der schrittweisen Verfeinerung missachtet. Da diese Art von Fehlern vom Typsystem erkannt werden können, sollen sie im Rahmen dieser Betrachtung als Typfehler behandelt werden.

```
1  //Feature1;
2  public class Nachricht{
3  }
4
5  //Feature2
6  public class refines Nachricht
7      public void send{
8          i++;
9          log();
10     }
11 }
12
13 //Feature3;
14 public class refines Nachricht{
15     int i;
16     public void log(){
17     //...
18     }
19 }
```

Quelltext 3.5: Typsicherheit bei drei Features

Typfehler sollen nur für die erstellte Variante erkannt werden. Typsicherheit für die gesamte Produktlinie stellt ein weites Forschungsgebiet dar, und soll hier nicht behandelt werden. Es wird auf die entsprechende Literatur verwiesen, zum Beispiel [HZS05, CP06, TBKC07, KA08, Thü10, AKGL10].

Methodenverfeinerung

In AHEAD und FeatureHouse bestehen die Möglichkeiten, neue Methoden einzuführen oder bestehende Methode zu verfeinern. Aus einer verfeinerten Methode kann auf die originale Methode mittels eines originalen Methodenaufruf zugegriffen werden. Dieser wird bei AHEAD mit `Super()` und FeatureHouse mit `original()` realisiert und darf nur innerhalb einer Methode stehen, die eine andere Methode verfeinert.

Durch eine fehlerhafte Reihenfolge bei der Featurekomposition kann es passieren, dass eine Methodenverfeinerung ins Leere läuft, in dem beispielsweise die verfeinerte Me-

thode erst mit einem späteren Feature eingeführt wird. Oder die Implementierung ist fehlerhaft, was durch eine falsch geschriebene Methodensignatur passieren kann.

Der nachfolgende Quelltext 3.6 zeigt ein entsprechendes Beispiel. Das Feature A führt eine Methode *log* ein. Feature B verfeinert die Methode print, die aber erst durch das Feature C eingeführt wird. Der originale Methodenaufruf innerhalb der *print*-Methode läuft ins Leere. In dem Feature C soll eigentlich die Methode *log* verfeinert werden, doch durch einen Fehler bei der Implementierung wird die Methode *logger* verfeinert, die nicht in der Klasse Nachricht vorhanden ist.

```
1  layer A;
2  public class Nachricht{
3      public void log(){...    }
4  }
5
6  layer B;
7  public refines class Nachricht{
8      public void print(){
9          Super().print();
10     }
11 }
12
13 layer C;
14 public refines class Nachricht{
15     public void print(){...}
16
17     public void logger() {
18         Super().logger();
19     }
20 }
```

Quelltext 3.6: Beispiel zur Methodenverfeinerung

Damit sind einige der FOP spezifische Fehler beschrieben. Im nächsten Abschnitt wird gezeigt, wie die bestehenden Ansätze von AHEAD und FeatureHouse diese Fehler erkennen.

3.2 Fehlererkennung bei den bestehende Konzepte

Im Kapitel 2 wurden mit AHEAD und FeatureHouse zwei Ansätze für die FOP vorgestellt. Bei diesen Ansätzen wird der feature-orientierte Quelltext von den Kompositionsprogrammen in eine nativen Sprache übersetzt, die dann von dem entsprechenden Compiler übersetzt werden kann. In Falle von AHEAD besteht der feature-orientierte Quelltext aus einer erweiterten Java-Sprache und wird in eine Zwischendarstellung, die aus nativen Java-Quelltext besteht, umgewandelt (siehe Abschnitt 2.2.1 und Abschnitt 2.2.2). Dieser wird dann von einem Java-Compiler in Bytecode übersetzt.

In diesem Abschnitt wird nun vorgestellt, wie AHEAD und FeatureHouse die spezifischen FOP-Fehler erkennen. Damit wird gezeigt, wo es bei den bisherigen Ansätze Probleme und Verbesserungspotential gibt.

Softwareprodukte unterliegen einem kontinuierlichen Prozess der Entwicklung. Die hier gezeigten Ergebnisse beruhen auf der zurzeit aktuellen Version von FeatureHouse (Version vom 27.04.2010) und AHEAD (Version vom 19.02.2010). Zukünftige Versionen könnten andere Ergebnisse liefern.

3.2.1 AHEAD

FOP Syntaxfehler

AHEAD erkennt bei der Komposition Syntaxfehler und gibt eine Fehlermeldung aus, in der der Name der Datei und die Zeilennummer enthalten ist. Die Fehlermeldung bietet zusätzlich Informationen darüber, ob an der Fehlerstelle ein Schlüsselwort oder ein Bezeichner erwartet wird. Wird ein `Super()`-Aufruf innerhalb einer Methode verwendet, die sich in einer normalen Klasse befindet, gibt AHEAD eine Warnung aus und löscht den entsprechenden Aufruf.

Leerlaufende Klassenverfeinerungen

Wird eine Klassenverfeinerung komponiert, ohne dass die entsprechende Klasse existiert, gibt die aktuelle Version von AHEAD eine Fehlermeldung aus. In älteren Versionen wurde aus der Klassenverfeinerung auf Grund des Mixin-Ansatzes eine normale Klasse erzeugt.

Mehrfaches Einfügen von Klassen / Interfaces

AHEAD verwendet einen Mixin-Ansatz zur Umsetzung der Klassenverfeinerung. Dieser Mixin-Ansatz führt aber dazu, dass wenn eine Klasse mehrfach eingeführt wird, nur die letzte Definition der Klasse existiert. AHEAD erzeugt eine Warnung, wenn eine bestehende Definition einer Klasse durch eine andere überschrieben wird.

```
1 public class Nachricht{
2     Client absender;
3 }
```

Quelltext 3.7: Ergebnis des mehrfachen Einfügens von Klassen mit AHEAD

Aus dem Beispiel, dass im Quelltext 3.2 gezeigt wird, erzeugt AHEAD den Quelltext 3.7. In der Ausgabe befindet sich nur noch die Klasse, die von Feature C beschrieben wird. AHEAD gibt eine Warnung aus, das die Klasse *Nachricht* durch Feature C überschrieben wird.

Komposition von Feldern

Wie in dem Abschnitt 2.2.1 gezeigt wurde, realisiert AHEAD die Klassenverfeinerung durch einen Mixin-Ansatz, der eine Vererbungshierarchie verwendet. Enthält die Verfeinerung ein Feld, das in der originalen Klassen ebenfalls enthalten ist, entstehen durch die Komposition zwei Klassen, wobei die Verfeinerung von der originalen Klasse erbt.

In beiden Klassen gibt es das entsprechende Feld. Ein solcher Zusammenhang wird als *Variable Shadowing* (siehe Abschnitt 6.3 in [GJSB05]) bezeichnet, und stellt eine potentielle Fehlerquelle dar.

AHEAD erzeugt aus dem Quelltext 3.3 die folgende Klassenhierarchie, die im Quelltext 3.8 gezeigt wird. Es wird deutlich, dass die Variable i in beiden Klassen existiert und somit auch unterschiedliche Werte haben kann. Wird die Methode *print* aufgerufen, erfolgt als Ausgabe „5“ und „5“. Nur innerhalb des Features A hat die Variable i hat.

Dies muss dem Entwickler bei der Programmierung bekannt sein, da zwei Felder mit dem gleichen Name existieren, die innerhalb der Features unterschiedliche Werte haben könne.

```
1  abstract class Nachricht$$A {
2      int i;
3
4      public void print(){
5          i = 23;
6      }
7  }
8
9  public class Nachricht extends  Nachricht$$A  {
10     int i = 5;
11     public void print(){
12         System.out.println(i);
13         super.print();
14         System.out.println(i);
15     }
16 }
```

Quelltext 3.8: Komposition von zwei Felder mit AHEAD

Das Problem, dass unterschiedlichen Features Zugriff auf unterschiedliche Variablen haben, wurde von Apel et al. für verschiedenen FOP-Ansätze untersucht [ALK+09]. Für die FOP werden zusätzliche Zugriffsmodifikatoren vorgeschlagen, die die Sichtbarkeit und den Zugriff auf Variablen zwischen unterschiedlichen Features regeln.

Typfehler

AHEAD prüft die Variante nicht auf Typsicherheit. Aus diesem Grund kann das Kompositionsprogramm von AHEAD auch keine Warnungen / Fehler erzeugen, dass beispielsweise ein Feld nicht deklariert und initialisiert wurde bevor es verwendet wird. Diese Art von Fehlern wird vom Java-Compiler erkannt. Da der Java-Compiler keine Kenntnisse von den zuvor durchgeführten Transformationen hat, beziehen sich diese Typfehlermeldungen auf die Zwischendarstellung, die vom Kompositionsprogramm erzeugt werden. Dieses Problem wird im Abschnitt 3.3 noch einmal näher erläutert.

Methodenverfeinerung

Befindet sich ein originaler Methodenaufruf in einer Methode, die keine andere Methode verfeinert, erzeugt das Kompositionsprogramm keine Fehlermeldung. Wie zuvor im

Abschnitt 2.2.1 beschrieben, wird das große **Super()** durch ein kleines **super()** ersetzt. Der nachgeschaltete Java-Compiler erkennt dann, dass es in der originalen Klasse keine entsprechende Methode gibt, die mit dem super-Aufruf erreicht werden kann.

Der Quelltext 3.9 zeigt die Ausgabe, die AHEAD aus dem Beispiel erzeugt, das in Quelltext 3.6 vorgestellt wurde. Die **Super()**-Aufrufe wurden durch **super()**-Aufrufe ersetzt. Der Java-Compiler erzeugt entsprechende Fehlermeldungen, dass die Methoden *logger* und *print* nicht aufgerufen werden können.

```
1  abstract class Nachricht$$A{
2      public void log(){ /*...*/  }
3  }
4
5  abstract class Nachricht$$B extends  Nachricht$$A {
6      public void print(){
7          super.print();
8      }
9  }
10
11 public class Nachricht extends  Nachricht$$B {
12     public void print(){ /*...*/ }
13
14     public void logger() {
15         super.logger();
16     }
17 }
```

Quelltext 3.9: Ergebnis der Methodenverfeinerung und originalen Methodenaufrufe

Bei AHEAD wird der originale Methodenaufruf durch das Schlüsselwort **Super** und dahinter durch den Methodenaufruf beschrieben. Der Methodenaufruf kann aber nicht nur die verfeinerte Methode, sondern jede Methode in der originalen Klasse beschreiben. Dies führt nicht direkt zu einem Fehler, doch deutet dies eventuell auf einen ungewollten Fehler hin. Möchte der Entwickler auf eine Methode in der originalen Klassen zugreifen kann der Methodenaufruf direkt hin geschrieben werden.

3.2.2 FeatureHouse

In diesem Abschnitt wird gezeigt, wie FeatureHouse die FOP spezifischen Fehler erkennt. Es werden die gleichen Beispiele wie bei AHEAD verwendet, nur dass sie für FeatureHouse entsprechend angepasst sind. Der **Super()**-Aufruf wird durch den **original()**-Aufruf ersetzt und das Schlüsselwört *refines* wird nicht verwendet.

FOP Syntaxfehler

FeatureHouse erkennt bei der Komposition Syntaxfehler und erzeugt eine Fehlermeldung, die den Namen der Datei und die Zeilennummer enthält, an der der Fehler aufgetreten ist. Der **original()**-Aufruf wird bei FeatureHouse nicht als Schlüsselwort realisiert. FeatureHouse überprüft daher nicht, ob ein **original()**-Aufruf nur innerhalb einer Klassenverfeinerung verwendet wird.

Leerlaufende Klassenverfeinerungen

FeatureHouse überlagert FSTs für die Komposition von Features und verwendet keine neuen Sprachkonstrukte. Eine Klassenverfeinerung unterscheidet sich nicht von einer normalen Klasse. In dem Beispiel, aus dem Quelltext 3.1, kann ohne das Schlüsselwort *refines* nicht entschieden werden, ob Feature A Feature B verfeinern sollte oder umgekehrt. Dies ist insofern kritisch, da die Featurekomposition nicht kommutativ ist [ALMK08].

Mehrfaches Einfügen von Klassen / Interfaces

In FeatureHouse ist es nicht möglich, eine bestehende Klassen oder Interface komplett durch eine andere zu ersetzen. Bestehende Klassen werden durch weitere Klassen immer verfeinert. Das Beispiel aus dem Quelltext 3.2 wird von FeatureHouse zu dem Quelltext 3.10 komponiert.

```
1 public class Nachricht{
2         int i;
3         String inhalt;
4         Client absender;
5 }
```

Quelltext 3.10: Ergebnis des mehrfachen Einfügens von Klassen mit FeatureHouse

Thaker et al. beschreiben das mehrfache Einfügen von Klassen als potentielle Fehlerquelle, aus diesem Grund ist der FeatureHouse-Ansatz vorteilhaft, da es nicht möglich ist Klassen zu überschreiben [TBKC07].

Komposition von Feldern

In FeatureHouse werden Felder komponiert und Initialwerte werden übernommen. Das Problem des *Variablen Shadowing* tritt nicht auf. Quelltext 3.11 zeigt die Ausgabe von FeatureHouse, die aus dem Quelltext 3.3 generiert wurde. Die Feld i existiert nur einmal innerhalb der Klasse. Die *print*-Methode liefert die erwartete Ausgabe „5 23“.

```
1  public class Nachricht {
2          int i = 5;
3
4          public void print__wrappee__A(){    i = 23; }
5
6          public void print(){
7              System.out.println(i);
8              print__wrappee__A();
9              System.out.println(i);
10         }
11 }
```

Quelltext 3.11: Komposition von zwei Felder mit FeatureHouse

Haben bei der Komposition von Feldern beide einen Initialwert wird er Initialwert übernommen, der von der letzten Klassenverfeinerung stammt. Dies steht in Widerspruch mit den Kompositionsregeln, die in [AKL09] für FeatureHouse beschrieben werden. Die Kompositionsregel lautet, dass ein Initialwert nur übernommen wird, falls die Variable zuvor noch keinen Initialwert hatte.

In dem Fall, dass beide Felder Initialwerte besitzen, generiert FeatureHouse keine Warnung. Der Entwickler muss bei der Entwicklung wissen oder der Zwischendarstellung entnehmen, welcher Initialwert gesetzt wird.

```
1  //Feature A;
2  public class Nachricht{
3      List liste = new ArrayList();
4
5      public void history(String text){
6          liste.add(text);
7      }
8  }
9  //Feature B;
10 public class Nachricht{
11     List liste = null;
12     //...
13 }
14 /*-------------------------------------------------------------------------*/
15 //Komposition aus Feature A und Feature B
16 public class Nachricht{
17     List liste = null;
18
19     public void history(String text){
20         liste.add(text);
21     }
22 }
```

Quelltext 3.12: Kompositionsproblem von Feldern bei FeatureHouse

Quelltext 3.12 zeigt ein Beispiel für die Problematik, dass der Initialwert der letzten Klassenverfeinerung verwendet wird. Das Feature A enthält ein Feld *liste*, das mit einer ArrayList initialisiert wird. Die Methode *history* speichert in dem Feld *liste* eine Zeichenkette. Das Feature B setzt das Feld *liste* auf *null*. Wird die Methode *history* aufgerufen wird zur Laufzeit eine Null-Pointer Exception geworfen. Das in diesem Beispiel gezeigte Problem, ist kritisch, da es vom Compiler nicht als Fehler erkannt wird, sondern erst zur Laufzeit auftritt. Wird das Feature B zuerst komponiert, tritt dieser Fehler nicht mehr auf.

Typfehler

Diese Art von Fehlern wird bei der Komposition mit FeatureHouse nicht erkannt. Das Typsystem des Java-Compilers erkennt diese Art von Fehlern. Das Problem besteht, wie auch bei AHEAD, dass die Fehlermeldungen für die Zwischendarstellung gelten und erst auf den eigentlich feature-orientierten Quelltext abgebildet werden müssen.

Methodenverfeinerung

FeatureHouse verwendet zum Aufrufen der originalen Methoden ein `original()`-Aufruf. Wird ein `original()`-Aufruf innerhalb einer Methode verwendet, die keine Methode verfeinert, bleibt der `original()`-Aufruf als normaler Methodenaufruf stehen. Der Java-Compiler wird dann eventuell erkennen, dass es keine entsprechende Methode gibt. Im ungünstigen Fall kann es eine Methode mit passender Signatur geben, so dass diese Methode aufgerufen wird und das Programm ein anderes Verhalten aufweist.

Handelt es sich um eine verfeinerte Methode, wird der `original()`-Aufruf durch den entsprechenden Methodenaufruf ersetzt.

3.3 Abbildung von Fehlermeldungen auf den Quelltext

Die bestehenden Konzepte AHEAD und FeatureHouse verwenden ein zweistufiges Verfahren. Der feature-orientierte Quelltext wird von einem Kompositionsprogramm in nativen Quelltext umgewandelt. Dieser wird von einem Standardcompiler übersetzt. Zuvor wurden einige FOP-spezifische Fehler gezeigt und wie die bestehenden Konzepte diese Fehler handhaben. Dabei wurde erwähnt, dass einige Fehler nicht von den Kompositionsprogrammen sondern vom Compiler erkannt werden. Zusätzlich kann der Java-Compiler weitere „normale“ Javafehler erkennen und entsprechende Fehlermeldungen erzeugen.

