

Universität Passau  
Fakultät für Informatik und Mathematik

# Alternative Features in Colored Featherweight Java

Diplomarbeit

Autor:  
Malte Rosenthal

Aufgabensteller:  
Prof. Christian Lengauer, Ph.D.  
Lehrstuhl für Programmierung  
Universität Passau

Betreuer:  
Dr.-Ing. Sven Apel,  
Dipl.-Wirt.-Inform. Christian Kästner

23.07.2009

**Rosenthal, Malte:**

*Alternative Features in Colored Featherweight Java*

Diplomarbeit, Universität Passau, 2009.

---

---

# Zusammenfassung

---

Die Entwicklung von *Software-Produktlinien* ist eine anspruchsvolle Aufgabe. Da Objekt-Orientierte Programmierung in diesem Zusammenhang auf ihre Grenzen stößt, wurden dazu neue Techniken und Ansätze entwickelt. Diese Diplomarbeit basiert auf dem Eclipse-Plugin *CIDE*, einer Implementierung des *Annotationsansatzes*, bei dem Codefragmente als zu einem oder mehreren *Features* zugehörig annotiert werden. Konnte die syntaktische Korrektheit aller Varianten in CIDE noch mit einfachen Regeln gewährleistet werden, wurde die sprachsemantische Korrektheit bisher noch nicht überprüft. Diese Diplomarbeit entwickelt daher ein Typsystem für die Software-Produktlinien-Sprache *CFJ*, eine Erweiterung von *Featherweight Java* um CIDE-Annotationen.

Aus der Anwendungsdomäne der Software-Produktlinie ergeben sich diverse Restriktionen an die Features. So kann es die Bedingung geben, dass sich zwei Features *gegenseitig ausschließen*, d. h. in jeder Variante darf höchstens oder genau eines der beiden Features vorhanden sein. In diesem Fall spricht man von *alternativen Features*. Es ist mit bisherigen Mitteln in CIDE nicht in jedem Fall möglich, alternative Features zu implementieren. Daher wird in dieser Arbeit ein Mechanismus zum Anlegen und Editieren alternativer Codefragmente entwickelt. Außerdem wird das CFJ-Typsystem so erweitert, dass alternative Features zusätzlich berücksichtigt werden.



---

---

# Inhaltsverzeichnis

---

<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Abkürzungsverzeichnis</b>	<b>vii</b>
<b>1 Einleitung und Zielsetzung</b>	<b>1</b>
1.1 Korrektheit der Varianten im Annotationsansatz . . . . .	2
1.2 Alternative Features . . . . .	3
<b>2 Grundlagen</b>	<b>5</b>
2.1 SPL-Entwicklungsmodelle . . . . .	5
2.2 Implementierung von Software-Produktlinien . . . . .	6
2.3 CIDE . . . . .	8
2.3.1 Entwicklungsprozess . . . . .	8
2.3.2 Vergleich mit Präprozessoren . . . . .	12
2.3.3 Syntaktische Korrektheit und Granularität . . . . .	13
2.3.4 Sprachunabhängigkeit . . . . .	14
2.3.5 Zusammenfassung . . . . .	14
<b>3 Colored Featherweight Java</b>	<b>17</b>
3.1 Grundlagen . . . . .	18
3.2 Syntax . . . . .	20
3.3 Typsystem . . . . .	22
3.3.1 Wohlgeformtheit eines CFJ-Programms . . . . .	22
3.3.2 Einfluss des Featuremodells . . . . .	23
3.3.3 Formalisierung von Annotationen . . . . .	25
3.3.4 Einschränkung gültiger CFJ-Programme . . . . .	25
3.3.5 CFJ-Typregeln . . . . .	26
<b>4 Alternative Features in CFJ</b>	<b>37</b>
4.1 Problemstellung . . . . .	37
4.2 Andere Ansätze . . . . .	39
4.2.1 Alternative Methoden durch Overriding . . . . .	39
4.2.2 Erweiterung des Sprachumfangs . . . . .	40
4.2.3 Exkurs: Alternative Rückgabe-Statements in Java . . . . .	41
4.3 Prinzip der Meta-Expressions . . . . .	42
4.4 Realisierung in CIDE . . . . .	42
4.4.1 Featuremodell . . . . .	43
4.4.2 Entwicklungsprozess . . . . .	43
4.5 Überprüfung des gegenseitigen Ausschlusses aller Alternativen . . . . .	48

4.6	Typsystem . . . . .	51
4.6.1	Ausschluss von AST-Knoten für Alternativen . . . . .	51
4.6.2	Erweiterung der CFJ-Typregeln . . . . .	56
4.6.3	Rechtfertigung des Mechanismus für alternative Features .	56
<b>5</b>	<b>Zusammenfassung und Ausblick</b>	<b>59</b>
5.1	Zusammenfassung . . . . .	59
5.2	Ausblick . . . . .	60
	<b>Literaturverzeichnis</b>	<b>63</b>

---

---

# Abbildungsverzeichnis

---

2.1	Konfiguration eines Featuremodells in CIDE . . . . .	9
2.2	Färbung eines Codefragments in CIDE . . . . .	10
2.3	Gefärbte Codefragmente in CIDE mit Tooltip . . . . .	10
2.4	Auswahl von Features bei der Erzeugung einer Variante . . . . .	11
2.5	Erzeugte Variante des Projekts <i>Stack</i> . . . . .	11
2.6	Annotieren zweier Methodenparameter in CIDE . . . . .	13
3.1	Aufruf einer gefärbten Methode . . . . .	17
3.2	Einfluss des Featuremodells auf das Typsystem . . . . .	23
3.3	Alternative Deklarationen einer Methode . . . . .	25
3.4	Überschreiben alternativer Methoden . . . . .	29
3.5	Überschreiben von Methoden entlang der Vererbungshierarchie . .	30
3.6	Ungültiges Überschreiben einer Methode . . . . .	32
4.1	Alternative Methoden durch Overriding . . . . .	40
4.2	Alternative Rückgabe-Statements in Java . . . . .	41
4.3	Featuremodell mit alternativen Features . . . . .	43
4.4	Anlegen einer Alternative zu einer Methodendeklaration . . . . .	44
4.5	Eingabe eines Namens der Alternative . . . . .	44
4.6	Die neu angelegte Alternative . . . . .	45
4.7	Die neu angelegte Alternative – editiert und annotiert . . . . .	45
4.8	Wechseln der aktiven Alternative . . . . .	46
4.9	Alternative Features als Baumstruktur . . . . .	47
4.10	Beispiel für kombinatorische Explosion der Typprüfungen . . . . .	53





---

---

## Abkürzungsverzeichnis

---

<b>AST</b>	Abstract Syntax Tree
<b>CFJ</b>	Colored Featherweight Java
<b>CIDE</b>	Colored Integrated Development Environment
<b>CST</b>	Concrete Syntax Tree
<b>FFJ</b>	Feature Featherweight Java
<b>FJ</b>	Featherweight Java
<b>FOR</b>	Feature-Orientierte Refaktorisierung
<b>GPL</b>	Graph Product Line
<b>JDT</b>	Java Development Tools
<b>OOP</b>	Objekt-Orientierte Programmierung
<b>SPL</b>	Software-Produktlinie



---

---

# KAPITEL 1

---

## Einleitung und Zielsetzung

Heutige Software wird in vielen Fällen nicht mehr nur für einen einzelnen Kunden individuell entwickelt, sondern soll einen möglichst großen Kundenkreis ansprechen. Dies bedeutet nicht nur, dass Programme vom Benutzer auf seine eigenen Bedürfnisse hin eingestellt werden können müssen (z.B. unterschiedliches Erscheinungsbild des Programms), sondern führt auch dazu, dass ein Programm je nach Anwender unterschiedliche Funktionalität aufweisen muss. So gibt es beispielsweise verschiedene Varianten des Microsoft Windows Betriebssystems oder der Microsoft Office Büro-Software. Funktionalitäten, durch deren Auswahl sich die unterschiedliche Funktionalität der Varianten *für den Benutzer* ergibt, nennt man *Features* [KCH<sup>+</sup>90]. Ein Feature ist also eine Funktionalität des Programms, die für einen Benutzer von Bedeutung ist. So ist z. B. das Format der Dateien in Microsoft Office (z. B. binär oder als XML-Struktur) für den größten Benutzerkreis kein Feature, wohingegen die Tabellenkalkulation oder die Projektmanagement-Funktionalität als Features (ggfs. mit weiteren Subfeatures) bezeichnet werden können.

Trotz der hohen Variabilität solcher Softwaresysteme bauen alle Varianten auf einer gemeinsamen Basis auf. Diese Basis besteht v. a. aus Gemeinsamkeiten, die innerhalb der Anwendungsdomäne bestehen. So stellen alle Varianten von Microsoft Windows eine Schnittstelle zur Hardware zur Verfügung (die eigentliche Aufgabe eines Betriebssystems), wohingegen nur manche Versionen eine vollständige Benutzerverwaltung mit Zugriffsrechten enthalten.

Ein Software-System, das auf einer gemeinsamen Basis aufbaut und in unterschiedlichen Varianten vorliegen kann, nennt man *Software-Produktlinie (SPL)* [CN02, PBv05]. Eine SPL entsteht also aus einer Code-Basis durch Erweiterung um Features. Somit muss nicht jede mögliche Variante von Grund auf neu geschrieben werden.

Eine Schwierigkeit bei der Entwicklung einer SPL besteht darin, dass es bereits bei geringer Anzahl von Features nicht mehr in angemessener Zeit möglich ist, alle Varianten der SPL zu testen<sup>1</sup>. Außerdem stößt die in heutigen Softwareprojekten weit verbreitete *Objekt-Orientierte Programmierung (OOP)* [Mey97] im Hinblick auf das grundlegende Software-technische Konzept der *Trennung von Belangen* (engl. *separation of concerns*) [Par72] auf ihre Grenzen [TOHS99]. Beispiele für solche Probleme sind die *Tyrannie der dominanten Dekomposition* oder das *Extensibility Problem* [TOHS99]. Dabei ist eine Trennung von Belangen bei

---

<sup>1</sup> Bei  $n$  Features gibt es  $2^n$  mögliche Varianten, falls alle Kombinationen von Features erlaubt sind.

der Entwicklung einer SPL besonders wichtig, da diese ja gerade aus einer Menge von Belangen (Features) besteht.

Die Entwicklung einer SPL kann unter verschiedenen Voraussetzungen (siehe Abschnitt 2.1 auf Seite 5) und mit Hilfe unterschiedlicher Ansätze und Techniken (siehe Abschnitt 2.2 auf Seite 6) wie z. B. dem *Annotationsansatz* erfolgen. Sie wurden u. a. dazu entwickelt, die soeben angesprochenen Probleme z. B. von OOP zu lösen oder zumindest abzumildern. Apel und Kästner [AK09] geben dazu einen umfassenden Überblick, insbesondere zum Paradigma der *Feature-Orientierten Softwareentwicklung*, engl. *Feature-oriented software development (FOSD)*. Die vorliegende Arbeit basiert auf einer Implementierung des Annotationsansatzes innerhalb des Eclipse<sup>2</sup>-Plugins *Colored Integrated Development Environment (CIDE)* (siehe Abschnitt 2.3 auf Seite 8).

## 1.1 Korrektheit der Varianten im Annotationsansatz

Im Annotationsansatz werden Codefragmente innerhalb einer Code-Basis denjenigen Features zugeordnet, die sie implementieren. Diese Zuordnung erfolgt durch *Annotation* des Codefragments mit den entsprechenden Features. Eine Möglichkeit, ein Codefragment mit einem Feature zu annotieren, ist die Umschließung mit Präprozessor-Direktiven wie `#ifdef FEATURE1 ... #endif` (siehe Abschnitt 2.2 auf Seite 6). Eine Variante der SPL ist durch die Auswahl einer Menge von Features eindeutig definiert. Sie entsteht aus der Code-Basis, indem all diejenigen Codefragmente gelöscht werden, die mit nicht ausgewählten Features annotiert sind. Die Variante als ausführbares Programm entsteht dann durch Kompilieren des so entstandenen Quellcodes.

Besteht die Möglichkeit, beliebige Codefragmente zu annotieren, so können manche Varianten syntaktisch inkorrekte Programme sein. Annotiert man beispielsweise das Semikolon am Ende einer Anweisung mit *Feature1*, so sind alle Varianten, die *Feature1* nicht enthalten, syntaktisch inkorrekte Programme, weil dort das Semikolon fehlt. Kästner et al. haben Bedingungen an die Annotationen gestellt, die gewährleisten, dass jede mögliche Variante syntaktisch korrekt ist [KAK08] (siehe detailliertere Ausführungen im Abschnitt 2.3 auf Seite 8).

Die syntaktische Korrektheit eines Programms reicht in vielen Sprachen alleine jedoch nicht aus. So können manche Annotationen dazu führen, dass eine Variante Typfehler enthält (z. B. Aufruf einer Methode, die in der Variante nicht existiert). Konnte die syntaktische Korrektheit aller möglichen Varianten noch unabhängig von der zugrunde liegenden Sprache gewährleistet werden [KAT<sup>+</sup>09], ist dies bei der Überprüfung auf Typfehler nicht mehr möglich. Zu einem der Ziele dieser Diplomarbeit gehört es somit, ein Typsystem für die Programmiersprache *Featherweight Java (FJ)* [Pie02] – eine funktionale Teilmenge von Java – innerhalb von CIDE zu implementieren und durch Berücksichtigung von Annotationen auf ein Typsystem für *Colored Featherweight Java (CFJ)* [KA08b] zu erweitern. Dadurch, dass FJ nur eine geringe Menge der Java-Sprachkonstrukte umfasst,

---

<sup>2</sup> [www.eclipse.org](http://www.eclipse.org)

können somit Erkenntnisse, inwiefern ein Typsystem Feature-Annotationen berücksichtigen muss, einfacher gewonnen werden, als bei der Verwendung von Java als zugrunde liegende Programmiersprache.

## 1.2 Alternative Features

Features können in einer SPL unter verschiedenen Bedingungen, die sich aus der Anwendungsdomäne ergeben, existieren. So gibt es z. B. Features, die in einer Variante vorhanden sein *können*, aber nicht müssen (*optionale* Features). Es können aber auch komplexe Bedingungen wie *Ist Feature1 in einer Variante vorhanden, so muss in dieser Variante auch Feature2 vorhanden sein* gegeben sein. Die Menge aller vorhandenen Features und deren Beziehungen zueinander werden in einem *Featuremodell* beschrieben (siehe Abschnitt 2.3.1 auf Seite 8).

Auf einem weiteren Beispiel für eine Bedingung im Featuremodell basieren *alternative Features*. Alternative Features sind Features, die sich gegenseitig ausschließen. Ein Beispiel ist die Softwaresteuerung eines Getriebes in einem PKW. *Entweder* handelt es sich um eine Steuerung für ein manuelles Getriebe *oder* für ein automatisches Getriebe. Bei optionalen Sprachkonstrukten (im Sinne der Syntax) bedeutet dies, dass in jeder möglichen Variante *höchstens eine* der möglichen Alternativen vorhanden sein darf. Bei nicht-optionalen Sprachkonstrukten muss in jeder Variante *genau eine* der Alternativen vorhanden sein.

Bei optionalen Sprachkonstrukten, wie z. B. formalen Parametern einer Methodendeklaration, könnte man Alternativen dadurch angeben, dass man sie hintereinander schreibt und entsprechend annotiert. Als Beispiel betrachte man Listing 1.1<sup>3</sup>.

```
Object someMethod(#ifdef ALTERNATIVE1 SomeClass p1 #endif
                  #ifdef ALTERNATIVE2 OtherClass p1 #endif) { /* ... */ }
```

Listing 1.1: Beispiel für die Realisierung alternativer Methodenparameter

Schließen sich die Features *ALTERNATIVE1* und *ALTERNATIVE2* gegenseitig aus, so ist in jeder Variante höchstens eine der beiden Parameter vorhanden. Dabei heißt dieser Parameter stets *p1*, so dass beim Zugriff auf diesen Parameter keine Feature-Spezifika berücksichtigt werden müssen.

In CIDE ist es zum Erhalt der syntaktischen Korrektheit nur möglich, optionale Sprachkonstrukte zu annotieren. Es kann aber auch von Interesse sein, z. B. alternative Rückgabetypen zu einer Methode, also Alternativen zu nicht-optionalen Codefragmenten, anzugeben. So könnte eine Methode in der einen Variante z. B. einen *BufferedStream*, in einer anderen Variante einen *UnbufferedStream* zurückgeben<sup>4</sup>. Es muss also ein Mechanismus gefunden werden, Alternativen zu

<sup>3</sup> Zur besseren Lesbarkeit weichen wir etwas von der offiziellen Syntax in C/C++ Präprozessoren ab.

<sup>4</sup> Die Möglichkeit der Verwendung z. B. von *Java-Generics* oder *C++ Templates* soll hier nicht diskutiert werden, da der in dieser Arbeit entwickelte Mechanismus z. B. auch in OOP-Sprachen ohne solche Sprachkonstrukte funktionieren soll.

nicht-optionalen Codefragmenten anzugeben. Um diesen Mechanismus einheitlich zu verwenden (und aufgrund einer wichtigen *Grundsatzentscheidung*, die obiges Beispiel verbieten wird, siehe Abschnitt 3.3.4 auf Seite 25), wird er auch bei optionalen Sprachkonstrukten angewandt.

Neben der bereits erwähnten Implementierung eines Typsystems für FJ und CFJ steht an zentraler Stelle dieser Diplomarbeit die Implementierung eines Mechanismus zur Verwaltung alternativer Codefragmente. Dieser Mechanismus speichert Alternativen außerhalb der Codebasis ab, wobei das Anlegen, Editieren und Annotieren einer Alternative innerhalb der Codebasis im Editor erfolgen kann. Außerdem müssen auch die Überprüfungen, die die syntaktische Korrektheit aller Varianten der SPL garantieren, erweitert werden. Auch das implementierte CFJ-Typsystem muss angepasst werden, um Alternativen zusätzlich zu berücksichtigen.

Es gibt weitere Möglichkeiten, alternative Features innerhalb von CIDE zu realisieren, die in dieser Arbeit diskutiert und mit dem implementierten Mechanismus verglichen werden sollen. Manche Vorgehensweisen können dabei nicht zu jeder Art von Codefragment Alternativen angeben, haben jedoch den Vorteil, dass die Typprüfungen einfacher und schneller durchgeführt werden können.

Alle Implementierungen erfolgten innerhalb des bestehenden Eclipse-Plugins CIDE. Dabei mussten nahezu alle wichtigen Bereiche des Plugins angepasst oder erweitert werden. Daher beleuchtet diese Diplomarbeit die eher technischen und algorithmischen Aspekte der Typsysteme und verzichtet auf die streng formale Darstellung der Typregeln, jedoch ohne auf die nötige Präzision zu verzichten.

---

---

# KAPITEL 2

---

## Grundlagen

Wie eingangs bereits erwähnt, basiert die vorliegende Diplomarbeit auf dem Eclipse-Plugin CIDE, das einen Annotationsansatz zur Entwicklung einer SPL verfolgt. Dazu sollen in diesem Kapitel die notwendigen Grundlagen vermittelt werden, damit die eigentlichen Ausführungen zu den Typsystemen und alternativen Features innerhalb dieses Rahmens erfolgen können.

### 2.1 SPL-Entwicklungsmodelle

Die Entwicklung einer SPL kann unter verschiedenen Voraussetzungen erfolgen. Zum einen kann ein Programm erst im Laufe seiner Entstehung oder auch erst nach seiner Fertigstellung zu einer vollwertigen SPL entwickelt werden, z. B. wenn sich der Markt innerhalb dieser Domäne verändert, entsprechende unternehmerische Entscheidungen dazu führen oder Gemeinsamkeiten verschiedener Produkte erst zu einem späteren Zeitpunkt erkannt werden. Zum anderen kann eine SPL bereits von Anfang an als SPL entworfen und entwickelt werden. Grundsätzlich sind bei der Beurteilung folgender Modelle gerade diese Voraussetzungen zu berücksichtigen.

Krueger [Kru02] unterscheidet zwischen drei sog. *Entwicklungsmodellen* (engl. *adoption models*). Beim *proaktiven* Entwicklungsmodell wird die SPL bereits zu Beginn auch als solche geplant. Dieser dem konventionellen Wasserfallmodell [Bal96] ähnliche Ansatz bringt Nachteile mit sich. So stellt der anfängliche Entwurf des Software-Systems einen erheblichen Aufwand dar, dem jedoch noch kein fertiges Produkt gegenübersteht. Dieser Aufwand zahlt sich aber in vielen Fällen in einer deutlich erhöhten Effizienz in der Implementierung des Programms aus. Nichtsdestotrotz legt man sich bei diesem Verfahren bereits sehr früh auf bestimmte Funktionalität und Strukturen fest, so dass auf nicht vorausgesehene Bedürfnisse oft nur in Verbindung mit hohen Kosten reagiert werden kann. Hinzu kommt, dass sich viele Programme erst zu einem späten Zeitpunkt zu einer SPL entwickeln, so dass im proaktiven Entwicklungsmodell ein großer Teil der Anwendung von Grund auf neu entworfen und entwickelt werden muss.

Diese Nachteile motivieren die Nutzung des *reaktiven* Entwicklungsmodells. Bei diesem iterativen Verfahren wird ausgehend vom bestehenden Produkt auf neue Anforderungen entsprechend reagiert. Dies ist z. B. dann notwendig, wenn im Rahmen einer proaktiven Vorgehensweise nicht mit bestimmten Erweiterungen gerechnet wurde.

Im Gegensatz zum proaktiven Ansatz steht das *extraktive* Entwicklungsmodell. Es geht davon aus, dass eine oder mehrere bereits entwickelte Anwendungen (engl. *legacy applications*) vorliegen, die als SPL umgesetzt werden sollen, ohne das Programm von Grund auf neu zu entwickeln. Dieser Prozess der Dekomposition eines Programms in Features wird auch als *Feature-Orientierte Refaktorisierung (FOR)* bezeichnet [LBL06]. Die Schwierigkeit und Komplexität besteht darin, bereits existierende Features in der Altanwendung zu identifizieren und zusätzlich als ein zusammengehöriges Inkrement innerhalb der SPL darzustellen. Im Abschnitt 2.3 auf Seite 8 werden wir diesen Aspekt noch detaillierter beleuchten. Ein großer Vorteil ist, dass die bestehende Anwendung ursprünglich nicht als SPL entwickelt werden musste und die Kosten somit deutlich unter denen eines proaktiven Ansatzes liegen können.

Die Probleme, die bei der Überführung eines konventionellen, nicht als SPL geplanten Softwareprojekts, in eine SPL entstehen können, bezeichnet man auch als *Adoptionsbarriere*. Krueger stellt in [CK02] fest, dass leichtgewichtige extraktive Technologien tendentiell die Adoptionsbarriere verringern. Clements hingegen argumentiert in [CK02], dass eine weitsichtige Festlegung der SPL-Domäne, auf die ein Produkt abzielt, in einer proaktiven Vorgehensweise dazu führen kann, dass die Erschließung neuer Markbereiche mit deutlich reduziertem Aufwand erfolgen kann.

Wie schon erwähnt, spielt die Tatsache, ob bereits eine bestehende Software existiert, eine große Rolle bei der Beurteilung der Entwicklungsmodelle. In vielen Fällen ist eine Migration der bestehenden Programme notwendig, so dass eine Neuentwicklung grundsätzlich nicht in Frage kommt. Krueger betont, dass je nach Anwendungsfall auch der gleichzeitige Einsatz mehrerer Entwicklungsmodelle sinnvoll sein kann.

## 2.2 Implementierung von Software-Produktlinien

Bisher haben wir nur Prozessmodelle zur Entwicklung einer SPL betrachtet. Entscheidend sind aber auch Technologien zur Realisierung solcher Ideen. Kästner et al. [KAK08] ordnen die meisten Technologien einer der folgenden beiden Gruppen zu.

### Kompositionsansatz

Ein Feature einer SPL wird im Kompositionsansatz als zusammengehörige Einheit implementiert. Eine Variante der SPL, die ja durch eine Auswahl von Features definiert ist, wird erstellt, indem die entsprechenden Einheiten der ausgewählten Features *komponiert* werden. Diese Komposition erfolgt abhängig von der zugrunde liegenden Technik.

Kompositionsansätze wurden bereits ausgiebig erforscht, wodurch eine Reihe von Ansätzen und Techniken entstanden sind. Beispiele sind Komponenten [SGM02], Frameworks [JF88], mixin layers [SB02], AHEAD [BSR04] sowie die



AHEAD-Weiterentwicklung FEATUREHOUSE [AKL09a]. Weitere Beispiele sind mehrdimensionale Trennung von Belangen [TOHS99] und Aspekte [KLM<sup>+</sup>97].

### Annotationsansatz

Beim Annotationsansatz werden die einem oder mehreren Features zugeordneten Codefragmente innerhalb der gemeinsamen Codebasis implementiert. Die Zuordnung (*Annotation*) eines Codefragments zu den entsprechenden Features muss in irgendeiner Form explizit erfolgen. Eine Variante entsteht im Gegensatz zum Kompositionsansatz nicht durch Verknüpfung der zu den Features zugehörigen Einheiten, sondern durch Projektion der Codebasis auf die entsprechende Featureauswahl. Kästner und Apel [KA08a] stellen fest, dass in der Praxis meistens ein Annotationsansatz zur Implementierung einer SPL verwendet wird.

Standardbeispiel eines Annotationsansatzes sind Präprozessor-Direktiven wie z. B. `#ifdef` und `#endif`, mit denen man Codefragmente abhängig von Bedingungen zur Kompilierungszeit aus- und einblenden kann. Andere Annotationsansätze sind Explizite Programmierung [B<sup>+</sup>02], Frames/XVCL [JBZZ03], Spoon [Paw06], GEARS [Kru02], Software Pläne [CPR07], Metaprogrammierung mit Traits [TR07] usw. Grundlage dieser Arbeit ist CIDE, das im Abschnitt 2.3 auf Seite 8 detaillierter betrachtet wird.

Annotationsansätze sind offensichtlich besonders für das extraktive Entwicklungsmodell geeignet. So muss man Features nicht, wie im Kompositionsansatz, aus der bestehenden Codebasis herauslösen und als eigenständige Einheiten darstellen. Stattdessen kann man die entsprechenden Codefragmente an der jeweiligen Position im Quellcode belassen und muss sie lediglich als den jeweiligen Features zugehörig markieren. Dies ermöglicht auch, die bestehende Software während der Umstellung auf eine vollwertige SPL noch auf konventionelle Art und Weise zu warten oder weiterzuentwickeln.

### Integration beider Ansätze

Apel et al. [AKL09b] stellen fest, dass die Vor- und Nachteile beider Ansätze komplementär sind. Somit bot es sich an, beide Ansätze zu integrieren und von den Vorteilen beider Herangehensweisen zu profitieren. Soll aus einer bestehenden Anwendung eine SPL entwickelt werden, so kann dies in einem ersten Schritt extraktiv mit einem Annotationsansatz geschehen. Sobald die Refaktorisierung abgeschlossen ist, kann man zu einer proaktiven Vorgehensweise wechseln, die sich auf Grund der weitsichtigen Planung oft einfach mit Hilfe von Kompositionsansätzen realisieren lässt.

Apel et al. [AKL09b] erwähnen außerdem, dass eine rein leichtgewichtige Herangehensweise per Annotationsansatz die Softwarequalität und Architekturstabilität verschlechtern kann. In Kombination mit einem Kompositionsansatz werde sie jedoch nicht beeinträchtigt.

## 2.3 CIDE

Diese Diplomarbeit basiert auf dem Eclipse-Plugin CIDE<sup>1</sup>. Es entstand aus einer Fallstudie von Kästner et al., in der das eingebettete Datenbanksystem *Oracle Berkeley DB Java Edition (JE)*<sup>2</sup> mit Hilfe von *AspectJ* [KHH<sup>+</sup>01] in Features dekomponiert wurde [Käs07, KAB07]. Es wurde festgestellt, dass die Dekomposition teilweise umständlich war und dass sich gerade die Lesbarkeit und Wartbarkeit des resultierenden AspectJ-Programms verschlechtern.

Motiviert durch diese Erkenntnisse wurde CIDE entwickelt [KAK08]. Es realisiert eine extraktive Herangehensweise mit einem Annotationsansatz, da sich ein solches Vorgehen gerade für die Refaktorisierung einer Anwendung in Features besonders anbietet (siehe Abschnitte 2.1 und 2.2). Diese Refaktorisierung sollte so einfach sein, wie sie es auf dem Blatt Papier war, auf dem zu einem oder mehreren Features zugehöriger Code einfach farblich markiert wurde. Diese eigentlich naheliegende Idee wurde grundlegend und namensgebend für CIDE, so dass die *Annotation* von Codefragmenten durch *Färbung* erfolgte.

### 2.3.1 Entwicklungsprozess

Die Erklärung und Veranschaulichung der Arbeitsweise mit CIDE soll nun anhand eines Beispiels [KAK08] in Java erfolgen. Es soll ein *Stack* mit zwei optionalen, voneinander unabhängigen Features<sup>3</sup> *Locking* und *Logging* implementiert werden.

In einem ersten Schritt muss das *Featuremodell* festgelegt werden. Es beschreibt die Menge aller vorhandenen Features sowie deren Beziehungen zueinander und definiert somit die Menge aller gültigen Varianten der SPL [KCH<sup>+</sup>90]. So können manche Feature z. B. optional sein (es darf Varianten ohne dieses Feature geben) oder sich gegenseitig ausschließen (eine Variante darf höchstens eines der beiden Features enthalten). Neben dem trivialen Featuremodell, das einfach aus einer Liste vorhandener Features besteht, unterstützt CIDE auch *pure::variants* [BPSP04] und *guidsl* [Bat05]. Im weiteren Verlauf dieser Arbeit werden wir *guidsl* verwenden, nicht zuletzt weil CIDE auf Implementierungen des ebenfalls an der Universität Magdeburg entwickelten Eclipse-Plugins *FeatureIDE* [LAMS05, KTS<sup>+</sup>09] zurückgreifen kann.