Der Compiler hat bei diesen zweistufigen Ansätzen keine Kenntnis von den zuvor gemachten Transformationen. Tritt bei Übersetzen ein Fehler auf bezieht sich die Fehlermeldung auf die vom Kompositionsprogramm erstellte Zwischendarstellung.

In Abbildung 3.1 wird die Abbildung der Fehlermeldung auf die entsprechenden Quelltextdateien nochmal grafisch dargestellt. Einige FOP-Fehler werden von AHEAD oder FeatureHouse erkannt und werden direkt auf die passenden Quelltextdateien abgebildet. Werden vom Java-Compiler Fehlermeldungen erzeugt, beziehen sich diese auf die Zwischendarstellung. Die Abbildung von der Zwischendarstellung auf den feature-orientierten Quelltext muss dann von einem Entwickler oder einer Integrated Development Environment (IDE) übernommen werden.

In dem Quelltext 2.5 und Quelltext 2.8 wurden die Zwischendarstellung von AHEAD und FeatureHouse gezeigt. In diesen einfachen Beispiel ist Abbildung der Fehler auf den feature-orientierten Quelltext noch gut realisierbar. Es ist aber sicherlich vorstellbar, dass bei umfangreicheren Programmen, mit tiefen Vererbungshierarchien und vielen Features und Verfeinerungen, eine Abbildung von Fehlermeldung zu zusätzliche Aufwand führt. Hinzu kommt, dass die Zwischendarstellung von der jeweiligen Implementierung von AHEAD und FeatureHouse beeinflusst wird (Methoden werden umbenannt, Vererbungshierarchie, usw.).

Der Entwickler wird dazu gezwungen, sich mit der Zwischendarstellung auseinander zu setzen, um die Fehler und Warnungen auf den feature-orienterten Quelltext abzubilden.

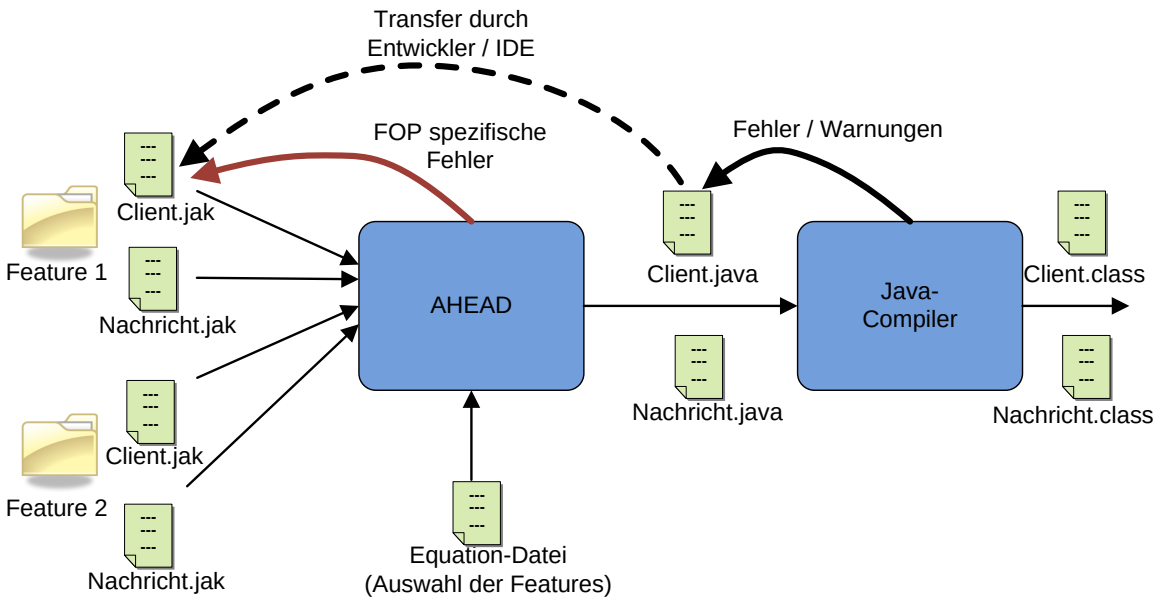

Abbildung 3.1: Abbildungen der Fehlermeldungen auf die Quelltextdateien

Dadurch entsteht ein erhöhter Aufwand, was die Kosten für die Entwicklung und die Wartung erhöht.

In Abbildung 3.1 wird zusätzlich gezeigt, dass die Abbildung der Fehlermeldungen von der Zwischendarstellung auf den feature-orienterten Quelltext von einer IDE erledigt werden kann. Mit FeatureIDE existiert eine Umsetzung, die im nachfolgenden Kapitel kurz vorgestellt wird.

3.3.1 Fehlermeldung in FeatureIDE

Bei FeatureIDE [LAMS05, KTS+09] handelt es sich um ein Eclipse Plug-in, dass die Kompositionsprogramme von AHEAD und FeatureHouse integriert und eine IDE zur Verfügung stellt.

In Abbildung 3.2 werden zwei Bildausschnitte von FeatureIDE gezeigt, die den Feature-Modell Editor und die Featureauswahl vorstellen. Die beiden Ausschnitte sollen einen kleinen Einblick geben, wie eine IDE die FOP durch Werkzeuge unterstützen kann. Mit dem Feature-Modell Editor können Feature-Modelle grafisch erstellt werden. Bei der Auswahl einer gültigen Featurekombination wird der Entwickler mittels grafischer Auswahlmenüs unterstützt. Ein Kollaborationsdiagramm kann ebenfalls angezeigt werden, das eine grafisch Übersicht, über die Verfeinerung der Klassen, darstellt.

FeatureIDE enthält einen Quelltexteditor, der Fehler und Warnungen an den richtigen Stellen im Quelltext anzeigt. Die Zwischendarstellung jeder Klasse enthält Informationen aus welchen Features sie komponiert wurden. Damit kann FeatureIDE feststellen auf welche Quelltextdatei der Fehler abgebildet werden muss. Die richtige Position innerhalb einer Klasse erfolgt mit einer textuellen Suche und dem Abzählen von Pro-

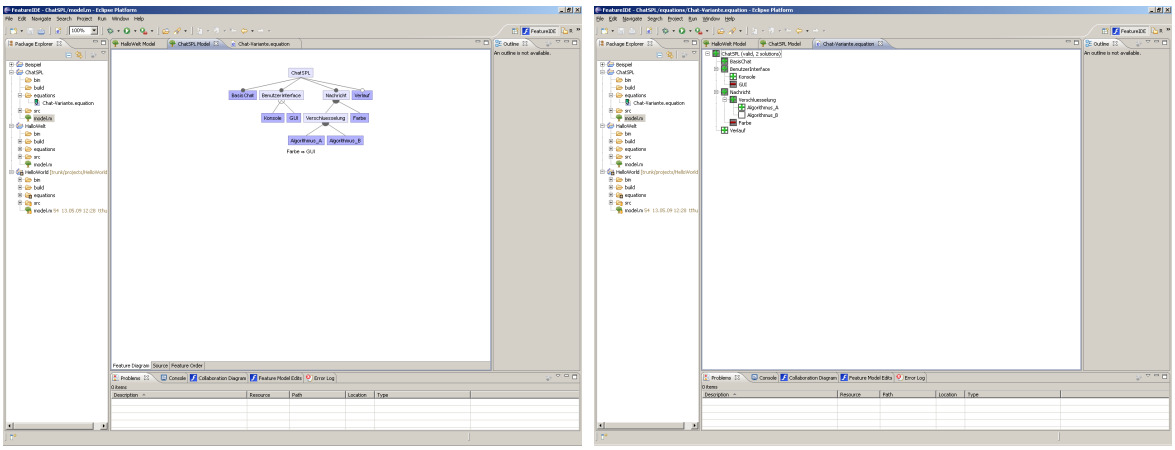

Abbildung 3.2: Zwei Auschnitte von FeatureIDE

grammzeilen. Tritt beispielsweise in der Zwischendarstellung der Fehler in Methode *A* in der vierten Zeile auf, wird diese Zeile im feature-orientierten Quelltext gesucht.

Dieser Ansatz liefert aus praktischer Sicht gute Ergebnisse, doch lassen sich Situationen finden, in der die Fehlermeldungen nicht an die richtigen Stelle gesetzt werden. In der Abbildung 3.3 wurde vor der Layer-Anweisung ein Kommentar eingefügt, was bei Quelltextdateien durchaus üblich sein kann. Dies führt dazu, dass die richtige Zeile für die Fehleranzeige nicht richtig berechnet werden kann, da durch den Kommentar die Zwischendarstellung verändert wird. Durch den einzeiligen Kommentar verschiebt sich die Zwischendarstellung um eine Zeile, was dazu führt, dass der Fehler eine Zeile zu tief angezeigt wird.

Dieses Beispiel soll die Leistungsfähigkeit von FeatureIDE nicht negativ bewerten. Die Suche kann angepasst werden und eventuell wurde dies in einer neueren Version von FeatureIDE schon erledigt. Das Beispiel soll aber zeigen, dass diese Art der Umsetzung der Abbildung von Fehlermeldungen gute Ergebnisse liefern kann, aber das Ganze auf textueller Suche und Abzählen von Programmzeilen beruht. Dies kann anfällig für Fehler sein. Ändert sich die Zwischendarstellung, auf Grund einer neueren Version des Kompositionsprogramms, muss auch die Abbildung der Fehlermeldungen angepasst werden.

3.4 Zusammenfassung der Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Fehlererkennung von AHEAD und FeatureHouse, bei den zuvor gezeigten FOP spezifischen Fehler, zusammengefasst. Dazu bietet die Tabelle 3.1 einen Überblick der Ergebnisse. FeatureIDE verwendet als Kompositionsprogramm AHEAD oder FeatureHouse, so dass die Ergebnisse bei der Fehlererkennung übernommen wurden.

FOP spezifische Syntaxfehler werden von AHEAD und FeatureHouse erkannt und passende Fehlermeldungen werden erzeugt.

Leerlaufende Klassenverfeinerugen werden in der aktuellen Version von AHEAD erkannt. Bei FeatureHouse tritt dieser Fehler auf Grund des Konzeptes nicht auf. Es kann

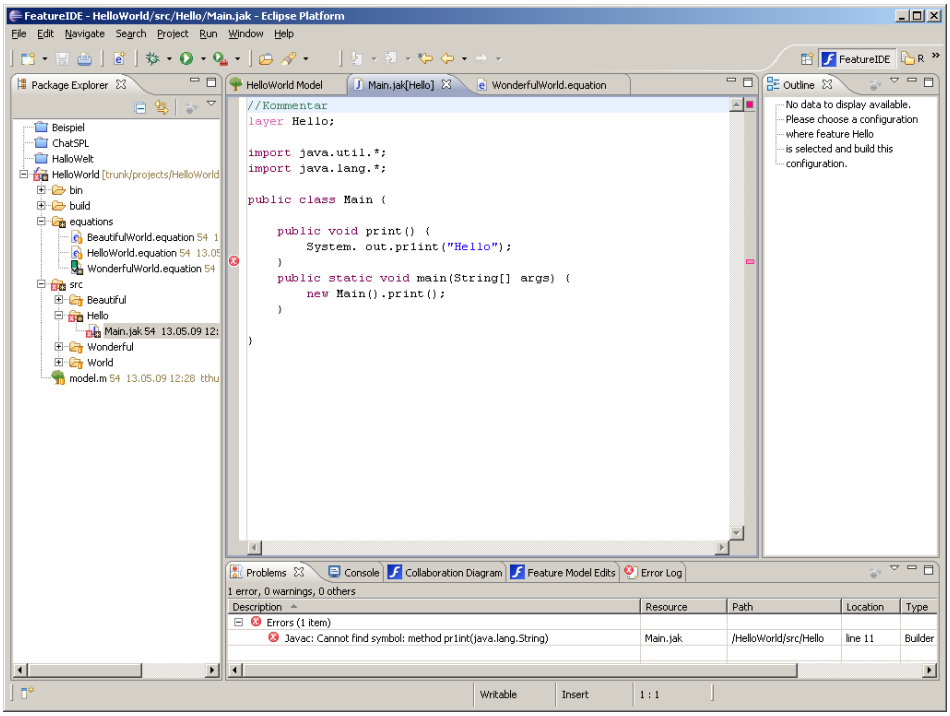

Abbildung 3.3: FeatureIDE: Problem bei der richtigen Anzeige der Fehlermeldung

	AHEAD	FeatureHouse	FeatureIDE
FOP Syntaxfehler	+	+	+/+
Leerlaufende Klassen-verfeinerung	+	o	+/o
Mehrfaches Einführen von Klassen / Interfaces	+	+	+/+
Komposition von Feldern	-	-	-/-
Typfehler	-	-	-/-
Methodenverfeinerung	o	o	o/o
Abbildung der Fehler auf den Quelltext	-	-	+
- = negativ, o = neutral , + = positiv			

Tabelle 3.1: Vergleich der Erkennung FOP spezifischer Fehler mit AHEAD, FeatureHouse und FeatureIDE

aber nicht entschieden werden, ob eine Klasse eine Verfeinerung ist oder nicht. Die Leistung wird als neutral bewertet, da im Vergleich zu AHEAD keine Warnung erzeugt werden.

Mehrfaches Einführen von Klassen ist bei AHEAD möglich. Je nach AHEAD-Version wird eine Fehlermeldung erzeugt. FeatureHouse umgeht diese potentielle Fehlerquelle, da mehrfaches Einführen von Klassen nicht möglich ist. Existieren zwei Klassen mit dem gleichen Namen werden sie zu einer Klasse komponiert.

Die Komposition von Feldern bei AHEAD führt dazu, dass in jedem Feature eine Variable existiert, die unterschiedliche Werte haben können. Dies ist kann zu Fehlern führen und wird daher mit negativ bewertet.

FeatureHouse übernimmt, im Gegensatz zu der beschriebene Formalisierung, den Initialwert des letzten Features und gibt keine Fehlermeldung aus. Daher wird die Felderkomposition mit negativ bewertet.

Typfehler werden weder von AHEAD noch von FeatureHouse erkannt. Der nachfolgende Java-Compiler kann diese Fehler erkennen, dies führt aber direkt zu dem nächsten Problem, der Abbildung der Fehlermeldungen auf den Quelltext. Hierbei muss der Entwickler die Abbildung der Fehlermeldungen, auf den eigentlichen Quelltext, selber durchführen und sich intensiv mit der Zwischendarstellung und den Implementierungsdetails der jeweiligen Kompositionsprogramme auseinander setzen. Auf Grund dieses Zusatzaufwandes wird die Leistung negativ bewertet.

Die originalen Methodenaufrufe können bei AHEAD zu dem Problem führen, dass eine andere Methode aufgerufen wird, als die verfeinerte Methode. Bei FeatureHouse kann ein ähnliches Problem auftreten, wenn ein originaler Methodenaufruf innerhalb einer Methode verwendet wird, der keine andere Methode verfeinert. In diesem Fall wird der originale Methodenaufruf nicht umgewandelt.

Da bei AHEAD und FeatureHouse fehlerhafte Methodenverfeinerungen vom Typsystem erkannt werden, wird die Leistung als neutral bewertet.

Die Abbildung der Fehler auf den Quelltext muss bei AHEAD und FeatureHouse manuell durch den Entwickler erfolgen, was zu zusätzlichem Aufwand führt und daher einen Nachteil darstellt. Hier zeigt FeatureIDE seine Stärke und bildet Fehlermeldungen automatisch auf den richtigen Quelltext ab.

Die Tabelle 3.1 zeigt, dass die bisherigen Ansätze mit dem zweistufigen Verfahren, aus Kompositionsprogramm und Compiler, besonders im Bereich der Abbildung der Fehlermeldungen noch verbessert werden kann. Auch im Bereich der semantischen Überprüfung gibt es Potentiel für Verbesserungen.

Ein nativer FOP-Compiler benötigt kein separates Kompositionsprogramm und kann zusätzliche Fehlerüberprüfungen durchführen. Im nächsten Abschnitt werden die Vorteile und Nachteile eines nativen FOP-Compilers detaillierter vorgestellt.

3.5 Nativer FOP-Compiler

Zuvor wurden einige spezifische FOP-Fehler vorgestellt und es konnte gezeigt werden, dass die bisherigen Ansätze nicht immer befriedigende Ergebnisse liefern. Des Weiteren

führt das zweistufige Verfahren, welches bei AHEAD und FeatureHouse verwendet wird, dazu, dass die Abbildung von den Fehlermeldungen zu zusätzlichen Aufwand führt. Mit FeatureIDE wurde eine Integration der beiden Kompositionsprogramme in eine IDE gezeigt, die die Abbildung der Fehlermeldungen automatisiert. FeatureIDE kann das Problem der Fehlerabbildung mindern, aber die Behandlung von FOP spezifischen Fehlern bleibt die gleiche. In diesem Abschnitt wird daher gezeigt, wie ein nativer FOP-Compiler den Bereich der Fehlererkennung erweitern kann und welche weiteren Auswirkungen dabei entstehen.

Die Abbildung 3.4 zeigt einen schematischen Aufbau des nativen FOP-Compilers. Ähnlich den bisher vorgestellten Ansätzen, werden zunächst die Feature komponiert und im Anschluss findet die Übersetzung in Bytecode statt.

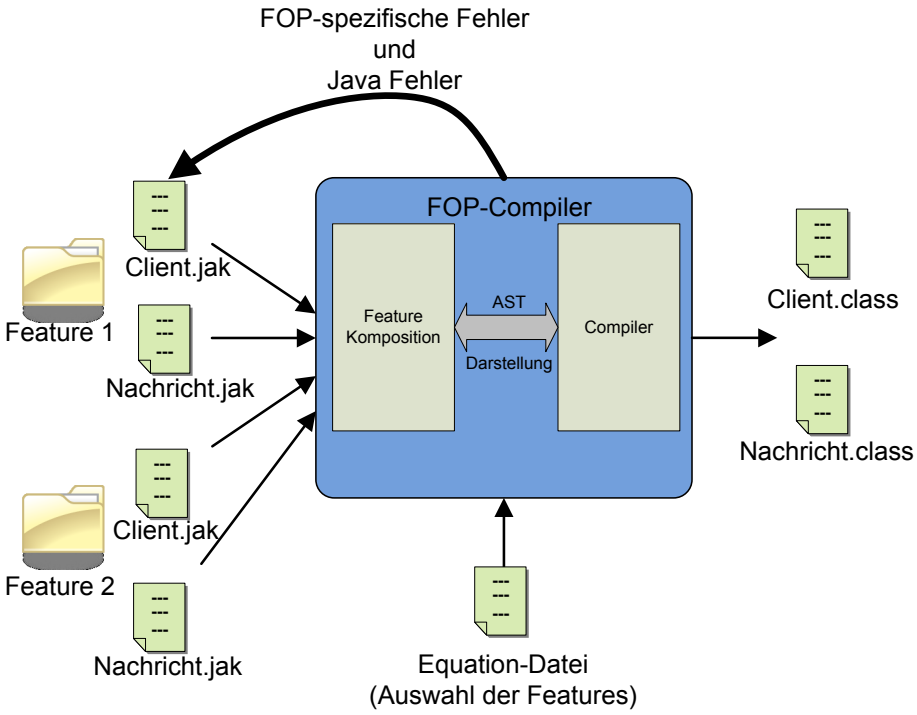

Abbildung 3.4: Schematischer und konzeptionellen Aufbau des nativen FOP-Compilers

Im Gegensatz zu AHEAD oder FeatureHouse wird im nativen FOP-Compiler nur eine Darstellung des Programms in Form eines AST verwendet. Das bedeutet, dass die Featurekomposition auf die gleichen Informationen zurückgreifen kann, wie auch der Compiler. Zusätzlich können während der Featurekomposition Funktionen des Compilers (z.B. das Typsystem) verwendet werden. Dies kann für weitere Fehlerüberprüfungen genutzt werden, die die Komposition sicherer gestalten können. Eine Zwischendarstellung, in Form von nativem Quelltext, wird nicht benötigt. Die einheitliche interne Darstellung und das Funktionen des Compiler verwendet werden können, stellen einen bedeutenden Vorteil gegenüber den bisherigen Ansätzen dar. Welche Nutzen aus diesem Vorteil gewonnen werden können, soll im Anschluss diskutiert werden.

In [TBKC07, AKL08] wurden Ansätze gezeigt, um die Featurekomposition sicher zu gestalten und hierfür beispielsweise Typsysteme zu verwenden. Die Ansätze beruhen

unter anderem darauf, dass Features nicht isoliert, sondern die gesamte SPL mit allen Features betrachtet werden muss. Aus diesem Grund wird in dieser Arbeit ein Ansatz verfolgt, bei dem der native FOP-Compiler zunächst alle Features einliest und dann einen AST der gesamten Produktlinien erstellt. Dieser AST kann die Grundlage für weitere Überprüfungen der SPL bilden. Erst nach diesen möglichen Überprüfungen, werden Features entfernt, die nicht ausgewählt worden sind.

Zuvor wurden einige FOP spezifische Fehler vorgestellt. Der Compiler kann so implementiert werden, dass diese Fehler passend erkannt und entsprechende Fehlermeldung erzeugt werden. Dazu können während der Komposition weitere Überprüfungen implementiert werden, die beispielsweise auch das Typsystem des Compilers verwenden. Auch können neue Arten von Fehlermeldungen implementiert werden, die sich beispielsweise auf mehrere Quelltextdateien beziehen. Die vorhandene Fehlerüberprüfung des Compilers überprüft die Variante auf semantische Fehler. Nähere Details zu der Implementierung der Fehlerüberprüfung befindet sich im Kapitel 4.

Tritt beim FOP-Compiler ein Fehler beim Übersetzen oder bei der Featurekomposition auf, kann auf Grund der einheitlichen Darstellungsform und der Kenntnisse, welche Features komponiert wurden, zurück verfolgt werden aus welchem Feature der fehlerbehaftete Quelltext kommt. Damit beziehen sich Fehlermeldungen immer auf den fehlerhaften Quelltext und das Problem der Fehlerabbildung tritt nicht auf.

Die Werkzeugunterstützung der FOP kann durch einen nativen FOP-Compiler verbessert werden. Der Compiler kann Informationen aus der Variante und der SPL extrahieren und diese Werkzeugen zur Verfügung stellen. Ein Ziel kann die Integration des Compilers in FeatureIDE sein. FeatureIDE könnte beispielweise die Informationen aus dem gesamten AST der SPL, nutzen um Kollaborationsdiagramme zu erstellen oder eine komfortable Autovervollständigunsfunktion zu bieten.

Ein Debugger ist heute ein wichtiges Werkzeug zur Entwicklung von Software. Bei den bisherigen Ansätzen konnte das Debuggen nur auf der Zwischendarstellung erfolgen. Wie zuvor angesprochen, kann die Zwischendarstellung bei vielen Features sehr umfangreich und schwer zu lesen sein. Ein nativer FOP-Compiler bietet die Möglichkeit an, einen Debugger zu entwickeln, der direkt auf dem feature-orientierten Quelltext arbeitet. Der Compiler kann als Basis für die entsprechende Debugger-Entwicklung dienen.

Neben den Vorteilen, die bis jetzt genannt wurden, sind eventuelle Nachteile zu berücksichtigen. Zunächst einmal entsteht ein höherer Implementierungsaufwand. Selbst wenn ein Compiler-Framework zur Erstellung des native FOP-Compilers verwendet wird, kann die Implementierung und die Wartung schwieriger sein, als bei einem Kompositionsprogramm. Ein Entwickler muss sich mit der Featurekomposition und dem Compiler-Framework auseinander setzen um den FOP-Compiler zu warten oder neue Features hin zu zufügen.

Ein anderer Nachteil, der bei der Verwendung eines nativen FOP-Compilers entsteht, ist, dass dieser Compiler nur für eine Programmiersprache funktioniert. FeatureHouse und AHEAD unterstützen neben der erweiterten Java-Sprache weitere Programmiersprachen. Der Ansatz von FeatureHouse zeigt durch seine Formalisierung, dass

viele Sprachen feature-orientiert verwendet werden können. Der Prototyp, der im Rahmen dieser Arbeit erstellt wurde, kann nur die erweiterte Java-Sprache verarbeiten, da das verwendete Compiler-Framework einen erweiterbaren Java-Compiler zur Verfügung stellt.

Zusammengefasst konnte gezeigt werden, dass die Verwendung eines nativen FOP-Compilers Vorteile mit sich bringt. Besonders zusätzliche Überprüfungen und das auf eine Zwischendarstellung verzichtet werden kann, sprechen für einen nativen Compiler. Im nächsten Kapitel wird die Implementierung eines FOP-Compiler Prototyps vorgestellt. Im Kapitel 5 wird die Leistungsfähigkeit dieses Prototyps evaluiert.

4. Implementierung

Ein Ziel dieser Arbeit ist die Entwicklung eines Prototyps für einen nativen FOP-Compiler. In diesem Kapitel wird die Implementierung eines solchen Compilers mit dem JastAddJ Framework näher beschrieben. Der Compiler soll kompatibel zu der Syntax von AHEAD und FeatureHouse sein, damit unter anderem bestehende Projekte verwendet werden können.

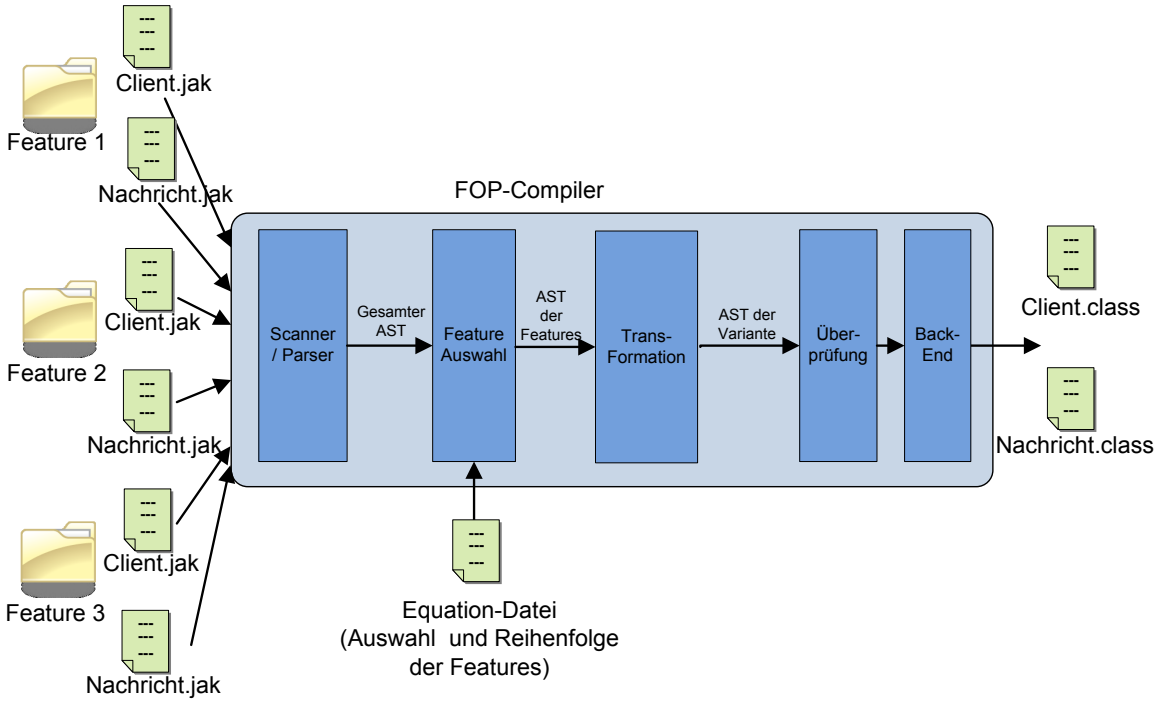

Abbildung 4.1: Schematischer Aufbau des nativen FOP-Compilers

Die Abbildung 4.1 zeigt den schematischen Aufbau des Compilers. Intern ist der Compiler in mehrere Schritte unterteilt, die im Folgenden kurz vorgestellt und in den nachfolgenden Kapiteln detaillierter diskutiert werden.

Der Compiler liest mit Hilfe des Scanners zunächst alle Features ein. Der Parser erzeugt daraus einen AST aller Features. Die Änderungen am Scanner und am Parser werden im Abschnitt 4.1 beschrieben.

Zusätzlich wird eine Equation- oder Expressiondatei eingelesen, in der die Reihenfolge und die Auswahl der Features beschrieben ist. Diese Informationen werden genutzt, um zu entscheiden, welche Teile des ASTs, die zu einem nicht ausgewählten Feature gehören, gelöscht werden können. Des Weiteren wird die Reihenfolge festgelegt, in der die Features komponiert werden. Dies wird in Abschnitt 4.2 gezeigt.

Im Anschluss daran wird im Abschnitt 4.3 die AST-Transformation beschrieben, die die Komposition der Features darstellt. Dazu wird auf das Konzept, der Überlagerung von FSTs, welches von FeatureHouse verwendet wird, zurückgegriffen. Der Hauptteil der Transformation stellt die Implementierung der Kompositionsregeln und das Erkennen von FOP-spezifischen Fehlern dar. Das Ergebnis ist der AST einer Variante.

Im Anschluss wird die Variante auf weitere semantische Fehler überprüft. Zusätzliche Erweiterungen für die FOP werden im Abschnitt 4.4 gezeigt. Nach der Überprüfung erzeugt das, vom Compiler-Framework vorgegebene, Back-End den Bytecode.

Im nächsten Abschnitt wird die Implementierung der Sprachkonstrukte für die FOP mit dem JastAddJ Framework gezeigt. Hierzu gehören die Änderungen am Scanner und Parser und die Beschreibung der neuen AST-Knoten.

4.1 Sprachkonstrukte für die FOP

Für die Umsetzung der FOP verwenden AHEAD und FeatureHouse neue Sprachkonstrukte und Schlüsselwörter. Damit der native FOP-Compiler kompatibel zu diesen beiden Vertretern ist, müssen die entsprechenden Sprachkonstrukte implementiert werden. Für ein neues Sprachkonstrukt können folgende Änderungen notwendig sein:

- Hinzufügen neuer Schlüsselwörter
- Hinzufügen von neuen oder Modifizieren von bestehenden Produktionsregeln
- Beschreibung der AST-Knoten in der JastAdd spezifischen Grammatik
- Hinzufügen von weiteren Konstruktoren in die AST-Knoten mittels Aspekte

In Abhängigkeit der zu implementierenden Sprachkonstrukte sind nicht unbedingt neue Schlüsselwörter, AST-Knoten oder Konstruktoren notwendig. Änderungen an der Grammatik sind immer notwendig, da sonst kein neues Sprachkonstrukt implementiert werden kann.

Die Implementierung der Klassen- und Interfaceverfeinerung wird im nachfolgenden Abschnitt detailliert gezeigt, um das Vorgehen zu verdeutlichen, wie neue Sprachkonstrukte in JastAdd implementiert werden können.

4.1.1 Klassen- und Interfaceverfeinerung

In diesem Abschnitt wird die Implementierung der Klassen- und Interfaceverfeinerung detailliert gezeigt, um das Vorgehen bei der Implementierung von neuen Sprachkonstrukten zu zeigen. Das Sprachkonstrukt der Klassen- und Interfaceverfeinerung erlaubt es, bestehende Klassen oder Interfaces durch neue Methode und Felder zu verfeinern. Da die Implementierung von Klassen- oder Interfaceverfeinerungen sehr ähnlich sind, wird dies zusammen vorgestellt.

Der erste Schritt stellt das Hinzufügen des Schlüsselwortes *refines* in den Scanner dar.

Für die Änderungen am Parser muss zunächst festgelegt werden, an welcher Stelle im AST Verfeinerungsknoten auftreten können. Die Verfeinerungen wurden so Implementiert, dass sie auf gleicher Ebene wie eine Klassen- oder Interfacedefinition stehen. Damit kann auf bestehende Knoten für die Körper der Verfeinerungen zurück gegriffen werden, was zu einem reduzierten Aufwand bei der Implementierung führt. Für die Klassenverfeinerung wurde als unterliegender Knoten der bestehende Knoten vom Typ *Classbody* gewählt und für die Interfaceverfeinerung entsprechend ein Knoten vom Typ *Interfacebody*. So kann innerhalb einer Verfeinerung alles stehen, was auch in einer Klasse oder einem Interface stehen kann. In Abbildung 4.2 wird dieser Zusammenhang grafisch dargestellt. Die Abbildung zeigt einen Ausschnitt aus dem vereinfachten AST, der schon in Abbildung 2.10 gezeigt wurde. Die neuen Knoten sind in der Farbe Rot dargestellt.

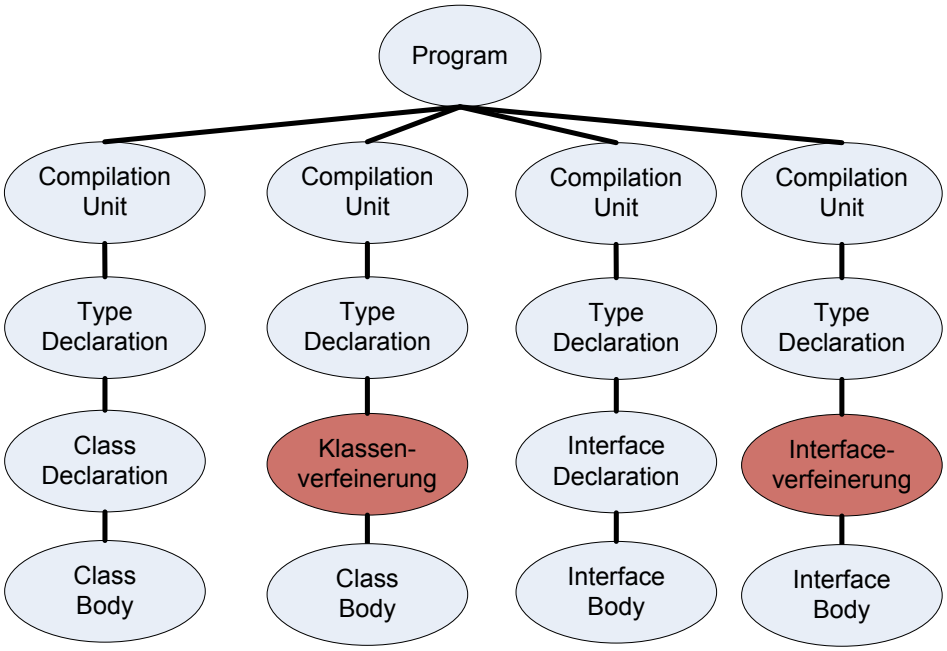

Abbildung 4.2: Position der Klassen- und Interfaceverfeinerung im AST

In der Grammatik der Java-Sprache muss der Aufbau der Verfeinerung beschrieben werden. Die neuen Produktionsregeln werden in Quelltext 4.1 detailliert gezeigt. Auf Grund der Ähnlichkeit von Klassen- und Interfaceverfeinerung wird zur besseren Im-

plementierung ein abstrakter Verfeinerungsknoten eingeführt, der die Gemeinsamkeiten abstrahiert.