Die Festlegung der beiden Features *Locking* und *Logging* in unserem Beispiel kann in einem grafischen Editor erfolgen (siehe Abbildung 2.1). Im Kontextmenü eines jeden Features kann man angeben, ob es obligatorisch (engl. mandatory) ist oder nicht. An den nicht ausgefüllten Kreisen über den beiden Features kann man erkennen, dass beide Features optional sind.

Die Programmierung und Färbung der Klasse *Stack* erfolgt nun im *Colored Text Editor*. Ein Codefragment wird einem Feature zugeordnet, indem man es

---

<sup>1</sup> [wwwiti.cs.uni-magdeburg.de/iti\\_db/forschung/cide](http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/cide)

<sup>2</sup> [www.oracle.com/technology/products/berkeley-db/je](http://www.oracle.com/technology/products/berkeley-db/je)

<sup>3</sup> Zwei Features heißen *voneinander unabhängig*, falls die Existenz bzw. das Fehlen des einen Features in einer Variante die Existenz bzw. das Fehlen des anderen Features in dieser Variante nicht notwendigerweise impliziert.

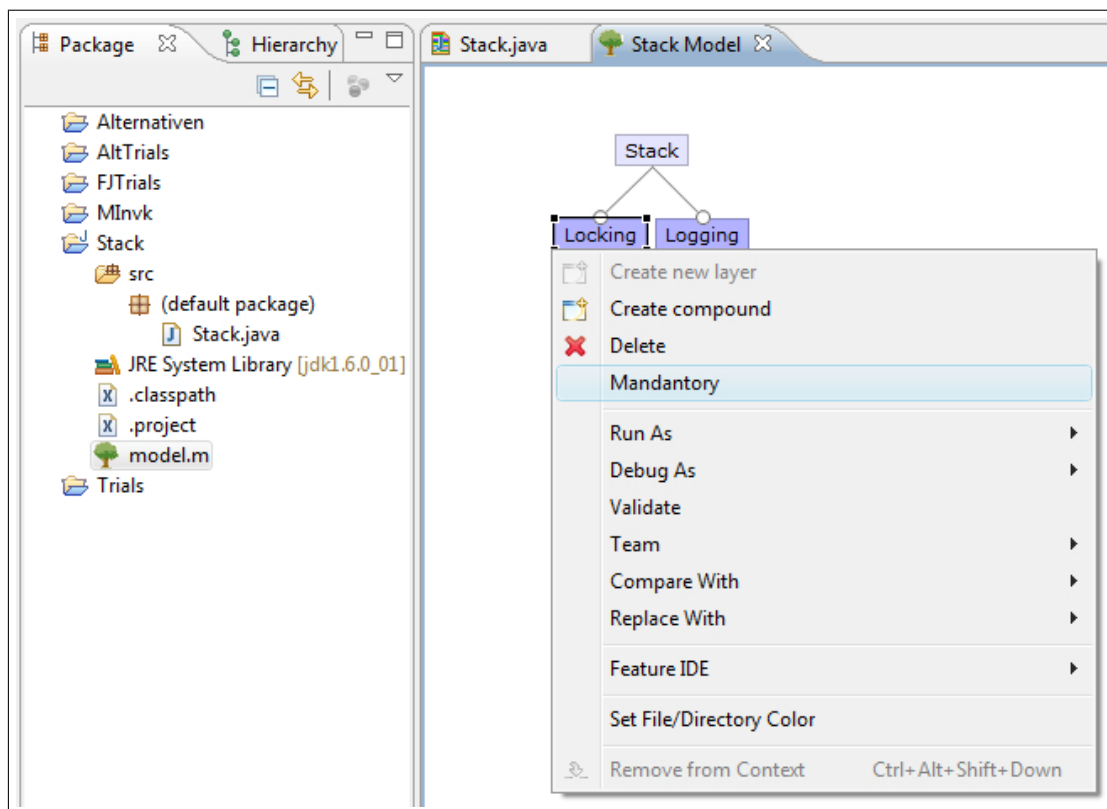


Abbildung 2.1: Konfiguration eines Featuremodells in CIDE

markiert und im Kontextmenü das entsprechende Feature auswählt (siehe Abbildung 2.2). Diese Zuordnung erkennt man nun daran, dass das gefärbte Codefragment mit einer Hintergrundfarbe hinterlegt ist (siehe Abbildung 2.3). Ist ein Codefragment mehrfach gefärbt (im Beispiel die Zeile `log("Lock failed.");`), so ist die Hintergrundfarbe eine Überlagerung aller beteiligten Farben. Somit kann man ausschließlich anhand der Färbung nicht erkennen, welchen Features ein Codefragment zugeordnet ist; dies zeigt CIDE in einem Tooltip an. Anhand der Hintergrundfarbe lässt sich aber nichtsdestotrotz zusammengehöriger Feature-Code erkennen.

Zur Erzeugung einer Variante wählt man in einem Dialog die gewünschten Features aus (siehe Abbildung 2.4). Dabei wird überprüft, ob diese Auswahl im Sinne des Featuremodells gültig ist. So wird z. B. bei einer gleichzeitigen Auswahl zweier sich gegenseitig ausschließender Features ein Fehler ausgegeben. Die Variante wird dann als eigenes Eclipse-Projekt angelegt. Ein Codefragment wird genau dann in die Variante übernommen, falls all seine Features in der Variante vorhanden sind. In unserem Beispiel bedeutet dies, dass der gesamte rot gefärbte Code (Feature *Locking*) ausgeblendet wird, falls wir für die Variante nur das Feature *Logging* wählen (siehe Abbildung 2.5). Auch die Zeile `log("Lock failed.");` existiert in dieser Variante nicht, da sie wohl mit dem Feature *Logging*, aber zusätzlich auch mit dem Feature *Locking* annotiert ist.

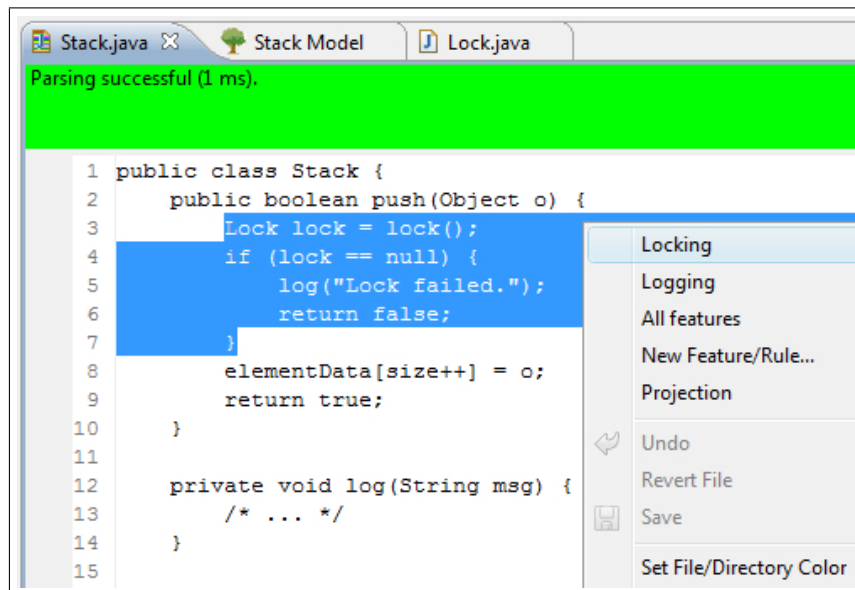


Abbildung 2.2: Färbung eines Codefragments in CIDE

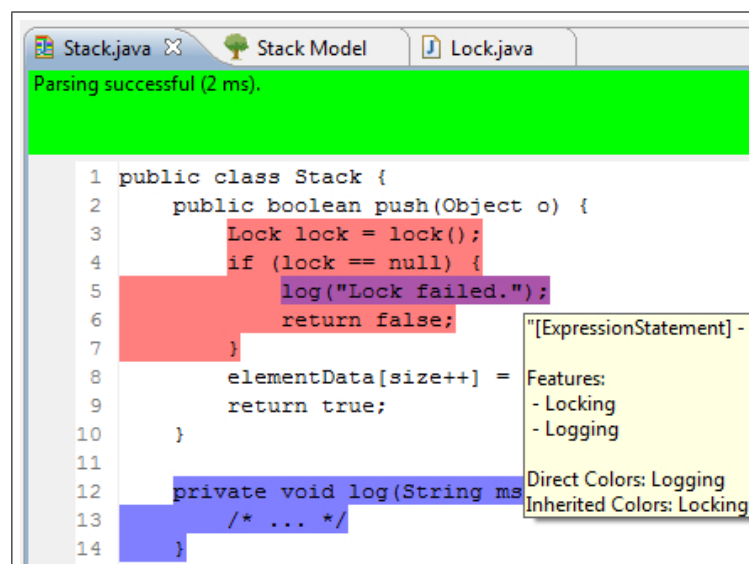


Abbildung 2.3: Gefärbte Codefragmente in CIDE mit Tooltip

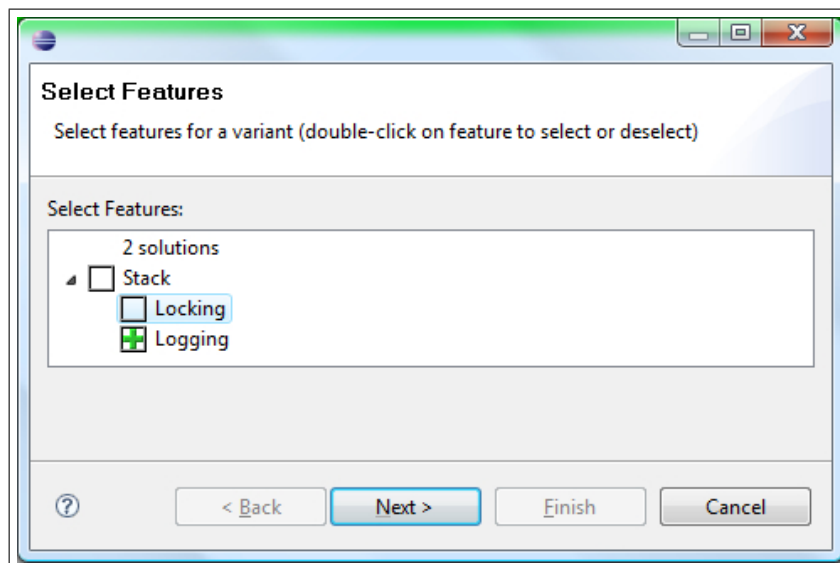


Abbildung 2.4: Auswahl von Features bei der Erzeugung einer Variante

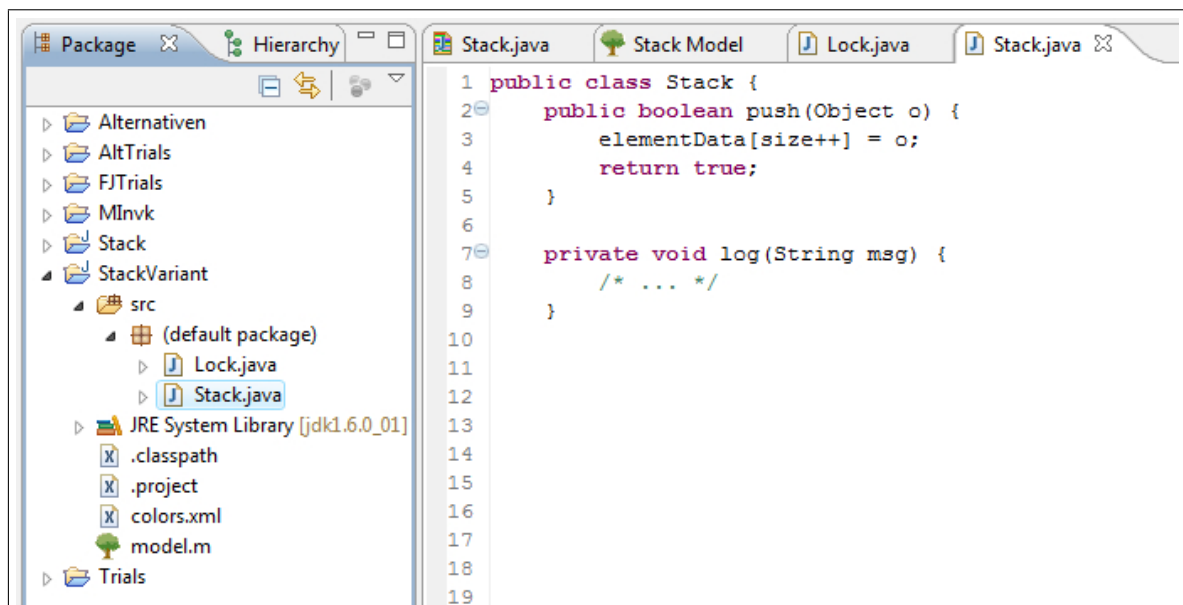


Abbildung 2.5: Erzeugte Variante des Projekts *Stack*

### 2.3.2 Vergleich mit Präprozessoren

Die Annotation von Codefragmenten durch Färbung ähnelt am ehesten einer Vorgehensweise mit **#ifdef**-Direktiven, wie sie in C und C++ möglich sind. Tatsächlich lässt sich obiges Beispiel auch damit ausdrücken (siehe Listing 2.1).

```
public class Stack {
    public boolean push(Object o) {
        #ifdef LOCKING
            Lock lock = lock();
            if (lock == null) {
                #ifdef LOGGING
                    log("Lock failed.");
                #endif
                return false;
            }
        #endif
        elementData[size++] = o;
        return true;
    }
    #ifdef LOGGING
        private void log(String msg) {
            /* ... */
        }
    #endif
}
```

Listing 2.1: Stack-Beispiel mit **#ifdef**-Direktiven

Dieses kleine Beispiel zeigt, dass die Darstellung einer Annotation durch Hintergrundfarben übersichtlicher ist als mit Hilfe von Präprozessor-Direktiven, dadurch dass die Annotationen nicht mit dem eigentlichen Quellcode vermischt, sondern extern gespeichert und dem Benutzer nur über den *Colored Text Editor* angezeigt werden. Daher nennt sich das Paradigma, das CIDE verfolgt, auch *virtuelle Trennung von Belangen* (engl. *virtual separation of concerns*) [KAT<sup>+</sup>08].

Ein weiterer Vorteil von CIDE gegenüber dem Präprozessor ist, dass in CIDE nicht jedes beliebige Codefragment annotiert werden kann. Mit einem Präprozessor ist es möglich, z. B. ein einzelnes Komma zu annotieren. Diese Möglichkeit ist aber auch notwendig, wenn man z. B. einzelne Parameter einer Methodendeklaration annotieren möchte (siehe Listing 2.2 [KAK08]).

```
public void someMethod(
    #ifdef FEATURE_1
        Object p1
        #ifdef FEATURE_2
            ,
        #endif
    #endif
    #ifdef FEATURE_2
        Object p2
    #endif
) { /* ... */ }
```

Listing 2.2: Annotieren zweier Methodenparameter mit **#ifdef**-Direktiven

Das Komma darf nur dann existieren, wenn beide Parameter – in diesem Fall also beide Features – vorhanden sind. Diese Tatsache muss man bei Benutzung eines Präprozessors explizit kodieren. In CIDE genügt es, beide Parameter (ohne Komma) entsprechend zu färben (siehe Abbildung 2.6).

```
public void someMethod(Object p1, Object p2) { /* ... */ }
```

Abbildung 2.6: Annotieren zweier Methodenparameter in CIDE

### 2.3.3 Syntaktische Korrektheit und Granularität

Somit besteht in CIDE keine Notwendigkeit, Kommata zu annotieren, so dass es auch gar nicht möglich ist. Allgemein kann man in CIDE nur solche Codefragmente annotieren, die optional im Sinne der Syntax sind. Intern werden in CIDE nämlich keine Codefragmente (eindeutig identifizierbar z. B. durch Offset und Länge innerhalb der Datei) annotiert, sondern Knoten des *abstrakten Syntaxbaums*, engl. *Abstract Syntax Tree (AST)*. Der AST abstrahiert im Gegensatz zum *konkreten Syntaxbaum*, engl. *Concrete Syntax Tree (CST)* von der konkreten Syntax (z. B. Kommata oder Semikolon am Ende einer Zeile) und enthält nur die strukturellen Elemente des Quellcodes. Trotzdem lässt sich der Quellcode aus dem AST mit Ausnahme der ursprünglichen Formatierung reproduzieren. Ein Knoten des AST enthält in der CIDE-Implementierung Informationen darüber, ob er optional ist (z. B. Parameter einer Methodendeklaration) oder nicht (z. B. Rückgabebetyp einer Methode). Gäbe es die Möglichkeit, nicht-optionale Codefragmente zu annotieren, so wäre es grundsätzlich möglich, Varianten zu erzeugen, die syntaktisch inkorrekt sind (z. B. muss zu einer Methode immer ein Rückgabebetyp angegeben werden) – nämlich genau solche, in denen mindestens eines der annotierten Features nicht vorkommt. Daher kann man in CIDE nur optionale AST-Knoten färben.

Eine weitere wichtige Regel, die in CIDE gewährleistet wird, ist die sog. *Teilbaum-Regel* (engl. *subtree rule*) [KAK08]. Sie besagt, dass mit einem AST-Knoten auch alle Knoten des von ihm aufgespannten Teilbaums des AST gefärbt werden. Annotiert man also z. B. eine Methodendeklaration, so sind auch alle formalen Parameter und der gesamte Methodenrumpf entsprechend annotiert. Alle Kindknoten *erben* die Farben des Vaterknotens. Sie können aber zusätzlich mit weiteren Features annotiert werden.

Diese beiden Regeln (nur optionale AST-Knoten können gefärbt werden und die Teilbaum-Regel) garantieren, dass jede mit CIDE erzeugte Variante ein syntaktisch korrektes Programm im Sinne der zugrunde liegenden Sprache ist [KAT<sup>+</sup>09]. Es müssen also zusätzlich zum eigentlichen Parsen des Quellcodes keine weiteren Überprüfungen durchgeführt werden, um die syntaktische Korrektheit aller Varianten zu gewährleisten. Stattdessen wird der Vorgang des Annotierens durch die Entwicklungsumgebung entsprechend gesteuert.

CIDE ermöglicht für die Annotationen also die nützliche *Granularität*, z. B. einzelne formale Parameter einer Methodendeklaration annotieren zu können, verbietet aber die Fehlerquellen, die durch die sehr feine, in der Praxis nicht benötigte Granularität (z. B. Annotieren von Kommata) von Präprozessor-Direktiven entstehen können. Kästner et al. [KAK08] bemerken, dass die in CIDE ermöglichte Granularität tatsächlich von Vorteil ist. So sind in Kompositionsansätzen oft

Workarounds notwendig (z. B. Einführung von Hook-Methoden), die den Quellcode unnötig unübersichtlich und kompliziert machen, z. B. wenn in bestehenden Methoden zusätzliche Statements eingefügt werden müssen. Ist der Bedarf an feingranularen Feature-Annotationen bei der Refaktorisierung von bestehenden Anwendungen in Features wohl am größten, kamen die Vorteile von CIDE aber auch bei Programmen, die von Grund auf als SPL entwickelt wurden, zum Tragen. So wurde die bekannte *Graph Product Line (GPL)* [LHB01], die bereits mit verschiedenen Kompositionsansätzen implementiert wurde, nun auch mit CIDE implementiert. Es wurde festgestellt, dass mit Hilfe feingranularer Annotationen Code-Replikationen erheblich verringert werden können [KAK08].

### 2.3.4 Sprachunabhängigkeit

Erste Versionen von CIDE unterstützten nur *Java* als zugrunde liegende Programmiersprache. Grund dafür war u. a., dass bestehende Funktionalitäten von Eclipse wiederverwendet werden konnten. So kann man beispielsweise den sehr komfortablen Java-Editor verwenden, um mit CIDE zu arbeiten. Außerdem stehen mit den *Java Development Tools (JDT)*<sup>4</sup> mächtige Bibliotheken zur Verfügung, die u. a. einen kompletten AST einer Java-Datei bereitstellen.

Eine komplette SPL besteht aber in den wenigsten Fällen ausschließlich aus Quellcode einer einzelnen Programmiersprache. Vielmehr besteht ein SPL-Projekt aus Quellcode unterschiedlicher Programmiersprachen, Entwürfen, Spezifikationen, Grammatiken, Skripten zur Entwicklung und Auslieferung, Dokumentationen usw. Diese Tatsache nennt man *Prinzip der Uniformität* (engl. *principle of uniformity*) [BSR04]. Es besagt, dass alle Arten von Artefakten einer SPL *einheitlich* bei der Entwicklung verwendet werden können sollen. Aus diesem Grund wurde CIDE auf weitere Typen von Artefakten erweitert [KAT<sup>+</sup>09]. Dabei war es wünschenswert, den Mechanismus zur Annotierung von Codefragmenten wiederverwenden zu können. Gleichsam gewährleistet die Einhaltung obiger beider Regeln weiterhin, dass jede mögliche Variante syntaktisch korrekt ist – unabhängig von der zugrunde liegenden Sprache.

Neben Java unterstützt CIDE aktuell bereits eine große Anzahl von Sprachen, darunter u. a. FJ, C, C#, (X)HTML, XML und Haskell. Neben den bereits genannten Fallstudien zu Berkeley DB und GPL wurden auch zwei funktionale Haskell-Programme mit CIDE erfolgreich in Features dekomponiert [AKGL09].

### 2.3.5 Zusammenfassung

Abschließend in diesem Kapitel sollen der Vollständigkeit halber noch einige weitere Features von CIDE erwähnt werden:

- Export-Funktionalität in Feature-Module (Jak und AspectJ [KKB07] sowie FEATUREHOUSE [AKL09a])

---

<sup>4</sup> [www.eclipse.org/jdt](http://www.eclipse.org/jdt)



- Einfache Navigation zwischen Codefragmenten, die zu den gleichen Features gehören (Erhöhung der Feature-Kohäsion und Abschwächung des Feature-Traceability-Problems) [KAK08, KTA08]
- Projektion des Quelltextes auf eine Auswahl von Features (ähnlich Code Folding)
- Analyse von Feature-Interaktionen [KKB08]

Zusammenfassend kann CIDE als vollwertiges SPL-Programm bezeichnet werden, wenn auch nur prototypisch implementiert. Die zugrunde liegenden Konzepte erleichtern nicht nur die Feature-orientierte Dekomposition einer Anwendung in Features, sondern ermöglichen auch eine komplette Neuentwicklung einer SPL von Grund auf. Neben dem Grundkonzept des *Färbens* spielt auch die Unterstützung des Programmierers durch CIDE als Entwicklungsumgebung eine große Rolle. Gerade die Nachteile, die ein Annotationsansatz mit sich bringt (z. B. fehlende Feature-Kohäsion durch verstreuten Feature-Code), lassen sich durch geeignete Unterstützung durch die Entwicklungsumgebung abmildern.



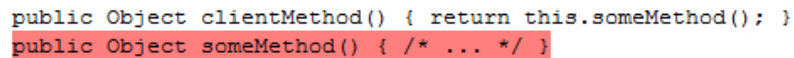
---

## KAPITEL 3

---

### Colored Featherweight Java

CIDE ist ein SPL-Programm, das die syntaktische Korrektheit aller möglichen Varianten gewährleistet. Einleitend wurde aber schon erwähnt, dass die syntaktische Korrektheit eines Programms alleine i. A. nicht ausreicht. Als Beispiel betrachte man das annotierte Java-Codefragment in Abbildung 3.1.



```
public Object clientMethod() { return this.someMethod(); }  
public Object someMethod() { /* ... */ }
```

Abbildung 3.1: Aufruf einer gefärbten Methode

Es wird die Methode `someMethod()` aufgerufen, die rot gefärbt ist. In Varianten, die das rote Feature nicht enthalten, existiert diese Methode also nicht, obwohl der Methodenaufruf existiert. In dieser Variante gibt es also einen Typfehler. Dieses Beispiel zeigt, dass durch die Existenz von Annotationen Typfehler entstehen können, obwohl der Basiscode ohne Annotationen ein lauffähiges Java-Programm ist.

Schon Kästner et al. [KAT<sup>+</sup>08] merken an, dass ein Bedarf an Überprüfungen der *Sprachsemantik* auch unter Berücksichtigung von Annotationen besteht. Dabei soll möglichst vermieden werden, alle möglichen Varianten zu erstellen und zu überprüfen, da sonst eine kombinatorische Explosion droht. Bei  $n$  vorhandenen Features gibt es  $2^n$  mögliche Varianten, falls es im Featuremodell keinerlei Restriktionen gibt. Aber auch in praktischen Beispielen, in denen viele Features voneinander abhängig sind, erreicht man schon mit wenigen Features eine sehr große Anzahl zu überprüfender Varianten.

Die grundsätzliche Problematik bei der Implementierung eines Typsystems ist die Tatsache, dass die Sprachsemantik in vielen Sprachen nur informell spezifiziert ist. So gibt es zu Java z. B. eine recht informelle Spezifikation im Umfang von 688 Seiten [GJSB05]. Alle dort aufgelisteten Regeln müssten dahingehend überprüft werden, inwieweit die Existenz von Annotationen eine Verletzung dieser Regeln in einer der Varianten herbeiführen kann.

Um einen ersten Eindruck darüber zu gewinnen, welche zusätzlichen Typprüfungen prinzipiell notwendig sind, um Annotationen im Typsystem zu berücksichtigen, wurden einige der häufigsten Fehlerursachen in Java unter Berücksichtigung von Annotationen in CIDE überprüft [KAT<sup>+</sup>09], so auch obiges Beispiel aus Abbildung 3.1. Jedoch wurden nicht alle Typregeln überprüft, so dass das so implementierte Typsystem nicht vollständig ist.

Grundsätzlich besteht der Anspruch, ein Typsystem vollständig zu implementieren. Bei SPL-Sprachen wie CFJ bedeutet diese Vollständigkeit, dass alle möglichen Varianten *wohlgeformt* (engl. *well-typed*) im Sinne des Typsystems der zugrunde liegenden Sprache sind. Zur Erforschung eines solchen Typsystems bietet sich als Grundlage die Programmiersprache FJ an, eine funktionale Teilmenge von Java, zu der es einen sehr überschaubaren Typenkalkül gibt, der die Typsicherheit in Form eines *Fortschritts- und Erhaltungstheorems* (engl. *progress and preservation theorem*) [Pie02] beweisbar gewährleistet. Auf solche Beweise der Typsicherheit gehen wir im weiteren Verlauf dieser Arbeit jedoch nicht genauer ein, sondern konzentrieren uns auf die Typregeln sowie deren Implementierung.

Dieses Kapitel beginnt somit mit einigen Grundlagen zu FJ und seiner Erweiterung um Annotationen zu CFJ (Abschnitt 3.1). Es folgen die Definitionen der Syntax (Abschnitt 3.2) und des Typsystems (Abschnitt 3.3) von CFJ.

## 3.1 Grundlagen

FJ ist eine minimale Programmiersprache, die dazu entwickelt wurde, Javas Typsystem einfacher formal analysieren zu können [Pie02]. Sie entstand aus Java dadurch, dass all diejenigen Sprachkonstrukte, die zu diesem Zweck nicht notwendig waren und vielmehr sogar davon ablenkten, weggelassen wurden. Somit konnte man sich auf die eigentliche Aufgabe konzentrieren. Außerdem wurde FJ zur Analyse von Java-Erweiterungen wie Generics [IPW01] oder inneren Klassen [IP02] verwendet. Auch hier hatte man den Vorteil, sich ausschließlich um die Behandlung der interessierenden Sprachkonstrukte kümmern zu können, ohne einen großen Aufwand durch die Berücksichtigung anderer Elemente der Sprache bewältigen zu müssen.

FJ ist eine funktionale Teilmenge von Java, d. h. jedes FJ-Programm ist ein lauffähiges Java-Programm mit identischem Verhalten. Sind Details der Evaluierung eines FJ-Programms also für manche Arbeiten nicht von Belang (wie z. B. auch in unserem Fall), so kann man zur Kompilierung und Ausführung einfach einen Java-Compiler verwenden. Trotz aller Beschneidungen ist FJ eine Turing-vollständige Sprache, d. h. jede im Sinne der Church'schen These berechenbare Funktion lässt sich als FJ-Programm darstellen.

Die SPL-Sprache CFJ wurde in [KA08b] als Erweiterung von FJ um CIDE-Annotationen entwickelt. Dabei sind die Annotationen kein Bestandteil der Syntax, sondern werden extern in einer *Annotations-Tabelle* verwaltet. Obwohl die Syntax eines CFJ-Programms genau die FJ-Syntax ist, wird CFJ als eigene Sprache bezeichnet. Grund dafür ist, dass CFJ-Programme nie direkt ausgeführt werden. Stattdessen werden aus einer CFJ-Produktlinie Varianten erzeugt, die dann als FJ-Programme ausgeführt werden. Außerdem ist das Typsystem von CFJ deutlich komplexer als das von FJ. Hinzu kommt, dass der Begriff *Typsicherheit* bei SPL-Sprachen wie CFJ anders definiert ist als bei konventionellen Programmiersprachen wie FJ (siehe dazu Abschnitt 3.3 auf Seite 22).