Die Zeile 1 in Quelltext 4.1 beschreibt die Position im AST. Eine Verfeinerung steht auf gleicher Ebene wie eine Klassen- oder Interfacedefinition. Eine solche abstrakte Verfeinerung kann entweder eine Klassen- oder Interfaceverfeinerung sein (Zeile 2-3). In Zeile 5-7 wird dann der Aufbau der Klassenverfeinerung gezeigt. Zunächst der optionale Modifikator (public, private usw.), im Anschluss die beiden Schlüsselwörter *refines* und *class* und danach der Name der Klasse, die verfeinert werden soll. Danach folgen geschwungene Klammern, in denen ein Klassenblock (der Konstruktoren, Felder, Methoden, usw. enthalten kann) stehen kann.

Die Produktionsregel für die Interfaceverfeinerung (Zeile 11-15) ist der Klassenverfeinerung sehr ähnlich und unterscheidet sich durch das Schlüsselwort *interface* und das eine Interfaceverfeinerung anstelle eines Klassenblocks einen Interfaceblock verwendet.

```
1  TypeDecl type_declaration = refine_Stmt;
2  Refine_Stmt refine_Stmt = refine.r  {: return r; :}
3                  | refines_interface.r {: return r; :};
4
5  Refines refines =
6      modifiers.m? REFINES CLASS IDENTIFIER
7          LBRACE class_body_declarations   RBRACE
8      {:return new Refine_Class(new Modifiers(m), "REF_"+IDENTIFIER,
9                            class_body_declarations, IDENTIFIER); :}
10
11 Refines_interface refines_interface =
12     modifiers.m? REFINES INTERFACE IDENTIFIER
13         LBRACE interface_member_declarations   RBRACE
14     {:return new Refine_Interface(new Modifiers(m), "REF_"+IDENTIFIER,
15                             interface_member_declarations, IDENTIFIER); :};
```

Quelltext 4.1: Änderungen am Parser für die Klassen- und Interfaceverfeinerung

Die AST-Knoten werden vom Compiler-Framework generiert und müssen dafür in einer JastAdd spezifischen Grammatik beschrieben werden. In Quelltext 4.2 befindet sich die Beschreibung für die Klassen- und Interfaceverfeinerung.

Wie zuvor erläutert, haben die Klassen- und die Interfaceverfeinerung Gemeinsamkeiten, die durch einen abstrakten Knoten abstrahiert werden. Dieser abstrakte Knoten wird in Zeile 1 beschrieben und erbt von einem *ReferenceType*-Knoten, der wiederum von dem Knoten *TypDeclaration* erbt. Die beiden Knoten zur Klassen- (Zeile 2) und Interfaceverfeinerung (Zeile 3) erben von dem abstrakten Knoten und haben als zusätzlichen Knoten eine Zeichenkette. Diese Zeichenkette beschreibt die Klasse, oder das Interface, das verfeinert werden soll.

Zuletzt werden noch neue Konstruktoren in den AST-Knoten eingeführt, damit der Parser diese Knoten erzeugen kann. Die Konstruktoren werden mittels imperativen Aspekt eingefügt. Ein solcher Aspekt, kann wie in Quelltext 4.3 gezeigt, aussehen.

```
1 abstract Refine_Stmt : ReferenceType;
2 Refine_Class : Refine_Stmt ::= <Name:String>;
3 Refine_Interface : Refine_Stmt ::= <Name:String>;
```

Quelltext 4.2: Beschreibung der Knoten für die Klassen und Interfaceverfeinerung

```
1 aspect RefinesStmt {
2     public Refine_Class.Refine_Class(
3         Modifiers p0, String p1, List<BodyDecl> p2, beaver.Symbol p3) {
4         setChild(p0, 0);
5         setID(p1);
6         setChild(p2, 1);
7         setName(p3);
8     }
9 }
```

Quelltext 4.3: Imperativer Aspekt zum Einfügen der Konstruktoren für die Klassen und Interfaceverfeinerung

Mit der Klassen- und der Interfaceverfeinerung sind zwei wichtige neue Sprachkonstrukte für AHEAD implementiert, die es erlauben bestehenden Klassen oder Interfaces zu verfeinern. Des Weiteren wurde detailliert gezeigt, wie neue Sprachkonstrukte mit JastAdd realisiert werden können. Zunächst wurde das Schlüsselwort *refines* zum Scanner hinzugefügt. Die Java-Grammatik wurde durch weitere Produktionen erweitert. Neue AST-Knoten werden durch die JastAdd-Grammatik beschrieben und neue Konstruktoren werden mittels imperative Aspekte eingefügt.

4.1.2 Konstruktorverfeinerung

Mit Hilfe der Konstruktorverfeinerung können bestehenden Konstruktoren durch weitere Anweisungen verfeinert werden. AHEAD verwendet für die Konstruktorverfeinerung wieder das Schlüsselwort *refines*. Damit stellt die Konstruktorverfeinerung bei AHEAD ein weiteres Sprachkonstrukt dar, das implementiert werden muss.

In Abschnitt 2.2.1 wurde die Konstruktorverfeinerung vorgestellt. Dabei steht innerhalb einer Verfeinerung das Schlüsselwort *refines* und dahinter die Konstruktorsignatur, von dem Konstruktor, der verfeinert werden soll.

Das Schlüsselwort *refines* wurde schon durch die Klassenverfeinerung eingeführt und daher sind keine Änderungen am Scanner notwendig.

In Quelltext 4.4 werden die Produktionsregeln für die Konstruktorverfeinerung gezeigt. Eine Konstruktorverfeinerung wurde so implementiert, dass sie innerhalb einer Klassen stehen kann (Zeile 1), siehe auch Abbildung 4.3. In Zeile 3-5 wird dann das Sprachkonstrukt beschrieben. Dieses besteht aus dem Schlüsselwort *refines* und dahinter aus einer Konstruktorsignatur. Diese Signatur beschreibt, welcher Konstruktor verfeinert werden soll.

Aus dieser Implementierung folgt, dass eine solche Verfeinerung auch innerhalb einer Klasse stehen kann, die keine Klasse verfeinert. Die Entwickler des JastAdd-Frameworks

```
1 BodyDecl class_body_declaration = constructor_refinement;
2
3 Constructor_Refinement constructor_refinement =
4         REFINES IDENTIFIER LPAREN formal_parameter_list.l? RPAREN
5         LBRACE block_statements? RBRACE
6         {...};
```

Quelltext 4.4: Erweiterungen des Parser für die Konstruktorverfeinerung

geben in ihren Präsentationsfolie den Ratschlag, dass der Parser möglichst einfach gehalten werden soll. Daher wird die Erkennung, dass eine Konstruktorverfeinerung innerhalb einer normalen Klasse steht, nicht vom Parser erledigt, sondern findet an späterer Stelle statt (siehe Abschnitt 4.4).

Die nächsten Schritte stellen wieder die Beschreibung des AST-Knotens und das Hinzufügen eines Konstruktors mittels Aspekt dar. Der Aufbau der Beschreibung unterscheidet sich nicht sonderlich von der Klassenverfeinerung und wird aus diesem Grund hier nicht detailliert gezeigt.

4.1.3 Originaler Methodenaufruf

Wenn eine Methode von einer anderen Methode verfeinert wird, kann bei AHEAD mit Hilfe der **`Super`**`()`-Anweisung und bei FeatureHouse mit Hilfe der **`original`**`()`-Anweisung auf die originale Methode zugegriffen werden. Im Gegensatz zu der Implementierung in FeatureHouse wird der **`original`**`()`- Aufruf mit Hilfe eines Schlüsselwortes realisiert. Dies hat den Vorteil, dass besser erkannt werden kann, ob ein **`original`**`()`-Aufruf innerhalb einer Methode verwendet wird, die keine andere Methode verfeinert.

Obwohl die Syntax der beiden Sprachkonstrukte unterschiedlich ist, erfüllen beide die gleiche Funktion und können sehr ähnlich implementiert werden. Aus diesem Grund werden die beiden Konstrukte zusammen vorgestellt.

Der erste Schritt bildet wieder das Hinzufügen der neuer Schlüsselwörter. In diesem Fall *Super* und *original*.

Die Position der originalen Methodenaufrufe wird in Abbildung 4.3 gezeigt. Die originalen Methodenaufrufe wurden als Ausdrücke (Expressions) implementiert, so dass sie innerhalb einer Methode an verschiedenen Stellen auftreten können.

In Quelltext 4.5 werden in der Methode *neuerClient* verschiedene Positionen eines originalen Methodenaufrufes gezeigt. Würden die originalen Methodenaufrufe als Anweisung (Statement) implementiert, wäre auf Grund der Java-Grammatik, nur die erste Position gültig. Als Ausdrücke sind die anderen Positionen ebenfalls gültig.

Der nächste Schritt stellt wieder die Beschreibung der neuen Produktionsregeln für den Parser dar. Die Regeln im Quelltext 4.6 beschreiben den syntaktischen Aufbau und dass es sich bei den originalen Methodenaufrufe um Ausdrücke (Expressions) handelt.

Da die Funktion der beiden Varianten der originalen Methodenaufrufe sehr ähnlich ist, wird für die beiden Knoten ein abstrakter Knoten verwendet von dem beide Varianten erben. Die genaue Knotenbeschreibung befindet sich in Quelltext 4.7.

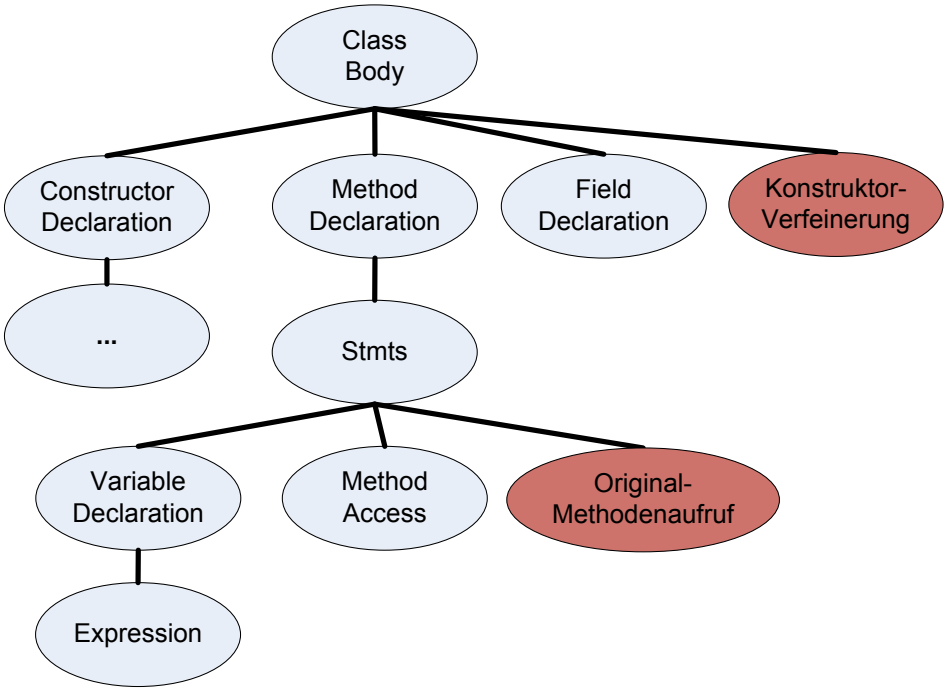

Abbildung 4.3: Position der originalen Methodenaufrufe und Konstruktorverfeinerungen im AST

```
1  public class Server{
2
3      ArrayList client;
4
5      public ArrayList neuerClient(){
6              original();
7              ArrayList liste = original();
8              client = original();
9              return original();
10     }
11
12 }
```

Quelltext 4.5: Unterschiedliche Positionen für einen originalen Methodenaufruf

Zum Erzeugen des Knotens ist ein passender Konstruktor notwendig. Dieser wird mittels imperativen Aspekt in den AST-Knoten eingeführt.

```
1 Expr expression = refsuper;
2 Expr expression = original_call;
3
4 OriginalMethodCall refsuper=
5     REFSUPER LPAREN type_list? RPAREN DOT method_invocation
6     {...};
7 OriginalMethodCall original_call=
8     ORIGINAL LPAREN argument_list.l? RPAREN
9     {...};
```

Quelltext 4.6: Produktionsregeln für die originalen Methodenaufrufe für AHEAD und FeatureHouse

```
1 abstract OriginalMethodCall : Expr;
2 RefSuper: OriginalMethodCall;
3 Original : OriginalMethodCall ::= Argument_list:List ;
```

Quelltext 4.7: Beschreibung der Knoten für die originalen Methodenaufrufe

4.1.4 Layer-Anweisung

Die Layer-Anweisung dient in AHEAD dazu, dem Quelltext ein Feature zuordnen zu können. Die Layer-Anweisung besteht aus dem Schlüsselwort *layer*, dem Featurenamen und einem Semikolon.

Zum Scanner wird das Schlüsselwort *layer* hinzugefügt.

Eine Quelltextdatei wird von JastAdd als Knoten vom Typ *CompilationUnit* dargestellt. Aus diesem Grund betrifft die Layer-Anweisung nur den *CompilationUnit*-Knoten. Die bestehende Produktionsregel des Knotens wird um die Möglichkeit der Layer-Anweisung erweitert, so wie es in Quelltext 4.8 durch Fettdruck dargestellt ist.

```
1 CompilationUnit compilation_unit =
2    LAYER IDENTIFIER SEMICOLON
3    package_declaration.p  import_declarations.i? type_declarations.t?
4             {...}
```

Quelltext 4.8: Änderung am Parser für die Layer-Anweisung

Ein neuer Knoten wird nicht benötigt. Für den Knoten *CompilationUnit* ist zusätzlich ein neuer Konstruktor notwendig, der mittels imperativen Aspekt eingefügt wird. Mit diesen Änderungen akzeptiert der Scanner und Parser eine Layer-Anweisung am Anfang einer Quelltextdatei und speichert den entsprechenden Featurenamen als Zeichenketten im AST-Knoten.

Damit sind alle Sprachkonstrukte, die von AHEAD und FeatureHouse benutzt werden, implementiert. Der Parser ist nun in der Lage, eine AST mit den neuen Sprachkonstrukte zu erzeugen.

4.2 Auswahl der Features

Im vorherigen Abschnitt wurde die Implementierung der neuen Sprachkonstrukte vorgestellt. In Abbildung 4.1 wurde gezeigt, dass der Parser einen AST von der kompletten SPL erzeugt. Der Grund warum auch nicht ausgewählte Features und Quelltextdateien eingelesen werden ist, dass der komplette AST viele Informationen über die SPL enthält. Für spätere Erweiterungen können die Informationen zur erweiterten Werkzeugunterstützung oder Fehlerdiagnose genutzt werden.

Mit Hilfe der Equation-Datei wird eine Auswahl und die Reihenfolge der Features festgelegt. Werden Features für eine Variante nicht ausgewählt, müssen Teile des ASTs gelöscht werden. Die Reihenfolge bestimmt, welches Feature ein anderes Feature verfeinert.

Eine Equation- oder eine Expressiondatei enthält die Namen und die Reihenfolge der Features. Die Dateien bestehen aus einer Liste der Featurenamen, die für die Variante ausgewählt wurden. Die Reihenfolge, in der die Featurenamen angeordnet sind, legt gleichzeitig auch die Reihenfolge der Featurekomposition fest.

Der Unterschied zwischen einer Equationdatei (AHEAD) und einer Expressiondatei (FeatureHouse) besteht im Aufbau der Datei. Eine Expressiondatei kann am Anfang noch eine Kommentarzeile enthalten, und die Features werden durch Tabulatoren getrennt. Bei einer Equationdatei werden die Features durch Zeilenumbrüche getrennt. Der in dieser Arbeit vorgestellte Prototyp kann beide Dateien einlesen und auswerten.

Nachdem der Parser den AST erstellt hat, wird die Equation- oder Expressiondatei ausgelesen. Eine Methode legt im AST-Knoten *Program* eine Liste an, in der die ausgewählten Features und die Reihenfolge festgelegt sind. Im Anschluss daran müssen Teile des AST entfernt werden, die zu nicht ausgewählten Features gehören. Jede Klassen- oder Interfaceverfeinerung lässt sich hierbei immer auf einen AST-Knoten vom Typ *CompilationUnit* abbilden. Daraus folgt, dass ein Feature aus einer Menge von AST-Knoten vom Typ *CompilationUnit* besteht.

Die Auswahl der Features wird nun so realisiert, dass jeder CompilationUnit-Knoten überprüft wird, ob er zu einem ausgewählten Feature gehört. Wenn dies nicht der Fall ist, wird er entfernt.

Der nächste Schritt stellt die Implementierung der Reihenfolge der Featurekomposition dar. Für die Umsetzung wird auf die Formalisierung, die bei FeatureHouse verwendet wird, zurück gegriffen. Dabei wurden FSTs rekursiv überlagert. Zunächst werden alle Packages in einer Liste gespeichert. Zu jedem Package gehört wiederum eine Liste von Klassen, Interfaces oder Verfeinerungen. In der zuletzt genannten Liste sind die Objekte so angeordnet, in der sie auch komponiert werden.

Die Abbildung 4.4 stellt den Schritt der Featureauswahl und die Reihenfolge nochmal an Hand eines Beispiels grafisch dar. Im oberen Teil der Grafik wird ein Program gezeigt, in dem noch alle Feature enthalten sind (Feature 1-4). Die Equation-Datei beschreibt, dass die Feature 1, Feature 2 und Feature 3 in dieser Reihenfolge komponiert werden sollen. Feature 4 ist nicht ausgewählt.

Mit diesen Informationen erfolgen dann, wie im unteren Teil der Grafik zu sehen ist, die Komposition der Features von links nach rechts. Das nicht ausgewählte Feature 4 und die dazu gehörigen CompilationUnit wird gelöscht und spielt bei der Komposition keine Rolle mehr. Zunächst wird Feature 1 mit Feature 2 und im Anschluss Feature 2 mit Feature 3 komponiert. Dabei werden auch die unterschiedliche Packages berücksichtigt und nur Klassen im gleichen Package werden komponiert.

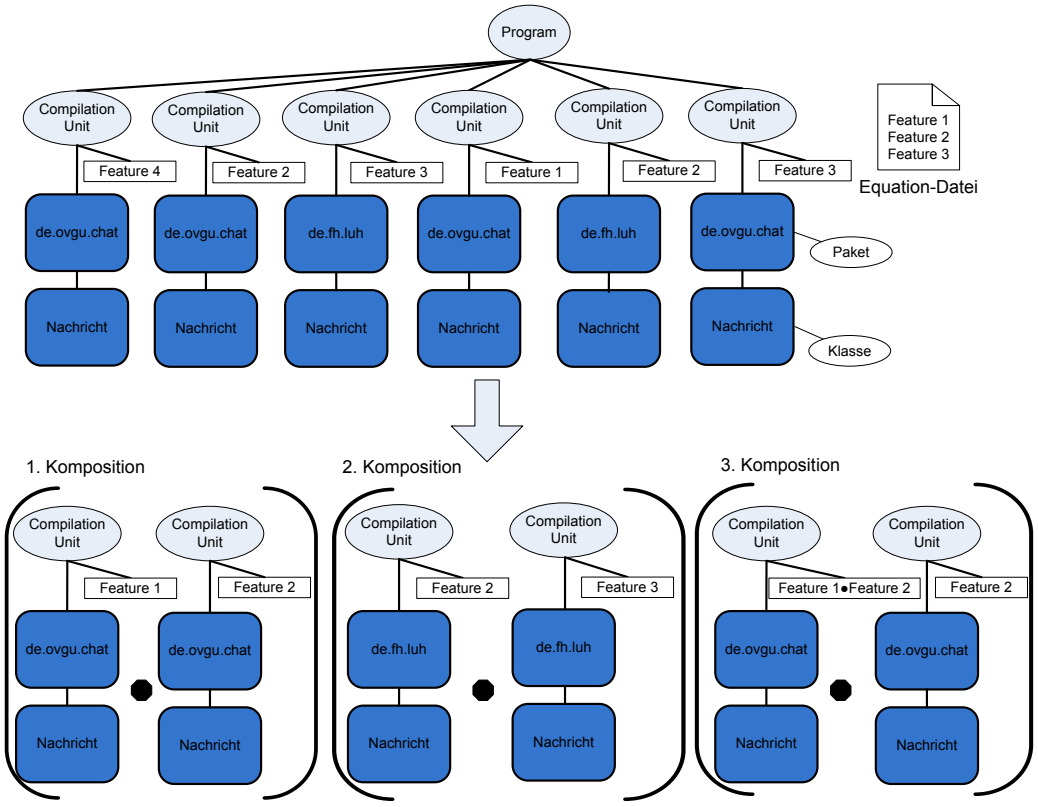

Abbildung 4.4: Beispiel zur Auswahl und Reihenfolge bei der Komposition von Features

4.3 Transformation des AST

Der nächste Teil des Compilers enthält den AST aller Features und die Reihenfolgen, die festlegt, welche Klassen von welcher Klasse verfeinert wird. In diesem Abschnitt wird nun die eigentliche Featurekomposition erläutert. Grundlage hierbei bildet die Formalisierung von FeatureHouse [ALMK08]. Jedes Feature wird durch eine Menge von Klassen oder Klassenverfeinerung dargestellt. Im Compiler werden diese als AST dargestellt. Nahe liegend ist hier, die Featurekomposition durch Überlagerung der ASTs zu realisieren.

Der AST, der zu einer Klasse gehört, kann daher wie ein FST betrachtet werden. Die Implementierung der Featurekomposition stellt somit nur noch die Implementierung der einzelnen Kompositionsregeln für zwei Knoten vom gleichen Typ dar. In [AKL09] sind für Java Kompositionsregeln beschrieben, die für diese Arbeit teilweise übernommen und angepasst wurden. Auf Grund der Kompatibilität zu AHEAD und FeatureHouse

sind die Regeln so zu implementieren, dass das vom Compiler erstellte Programm das gleiche Verhalten ausweist wie ein Programm, dass mit AHEAD oder FeatureHouse erstellt wurde. Des Weiteren wurden die Regeln so angepasst, dass die Implementierung mit dem JastAddJ-Framework möglichst einfach gestaltet werden konnte. Diese Regeln werde in den nachfolgenden Abschnitten im Einzelnen beschrieben.

Felder

Enthält eine Klassenverfeinerung ein Feld, wird zunächst geprüft, ob in der original Klasse ein Feld mit dem gleichen Namen enthalten ist. Ist dies nicht der Fall, wird dieses Feld in die original Klasse übernommen. Enthält die original Klasse bereits ein solches Feld, bestehen mehrere Möglichkeiten:

- Das Feld in der original Klasse und das Feld in der Klassenverfeinerung haben keinen Initialwert. In diesem Fall wird nichts geändert.
- Das Feld in der original Klasse hat keinen Initialwert, aber das Feld in der Klassenverfeinerung besitzt einen. In diesem Fall wird der Initialwert übernommen.
- Das Feld in der original Klasse hat einen Initialwert, das Feld in der Klassenverfeinerung aber nicht. In diesem Fall wird nichts verändert.
- Beide Felder besitzen einen unterschiedlichen Initialwert. Dies führt zu einem Fehler beim der Featurekomposition.

import-Anweisungen

Enthält eine Klassenverfeinerung import-Anweisungen werden diese in die originale Klasse übernommen. Dies ist notwendig, damit nach der Komposition alle benötigten Klassen eingebunden werden können. Doppelte Anweisungen werden ignoriert.

Implementierte Interfaces

Enthält eine Klassenverfeinerung implementierte Interfaces werden diese in die original Klassen übernommen. Eine Klassenverfeinerung kann sich beispielsweise bei einem Listener registrieren und ein entsprechendes Interface implementieren. Bei der Komposition müssen nicht nur die Methoden übernommen werden, sondern auch das Interface damit es bei der Registrierung zu keinem Typfehler kommt. Doppelte Interfaceeinträge werden nicht übernommen.

Innere Klassen

Innere Klassen werden in die originale Klasse übernommen, damit sie dort auch genutzt werden. Existiert eine innere Klasse bereits in der originalen Klasse wird zur Zeit ein Fehler erzeugt. In zukünftigen Versionen kann eine Verfeinerung von inneren Klassen unterstützt werden.

Konstruktor

Enthält eine Klassenverfeinerung einen Konstruktor, der in der originalen Klasse nicht enthalten ist, wird dieser übernommen. Damit können neue Konstruktoren von Features eingeführt und verwendet werden. Gibt es in der originalen Klasse einen Konstruktor mit der gleichen Signatur, wird der Inhalt aus der Verfeinerung an das Ende des Konstruktors in der originalen Klasse gehangen. Das gleiche gilt, wenn eine Konstruktorverfeinerung, wie in Abschnitt 4.1.2 gezeigt wurde, verwendet wird. Dadurch können bestehenden Konstruktoren um weitere Anweisungen erweitert werden.

Methoden

Enthält die Klassenverfeinerung eine Methode, die in der original Klasse nicht vorhanden ist, wird sie übernommen. Bei einer bereits vorhandenen Methode wird die originale Klasse umbenannt und die verfeinerte Klasse wird eingefügt. Das Umbenennen der originalen Methoden erfolgt, damit beide Methoden in der originalen Klassen zur Verfügung stehen. Die originale Methode wird für einen eventuell vorhandenen originalen Methodenaufruf benötigt. Innerhalb des Knotens der Methodendeklaration wird mittels imperativen Aspekts ein Feld angelegt, das auf die umbenannte originale Methode zeigt. Nachdem die Methode umbenannt wurde, wird das Feld gesetzt.

Wenn eine bereits vorhandene Methode überschrieben wird, besteht die Möglichkeit, dass innerhalb einer Methode ein originaler Methodenaufruf steht. Dieser kann entweder durch ein `original()`-Aufruf oder durch ein `Super()`-Aufruf umgesetzt werden. Dafür muss zunächst die gesamte Methode nach so einem Aufruf durchsucht werden. Hierbei muss der gesamte AST der Methode durchsucht werden, da tiefe Verschattelungen durch Blöcke oder if-else Anweisungen möglich sind.

Wird ein originaler Methodenaufruf gefunden, muss dieser durch den passenden Methodenaufruf ersetzt werden. Der `original()`-Aufruf wird durch einen Methodenaufruf ersetzt. Dieser wird mit Hilfe der Referenz, auf die umbenannte originale Methode, erzeugt.

Beim `Super()`-Aufruf steht hinter dem Schlüsselwort *Super* ein Methodenaufruf. Dieser Methodenaufruf wird mit dem originalen Methodenaufruf verglichen. Besteht ein Unterschied bedeutet dies, dass nicht die originale Methode aufgerufen wird, sondern irgendeine andere Methode. Da dies in den meisten Fällen nicht so gedacht ist, und es sich wahrscheinlich um einen Fehler handelt, wird eine Warnung ausgegeben.

JastAdd spezifische Implementierungsdetails

Zuvor wurden die wichtigsten Kompositionsregeln und deren konzeptionellen Implementierung beschrieben. Bevor nun die Überprüfung des AST beziehungsweise des Programms vorgestellt wird, werden noch zwei Implementierungsdetails erläutert. Dazu gehört das Zwischenspeichern von Werten in den AST-Knoten und das Erzeugen des Bytecodes aus den CompilationUnits. Diese Details sollten beachtet werden, wenn der Compiler erweitert werden soll, da es sonst zu Fehlern bei den nachfolgenden Überprüfungen kommt.

Das Compiler-Framework baut die Knoten des AST so auf, dass Werte zwischengespeichert und nur einmal berechnet werden. Wird beispielsweise in dem AST-Knoten *MethodDecl* die Methode `getSignature()` aufgerufen, wird die Zeichenkette nur beim ersten Mal berechnet. Dies sorgt dafür, dass umfangreichere Berechnungen nur einmal durchgeführt werden müssen. Beispielsweise existiet innerhalb einer Klassendeklaration eine Liste, die alle Methodensignaturen der entsprechenden Klasse enthält. Eine solche Liste ist zum Zeitpunkt der Featurekomposition mit den Methodensignaturen gesetzt.

Werden bei der Komposition neue Methoden und Felder in eine Klasse eingefügt, wird diese Liste nicht aktualisiert, da die zwischengespeicherte Liste weiter verwendet wird. Das führt dazu, dass der Compiler ein neu eingefügtes Feld nicht kennt, obwohl dieses im AST vorhanden ist.

Nach der Transformation / Komposition des AST müssen zwischengespeicherten Werte gelöscht werden. Hierzu gibt es im AST-Knoten eine `flushCaches()`-Methode, die dafür sorgt dass alle Werte gelöscht und beim nächsten Aufruf neu berechnet werden. Es sollte darauf geachtet werden, dass nach Veränderungen am AST die `flushCaches()`-Methode aufgerufen wird, damit keine Fehler durch nicht mehr aktuelle Werte entstehen.

Nach der Transformation findet eine Überprüfung aller Knoten statt. Hierzu würden auch die Klassenverfeinerungen zählen. Diese Knoten brauchen an dieser Stelle nicht mehr überprüft zu werden, da sie für den weiteren Verlauf nicht mehr benötigt werden. Des Weiteren soll von den Klassenverfeinerungen auch kein Bytecode erzeugt werden. Um dies zu unterdrücken gibt in dem CompilationUnit-Knoten ein boolesches Feld mit dem Namen *fromSource*. Wird dieses Feld auf *false* gesetzt wird die CompilationUnit nicht weiter überprüft und es wird auch kein Bytecode erzeugt.

4.4 Überprüfung

Während der Komposition wurde das Programm auf FOP spezifische Fehler überprüft, zum Beispiel auf ins Leere laufende Klassenverfeinerungen oder fehlerhafte originale Methodenaufrufe. In diesem Abschnitt wird in der Überprüfung das Programm auf weitere semantische Fehler überprüft, die die Sprache Java betreffen. Dazu wird beispielsweise überprüft ob Variablennamen nicht doppelt vergeben wurden oder ob eine Variable erst initialisiert wurde bevor sie verwendet wird. Bei der Zuweisung von Werten zu einer Variable wird überprüft, ob der Wert kompatibel zu der Variable ist. Hierzu wird ein Typsystem [Pie02] verwendet. Zu den weiteren Aufgaben des Typsystems gehört die Ausnahmebehandlung (Exception-Handling) und die Überprüfung ob Anweisungen vorhanden sind, die nicht erreichbar sind.

Im JastAddJ-Framework wird die Überprüfung so realisiert, dass jeder AST-Knoten unterschiedliche Prüfungen implementieren kann. Bei der Überprüfung wird dann der komplette AST durchlaufen und jeder Knoten führt die entsprechenden Überprüfungen durch. Die Transformation, durch die die Klassenverfeinerung realisiert wird, sorgt dafür, dass der AST an diesem Punkt ein normales Java-Programm enthält. Damit sind für die Überprüfung nur noch wenige Änderungen notwendig, da die Implementierung der Überprüfung vom Framework bereitgestellt wird. Die Anpassungen, die noch gemacht werden müssen, werden in den folgenden Abschnitten diskutiert.

4.4.1 Erkennung falsch platzierter Konstruktorverfeinerungen und originaler Methodenaufrufe

Im Abschnitt 4.1 wurden die neuen Sprachkonstrukte, wie die Konstruktoverfeinerung oder die originalen Methodenaufrufe so implementiert, dass sie in jeder Klasse beziehungsweise Methode stehen können. Der Grund dafür war, dass die Änderungen am Parser möglichst einfach gehalten worden sind und alles Weitere später überprüft wird. Mit Hilfe der Überprüfung können nun falsch positionierte originale Methodenaufrufe und Konstruktorverfeinerung erkannt werden.

Wie zuvor beschrieben, ist die Überprüfung so aufgebaut, dass der gesamte AST durchlaufen wird und jeder Knoten unterschiedliche Überprüfungen durchführen kann. Während der Transformation werden alle Konstruktoverfeinerungen und original Methodenaufrufe ersetzt. Bei der Konstruktorverfeinerung wird der Inhalt der Verfeinerung an den verfeinerten Konstruktor gehangen. Der originale Methodenaufruf wird in einen Methodenaufruf umgewandelt.

Wird bei der Überprüfung ein Knoten vom Typ Konstruktorverfeinerung oder ein originaler Methodenaufruf erreicht, deutet dies darauf hin, dass diese Sprachkonstrukte an einer Stelle verwendet wurden, an der sie nicht stehen dürfen. Dazu wird in den beiden Knoten einen Überprüfung implementiert, die sofort eine Fehlermeldung erzeugt, dass die entsprechende Konstrukte an dieser Stelle falsch platziert sind.

4.4.2 Positionsangaben der Fehlermeldungen

Im Kapitel 3 wurde gezeigt, dass sich durch den Einsatz eines nativen FOP-Compilers Fehlermeldungen auf den feature-orientierten Quelltext beziehen. Um dieses zu erreichen, müssen die Zeile und die Datei bekannt sein, die den Fehler produziert hat.

Das JastAddJ-Framework bietet in jedem AST-Knoten eine Error- und Warningmethode an. Jeder AST-Knoten besitzt zusätzlich auch die Information zu welcher Position (Zeilennummer) im Quelltext er gehört. Wird eine solche Methode vom einem Knoten aufgerufen, wird der AST nach oben durchlaufen bis er den Knoten *CompilationUnit* erreicht. In diesem Knoten befindet sich die Information, zu welcher Datei die CompilationUnit gehört.

Bei der Transformation des ASTs werden Knoten aus einer CompilationUnit (Inhalte einer Klassenverfeinerung) in eine andere CompilationUnit (die verfeinerte Klasse) verschoben. Tritt nun ein Fehler auf, ist die Zeilennummer noch richtig, aber bei der Bestimmung der Quelltextdatei wird die verfeinerte Klasse und nicht die Klassenverfeinerung angezeigt.

Damit die richtige Quelltextdatei angezeigt wird, wird bei der Transformation des ASTs die originale CompilationUnit gespeichert. Wird nun einen Fehlermeldung erzeugt, wird zunächst geprüft, ob es sich um einen transformierten Knoten handelt und eine originale CompilationUnit gespeichert ist. Trifft dies zu, wird die Quelltextdatei aus der originalen CompilationUnit ausgelesen und an die entsprechende Fehlermeldung übergeben.