### Sprachumfang von (C)FJ

Der größte Unterschied zwischen FJ und Java ist das Fehlen von Mehrfachzuweisungen. In FJ werden die Felder einer Klasse einmalig beim Erzeugen einer Instanz im Konstruktor initialisiert und nie mehr verändert. Daher wird FJ auch als *funktionale* Teilmenge von Java bezeichnet. Außerdem fehlen beispielsweise auch folgende Java-Spracheigenschaften:

- Kontrollfluss-Anweisungen wie **if**, **for** oder **while**
- Interfaces
- **null**-Pointer
- Primitive Datentypen wie **int**, **double** oder **boolean**
- Abstrakte Methoden und Klassen
- Innere Klassen
- Überschatten von Feldern einer unmittelbaren oder mittelbaren Superklasse (*Overshadowing*)
- Zugriffsmodifikatoren wie **public** oder **private**
- Ausnahmen

Der Konstruktor einer Klasse ist durch die Felder der Klasse und aller Superklassen eindeutig festgelegt. Bei der Instanziierung eines Objekts müssen alle Felder der Klasse und der Superklassen initialisiert werden, und zwar in der Reihenfolge der Felder in der Vererbungshierarchie abwärts, beginnend bei `Object`. Somit hat der Programmierer eigentlich keinen Einfluss auf die Gestaltung des Konstruktors. Er wurde lediglich zur Kompatibilität zu Java eingeführt [Pie02]. Für viele Betrachtungen kann man den Konstruktor also vernachlässigen, wie wir es z. B. in den Abschnitten 3.3 und 4.6 auch machen werden.

In FJ ist es wie in Java möglich, eine Klasse (*Subklasse*) von einer anderen Klasse (*Superklasse*) mittels des Schlüsselworts **extends** zu vererben. Dabei dürfen auch Methoden einer unmittelbaren oder mittelbaren Superklasse *überschrieben* werden (*Overriding*), d. h. zu einer Methode darf es in einer Subklasse eine Methode mit gleichem Namen, gleichem Rückgabetyt und gleichen Parametertypen in gleicher Reihenfolge geben. Jedoch ist in FJ das Überladen von Methoden (*Overloading*) nicht erlaubt, so dass sich zusammen mit dem Overriding folgende zwei Regeln ergeben:

1. In einer Klasse darf es keine zwei Methoden mit gleichem Namen geben.
2. Gibt es in einer unmittelbaren oder mittelbaren Superklasse eine Methode mit gleichem Namen, so müssen die Rückgabetyt und Parametertypen übereinstimmen.

Ein Methodenrumpf besteht nur aus einem **return**-Statement bestehend aus dem **return**-Schlüsselwort und einem *Term*. In FJ gibt es lediglich fünf Arten von Termen, die innerhalb eines **return**-Statements verwendet werden können, wobei eine Verschachtelung möglich ist:

- Erzeugen von Objekten mit **new**
- Aufruf von Methoden mit Übergabe von Parametern
- Zugriff auf Felder von Objekten
- Casten eines Ausdrucks
- Zugriff auf einen formalen Parameter (Variable) einer Methode

Im folgenden Abschnitt werden wir die Syntax von (C)FJ exakt definieren und einige Beispiele geben.

## 3.2 Syntax

Die Syntax von FJ erklären wir anhand der Grammatik (siehe Listing 3.1)<sup>1</sup>.

```
TypeDeclaration :
  "class" <IDENTIFIER> "extends" ExtendedType "{"
    (VarDeclaration)*
    ClassConstructor
    (MethodDeclaration)*
  "}";

ExtendedType : <IDENTIFIER> | "Object";
VarDeclaration : Type <IDENTIFIER> ";";

ClassConstructor :
  Type "(" (FormalParameterList)? ")" "{"
    "super" "(" (IdentifierList)? ";";
    (FieldAssign)*
  "}";

MethodDeclaration :
  Type <IDENTIFIER> "(" (FormalParameterList)? ")" "{"
    "return" Expression ";";
  "}";

Type : <IDENTIFIER> | "Object";
FormalParameterList : &LI FormalParameter ("," &LI FormalParameter)*;
IdentifierList : &LI <IDENTIFIER> ("," &LI <IDENTIFIER>)*;
FieldAssign : "this" "." <IDENTIFIER> "=" <IDENTIFIER> ";";
FormalParameter : Type <IDENTIFIER>;

Expression :
  MethodInvoke
  | FieldInvoke
  | <IDENTIFIER>
  | AllocationExpression
  | CastExpression
  | NestedExpression
  | "this";
```

<sup>1</sup> Das Format dieser Grammatik ähnelt der Backus-Naur Form und wird in der Implementierung von CIDE in dieser Form verwendet.

```

MethodInvoke : InvokeTarget "." <IDENTIFIER> "(" (ExpressionList)? ")";
FieldInvoke : InvokeTarget "." <IDENTIFIER>;
AllocationExpression : "new" <IDENTIFIER> "(" (ExpressionList )? ")";
CastExpression : "(" Type ")" Expression;
NestedExpression : "(" Expression ")";
InvokeTarget: AllocationExpression | NestedExpression | <IDENTIFIER> | "this";
ExpressionList : &LI Expression ("," &LI Expression)*;

```

Listing 3.1: Grammatik von FJ

Ein Featherweight Java Programm besteht aus einer Menge von Typdeklarationen (TypeDeclaration), die die Klassen repräsentieren. Bei einer solchen Deklaration einer Klasse muss man immer einen Supertypen (ExtendedType) angeben, auch wenn er Object ist. Außerdem besteht eine Klasse aus einer Menge von Variablen-Deklarationen (VarDeclaration), einem einzigen Konstruktor (ClassConstructor) sowie aus einer Menge von Methoden-Deklarationen (MethodDeclaration) – in genau dieser Reihenfolge. Hier erkennt man, dass ein Konstruktor zwingend vorgeschrieben ist.

Im Konstruktor muss an erster Stelle immer ein `super`-Aufruf stehen. Im Anschluss folgt eine Liste von Feldzuweisungen (FieldAssign). Außerdem erkennt man in der Grammatik, dass ein Methodenrumpf ausschließlich aus einem **return**-Statement besteht. Zurückgegeben werden kann ein Term (Expression), der ein Methodenaufruf (MethodInvoke), ein Feldzugriff (FieldInvoke), ein Variablenzugriff (<IDENTIFIER>), eine Objekterzeugung (AllocationExpression) oder ein Cast-Ausdruck (CastExpression) sein kann.

Das Ziel eines Methodenaufrufs oder Feldzugriffs (InvokeTarget) muss immer angegeben werden, auch wenn es sich um **this** handelt.

Listing 3.2 gibt einige Beispiele für Klassen in FJ an [Pie02].

```

class A extends Object { A() { super(); } }

class Pair extends Object {
    Object fst;
    Object snd;

    Pair(Object fst, Object snd) {
        super();
        this.fst = fst;
        this.snd = snd;
    }

    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}

class SuperClass extends Object {
    Object o1;
    Object o2;

    SuperClass(Object o1, Object o2) {
        super();
        this.o1 = o1;
        this.o2 = o2;
    }
}

class SubClass extends SuperClass {
    Object o;
}

```

```

SubClass(Object o1, Object o2, Object o) {
    super(o1, o2);
    this.o = o;
}

```

Listing 3.2: Beispiele für FJ-Klassen

Der Rumpf der Methode `setfst` der Klasse `Pair` besteht nur aus einem **return**-Statement bestehend aus dem Schlüsselwort **return** sowie dem Objekterzeugungsterm `new Pair(newfst, this.snd)`. Der zweite Parameter `this.snd` ist ein Feldzugriffs-Term. Am Beispiel der Klasse `SubClass` kann man sehen, wie der Konstruktor gebildet werden muss, wenn Felder einer Superklasse geerbt werden.

Es fällt auf, dass es keine statischen Methoden gibt. Somit kann es auch keine statische `main`-Methode geben, so dass sich die Frage stellt, was der Einstiegspunkt in ein FJ-Programm ist. Zu diesem Zweck nimmt man an, dass zusätzlich zu den Klassen noch ein Term angegeben wird, der beim Start des Programms ausgewertet wird. Dieser Term kann also als Rumpf der `main`-Methode interpretiert werden.

Abschließend soll diskutiert werden, aus welchem Grund Mehrfachzuweisungen in FJ nicht möglich sind. Der Grund ist, dass somit eine einfache, Syntaxgetriebene Evaluierung möglich ist. Der Term `new Pair(new A(), new B()).snd` kann damit, ohne einen Heap modellieren zu müssen, zu `new B()` evaluiert werden, da man weiß, dass sich das mit `new B()` initialisierte Feld `snd` nicht mehr ändern wird [Pie02]. Der Programmtext entspricht also dem Programzustand.

## 3.3 Typsystem

Bevor wir das Typsystem von CFJ im Detail betrachten, müssen einige Grundlagen und Voraussetzungen besprochen werden.

### 3.3.1 Wohlgeformtheit eines CFJ-Programms

Ein Programm heißt *wohlgeformt*, falls das Typsystem der zugrunde liegenden Programmiersprache keine Fehler innerhalb dieses Programms meldet. Ein Typsystem besteht aus syntaktischen Methoden zur automatischen Überprüfung von Programmen. Ziel dabei ist es, fehlerhaftes Programmverhalten auszuschließen [Pie02]. Eine Aufgabe im Java-Typsystem ist es z. B., bei der Deklaration einer Variable zu überprüfen, ob der Typ der Variable auch existiert. Würde diese Typprüfung zur Kompilierungszeit nicht durchgeführt, bestünde die Gefahr, dass erst bei der Ausführung des Programms ein Fehler auftritt.

CFJ-Programme werden nicht direkt ausgeführt. Aus einem solchen Programm können Varianten gebildet werden, die dann als FJ-Programme ausgeführt werden. Daher muss die Definition eines Typsystems für CFJ-Programme etwas abgeändert werden. Wir bezeichnen solche Programmiersprachen nun vereinfachend als *SPL-Sprachen*. Nicht-SPL-Sprachen nennen wir *konventionelle Programmiersprachen*.



Wir definieren ein CFJ-Programm als **wohlgeformt im Sinne des CFJ-Typsyste**ms, falls jede im Rahmen des Featuremodells gültige, daraus erzeugbare Variante im Sinne des FJ-Typsyste

ms wohlgeformt ist. Kästner und Apel [KA08b] formulieren dies kurz als: *Die Erzeugung von Varianten erhält die Typsicherheit (engl. variant generation preserves typing).*

Bei SPL-Sprachen ist die Wohlgeformtheit einer SPL also vom Typsystem der zugrunde liegenden Programmiersprache abhängig. Wir erlauben also nicht, ein beliebiges Typsystem für CFJ angeben zu können und dann von Wohlgeformtheit eines CFJ-Programms zu sprechen. Vielmehr muss ein CFJ-Typsystem per Definition gewährleisten, dass jede gültige Variante als FJ-Programm wohlgeformt ist.

### 3.3.2 Einfluss des Featuremodells

Die Tatsache, dass in einem wohlgeformten CFJ-Programm jede im Sinne des Featuremodells *gültige* Variante wohlgeformt sein muss, führt dazu, dass bei Typprüfungen eines CFJ-Programms das Featuremodell miteinbezogen werden muss. Als Beispiel betrachte man Abbildung 3.2.

```
public Object clientMethod() { return this.someMethod(); }
public Object someMethod() { /* ... */ }
```

Abbildung 3.2: Einfluss des Featuremodells auf das Typsystem

Dieses Codefragment ist nicht wohlgeformt, falls das blaue Feature auch ohne das rote Feature existieren kann. Dann wird nämlich die Methode `someMethod()` aufgerufen, obwohl sie gar nicht deklariert wurde. Kann man aus dem Featuremodell hingegen schließen, dass das blaue Feature das rote Feature impliziert (*Blau*  $\Rightarrow$  *Rot*), so kann dieser Fehler nicht passieren. Aus diesem Grund müssen die Bedingungen und Restriktionen des Featuremodells im Typsystem berücksichtigt werden.

Featuremodelle können auf unterschiedliche Art und Weise beschrieben werden. Eine weit verbreitete Form sind *Featurediagramme* [KCH<sup>+</sup>90], die das Featuremodell als Baumstruktur beschreiben. Eine andere Möglichkeit ist, ein Featuremodell durch *aussagenlogische Formeln* zu definieren [Bat05]. Eine aussagenlogische Formel ist eine Verknüpfung von *Aussagenvariablen* mit *Junktoren* [EFT96]. Beispiele für solche Junktoren sind  $\neg$  (nicht),  $\wedge$  (und zugleich),  $\vee$  (oder auch),  $\Rightarrow$  (impliziert) oder  $\Leftrightarrow$  (äquivalent). Die Aussagenvariablen werden als Wahrheitswerte interpretiert, so dass eine aussagenlogische Formel bei einer Belegung aller Aussagenvariablen mit Wahrheitswerten auch zu einem Wahrheitswert ausgewertet werden kann<sup>2</sup>.

In der Anwendung für Featuremodelle wird jedes Feature mit einer Aussagenvariable identifiziert. Die Auswahl von Features für eine Variante entspricht einer

<sup>2</sup> Wir legen für weitere Ausführungen eine intuitive Semantik der Logik zugrunde. Formal vollständige Ausführungen findet man z. B. in [EFT96]

Belegung dieser Variablen mit den Wahrheitswerten *true* (Feature wurde ausgewählt) oder *false* (Feature wurde nicht ausgewählt). Zwei obligatorische Features lassen sich z. B. mit der Formel  $F_1 \wedge F_2$  ausdrücken. Soll in jeder Variante genau eines der beiden Features  $F_1$  und  $F_2$  vorhanden sein, so lässt sich dies mit dem *exklusiven Oder* (*XOR*) ausdrücken. Dabei ist  $F_1 \text{ XOR } F_2$  logisch äquivalent zu  $(F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2)$ . Soll die Existenz eines Features auch immer die Existenz eines anderen Features implizieren, kann man dies durch  $F_1 \Rightarrow F_2$  ausdrücken. Allgemein kann das Featuremodell aus beliebigen aussagenlogischen Formeln bestehen.

Eine Auswahl von Features ist *gültig* im Sinne des Featuremodells, falls jede aussagenlogische Formel des Featuremodells unter dieser Belegung zu *true* ausgewertet. Setzen wir all diese Formeln durch *Konjunktion* zu einer Formel zusammen, so können wir das Featuremodell mit einer einzigen aussagenlogischen Formel  $F$  über der Menge vorhandener Features identifizieren. Wählt man bei der Erstellung einer Variante z. B. zwei sich gegenseitig ausschließende Features zusammen aus, so evaluiert die Formel  $F_1 \text{ XOR } F_2$  zu *false*, so dass die Auswahl als ungültig erkannt wird.

Im Zusammenhang mit dem CFJ-Typsystem müssen oft Aussagen der Form *für alle gültigen Varianten gilt  $\alpha$*  für eine aussagenlogische Formel  $\alpha$  formuliert werden. Diese Aussage lässt sich umformulieren in *für alle Belegungen der Aussagenvariablen mit Wahrheitswerten evaluiert die Formel  $F \Rightarrow \alpha$  zu true*, oder kurz  *$F \Rightarrow \alpha$  ist eine Tautologie*, wobei  $F$  die aussagenlogische Formel des Featuremodells bezeichne. Diese Implikation entsteht dadurch, dass Belegungen, unter denen  $F$  zu *false* ausgewertet (d. h. ungültige Varianten) irrelevant sind.

Eine Überprüfung der *Allgemeingültigkeit* von  $F \Rightarrow \alpha$  durch Einsetzen aller  $2^n$  möglichen Belegungen bei  $n$  Features führt schon bei wenigen Features zu einer inakzeptablen Laufzeit. Daher musste nach alternativen Lösungen gesucht werden.