4.4.3 Fehlermeldungen für zwei Dateien

Bei der FOP werden Klassen von Klassenverfeinerungen erweitert. In einer Quelltextdatei wird die Klasse und in weiteren Dateien werden die Verfeinerungen beschrieben. Nun können bei der Klassenverfeinerung und im speziellen bei der Komposition, wie zuvor gezeigt, Fehler auftreten. Beispielsweise wenn in der originalen Klasse und in der Verfeinerung jeweils ein Feld mit dem gleichen Namen und Initialwert vorhanden ist. In diesem Fall bezieht sich der Fehler auf zwei unterschiedlichen Positionen in unterschiedlichen Quelltextdateien. Für diesen Fall wurde eine neue Art von Fehlermeldung implementiert, die sich auf zwei Quelltextdateien beziehen kann.

Mit der Überprüfung sind nun alle wichtigen Implementierungsdetails besprochen worden. Der Prototyp des Compilers akzeptiert die Syntax und die neuen Sprachkonstrukte von FeatureHouse und AHEAD. Die Auswahl und die Reihenfolge der Features werden mit einer Equation- oder Expresseiondatei bestimmt. Mit Hilfe einer Transformation des AST werden die Features komponiert. Die Überprüfung des Programms und das Back-End werden vom JastAddJ-Framework vorgegeben. Die Funktions- und Leistungsfähigkeit werden im Kapitel 5 gezeigt. Dort werden einige Beispielsprogramme mit dem Prototypen übersetzt und es findet ein Vergleich mit den bisherigen Ansätzen statt.

5. Evaluation

In diesem Kapitel wird der Prototyp des nativen FOP-Compilers evaluiert. Dazu wurden im Rahmen dieser Arbeit einige feature-orientierte Projekte ausgewählt und von dem Prototypen übersetzt. Die ausgewählten Projekte werden im Abschnitt 5.1 kurz vorgestellt. Die Ergebnisse der Laufzeitmesssung werden dann mit den Ergebnissen von AHEAD und FeatureHouse verglichen. Im Anschluss wird im Abschnitt 5.2 gezeigt, wie der Prototyp, die in Abschnitt 3.1 vorgestellten, FOP spezifischen Fehler erkennt. Danach erfolgt eine Bewertung über die Leistungsfähigkeit des Prototypen.

5.1 Verwendete Programme zur Evaluierung

In diesem Abschnitt werden die ausgewählten Projekte kurz vorgestellt, die für die Evaluation genutzt wurden.

5.1.1 Chat-SPL

Bei der Chat-SPL handelt es sich um ein kleines Chat-Programm, das mit AHEAD erstellt wurde. Das Programm besteht aus einer Client und einer Server Anwendung. Die Chat-SPL ist der gezeigten SPL im Kapitel 2 sehr ähnlich. Damit die Chat-SPL ein gutes Beispiel für diese Arbeit darstellt, wurden einige Features und ihre Anordnung verändert. Das Programm wurde im Rahmen der Vorlesung Erweiterte Programmierkonzepte für maßgeschneiderte Datenhaltung [KS09] an der Universität Magdeburg erstellt.

5.1.2 Graph-Produktlinie

Bei der Graph-Produktlinie (GPL) handelt es sich um eine bekannte SPL, die in zahlreichen Publikationen als Beispiel SPL dient [LhB01]. Die Graph-Produktlinie wurde für verschiedene Programmiersprachen umgesetzt. In dieser Arbeit wird die Java-Variante betrachtet. Als Kompositionsprogramm wird FeatureHouse verwendet.

5.1.3 TankWar

Bei diesem Programm handelt es sich um eine SPL, aus der verschiedene Varianten eines 2D Spiels erzeugt werden können. [1] Die SPL wurde mit FeatureIDE und AHEAD erstellt.

5.1.4 myViolett

Violett ist ein Opensource UML-Editor [2] . Bei myViolett handelt es sich um eine FOP-Version des UML-Editors, der im Rahmen eines Projektes an der Universität Texas in Austin in ein FeatureHouse Projekt portiert wurde [AL08].

5.1.5 GUIDSL

GUIDSL ist ein Teil von AHEAD und wurde selber mit AHEAD erstellt. Es dient dazu Feature-Modelle, die in der GUIDSL Grammatik erstellt wurden, zu analysieren und zu überprüfen [Bat05].

5.1.6 BerkleyDB

Bei BerkleyDB handelt es sich um ein Datenbanksystem für eingebettete Systeme von Oracle. Im Rahmen einer Fallstudie wurde diese Datenbank in Feature aufgeteilt [AKL09]. Dazu wurden die Features zunächst mit CIDE annotiert und dann in Feature Module überführt [KAK09].

5.1.7 Übersicht der verschiedenen Programme

In Tabelle 5.1 befindet sich eine Übersicht über die einzelnen Projekte. Bei der Chat-SPL, TankWar und GUIDSL handelt es sich um AHEAD-Projekte. Die Graph-Produktlinie (GPL), myViolett und BerkeleyDb sind Projekte, die mit FeatureHouse komponiert werden. Die Anzahl der Features beschreibt wie viele verschiedene Featuremodule vorhanden sind. Die Anzahl der Verfeinerung stellt ein Maß dar, wie häufig zwei Klassen komponiert werden. Ein Feature kann ein oder mehrere Verfeinerungen erhalten. Die Anzahl der Klassen beschreibt, wie viele unterschiedliche Klasse innerhalb des Projektes vorhanden sind. Die Anzahl der Programmzeilen stellt ein Maß für den Umfang des Projektes dar. Dieses Maß wurde mit dem Programm loc-counter [3] bestimmt. Dieses Programm misst die Anzahl physikalischer Programmzeilen. Kommentare werden bei der Zählung nicht berücksichtigt.

[1] Die SPL wurde von Lei Luo, Liang Liang und Songxuan Wu im Rahmen des Laborpraktikums mit dem Thema Implementation of Software Product Lines with FOP (2D Game) 2009 an der Universität Magdeburg erstellt.

[2] http://alexdp.free.fr/violetumleditor/page.php

[3] https://loc-counter.dev.java.net/

	Chat-SPL	GPL	TankWar	myViolett	GUIDSL	BerkeleyDB
Anzahl Features	8	26	60	88	26	99
Anzahl Verfeinerungen	9	56	60	89	102	620
Anzahl Klassen	8	16	20	67	130	284
Lines of Code	380	1997	3920	5168	7844	66343

Tabelle 5.1: Übersicht über die verwendeteten Programme

5.1.8 Beschreibung der Testplattform

Die nachfolgenden Messungen wurden auf einem DesktopPC mit einem AMD 3200+ (2,3GHz) und 2GB Arbeitsspeicher durchgeführt. Als Betriebssystem wurde Windows 7 (64Bit) Professional verwendet.

Die Laufzeitmessung des FOP-Compilers wurde mit der Java-Methode `System.currentTimeMillis()` umgesetzt. Bei der Ausführung des Compilers wird am Anfang die aktuelle Zeit gespeichert und am Ende ebenfalls. Die Differenz stellt die Laufzeit, die für die Übersetzung benötigt wurde, dar.

Die Laufzeit der bisherigen Ansätzen wurde mit Hilfe von ANT-Skripten realisiert. ANT-Skripte bieten Methoden an, die es ermöglichen, die Laufzeit zu bestimmen. Des Weiteren sorgen die Skripte dafür, dass die verschiedenen Werkzeuge (z.B. FeatureHouse, Java-Compiler) nacheinander aufgerufen werden. Der Aufbau der jeweiligen Skripte befindet sich im Anhang A.

5.2 Ergebnisse und Auswertung

In diesem Abschnitt sollen die Ergebnisse der Evaluierung betrachtet werden. Im nachfolgenden Abschnitt wird zunächst die Laufzeit betrachtet.

5.2.1 Laufzeituntersuchung

Ausführzeit von:	Kompositionsprogramm	Java-Compiler	FOP-Compiler
Chat-SPL	(1,04 + 5,03)s	1,56s	1,64s
GPL	2,86s	-	1,72s
TankWar	(1,45 + 9,241)s	1,63s	2,57s
myViolett	11,29s	2,56s	5,61s
GUIDSL	(6,91+25,3)s	3,81s	4,84s
BerkeleyDB	62,519s	-	9,06s*

Tabelle 5.2: Übersetzungszeit der verschiedenen Projekte

In diesem Abschnitt werden die Ergebnisse der Laufzeitmessung diskutiert. Die Ergebnisse der Messungen befindet sich in Tabelle 5.2.

Bei der GPL konnte das Übersetzen nicht fehlerfrei beendet werden. Der Java-Compiler meldet, dass an einigen Stellen **`original`**`()`-Methoden gefunden wurden, zu denen es

keine passende Methode gibt. Dies deutet darauf hin, dass FeatureHouse nicht alle `original()`-Aufrufe umwandelt. Der native FOP-Compiler wandelt alle original-Aufrufe um und es gibt keine Fehler beim Übersetzen.

BerkleyDB konnte ebenfalls nicht übersetzt werden. Der Composer läuft ohne Fehler durch, aber der nachgeschaltete Compiler meldete eine Vielzahl unterschiedlicher Fehler. Der Prototyp des nativen FOP-Compilers läuft dem entsprechend auch nicht fehlerfrei durch. Aus diesem Grund wurde beim Prototyp nach der AST-Transformation eine zusätzliche Zeit gemessen. Nach dieser Transformation sind alle Features komponiert. Dieser gemessene Wert kann aber auch nur bedingt mit dem Zeitwert von FeatureHouse verglichen werden, da beim FOP-Compiler nach der Komposition die Ergebnisse nicht in Dateien auf der Festplatte gespeichert werden.

Die Chat-SPL, TankWar und GUIDSL verwenden als Kompositionsprogramm den AHEAD-Composer. Dieser Composer erzeugt als Ausgabe jak-Dateien, die durch ein zusätzlichen Programm *jak2java* [4] in Java-Dateien umgewandelt werden müssen. Dieses Programm kann nur einzelne Dateien einlesen. Dies führt dazu, dass dieses Programm sehr häufig aufgerufen werden muss, was die lange Laufzeit erklären kann. Aus diesem Grund soll der Zeitwert für jak2java nicht sonderlich stark berücksichtigt werden. In der Tabelle 5.2 wird daher bei diesen Programmen die Ausführungszeit für die Komposition durch zwei Werte angegeben. Der erste Wert stellt die Zeit dar, die vom AHEAD-Composer benötigt wurde und der zweite Werte entsprechend die Zeit für das Programm *jak2java*.

Die Ergebnisse in Tabelle 5.2 zeigen deutlich, dass der Prototyp des nativen FOP-Compiler schneller ist als die Ansätze mit AHEAD und FeatureHouse. Ein Nachteil bei den bisherigen Ansätzen ist, dass unterschiedliche Programme sequenziell aufgerufen werden. Allein der Start eines Java-Programmes benötigt Zeit. Des Weiteren wird in den zweistufigen Verfahren an zwei Stellen eine Syntaxüberprüfung durchgeführt. Das Kompositionsprogramm und der Java-Compiler führen jeweils eine eigene Überprüfung durch.

Zusammengefasst kann gesagt werden, dass der Geschwindigkeitsgewinn gegenüber den zweistufigen Verfahren sehr deutlich ist. Laufzeit stellt aber nur ein Kriterium dar, an dem die Vorteile eines nativen FOP-Compilers gezeigt werden sollen. In dem nächsten Abschnitt wird gezeigt, wie der Prototyp die zuvor vorgestellten FOP-spezifischen Fehler erkennt.

5.2.2 FOP-spezifische Fehler

Im Abschnitt 3.1 wurden einige FOP-Fehler vorgestellt und es konnte gezeigt werden, dass die bestehenden Ansätze nicht immer befriedigende Ergebnisse liefern. Daraus entwickelte sich unter anderem die Motivation zur Entwicklung eines nativen FOP-Compilers. Aufbauend auf der Implementierung, die in Kapitel 4 diskutiert wurde, wird in diesem Abschnitt gezeigt, wie der Prototyp die FOP spezifischen Fehler erkennt. Damit sollen die Vorteile eines nativen FOP-Compilers gegenüber den zweistufigen Ansätzen gezeigt werden.

[4] http://userweb.cs.utexas.edu/~schwartz/ATS/fopdocs/j2j.html

FOP Syntaxfehler

Die Java-Grammatik wurde bei der Implementierung für die FOP erweitert, so dass die neuen Sprachkonstrukte erkannt werden. Bei fehlerhafte Syntax wird eine Fehlermeldung vom Parser erzeugt, der die Quelltextdatei angibt und aufzeigt, in welcher Zeile der Fehler aufgetreten ist.

Leerlaufende Klassenverfeinerungen

Der Prototyp des Compilers kann Klassenverfeinerungen verwenden, die, wie bei AHEAD, durch ein neues Sprachkonstrukt realisiert sind, oder das Verfahren von FeatureHouse anwenden.

Der Compiler wurde so implementiert, das zwei unterschiedliche Modi per Compilerflag ausgewählt werden können. Im AHEAD-Modus können Klassen nur durch ein refines class verfeinert werden. Tritt dabei eine Klassenverfeinerung auf, zu der keine Klassen existiert, wird eine Fehlermeldung erzeugt.

Im FeatureHouse-Modus werden Klassen durch andere Klassen verfeinert. Tritt eine Klasse zum ersten Mal auf kann nicht entschieden werden, ob es sich um eine Klassenverfeinerung handelt.

Mehrfaches Einfügen von Klassen / Interfaces

Beim mehrfachen Einfügen von Klassen oder Interfaces muss wieder zwischen den beiden Modi unterschieden werden. Beim AHEAD-Modus wird eine Warnung erzeugt, wenn eine Klasse oder ein Interface doppelt eingeführt wird.

Beim FeatureHouse-Modus findet eine Verfeinerung statt, wenn zwei Klassen oder Interfaces komponiert werden. Dadurch besteht nicht die Möglichkeit, bestehende Klassen komplett durch andere Klassendefinitionen zu ersetzen. Wie schon bei FeatureHouse ist dies vorteilhaft, da so eine potentielle Fehlerquelle nicht auftreten kann.

Komposition von Feldern

Bei der Komposition von Feldern können in Abhängigkeit der Initialwerte Fehler erzeugt werden. Hat keines der Felder einen Initialwert wird nichts verändert. Hat nur eines der Felder einen Initialwert, wird dieser Wert übernommen. Haben beide einen Initialwert wird ein Fehlermeldung erzeugt.

Da der Prototyp des Compilers für die Komposition ein Jampack-ähnlichen Ansatz verwendet, tritt das Problem des *Variablen Shadowing* nicht auf.

Typfehler

Typfehler werden vom Typsystem des Compilers nach der Komposition erkannt. Auf Grund dessen, dass kein mehrstufiges Verfahren verwendet wird, werden die Fehlermeldungen direkt auf den feature-orientierten Quelltext abgebildet.

Methodenverfeinerung

Wird ein originaler Methodenaufruf innerhalb einer Klasse oder Verfeinerung verwendet, die keine andere Methode verfeinert, wird eine Fehlermeldung erzeugt, die den Entwickler darüber informiert. Dabei ist es egal, ob ein `Super()`-Aufruf, oder ein `original()`-Aufruf verwendet wird.

Wird als originaler Methodenaufruf ein `Super()`-Aufruf verwendet, überprüft der Compiler, ob der Methodenaufruf die verfeinerte Methode beschreibt. Trifft dies nicht zu, wird ein Warnung ausgegeben.

5.2.3 Vergleich mit den bisherigen Ansätzen

In der Tabelle 5.3, werden die Ergebnisse nochmals zusammengefasst dargestellt. Die Tabelle baut auf der Tabelle 3.1 auf und wurde dabei um drei Zeilen (Laufzeit, Verwendung verschiedener Sprachen und Erweiterungspotential) erweitert.

Bei der Fehlererkennung werden FOP-spezifische Fehler erkannt und passende Fehlermeldungen erzeugt.

Bei der Komposition von Feldern tritt das Problem des *Variablen Shadowing* nicht auf und bei zwei Initialwerten wird eine Fehler erzeugt.

Typfehler werden direkt vom Typsystem des Compilers erkannt und Fehlermeldungen werden direkt auf den entsprechenden Quelltext abgebildet.

Bei den zweistufigen Ansätzen werden falsch platzierte originale Methodenaufrufe eventuell erst vom Typsystem erkannt. Der Prototyp des Compilers erkennt diesen Fehler direkt und gibt eine Fehlermeldung aus, die beschreibt, dass der Aufruf an der falschen Stelle verwendet wurde. Beim `Super()`-Aufruf wird zusätzlich überprüft, ob der Methodenaufruf auch die verfeinerte Methode beschreibt.

Im Abschnitt 5.2.1 konnte gezeigt werden, dass die Laufzeit des Prototyp kürzer ist, als bei den zweistufigen Verfahren.

Der Nachteil, der durch den Einsatz des Compilers entsteht, ist, dass nur die Programmiersprache Java unterstützt wird. AHEAD unterstützt neben JAK, als erweiterte Java-Sprache, weitere Sprache unterschiedlicher Paradigmen. FeatureHouse bietet ein Framework an, mit dem leicht weitere Sprache implementiert werden können [AKL09]. Der Prototyp bietet eine solche Möglichkeit nicht und ist für weitere Sprachen nicht erweitertbar. Dies liegt daran, dass der Compiler mit dem JastAddJ-Compilerframework erstellt wurde und dieser einen erweiterbaren Java-Compiler bietet.

Der Quelltext von FeatureHouse und AHEAD ist verfügbar, so dass Erweiterungen möglich sind. Der Prototyp des nativen FOP-Compilers bietet im Gegensatz zu AHEAD und FeatureHouse mehr Erweiterungspotential. Der Prototyp wurde mit dem JastAddJ-Framework erstellt. Das Framework zeichnet sich dadurch aus, dass Änderungen modular implementiert werden können. Der erstellte Prototyp kann um weitere Überprüfungen erweitert werden und dafür beispielsweise das Typsystem verwenden. Durch

	AHEAD	FeatureHouse	nativer FOP-Compiler
FOP Syntaxfehler	+	+	+
Leerlaufende Klassen-verfeinerung	+	o	+
Mehrfaches Einführen von Klassen / Interfaces	+	+	+
Komposition von Feldern	-	-	+
Typfehler	-	-	+
Methodenverfeinerung	o	o	+
Abbildung der Fehler auf den Quelltext	-	-	+
Laufzeit	o	o	+
Verwendung verschiedener Sprachen	o	+	-
Erweiterungspotential	o	o	+
- = negativ, o = neutral , + = positiv			

Tabelle 5.3: Vergleich Erkennung FOP spezifischer Fehler mit bisherigen Ansätzen und einem nativen FOP-Compiler

modulare Erweiterungen kann er für weitere Forschungen im Bereich der SPL und der FOP verwendet werden. Hierfür sei auf das Kapitel 7 verwiesen.

Im Abschnitt 2.2.1 wurde zur aktuellen Version von AHEAD vermerkt, dass die Java-Version 1.5 und Packages nicht unterstützt werden. Der Prototyp des Compilers ist kompatibel zu AHEAD, beruht aber auf einem Java 1.5 Compiler. So können auch die Sprachkonstrukte der Java-Version 1.5 , wie zum Beispiel die erweiterte For-Schleife, verwendet werden.

Packages werden vom Prototypen des Compilers unterstützt und bei der Komposition berücksichtigt. Damit ist der Prototyp des FOP-Compilers nicht nur Kompatibel zu AHEAD sondern behebt auch gleichzeitig einige Nachteile der aktuellen Implementierung.

FeatureHouse zeichnet sich unter anderem dadurch aus, dass keine neuen Sprachkonstrukte benötigt werden. Bei den FOP-spezifischen Fehlern konnte aber beispielsweise nicht unterschieden werden, ob eine Klasse eine Verfeinerung darstellt oder nicht. Für zukünftige Projekte, können beispielsweise Klassenverfeinerungen mit *refines* verwendet werden. So können dann ins Leere laufende Klassenverfeinerungen erkannt werden. Gleichzeitig kann der **`original`**`()`-Aufruf verwendet werden, der vom Aufbau einfacher ist, als der **`Super`**`()`-Aufruf. Der Entwickler kann das Beste aus beiden Ansätzen verwenden. Der Nachteil dabei ist, dass durch diesen *Mischbetrieb*, die so erstellten Programme nicht mehr kompatibel zu AHEAD und FeatureHouse sind.

In diesem Kapitel wurde der Prototyp eines nativen FOP-Compilers evaluiert. Es konnte gezeigt werden, dass der Compiler entscheidende Vorteile gegenüber den bisherigen zweistufigen Verfahren hat. Der Compiler verwendet keine Zwischendarstellung in Form

von nativen Java-Quelltext, wo durch das Problem der Fehlerabbildung behoben ist. Zusätzlich kann während der Komposition Funktionen des Compilers (zum Beispiel das Typsystem) verwendet werden. Durch modulare Erweiterungen kann dieser Prototyp als Basis für weitere Forschungen im Bereich der FOP und SPL verwendet werden. Im nächsten Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst.

6. Zusammenfassung

Im Rahmen dieser Arbeit wurde untersucht, welche Vor- und Nachteile durch den Einsatz eines nativen FOP-Compiler entstehen. Zunächst wurden SPL als eine Technik der Softwareentwicklung beschrieben, die es erlauben Varianten aus einer Quelltextbasis und effizient Quelltext wieder zu verwenden. Für die Implementierung wurden einige mögliche Techniken vorgestellt, zu denen auch die FOP gehört. Die FOP eignet sich besonders für die Implementierung von SPL, da Quelltext, der ein Feature beschreibt, modular in separaten Modulen gespeichert wird. Aus einer Menge von Features kann dann eine Variante der SPL erzeugt werden.

Mit AHEAD und FeatureHouse wurden zwei prominente Vertreter dieses Programmierparadigmas vorgestellt. Diese beiden Vertreter verwenden ein zweistufiges Verfahren. Ein Kompositionsprogramm komponiert den feature-orientierten Quelltext und erzeugt nativen Quelltext. Dieser native Quelltext wird dann von einem Standardcompiler in die Zielsprache übersetzt. Die Kompositionsprogramme von AHEAD und FeatureHouse können verschiedene Programmiersprachen unterschiedlicher Programmierparadigmen verarbeiten.

In Rahmen dieser Arbeit wurde nur Java als erweiterte Programmiersprache betrachtet. FeatureHouse und AHEAD setzen das Paradigma der FOP durch die Möglichkeit der Klassenverfeinerung um. Ein Feature kann in eine Basisimplementierung neue Klassen einführen oder bestehende Klassen durch neue Methoden und Felder verfeinern.

Durch die Möglichkeit der Klassenverfeinerung entstehen neue FOP spezifische Fehler, die bei der Komposition der Features auftreten können. In dieser Arbeit wurden einige dieser möglichen Fehler vorgestellt und gezeigt, wie AHEAD und FeatureHouse diese Fehler erkennen. Es konnte gezeigt werden, dass einige der Fehler nicht erkannt werden und das durch die Umsetzung der Klassenverfeinerung zusätzliche Fehlerquellen entstehen. Ein anderes Problem, das durch den zweistufigen Ansatz entsteht, ist das Fehlermeldungen vom Compiler, sich auf die Zwischendarstellung beziehen, die vom Kompositionsprogramm erstellt wurde. Die Abbildung der Fehler auf den eigentlichen

feature-orientierten Quelltext muss von einem Entwickler oder einer IDE erledigt werden, was zu zusätzlichen Aufwand führt.

Auf Grund der nicht immer befriedigenden Fehlererkennung und die Notwendigkeit der Zwischenabbildung und der daraus resultierende Mehraufwand bei der Abbildung der Fehlermeldung, bildete die Motivation, zu untersuchen, wie ein nativer FOP-Compiler diese Aufgaben lösen kann.

Dazu wurde im Rahmen dieser Arbeit ein Prototyp eines nativen FOP-Compilers erstellt. Da die Implementierung eines kompletten Compilers eine sehr aufwändige und gleichzeitig fehleranfällige Aufgabe ist, wurde das Compiler-Framework JastAddJ verwendet. Bei JastAddJ handelt es sich um einen Java-Compiler, der beispielsweise mit einer Aspekttechnologie erweitert werden kann. Mit diesem Framework wurde ein Compiler erstellt, der kompatibel zu AHEAD und FeatureHouse ist. Durch die Implementierung mit JastAddJ, ist der erstellte Compiler auch weiterhin durch die Mechanismen des Frameworks erweiterbar und stellt somit einen guten Ausgangspunkt für zukünftige Forschungen und Projekte in Bereichen der FOP und SPL dar.

Zur Evaluierung des Prototyps wurden einige Projekte, die für AHEAD oder FeatureHouse erstellt wurden, übersetzt. Dies diente dazu, das Laufzeitverhalten und die Funktionsfähigkeit zu testen. Es konnte gezeigt werden, dass der Prototyp ein besseres Laufzeitverhalten aufweist, als die zweistufigen Ansätze, die AHEAD und FeatureHouse verwenden. Eine Zwischendarstellung in Form von Java-Quelltext wird nicht benötigt und somit tritt das Problem mit der Abbildung der Fehlermeldungen nicht auf.

Die FOP-spezifischen Fehler können besser erkannt werden, da bei der Komposition der Features auf Funktionen des Compilers, zum Beispiel das Typsystem, zurück gegriffen werden kann.

7. Ausblick

In dieser Arbeit wurde ein Prototyp eines nativen FOP-Compilers entwickelt. Es konnte gezeigt werden, dass durch die Verwendung eines solchen Compilers besonders Vorteile bei der Fehlerüberprüfung entstehen.

Der Prototyp des native FOP-Compilers wurde mit dem JastAddJ-Compilerframework entwickelt, das sich besonders durch die gute Erweiterbarkeit auszeichnet. Der Prototyp ist auch weiterhin gut erweiterbar, so dass er als Grundlage für weitere Forschungen im Bereich der FOP und SPL genutzt werden kann.

Als Vorteil des nativen FOP-Compilers wurde genannt, das vor und während der Komposition auf Funktionen des Compilers , wie zum Beispiel das Typsystem, zurückgegriffen werden kann.

Bevor die Features komponiert werden, steht die gesamte SPL als AST zur Verfügung. Der Compiler kann an diesem Punkt erweitert werden, so dass die gesamte SPL mit Hilfe des Typsystems auf Fehler überprüft wird. Ein Typsystem zur Fehlerüberprüfung der SPL zuverwenden wurde beispielsweise von [Thü10] vorgeschlagen. Zusätzlich zur Featurereihenfolge und der eigentlichen Quelltextbasis kann das Featuremodell eingelesen werden um an zusätzliche Informationen zu gelangen. Mit diesen Information kann versucht werden, Typfehler innerhalb der SPL zu finden.

In [ALS07] wurden die Möglichkeiten der *Aspectual Feature Modules* diskutiert, die eine Kombination von Features und Aspekten vorsieht. Mit einem nativen FOP-Compiler besteht die Möglichkeit, dieses Konzept umzusetzen. Mit dem *AspectBench Compiler for AspectJ (abc-Compiler)*, steht ein Compiler für die Aspekt-orientierte Programmierung zur Verfügung, der mit JastAddJ erstellt wurde [AET08]. Auf Grund, dass das gleiche Compiler-Framework verwendet wird, kann beispielsweise der abc-Compiler um die Möglichkeit der Klassenverfeinerung erweitert werden. Die Umsetzung der *Aspectual Feature Modules* kann durch eine Kombination der beiden Compiler realisiert werden.

Bei dem nativen FOP-Compiler handelt es sich zur Zeit um einen Prototyp, so dass sicherlich noch Zeit für weitere Implementierungen und Fehlerbehebung investiert werden

kann. Auch bietet der Compiler sicherlich noch viel Raum für weitere Optimierungen, so dass das Laufzeitverhalten noch verbessert werden kann.

Ein mögliches Anwendungszenario stellt die Integration des Compilers in FeatureIDE dar. Der Compiler kann der IDE den gesamten AST der SPL zur Verfügung stellen. Mit dem AST können in FeatureIDE weitere „komfort-Funktionen“ wie Autovervollständigung oder Kollaborationsdiagramme realisiert werden. Die verbesserte Fehlererkennung hilft bei der Entwicklung von Software in FeatureIDE, in dem spezifische FOP-Fehler erkannt werden. Mit Hilfe des nativen FOP-Compilers und dem Eclipse-Framework kann in FeatureIDE ein Debugger entwickelt werden. Dieser Debugger kann dann direkt auf dem feature-orientierten Quelltext arbeiten.

A. Anhang A

In diesem Abschnitt werden die ANT-Skripte gezeigt, die für die Zeitmessung im Kapitel 5 verwendet wurden.

Der Quelltext A.1 zeigt das ANT-Skript, das für die Zeitmessung von TankWar verwendet wurde und soll Stellvertretend für alle Projekte, die mit AHEAD erstellt wurden, stehen. Zunächst wird ein Timer initialisiert. Im Anschluss wird zunächst der AHEAD-Composer aufgerufen. Dieser komponiert die Features, erzeugt aber als Ausgabe Jak-Dateien, die im Anschluss von dem Programm *jak2java* in Javadateien umgewandelt werden. Als letztes wird der Java-Compiler *javac* zum Compilieren verwendet.

Der Quelltext A.2 zeigt das ANT-Skript, das für die Zeitmessung von Projekten, die mit FeatureHouse komponiert werden, misst. Im Gegensatz zu der AHEAD-Variante wird nur der FeatureHouse-Composer aufgerufen und im Anschluss der *javac*-Javacompiler.

```
<project>
    <taskdef resource="net/sf/antcontrib/antlib.xml"/>
    <property name="expression"     value="TankWar"/>

    <target name="stoptime">
        <stopwatch name="timer1"/>
            <java jar="..\ahead\build\lib\composer.jar"
            fork="true">
        <arg line = "--equation␣${expression}.equation␣" />
        <arg line = "--target␣${expression}_build" />
        </java>
        <stopwatch name="timer1" action="elapsed"/>
        <for param="file">
            <path>
                <fileset dir="${expression}_build" includes="*.jak"/>
            </path>
            <sequential>
            <java jar="..\ahead\build\lib\jak2java.jar"
                fork="true">
            <arg line = "␣@{file}␣" />
            </java>
            </sequential>
        </for>
        <stopwatch name="timer1" action="elapsed"/>
        <javac srcdir="${expression}_build\"/>
        <stopwatch name="timer1" action="total"/>
        </target>

</project>
```

Quelltext A.1: ANT-Skript für die Featurekomposition mit AHEAD

```
<project>
    <taskdef resource="net/sf/antcontrib/antlib.xml"/>
    <property name="expression"     value="violet"/>

    <target name="stoptime">
        <stopwatch name="timer1"/>
            <java jar="..\FeatureHouse-2010-02-27.jar"
            fork="true">
        <arg line = "--expression␣${expression}.expression␣" />
        </java>
        <stopwatch name="timer1" action="elapsed"/>
        <javac srcdir="${expression}\"/>
        <stopwatch name="timer1" action="total"/>
    </target>
</project>
```

Quelltext A.2: ANT-Skript für die Featurekomposition mit FeatureHouse

B. Anhang B

In diesem Abschnitt soll die Funktionsweise des JastAdd-Frameworks anhand eines Beispiels detailliert gezeigt werden. Diese Art Anleitung kann für spätere Erweiterungen am Prototypen des nativen FOP-Compilers einen guten Einstieg in das Framework bieten.

PicoJava [1]stellt eine minimale Objekt-oriente Sprache dar, die nur Klassen, Vererbung und einfache Ausdrücke wie Variablen und Schleifen zulässt. Bei *PicoJava* handelt es sich daher um ein Beispielprojekt, das die Fähigkeiten des JastAdd Frameworks zeigen soll. Es existiert für diese Sprache nur ein Prüfprogramm (*PicoJavaChecker.java*), das ein Programm nur auf Syntax und semantische Fehler überprüft. Ein Back-End und dem entsprechend ein ausführbares Programm wird nicht erzeugt. Es wurde für diese Arbeit bewusst ein etwas umfangreicheres Beispiel gewählt, um möglichst viele Techniken des Frameworks zu zeigen. Diese Techniken wurden für die Implementierung in Kapitel 4 benötigt.

Der Scanner wird mit dem Programm JFlex [2] erzeugt. Als Schlüsselwörter erkennt der Scanner *class, extends, while* und dazu noch die Boolschen Ausdrücke *true* und *false*.

Als Parsergenerator wird Beaver [3] verwendet. Die Grammatik der PicoJava Sprache wird in Quelltext B.1 gezeigt. Die dort gezeigte Grammatik ist vom Aufbau und der Interpretation her, der Grammatik aus dem Quelltext 2.10 ähnlich. Anstelle des Pfeiles (->) wird eine Gleichheitszeichen (=) verwendet. Zusätzlich stehen hinter jeder Produktion, Anweisungen für die Erzeugung des ASTs. Die Grammatik beschreibt ein Programm, das aus einem Block besteht (Zeile 1). Ein Block besteht aus geschwungenen Klammern (LBRACE { , RBRACE }) und kann eine Liste von Block-Anweisungen (Statements) enthalten (Zeile 4-15). Diese Block-Anweisungen sind entweder Klassendeklarationen, Variablendeklarationen, oder Anweisungen (Zeile 17-19). Die Anweisungen werden dann nochmal in Zuweisungen (assign-statement) und Schleifen-Anweisungen

[1] http://jastadd.org/jastadd-tutorial-examples/picojava-checker
[2] http://jflex.de/
[3] http://beaver.sourceforge.net/

(while-statement) unterschieden (Zeile 21-22). Zeile 24-25 beschreibt den Aufbau einer Klassendeklaration. Diese besteht aus dem Schlüsselwort *class* einem Namen (*IDENTIFIER*) und einem optionalen *extends*-Block. Der *extends*-Block, der die Vererbung von Klassen implementiert wird in Zeile 27-30 beschrieben. Die restlichen Zeilen beschreiben in gleicher Art, den Aufbau von Variablendeklarationen, Zuweisungen, While-Schleifen und so weiter.

In Quelltext B.2 ist gültiges PicoJava Programm dargestellt, welches alle zuvor beschriebenen Sprachmittel enthällt. Der Parser erzeugt mit Hilfe, der zuvor gezeigten Grammatik, den AST. Hierfür stehen in der Grammatik aus Quelltext B.1 Anweisungen zur Erzeugung der Knoten. Hierbei handelt es sich um Java-Anweisungen (Aufrufen der Konstruktoren).