Die zu überprüfende Aussage ist logisch äquivalent zu *es gibt keine Belegung der Aussagenvariablen mit Wahrheitswerten, so dass die Formel  $\neg(F \Rightarrow \alpha)$  zu true evaluiert*. Dies ist gerade die negierte *Erfüllbarkeit* der Formel  $\neg(F \Rightarrow \alpha)$ .

Das Problem lässt sich also auf die Lösung des *Erfüllbarkeitsproblems der Aussagenlogik*, kurz *SAT*, zurückführen. Die Verwendung von SAT-Lösern im Zusammenhang mit Featuremodellen wurde ursprünglich von Batory [Bat05] vorgeschlagen. Positive Erfahrungen mit der Laufzeit des SAT-Lösers bei Verwendung innerhalb von Featuremodellen haben beispielsweise Thüm et al. in [TBK09] gemacht. Sie haben den SAT-Löser SAT4J<sup>3</sup> innerhalb ihrer *guidsl*-Implementierung verwendet. Sowohl Featuremodelle aus der Praxis als auch zufällig erzeugte Featuremodelle konnten in unter einer Sekunde, in extremen Fällen in wenigen Sekunden verarbeitet werden. Wir können in CIDE von einer gleichsam guten Laufzeit profitieren, da wir die *guidsl*-Bibliotheken von Thüm et al. verwenden.

---

<sup>3</sup> [www.sat4j.org](http://www.sat4j.org)

### 3.3.3 Formalisierung von Annotationen

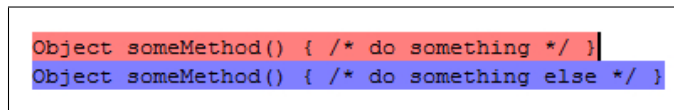
Die Annotation eines Codefragments muss im CFJ-Typsystem berücksichtigt werden. Durch die Notwendigkeit, das Featuremodell – formalisiert als aussagenlogische Formel – miteinzubeziehen, bietet es sich an, auch die Annotationen als aussagenlogische Formeln darzustellen.

In CIDE ist es möglich, ein Codefragment mit mehreren Features zu annotieren. Dies bedeutet, dass der entsprechende Quelltext in genau denjenigen Varianten vorhanden ist, die mindestens diese Features enthalten. Czarnecki und Pietroszek [CP06] ermöglichen sogar eine Annotation mit beliebigen aussagenlogischen Formeln und nennen sie *Anwesenheitsbedingung* (engl. *presence condition*). Für weitere theoretische Betrachtungen gehen wir davon aus, dass ein Codefragment in einer Art und Weise annotiert ist, dass sich bei einer konkreten Auswahl von Features ein Wahrheitswert *true* (Codefragment ist in der Variante vorhanden) oder *false* (Codefragment ist in der Variante nicht vorhanden) ermitteln lässt. Somit können wir eine Annotation mit den Features  $F_1, \dots, F_n$  in CIDE durch die aussagenlogische Formel  $F_1 \wedge \dots \wedge F_n$  darstellen.

Zusammenfassend können wir festhalten, dass die Aussage *ein Codefragment ist in einer Variante vorhanden* äquivalent ist zu *die Annotation des Codefragments evaluiert unter den Features der Variante zu true*. Dabei muss berücksichtigt werden, dass ein nicht annotiertes Codefragment immer zu *true* auswerten muss, da es ja in jeder möglichen Variante vorhanden ist.

### 3.3.4 Einschränkung gültiger CFJ-Programme

Im Laufe der Entwicklung des CFJ-Typsystems wurden wir beispielsweise mit dem Codefragment aus Abbildung 3.3 konfrontiert.



```
Object someMethod() { /* do something */ }
Object someMethod() { /* do something else */ }
```

Abbildung 3.3: Alternative Deklarationen einer Methode

Schließen sich die Features *Rot* und *Blau* gegenseitig aus (alternative Features), so weiß man, dass in jeder möglichen Variante höchstens eine der beiden Methodendeklarationen vorhanden ist. Vernachlässigt man die Annotationen, so ist das Beispiel kein gültiges Java-Programm, da es in einer Klasse keine zwei Deklarationen der gleichen Methode geben darf.

Mittelfristiges Ziel ist es, die Erkenntnisse in CFJ auf Java zu erweitern. Aufgrund der hohen Komplexität möchte man jedoch vermeiden, das komplette Java-Typsystem neu zu implementieren, um es dann auf Annotationen erweitern zu können. Daher ist es sehr erstrebenswert, ein bestehendes Java-Typsystem – in Eclipse z.B. in Form der JDT-Bibliotheken – wiederverwenden zu können. Dies ist aber nur dann möglich, wenn ein gefärbtes Java-Programm unter Vernachlässigung der Annotationen ein gültiges Java-Programm ist, denn ein bestehender

Java-Compiler geht von dem Quelltext aus, der im Editor steht. Obiges Beispiel darf also nicht ermöglicht werden.

Aus diesem Grund haben wir folgende grundsätzliche Entscheidung getroffen:

**Jedes annotierte Programm muss unter Vernachlässigung der Annotationen ein gültiges Programm im Sinne der zugrunde liegenden Sprache sein.**

Der Leser mag erkennen, dass somit die obige Möglichkeit, alternative Methodendeklarationen zu implementieren, verboten wurde. Diese Einschränkung ist aber kein Instrument dazu, den in dieser Arbeit entwickelten Mechanismus zur Verwaltung alternativer Features zu motivieren. Dieser Mechanismus wird nämlich auch benötigt, um Alternativen zu nicht-optionalen Sprachkonstrukten (z. B. alternative Rückgabetypen einer Methode) angeben zu können, wie wir in Kapitel 4 ab Seite 37 noch diskutieren werden.

Zusammenfassend wurde diese Grundsatzentscheidung getroffen, um die Unterstützung des Programmierers durch bestehende Entwicklungsumgebungen beibehalten zu können. Außerdem ist es somit möglich, bestehende Typsysteme *echt erweitern* zu können. Es besteht nicht der Bedarf, Regeln bestehender Typsysteme abzuändern<sup>4</sup>, sondern nur, neue Regeln hinzuzufügen. Dies erleichtert die zusätzliche Berücksichtigung von Annotationen in vorhandenen Typsystemen erheblich.

### 3.3.5 CFJ-Typregeln

Wir werden nun ein Typsystem für CFJ angeben, das die Wohlgeformtheit eines CFJ-Programms überprüft. Die folgenden Ausführungen basieren auf den Erkenntnissen in [KA08b]<sup>5</sup>.

Vorab definieren wir die *Subtyp-Relation*  $<$ : als die reflexive, transitive Hülle der **extends**-Relation, d. h. für alle Klassen  $C, D, E$  gilt:

- $C <: C$
- $C <: D, D <: E \Rightarrow C <: E$
- **class**  $C$  **extends**  $D \Rightarrow C <: D$

Die Subtyp-Relation ist unabhängig von Annotationen, da der Supertyp einer Klasse nicht gefärbt werden kann. Somit entspricht die Subtyp-Relation von CFJ der von FJ.

---

<sup>4</sup> Im obigen Beispiel hätten wir die Regel „Eine Klasse darf höchstens eine Deklaration zu einer Methode enthalten“ in einem bestehenden Typsystem deaktivieren müssen, was in JDT beispielsweise nicht möglich ist.

<sup>5</sup> Manche Typprüfungen werden von denen von Kästner und Apel abweichen, da wir z. B. die oben angesprochene Grundsatzentscheidung berücksichtigen. Viele folgender Ergebnisse sind zusammen mit Kästner erarbeitet worden und werden in eine Überarbeitung und Erweiterung von [KA08b] miteinfließen.

## Überprüfung einer Klasse

Für jede Klasse des CFJ-Programms müssen folgende Überprüfungen durchgeführt werden:

**(L.1)** Die Superklasse muss existieren. Dies muss zum einen unter Vernachlässigung der Annotationen, d. h. im Sinne von FJ, zutreffen. Zum anderen muss gelten: Wenn die Klasse in einer Variante vorhanden ist, so muss in dieser Variante auch die Superklasse vorhanden sein. Solch eine Implikation ist die häufigste Annotationsprüfung im CFJ-Typsystem. Die meisten Fehler beim Annotieren können nämlich auftreten, wenn andere AST-Knoten referenziert werden.

Wir werden diese Implikation an dieser Stelle einmalig formalisieren. Alle folgenden Regeln der Art *Wenn das eine Codefragment in einer Variante vorhanden ist, so muss in dieser Variante auch das andere Codefragment vorhanden sein* können analog formalisiert werden. Sei dazu  $\text{annot}(C)$  die Annotation des AST-Knotens  $C$  (in CIDE also z. B. die Konjunktion aller Features, mit denen der Knoten gefärbt wurde). In diesem Fall benötigen wir die Werte  $\text{annot}(C)$  und  $\text{annot}(SC)$  für die betrachtete Klasse  $C$  und die angegebene Superklasse  $SC$ . Diese Typregel überprüft, ob die Formel  $F \Rightarrow (\text{annot}(C) \Rightarrow \text{annot}(SC))$  eine Tautologie ist, wobei  $F$  weiterhin das Featuremodell in Form einer aussagenlogischen Formel bezeichne.

Einen Sonderfall stellt der Supertyp `Object` dar, da er nicht explizit im Quellcode als Klasse deklariert wird. Wir gehen also davon aus, dass die Klasse `Object` ohne Annotation existiert.

**(L.2)** Für jedes Feld muss sein angegebener Typ existieren. Dies muss zum einen unter Vernachlässigung der Annotationen zutreffen. Zum anderen muss gelten: Wenn das Feld in einer Variante vorhanden ist, so muss in dieser Variante auch der Typ als Klassendeklaration vorhanden sein.

**(L.3)** Es darf keine zwei Klassen mit gleichem Namen geben. In diesem Fall ist keine zusätzliche Berücksichtigung von Annotationen notwendig, denn wenn diese Regel bereits in der Codebasis gilt, dann gilt sie insbesondere in jeder Variante.

Dies ist ein Beispiel dafür, dass die getroffene Grundsatzentscheidung die Typregeln vereinfacht. Würde man nämlich zwei Klassen mit gleichem Namen unter der Voraussetzung, dass sie mit sich gegenseitig ausschließenden Features annotiert sind, erlauben, so müsste man bei der Suche nach einer Klasse immer nach *allen* Deklarationen suchen und überprüfen, ob sie alternativ gefärbt sind und welche der Alternativen für die aktuelle Überprüfung betrachtet werden muss. Diese Aufgabe gilt es im Zusammenhang mit alternativen Codefragmenten in Kapitel 4 zu diskutieren. Der Unterschied besteht jedoch darin, dass alternative Klassendeklarationen nicht in der Codebasis, sondern extern gespeichert sein werden.

**(L.4)** Es darf keine zwei Methoden mit gleichem Namen geben. Dies ist gerade das verbotene *Overloading* innerhalb einer Klasse. Auch hier ist keine zusätzliche Berücksichtigung von Annotationen notwendig.

(L.5) Eine selbst deklarierte Klasse darf nicht `Object` heißen.

(L.6) Die Klasse darf sich in keinem Vererbungszyklus befinden. Da die Subtyp-Relation unabhängig von den Annotationen ist, muss diese Typregel auch keine Annotationen berücksichtigen.

Die Vorgehensweise zur Überprüfung dieser Regel ist einfach: Ausgehend von der Klasse verfolgen wir die Vererbungshierarchie aufwärts, bis wir `Object` erreichen. Dabei merken wir uns in jedem Schritt, welche Klassen bereits besucht wurden. Treffen wir auf eine Klasse, die bereits besucht wurde, haben wir einen Zyklus gefunden und geben einen Fehler aus.

(L.7) In der Klasse darf es keine zwei Felder mit gleichem Namen geben. Außerdem darf es zu einem Feld der Klasse kein gleichnamiges Feld in einer unmittelbaren oder mittelbaren Superklasse geben (*Overshadowing*). Da diese Regel bereits in der Codebasis gelten muss, gilt sie auch in allen Varianten. Eine zusätzliche Berücksichtigung von Annotationen ist also nicht notwendig.

### Überprüfung des Konstruktors

Wie bereits erwähnt, lassen wir den Konstruktor für theoretische Betrachtungen außen vor, da er durch die Felder der Klasse und aller Superklassen eindeutig bestimmt ist und weil wir dadurch keine neuen Erkenntnisse bzgl. der Berücksichtigung von Annotationen in Typsystemen gewinnen. Nichtsdestotrotz wurden die notwendigen Überprüfungen implementiert. Zu erwähnen ist lediglich, dass die Überprüfung des Konstruktors einen guten technischen Überblick v. a. in der Aufteilung seiner Parameter in Initialwerte für Superklassen und Initialwerte für die instanziierte Klasse benötigt.

### Überprüfung einer Methode

Neben diesen Typprüfungen müssen für jede Methode einer jeden Klasse folgende Regeln überprüft werden:

(M.1) Der Rückgabotyp muss existieren. Dies muss zum einen unter Vernachlässigung der Annotationen zutreffen. Zusätzlich muss gelten: Wenn die Methode in einer Variante vorhanden ist, so muss in dieser Variante auch der Rückgabotyp als Klassendeklaration vorhanden sein.

(M.2) **Gültiges Overriding** Die Möglichkeit in FJ, Methoden zu überschreiben, führte anfangs bei der Entwicklung des Typsystems zu Schwierigkeiten, obwohl ein Overloading, das den Sachverhalt noch verkomplizieren könnte, sogar verboten ist. Grund für diese anfänglichen Probleme war u. a. das Beispiel aus Abbildung 3.4.

Davon abgesehen, dass sich die Features *Grün* und *Rot* gegenseitig ausschließen müssen, damit jede Variante ein lauffähiges FJ-Programm ist, stellt sich die Frage, welche Varianten von `someMethod` durch die Implementierung in `SubClass` überschrieben werden können. Diese Frage ist wichtig, da z. B. die Annotationen der

```

1 class SuperClass extends Object {
2     SuperClass() { super(); }
3
4     Object someMethod(Object p1) { return new ThisClass(); }
5
6     Object someMethod(Object p1, Object p2) { return new ThatClass(); }
7 }
8
9 class SubClass extends SuperClass {
10     SubClass() { super(); }
11
12     Object someMethod(Object p1, Object p2) { return this; }
13 }

```

"Object p1"

Features:

- Blau
- Rot

Direct Colors: Rot

Inherited Colors: Blau

Abbildung 3.4: Überschreiben alternativer Methoden

formalen Parameter von überschriebener und überschreibender Methode überprüft werden müssen, wie wir nachher noch sehen werden. Falls z. B. die Features *Blau* und *Grün* nicht gleichzeitig vorhanden sein können, wird die rote Variante von `someMethod` überschrieben, falls das Feature *Rot* vorhanden ist. Enthält eine Konfiguration aber das Feature *Rot* nicht, so liegt kein Overriding vor, obwohl `SubClass.someMethod` dann nur den Parameter `p2` enthält (denn die Features *Grün* und *Blau* schließen sich ja gegenseitig aus).