Die zu den Konstruktoren zugehörigen Klassen werden vom JastAdd Framework generiert. Die Knoten / Klassen werden in speziellen AST-Dateien beschrieben und verwenden als Endung **.ast*. Zu einem Knoten gehört ein Name, von welchen existierenden Knoten geerbt wird und welche Parameter bei der Erzeugung vom Parser übergeben werden. Die abstrakte Grammatik [4] in den AST-Dateien korrespondiert direkt mit der Klassenhierachie im AST. Damit kann der Begriff Knoten synonym mit dem Begriff Klasse verwendet werden. JastAdd bietet drei vorgegebene Knoten:

- AST-Knoten, dieser Knoten stellt grundlegende Eigenschaften und Methoden zur Verfügung. Hierzu zählen beispielsweise das Hinzufügen oder Entfernen von Kinder-Knoten
- List-Knoten, dieser Knoten stellt die Implementierung von Listen dar. Zusätzlich erbt dieser Knoten vom AST-Knoten
- OPT-Knoten, da Java keine variable Parameterlisten unterstützt, werden für optionale Parameter, OPT-Knoten als Wrapper verwendet. Bei der Erzeugung wird immer ein OPT-Knoten übergeben, der dann den optionalen Parameter enthalten kann. Dieser Knoten erbt ebenfalls vom AST-Knoten

Für die Sprache PicoJava werden die Knoten wie in Quelltext B.3 beschrieben. Der Ausdruck in Zeile 1 beschreibt, dass der Knoten *Program* einen Knoten *Block* als Kind hat. Ein Knoten *Block* kann optional Block-Anweisungen enthalten (gekennzeichnet durch das Sternchen *). Nichtterminalknoten werden durch Abstrakte-Knoten dargestellt (Zeile 3-6 und Zeile 11-13). Der Ausdruck in Zeile 4 besagt, dass der Knoten *Stmt* von dem Knoten *BlockStmt* erbt (Gekennzeichnet durch den Doppelpunkt :). In Zeile 7 wird der Knoten *ClassDecl* beschrieben. Der Knoten erbt von dem Knoten *TypeDecl*. Als Kinder kann eine Liste von Superklassen enthalten (Listen werden durch die eckigen Klammer [ ] gekennzeichnet). Der Ausdruck *Body:Block* bedeutet, dass der Knoten *ClassDecl* zusätzlich einen Knoten *Block* als Kind haben kann, der als Body benannt ist.

[4] http://jastadd.org/jastadd-reference-manual/abstract-syntax

Das letzte wichtige Konstrukt wird in Zeile 5 gezeigt. Der Knoten *Decl* erbt *BlockStmt* und hat als Kind einen String. Strings als Parameter werden in Spitzenklammern (<, >) gesetzt.
(Anmerkung: PicoJava bietet trotz der sehr eingeschränkten Sprachmittel, Typ-Überprüfung an. Hierfür werden weitere Knoten benötigt, die in der Grammatik nicht gezeigt wurden. Aus Gründen der Übersicht wird auf dieses Merkmal der Sprache PicoJava nicht näher eingegangen.)

JastAdd erzeugt nun die passenden Klassen und Funktionen die für den AST wichtig sein können. Als Beispiel ist die Klasse Block in Quelltext B.4 gezeigt, so wie sie vom Framework erzeugt wird. Die Methodenrümpfe wurden zur besseren Übersicht ausgeblendet. Die Knoten bieten für die weiterer Verwendung Methoden an, um auf Felder zu zugreifen oder Werte zu setzen. Des Weiteren existieren Methoden um Kinder zu AST hinzufügen oder zu entfernen.

```
1  Program goal =  block
2                  {: return new Program(block); :} ;
3
4  Block block =   LBRACE block_stmt_list_opt RBRACE
5                  {: return new Block(block_stmt_list_opt); :} ;
6
7  List block_stmt_list_opt =
8                  {: return new List(); :}
9              | block_stmt_list
10                 {: return block_stmt_list; :} ;
11
12 List block_stmt_list =  block_stmt
13                 {: return new List().add(block_stmt); :}
14             | block_stmt_list block_stmt
15                 {: return block_stmt_list.add(block_stmt); :} ;
16
17 BlockStmt block_stmt =  class_decl
18             | var_decl
19             | stmt ;
20
21 Stmt stmt =   assign_stmt
22             | while_stmt ;
23
24 ClassDecl class_decl =   CLASS IDENTIFIER extends_opt block
25                 {: return new ClassDecl(IDENTIFIER, extends_opt, block); :} ;
26
27 Opt extends_opt =
28                 {: return new Opt(); :}
29             | EXTENDS IDENTIFIER
30                 {: return new Opt(new Use(IDENTIFIER)); :} ;
31
32 VarDecl var_decl =  name IDENTIFIER SEMICOLON
33                 {: return new VarDecl(IDENTIFIER, name); :} ;
34
35 AssignStmt assign_stmt =    name ASSIGN exp SEMICOLON
36                 {: return new AssignStmt(name, exp); :} ;
37
38 WhileStmt while_stmt =  WHILE LPAREN exp RPAREN stmt
39                 {: return new WhileStmt(exp, stmt); :} ;
40
41 Exp exp =       name
42             | boolean_literal ;
43
44 Access name =   IDENTIFIER
45                 {: return new Use(IDENTIFIER); :}
46             | name DOT IDENTIFIER
47                 {: return new Dot(name, new Use(IDENTIFIER)); :} ;
48
49  Exp boolean_literal =  BOOLEAN_LITERAL
50                 {: return new BooleanLiteral(BOOLEAN_LITERAL); :} ;
```

Quelltext B.1: Auszug aus der PicoJava.parser Datei

```
{
  class A {
    boolean a;
    a = true;
    class AA {
      boolean aa;
    }
  }
  class B extends A {
    boolean b;
    b = a;
    A refA;
    B refB;
    refA = refB;
    refB.b = refA.a;
    class BB extends AA {
      boolean bb;
      bb = aa;
      while (b)
        b = a;
    }
  }
}
```

Quelltext B.2: Beispiel für ein gültiges PicoJava-Programm

```
Program ::= Block ;
Block ::= BlockStmt*;
abstract BlockStmt;
abstract Stmt: BlockStmt;
abstract Decl: BlockStmt ::= <Name:String>;
abstract TypeDecl:Decl;
ClassDecl: TypeDecl ::= [Superclass:IdUse] Body:Block;
VarDecl: Decl ::= Type:Access;
AssignStmt: Stmt ::= Variable:Access Value:Exp;
WhileStmt: Stmt ::= Condition:Exp Body:Stmt;
abstract Exp;
abstract Access:Exp;
abstract IdUse: Access ::= <Name:String>;
Use: IdUse;
Dot:Access ::= ObjectReference:Access IdUse;
BooleanLiteral : Exp ::= <Value:String>;
```

Quelltext B.3: Beschreibung der AST-Knoten für PicoJava

```
1  public class Block extends ASTNode implements Cloneable {
2      public Block() {...}
3
4      public Block(List p0) {...}
5
6      public Object clone() throws CloneNotSupportedException {...}
7
8      public ASTNode copy() {...}
9
10     public ASTNode fullCopy() {...}
11
12     public void flushCache() {...}
13
14     protected int numChildren() {...}
15
16     public boolean mayHaveRewrite() { ...}
17
18     public void setBlockStmtList(List list) {...}
19
20     public int getNumBlockStmt() {...}
21
22     public BlockStmt getBlockStmt(int i) {...}
23
24     public void addBlockStmt(BlockStmt node) {...}
25
26     public void setBlockStmt(BlockStmt node, int i) {...}
27
28     public List getBlockStmtList() {...}
29
30     public List getBlockStmtListNoTransform() {...}
31
32     public ASTNode rewriteTo() {...}
33 }
```

Quelltext B.4: Von JastAdd generierte Klasse Block

Literaturverzeichnis

[ADT07] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. On Refining XML Artifacts. In *Proceedings of the International Conference on Web Engineering (ICWE), volume 4607 of Lecture Notes in Computer Science*, pages 473–478. Springer-Verlag, Berlin / Heidelberg, 2007. (zitiert auf Seite 13)

[AET08] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity first: a case for mixing AOP and attribute grammars. In *Proceedings of International Conference on Aspect-oriented software development (AOSD)*, pages 25–35, New York, NY, USA, 2008. ACM Press. (zitiert auf Seite 75)

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173, Springer-Verlag, Berlin / Heidelberg, 2006. (zitiert auf Seite 30)

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Guest Column. (zitiert auf Seite 6 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 33)

[AKL08] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 101–112. ACM Press, October 2008. (zitiert auf Seite 33 und 45)

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society. (zitiert auf Seite 2, 16, 18, 31, 39, 58, 66 und 70)

[AL08] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proceedings of the*

International Symposium of Software Composition (SC), pages 20–35, Budapest, Hungary, March 2008. Springer-Verlag, Berlin / Heidelberg. (zitiert auf Seite 66)

[ALK+09] Sven Apel, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. An Orthogonal Access Modifier Model for Feature-Oriented Programming. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 27–33, New York, NY, USA, 2009. ACM Press. (zitiert auf Seite 36)

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, July 2008. (zitiert auf Seite 16, 19, 38 und 58)

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, Berlin / Heidelberg, September 2005. (zitiert auf Seite 13)

[ALS07] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34:162–180, 2007. (zitiert auf Seite 75)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer-Verlag, Berlin / Heidelberg, 2005. (zitiert auf Seite 6, 7 und 66)

[BCK05] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, Boston ; Munich [u.a.], 2005. (zitiert auf Seite 1 und 6)

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197, May 3–10, 2003. (zitiert auf Seite 1, 2, 12 und 13)

[CDS06] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), workshop on Role of software architecture for testing and analysis (ROSATEA)*, pages 53–63, New York, NY, USA, 2006. ACM Press. (zitiert auf Seite 8)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, New York, NY, USA, 2000. (zitiert auf Seite 6)

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA, 2006. ACM. (zitiert auf Seite 33)

[DA99] David Detlefs and Ole Agesen. Inlining of Virtual Methods. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 258–278, London, UK, 1999. Springer-Verlag, Berlin / Heidelberg. (zitiert auf Seite 18)

[Dij82] Edsger Wybe Dijkstra. *Selected writings on computing: a personal perspective*. Springer-Verlag New York, Inc., New York, NY, USA, 1982. (zitiert auf Seite 8)

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. (zitiert auf Seite 8)

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *Proceeding of the European Conference on Object-Oriented Programming (ECOOP)*, pages 144–169. Springer-Verlag, Berlin / Heidelberg, 2004. (zitiert auf Seite 24)

[EH07a] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Companion to the conference on Object-oriented programming systems and applications (OOPSLA)*, pages 884–885, New York, NY, USA, 2007. ACM Press. (zitiert auf Seite 24)

[EH07b] Torbjörn Ekman and Görel Hedin. The jastadd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007. (zitiert auf Seite 23)

[Ekm06] Torbjörn Ekman. *Extensible Compiler Construction*. PhD thesis, Department of Computer Science, Lund University, 2006. (zitiert auf Seite 23 und 25)

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999. (zitiert auf Seite 29)

[GE99] Ralf Hartmut Güting and Martin Erwig. *Übersetzerbau - Techniken, Werkzeuge, Anwendungen*. Springer-Verlag, Berlin / Heidelberg, 1999. (zitiert auf Seite 20 und 22)

[GJ06] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques (Monographs in Computer Science)*. Springer-Verlag, Berlin / Heidelberg, Secaucus, NJ, USA, 2006. (zitiert auf Seite 21)

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005. (zitiert auf Seite 36)

[HM03] Görel Hedin and Eva Magnusson. Jastadd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003. (zitiert auf Seite 23)

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin / Heidelberg, 2005. (zitiert auf Seite 33)

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. (zitiert auf Seite 11)

[KA08] C. Kastner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267, Washington, DC, USA, 2008. IEEE Computer Society. (zitiert auf Seite 33)

[KA09] Christian Kästner and Sven Apel. Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, September 2009. Refereed Column. (zitiert auf Seite 10)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software engineering (ICSE)*, pages 311–320, New York, NY, USA, 2008. ACM Press. (zitiert auf Seite ix, 9, 10 und 11)

[KAK09] Christian Kästner, Sven Apel, and Martin Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, October 2009. (zitiert auf Seite 66)

[KAL07] Martin Kuhlemann, Sven Apel, and Thomas Leich. Streamlining feature-oriented designs. In *Software Composition*, pages 168–175. Springer-Verlag, 2007. (zitiert auf Seite 14 und 18)

[KAS10] Christian Kästner, Sven Apel, and Gunter Saake. Virtuelle Trennung von Belangen (Präprozessor 2.0). In *Software Engineering 2010 – Fachtagung*

des GI-Fachbereichs Softwaretechnik, number P-159 in Lecture Notes in Informatics, pages 165–176. Gesellschaft für Informatik (GI), February 2010. (zitiert auf Seite 10)

[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, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, November 1990. (zitiert auf Seite 6)

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001. Springer-Verlag. (zitiert auf Seite 11)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer-Verlag, 1997. (zitiert auf Seite 8, 9 und 11)

[KR90] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C : mit dem C-Reference-Manual in deutscher Sprache - 2. Ausg., ANSI C*. Hanser, München [u.a.], 1990. (zitiert auf Seite 9)

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE Computer Society Press, 1994. (zitiert auf Seite 10)

[KS09] Christian Kästner and Gunter Saake. Vorlesung: Erweiterte Programmierkonzepte für maßgeschneiderte Datenhaltung (EPMD). University of Magdeburg, 2008-2009. (zitiert auf Seite ix, 12 und 65)

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Preprocessors 2.0*. PhD thesis, University of Magdeburg, School of Computer Science, 2010. (zitiert auf Seite ix und 8)

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society. (zitiert auf Seite 6 und 41)

[LAMS05] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool Support for Feature-Oriented Software Development: FeatureIDE: an Eclipse-based approach. In *Proceedings of the Object-Oriented Programming, Systems,*

Languages & Apllications (OOPSLA), workshop on Eclipse technology eXchange, pages 55–59, New York, NY, USA, 2005. ACM Press. (zitiert auf Seite 41)

[LhB01] Roberto E. Lopez-herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the Conference on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Messe Erfurt, Erfurt, Germany, Springer-Verlag, Berlin / Heidelberg, 2001. (zitiert auf Seite 65)

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *Proceedings of the International Symposium on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 68–77, New York, NY, USA, 2006. ACM Press. (zitiert auf Seite 33)

[LR09] Bernhard Lahres and Gregor Rayman. *Praxisbuch Objektorientierung: Das umfassende Handbuch*. Galileo Press, Bonn, 2. edition, 2009. (zitiert auf Seite 8)

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 138–152. Springer-Verlag, Berlin / Heidelberg, 2003. (zitiert auf Seite 23)

[NTJ06] Clémentine Nebut, Yves Le Traon, and Jean-Marc Jézéquel. System testing of product lines: From requirements to test cases. In *Software Product Lines*, page 447. Springer-Verlag, 2006. (zitiert auf Seite 8)

[Par72] David. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. (zitiert auf Seite 8)

[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 6)

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. (zitiert auf Seite 22 und 61)

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceeding of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer-Verlag, Berlin / Heidelberg, 1997. (zitiert auf Seite 1 und 12)

[Sar03] Jacob N. Sarvela. The bali language. Verfügbar unter: http://userweb.cs.utexas.edu/users/schwartz/ATS/fopdocs/bali.pdf, May 2003. (zitiert auf Seite 16)

[SB02] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002. (zitiert auf Seite 14 und 18)

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley Longman, Amsterdam, 2nd ed. (15. november 2002) edition, 2002. ISBN-13: 978-0201745726. (zitiert auf Seite 10)

[SLA08] Ravi Sethi, Monica S. Lam, and Alfred V. Aho. *Compiler. Prinzipien, Techniken und Tools (Pearson Studium): Prinzipien, Techniken und Werkzeuge*. Pearson Education, München, 2. Auflage edition, January 2008. (zitiert auf Seite 20)

[Spe92] Henry Spencer. ifdef Considered Harmful, or Portability Experience with C News. In *Proceeding of the Summer'92 USENIX Conference*, pages 185–197, 1992. (zitiert auf Seite 10)

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, 2006. (zitiert auf Seite 11)

[Sys06] Andreas Syska. *Produktionsmanagement*. Betriebswirtschaftlicher Verlag Dr. Th. Gabler GWV Fachverlage GmbH, Wiesbaden, 2006. (zitiert auf Seite 5)

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the international conference on Generative programming and component engineering (GPCE)*, pages 95–104, New York, NY, USA, 2007. ACM Press. (zitiert auf Seite 2, 29, 30, 31, 33, 38 und 45)

[Thü10] Thomas Thüm. A Machine-Checked Proof for a Product-Line Aware Type System. (Diplomarbeit), University of Magdeburg, 2010. (zitiert auf Seite 33 und 75)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119, New York, NY, USA, 1999. ACM Press. (zitiert auf Seite 8)

[UGKB08] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 249–258, Washington, DC, USA, 2008. IEEE Computer Society. (zitiert auf Seite 8)

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical report, University of Amsterdam, Programming Research Group, 1997. (zitiert auf Seite 20)

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971. (zitiert auf Seite 33)

[ZO01] Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP), workshop on Multiparadigm Programming with Object-Oriented Languages*, pages 61–80. ACM Press, 2001. (zitiert auf Seite 23)

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

Magdeburg, den 24.05.2010