Solche Beispiele kann man entlang der Vererbungshierarchie beliebig verkomplizieren. Der Leser mag bemerkt haben, dass das vorliegende Beispiel gegen unsere Grundsatzentscheidung verstößt. Es war eine weitere Motivation dafür, diese Entscheidung zu treffen, denn es erleichtert die zu tätigen Überprüfungen beim Overriding.

Die übrigen, mit der Grundsatzentscheidung verträglichen Fälle, können jedoch ähnlich komplex sein. Wir betrachten als Beispiel Abbildung 3.5.

In jeder Klasse entlang der Vererbungshierarchie wurde die Methode `someMethod` implementiert. Aufgrund der Grundsatzentscheidung und dem in FJ verbotenen Overloading wäre es z. B. nicht erlaubt, in der Klasse `SuperClass` den Parameter `o` wegzulassen. Es bleibt die Frage, welche Methode z. B. von `SubClass.someMethod` überschrieben wird. Je nach Featuremodell und je nach konkreter Feature-Auswahl kann es Fälle geben, in denen die Methode `SuperClass.someMethod`, aber auch `SuperSuperClass.someMethod` überschrieben wird.

Das Vorgehen zur Überprüfung des Overridings in FJ ist einfach: Ausgehend von `SubClass.someMethod` verfolgt man die Vererbungshierarchie so lange aufwärts, bis man eine Methode mit dem Namen `someMethod` gefunden hat. Das Overriding ist genau dann gültig, wenn Rückgabetypen und Parametertypen exakt übereinstimmen. Die Vererbungshierarchie muss nicht weiter aufwärts verfolgt

```

1 class SuperSuperClass extends Object {
2     SuperSuperClass() { super(); }
3
4     Object someMethod(Object o) { return new ThisClass(); }
5 }
6
7 class SuperClass extends SuperSuperClass {
8     SuperClass() { super(); }
9
10    Object someMethod(Object o) { return new ThatClass(); }
11 }
12
13 class SubClass extends SuperClass {
14     SubClass() { super(); }
15
16    Object someMethod(Object o) { return this; }
17 }

```

Abbildung 3.5: Überschreiben von Methoden entlang der Vererbungshierarchie

werden, da `SubClass.someMethod` immer die gefundene Methode überschreibt. Wegen unserer Grundsatzentscheidung führen wir diese Überprüfung auch im CFJ-Typsystem durch.

Im Zusammenhang mit Annotationen ist der Sachverhalt komplizierter. Hier kann es passieren, dass die soeben gefundene Methode in manchen Varianten nicht vorhanden ist, so dass in diesem Fall eine andere, sich in der Vererbungshierarchie weiter oben befindliche Methode, überschrieben werden könnte. Es müssen also weitere Überprüfungen gemacht werden.

Bevor wir die Vorgehensweise zum Finden all derjenigen Methoden, die überschrieben werden können, besprechen, soll geklärt werden, welche Eigenschaften von überschriebener Methode  $m$  und überschreibender Methode  $m'$  überprüft werden müssen. Von Belang sind nur diejenigen Varianten, in denen beide Methoden vorhanden sind. In diesen Fällen muss der  $i$ -te Parameter  $p_i$  von  $m$  in *genau den gleichen Varianten* vorhanden sein wie der  $i$ -te Parameter  $p'_i$  von  $m'$ . Das bedeutet: Wenn der  $i$ -te Parameter von  $m$  vorhanden ist, dann muss auch der  $i$ -te Parameter von  $m'$  vorhanden sein *und umgekehrt*. Wäre dies nicht der Fall, so gäbe es Varianten, in denen sich die Signaturen von überschriebener und überschreibender Methode unterscheiden würden, was in FJ verboten ist.

Formal muss für jeden formalen Parameter überprüft werden, ob folgende Formel eine Tautologie ist:  $F \Rightarrow ((annot(m) \wedge annot(m')) \Rightarrow (annot(p_i) \Leftrightarrow annot(p'_i)))$

Dies ist ein Beispiel dafür, dass Restriktionen in der Sprache auch dazu führen können, dass Typprüfungen komplizierter werden. In FJ hatte das Verbot des Overloadings den Vorteil, dass bei der Suche nach einer Methode (z. B. bei der Überprüfung eines Methodenaufrufs) nur auf den Namen und nicht auf die ganze Signatur geachtet werden musste. Die zusätzliche Überprüfung auf gültiges Overriding war trotzdem einfach möglich. Im CFJ-Typsystem müssen wir nun jedoch bei zusätzlicher Berücksichtigung von Annotationen komplexere Überprüfungen zur Gewährleistung eines gültigen Overridings durchführen.



Abschließend ist noch zu klären, bei welchen Methoden diese Überprüfung der Parameter-Annotationen gemacht werden muss. Grundsätzlich ist klar, dass dies bei all denjenigen Methoden geschehen muss, zu der es eine Variante gibt, in der sie *unmittelbar* im Sinne von FJ überschrieben wird. Die Schwierigkeit besteht darin, dies algorithmisch zu formulieren. Eine mögliche Lösung ist folgende Vorgehensweise. Sei dazu  $m'$  die überschreibende Methode: Wir durchlaufen alle Methoden  $m$  mit gleichem Namen in der Vererbungshierarchie aufwärts. Es werden nur diejenigen Methoden betrachtet, die zusammen mit  $m'$  vorhanden sein können. Dies ist ein Standardproblem für den SAT-Löser, denn es muss die *Erfüllbarkeit* von  $F \wedge \text{annot}(m) \wedge \text{annot}(m')$  überprüft werden. Sei  $m$  nun also eine Methode, die zusammen mit  $m'$  vorhanden sein kann. Sei  $M$  die Menge aller Methoden  $m''$  mit folgenden Eigenschaften:

- Der Name von  $m''$  ist gleich dem Namen von  $m'$
- $m''$  befindet sich in der Vererbungshierarchie *echt* zwischen  $m'$  und  $m$ . Steht  $m'$  also z.B. in Klasse  $A$  und  $m$  in Klasse  $C :> A$ , so durchsuchen wir alle Klassen  $B$  mit  $A <: B, B <: C$  und  $B \neq A, C$ .

Nun muss überprüft werden, ob es unter all den Varianten, in denen  $m$  und  $m'$  gleichzeitig vorhanden sind, eine gültige Variante gibt, in der *alle* Methoden  $m'' \in M$  *nicht* vorhanden sind. Nur dann wird  $m$  von  $m'$  *unmittelbar* überschrieben. Formal wird die *Erfüllbarkeit* folgender Formel geprüft:  $F \wedge \text{annot}(m) \wedge \text{annot}(m') \wedge \bigwedge_{m'' \in M} \neg \text{annot}(m'')$

Zur Veranschaulichung betrachten wir noch einmal Abbildung 3.5 unter der Annahme, dass alle drei Features optional und unabhängig voneinander sind. Wir möchten überprüfen, ob die Methode `SubClass.someMethod` ein gültiges Overriding ist. Im ersten Schritt wird das Overriding im Sinne von FJ überprüft. Es wird die Methode `SuperClass.someMethod` gefunden und festgestellt, dass Rückgabotyp und Parametertyp übereinstimmen.

Im nächsten Schritt werden alle Methoden ermittelt, die gleichzeitig mit der überschreibenden Methode vorhanden sein können. Aufgrund unserer Annahme kommen beide Methoden in den Superklassen in Frage. Bei der Betrachtung von `SuperClass.someMethod` gibt es keine Klasse zwischen `SubClass` und `SuperClass`, so dass die Methode überschrieben werden kann. Dadurch, dass kein Parameter separat gefärbt wurde, ist klar, dass beide Parameter in genau den gleichen Varianten, in denen beide Methoden existieren, vorhanden sind. Wäre aber einer der beiden Parameter mit einem vierten, unabhängigen Feature annotiert, würde ein Typfehler ausgegeben werden, da in allen Varianten, die dieses Feature nicht enthalten, die Anzahl der Parameter nicht übereinstimmen würde.

Interessanter ist die Betrachtung von `SuperSuperClass.someMethod`. Die Methode `SuperClass.someMethod` liegt echt zwischen den Klassen `SubClass` und `SuperSuperClass`. Außerdem gibt es Varianten, in denen `SuperClass.someMethod` nicht vorhanden ist, nämlich alle Varianten ohne das Feature *Rot*. Aus diesem Grund müssen auch die Parameter-Annotationen von `SubClass.someMethod` und `SuperSuperClass.someMethod` verglichen werden.

Abbildung 3.6 zeigt das Beispiel, in dem zusätzlich der Parameter der Methode `someMethod` in der Klasse `SuperSuperClass` mit einem vierten Feature gefärbt wurde. In beiden der anderen Methoden wird erkannt, dass dadurch ein Typfehler vorliegt.

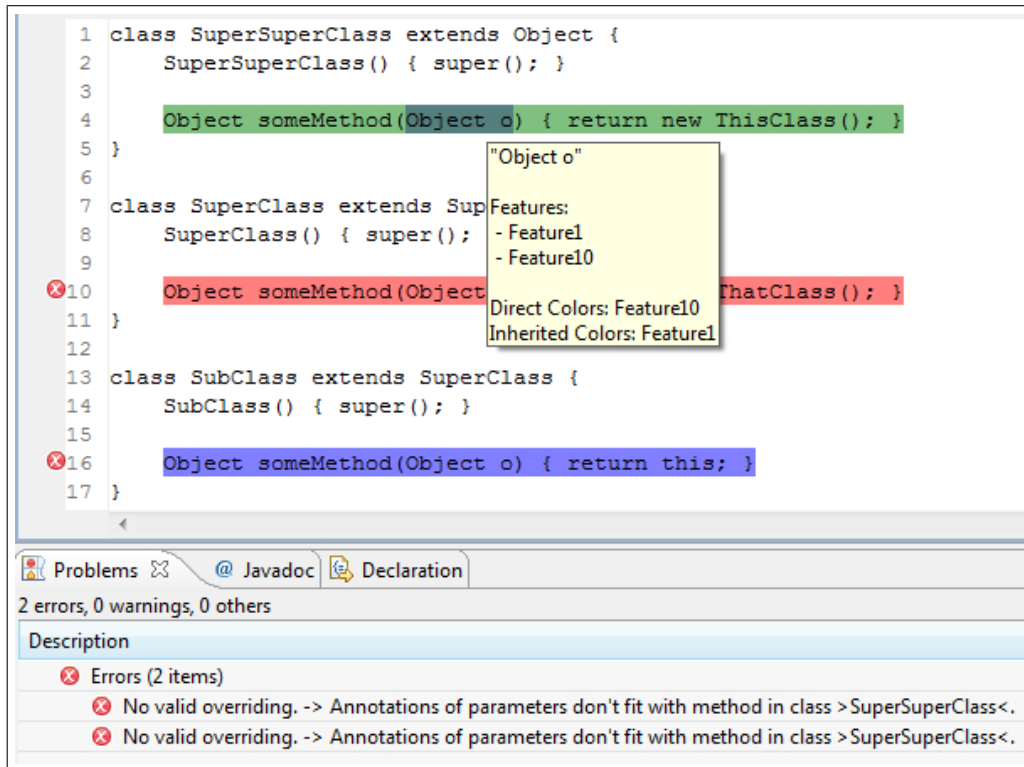


Abbildung 3.6: Ungültiges Überschreiben einer Methode

Wir fahren mit den restlichen Typprüfungen für eine Methode fort.

(M.3) Der Typ eines jeden formalen Parameters muss existieren. Dies muss zum einen unter Vernachlässigung der Annotationen zutreffen. Außerdem muss gelten: Wenn ein formaler Parameter in einer Variante vorhanden ist, so muss in dieser Variante auch der Typ als Klassendeklaration vorhanden sein.

(M.4) Es darf keine zwei formalen Parameter mit gleichem Namen geben. Diese Überprüfung erfolgt unabhängig von der Annotierung.

(M.5) Der Term des **return**-Statements muss wohlgeformt sein. Dazu wird der Typ dieses Terms bestimmt, wie im Anschluss erläutert. Kann dieser Typ nicht bestimmt werden, so ist diese Regel verletzt.

(M.6) Der Typ des Terms des **return**-Statements muss ein Subtyp des Rückgabetyps im Sinne der oben angegebenen Subtyp-Relation  $<:$  sein. Sobald der Typ des Terms in (M.5) ermittelt wurde, kann diese Regel unabhängig von Annotationen überprüft werden, denn der Rückgabetypp einer Methode ist in jeder Variante der gleiche, da er als nicht-optionales Sprachkonstrukt nicht annotiert werden kann.

### Bestimmung des Typs eines Terms

Die restlichen Typregeln beziehen sich auf fünf Arten von Termen, die innerhalb eines **return**-Statements einer Methode stehen können. Solche Terme können ineinander verschachtelt werden. So kann z. B. als Aufrufparameter eines Konstruktors ein Methodenaufruf stehen. Für einen Term muss sein Typ bestimmt werden.

**Variablenzugriff** Die Variable muss als formaler Parameter der Methode existieren. Dies muss zum einen unter Vernachlässigung der Annotationen zutreffen. Außerdem muss gelten: Wenn der Variablenzugriff in einer Variante vorhanden ist, so muss in dieser Variante auch der formale Parameter vorhanden sein. Sind beide Bedingungen erfüllt, so ist der Typ des Terms gleich dem Typ des formalen Parameters.

**Feldzugriff** Ein Feldzugriff der Form `invokeTarget.field` besteht aus einem Aufrufziel (`invokeTarget`), das wiederum ein Term ist, sowie aus dem Feld des Aufrufziels (`field`). Es müssen folgende Überprüfungen gemacht werden:

(F.1) Das Aufrufziel muss wohlgeformt sein, d. h. einen Typ  $T$  haben.

(F.2) Das Feld muss in der Klasse  $T$  oder einer Superklasse von  $T$  existieren. Dies muss zum einen unter Vernachlässigung der Annotationen zutreffen. Zum anderen muss gelten: Wenn der Feldzugriff in einer Variante vorhanden ist, so muss in dieser Variante auch das Feld in der Klasse  $T$  oder einer Superklasse von  $T$  vorhanden sein.

Sind beide Bedingungen erfüllt, so ist der Typ des Terms gleich dem Typ des gefundenen Feldes. Dieses Feld kann unabhängig von Annotationen ermittelt werden, da das Überschatten von Feldern verboten ist und somit in der Vererbungshierarchie des Aufrufziels höchstens ein Feld mit dem entsprechenden Namen existiert.

**Methodenaufruf** Ein Aufruf einer Methode der Form `invokeTarget.method` (`term_1, ..., term_n`) besteht aus einem Aufrufziel (`invokeTarget`) – wiederum ein Term –, einer Methode des Aufrufziels (`method`) sowie aus einer Liste von Termen (`term_1, ..., term_n`), die als Parameter übergeben werden. Es müssen folgende Überprüfungen gemacht werden:

(I.1) Das Aufrufziel muss wohlgeformt sein, d. h. einen Typ  $T$  haben.

(I.2) Jeder Term der übergebenen Liste von Termen muss wohlgeformt sein, d. h. Typen  $T_1, \dots, T_n$  haben.

(I.3) Die aufgerufene Methode muss existieren. Da aber Overriding und Annotationen überschriebener Methoden möglich sind, kann es analog zum Overriding je nach Feature-Auswahl verschiedene Methoden  $m_1, \dots, m_k$  geben, die aufgerufen werden. Die dazu in Frage kommenden Methodendeklarationen werden analog zum Overriding bestimmt, wobei die Annotation der überschriebenen Methode durch die Annotation des Methodenaufrufs ersetzt werden muss.

Zusätzlich muss überprüft werden, ob es unter all denjenigen Varianten, in denen der Methodenaufruf vorhanden ist, eine Variante gibt, in der all diese

in Frage kommenden Methoden nicht vorhanden sind. Ist dies der Fall, so gibt es eine Variante, in der die aufgerufene Methode nicht existiert und es muss ein Fehler ausgegeben werden. Die Überprüfung erfolgt mit einem SAT-Löser, indem auf die *Erfüllbarkeit* der Formel  $F \wedge \text{annot}(I) \wedge \bigwedge_{i \in 1..k} \neg \text{annot}(m_i)$  getestet wird, wobei  $I$  den Methodenaufruf bezeichne.

All diese gefundenen Methoden haben gleichen Rückgabotyp und gleiche Parametertypen, denn sonst hätte bereits die Überprüfung des Overridings in Regel (M.2) einen Fehler ausgegeben. Der Typ des Methodenaufrufs ist also der Rückgabotyp der Methoden, falls die anderen Überprüfungen keinen Fehler zurückgeben.

(I.4) Die Anzahl der Terme, die übergeben werden, und die Anzahl der formalen Parameter der Methoden müssen übereinstimmen.

(I.5) Die Typen  $T_1, \dots, T_n$  der übergebenen Terme müssen Subtypen der Typen der formalen Parameter der Methoden sein.

(I.6) Für jede der in (I.3) ermittelten Methoden muss gelten: In allen Varianten, in denen der Methodenaufruf und die Methode gleichzeitig vorhanden sind, müssen der  $i$ -te übergebene Term und der  $i$ -te formale Parameter der Methode in *genau den gleichen Varianten* vorhanden sein. Diese Überprüfung wurde bereits im Zusammenhang mit Overriding beim Vergleich der Parameter-Annotationen von überschriebener und überschreibender Methode erklärt.

**Objekterzeugung** Ein Term `new SomeClass(term_1, ..., term_n)`, der ein Objekt mittels `new` erzeugt, besteht aus dem Namen der Klasse (`SomeClass`) sowie aus einer Liste von Termen (`term_1, ..., term_n`), die als Parameter übergeben werden. Bei einer Objekterzeugung müssen alle Felder der Klasse und aller Superklassen in der richtigen Reihenfolge initialisiert werden. Eine korrekt sortierte Liste dieser Felder erhält man, wenn man ausgehend von der instanziierten Klasse die Felder der Klasse von unten nach oben immer vorne an die Liste anhängt und so die Vererbungshierarchie hinaufsteigt. Zur Veranschaulichung betrachte man Listing 3.3. Als Felder der Klasse `SubClass` wird die Liste `[p1, p2, o1, o2]` zurückgegeben.

```
class SuperClass extends Object {
    Object p1;
    Object p2;

    SuperClass(Object p1, Object p2) {
        super();
        this.p1 = p1;
        this.p2 = p2;
    }
}

class SubClass extends SuperClass {
    Object o1;
    Object o2;

    SubClass(Object p1, Object p2, Object o1, Object o2) {
        super(p1, p2);
    }
}
```

```

    this.o1 = o1;
    this.o2 = o2;
  }
}

```

Listing 3.3: Beispiel zur Ermittlung aller Felder einer Klasse

Folgende Überprüfungen müssen durchgeführt werden:

**(N.1)** Die Klasse muss existieren. Zum einen muss dies unter Vernachlässigung der Annotationen zutreffen. Zum anderen muss gelten: Wenn der Term in einer Variante vorhanden ist, so muss in dieser Variante auch die Klasse vorhanden sein.

**(N.2)** Jeder Term der übergebenen Liste von Termen muss wohlgeformt sein, d. h. Typen  $T_1, \dots, T_n$  haben.

**(N.3)** Die Anzahl der Terme, die übergeben werden, und die Länge der Liste aller Felder der Klasse und aller Superklassen müssen übereinstimmen.

**(N.4)** Die Typen  $T_1, \dots, T_n$  der übergebenen Terme müssen Subtypen der Typen der Felder sein.

**(N.5)** In allen Varianten, in denen der Term vorhanden ist, müssen der  $i$ -te übergebene Term und das  $i$ -te Feld aus obiger Liste in *genau den gleichen Varianten* vorhanden sein. Wäre dies nicht der Fall, so würde es Felder geben, die nicht initialisiert werden oder es würden Terme übergeben werden, zu denen es kein Feld gibt, das sie initialisieren können.

Sind all diese Bedingungen erfüllt, so ist der Typ des Terms gerade die Klasse, die erzeugt wird.

**Cast** Ein Cast-Term von der Gestalt `(SomeClass) term` besteht aus einer Klasse `(SomeClass)` sowie einem Term `(term)`. Es muss folgendes überprüft werden:

**(C.1)** Die Klasse muss existieren. Zum einen muss dies unter Vernachlässigung der Annotationen zutreffen. Zum anderen muss gelten: Wenn der Cast-Ausdruck in einer Variante vorhanden ist, so muss in dieser Variante auch die Klasse vorhanden sein.

**(C.2)** Der Term, der gecasted werden soll, muss wohlgeformt sein, d. h. einen Typ  $T$  haben.

**(C.3)** Der Cast muss legal sein, d. h. soll ein Term vom Typ  $T$  auf einen Typ  $C$  gecasted werden, so muss  $T <: C$  (*Upcast*) oder  $C <: T$  (*Downcast*) gelten.

Sind alle Bedingungen erfüllt, so ist der Typ des Cast-Ausdrucks gerade die Klasse, auf die gecasted wird.

All diese Typregeln wurden innerhalb von CIDE implementiert und erfolgreich getestet. Kästner und Apel [KA08b] haben für die dort angegebenen Typregeln (die unsere Grundsatzentscheidung noch nicht berücksichtigen) den Beweis dafür skizziert, dass alle Varianten, die man aus einer im Sinne dieses CFJ-Typsystems wohlgeformten SPL erzeugen kann, auch wohlgeformt im Sinne des

FJ-Typsystems sind<sup>6</sup>. Sie bemerken, dass der Beweis sehr der Vorgehensweise bei der Erweiterung des FJ-Typsystems entspricht: Zu jeder FJ-Typregel muss man sich überlegen, wie sie durch Annotationen verletzt werden kann. Z. B. kann ein Feldzugriff nicht nur dadurch ungültig sein, dass das Feld nicht existiert (FJ-Typregel), sondern auch dadurch, dass Feldzugriff und Feld „falsch“ annotiert sind.

---

<sup>6</sup> In der bereits angesprochenen Überarbeitung und Erweiterung von [KA08b] werden Kästner et al. diesen Beweis an die überarbeiteten CFJ-Typregeln, die u. a. unsere Grundsatzentscheidung berücksichtigen werden, anpassen.

---

# KAPITEL 4

---

## Alternative Features in CFJ

Auch wenn alternative Features bei der Dekomposition einer bestehenden Anwendung in Features – dem ursprünglichen Anwendungsgebiet von CIDE – eher selten vorkommen, finden sie bei der Neuentwicklung einer SPL Anwendung [KA08b]. So kann es z. B. in einer Datenbank-SPL verschiedene Mechanismen zur persistenten Speicherung der Daten geben, wobei es in einer Variante dieser Datenbank *genau eine* dieser Mechanismen geben muss.

Dieses Kapitel beginnt mit einer Erklärung der Probleme, die diese Arbeit im Hinblick auf alternative Features löst. Dabei wird insbesondere die Grundsatzentscheidung, die wir im Abschnitt 3.3.4 auf Seite 25 getroffen haben, eine wichtige Rolle spielen. Es wird folgen, dass die bestehenden Möglichkeiten in CIDE nicht ausreichen, um alternative Features für unsere Bedürfnisse zu realisieren. Bevor wir einen Mechanismus vorstellen, der alternative Features innerhalb von CIDE ermöglicht, sprechen wir andere Ansätze an, mit denen – teilweise unter bestimmten Einschränkungen – alternative Features implementiert werden können.

Es folgen Ausführungen zu dem Mechanismus, der in CIDE implementiert wurde. Neben dem zugrunde liegenden Prinzip erklären wir, wie dieser Mechanismus in CIDE realisiert wurde. Im Anschluss erläutern wir, wie weiterhin sowohl die syntaktische als auch die sprachsemantische Korrektheit aller Varianten gewährleistet wird. Dazu werden wir das bestehende CFJ-Typsystem als Basis verwenden und entsprechend erweitern. Dabei wird die Frage diskutiert, zu welchen Arten von AST-Knoten tatsächlich Alternativen angegeben werden können sollen.

### 4.1 Problemstellung

Einleitend in dieser Arbeit haben wir bereits eine Möglichkeit angesprochen, alternative formale Parameter einer Methodendeklaration zu realisieren (siehe Abschnitt 1.1 auf Seite 2). In Listing 1.1 werden dort die beiden Alternativen einfach nebeneinander geschrieben und mit den entsprechenden alternativen Features annotiert. Dadurch, dass sich die beiden Features gegenseitig ausschließen, können in einer gültigen Variante nicht beide Alternativen gleichzeitig vorhanden sein. Jede Variante ist also wohlgeformt.

Das Beispiel in Abbildung 3.3 im Abschnitt 3.3.4 über unsere Grundsatzentscheidung (Seite 25) verfolgt den gleichen Ansatz. Zwei alternative Methodendeklarationen (d. h. die Methoden haben gleichen Namen, gleichen Rückgabotyp und gleiche Parametertypen, aber unterschiedliche Implementierungen) wurden untereinander geschrieben und mit zwei alternativen Features gefärbt. Dieser

Codeausschnitt ist syntaktisch korrekt, denn es handelt sich um zwei Methoden-deklarationen. Im CFJ-Typsystem *könnte* man überprüfen, ob sich die beiden Annotationen gegenseitig ausschließen und könnte somit gewährleisten, dass in jeder Variante höchstens eine der beiden Alternativen vorhanden ist.

Dieses Beispiel haben wir aber durch unser grundsätzliches Bestreben, bestehende Typsysteme wiederverwenden zu können, in einer Grundsatzentscheidung untersagt. Somit kann der Ansatz aus diesem Beispiel und aus dem der Einleitung nicht verwendet werden, um alternative Codefragmente zu implementieren.

Dies ist jedoch nicht der einzige Grund, warum wir einen eigenen Mechanismus zur Verwaltung alternativer Features implementiert haben. Die Strategie des Nebeneinander- oder Untereinanderschreibens ist bei nicht-optionalen Codefragmenten nämlich bereits syntaktisch nicht möglich. Man kann beispielsweise alternative Rückgabetypen einer Methode nicht dadurch implementieren, dass man `SomeClass OtherClass someMethod() /*...*/` schreibt und die Rückgabetypen `SomeClass` und `OtherClass` entsprechend färbt<sup>1</sup>. Falls man also auch Alternativen zu nicht-optionalen Codefragmenten angeben können möchte, besteht unabhängig von unserer getroffenen Grundsatzentscheidung der Bedarf eines eigenen Mechanismus.

Zur Notwendigkeit, Alternativen zu nicht-optionalen Codefragmenten angeben zu können, kann nur schwer eine Aussage gemacht werden. Grund dafür ist, dass es an Fallstudien mangelt, die den Bedarf an derart feingranularen Alternativen untersuchen. Solche Fallstudien können auch erst dann durchgeführt werden, wenn es Werkzeuge dazu gibt. Daher haben wir uns dazu entschlossen, zunächst einen Mechanismus zur Realisierung alternativer Features zu entwickeln, ohne den genauen Bedarf zu kennen. Mit Hilfe dieser CIDE-Erweiterungen können alternative Features detaillierter erforscht werden.

Außerdem wurde auch im Rahmen dieser Arbeit keine Fallstudie durchgeführt, da FJ keine Programmiersprache ist, die die realistischen Programmierbedingungen der Praxis widerspiegelt. Auch wenn FJ eine Turing-vollständige Sprache ist, wird man damit keine praktisch relevanten Programme schreiben.

Der hier vorgestellte Mechanismus zur Realisierung alternativer Codefragmente ist unabhängig von der zugrunde liegenden Programmiersprache, funktioniert also insbesondere auch für Java. Lediglich das Typsystem ist (C)FJ-spezifisch. Somit könnte man den Mechanismus auch grundsätzlich für eine Java-Fallstudie verwenden. Jedoch waren wir uns bereits in FJ zunächst im Unklaren darüber, zu welcher Art von AST-Knoten man überhaupt Alternativen angeben können soll und mit welchem Aufwand dies vom Typsystem überprüft werden kann. Daher haben wir uns zunächst auf FJ konzentriert und im Zusammenhang mit dem Typsystem tatsächlich festgestellt, dass es bei den meisten Arten von AST-

---

<sup>1</sup> Bei Verwendung von C/C++ Präprozessor-Direktiven ist es tatsächlich möglich, `#ifndef ALTERNATIVE1 SomeClass #endif`  
`#ifndef ALTERNATIVE2 OtherClass #endif someMethod() /*...*/`  
zu schreiben, da Präprozessor-Direktiven Bestandteil der Syntax sind und im Gegensatz zu den Feature-Annotationen in CIDE nicht außerhalb der Codebasis gespeichert werden.



Knoten problematisch ist, Alternativen zu ermöglichen (siehe Abschnitt 4.6.1 auf Seite 51).

Der hier entwickelte Mechanismus hat aber auch für optionale Codefragmente Vorteile. Beim einfachen Nebeneinander- oder Untereinanderschreiben alternativer Codefragmente wird nämlich nicht direkt deutlich, dass es sich um alternative Codefragmente handelt. Dies könnte man wohl innerhalb einer Entwicklungsumgebung implementieren, indem man die Codebasis nach nebeneinander oder untereinander stehenden Codefragmenten durchsucht, die mit alternativen Features annotiert sind. Die so gefunden Codefragmente könnte man im Editor entsprechend kennzeichnen. Das Durchsuchen der gesamten Codebasis nach alternativen Codefragmenten kann jedoch sehr aufwendig sein. In unserem Ansatz werden Alternativen einfach direkt als solche gekennzeichnet und angezeigt.

## 4.2 Andere Ansätze

Bevor wir den in dieser Arbeit entwickelten Mechanismus zur Verwaltung alternativer Codefragmente vorstellen, diskutieren wir andere Möglichkeiten, alternative Features zu realisieren [KA08b]. Dabei werden sich Einschränkungen und Nachteile ergeben, die die Entwicklung eines eigenen Mechanismus zusätzlich motivieren.

### 4.2.1 Alternative Methoden durch Overriding

Betrachten wir erneut die Anforderung, Alternativen zu Methodendeklarationen angeben zu können. Unsere Grundsatzentscheidung verbietet, die Alternativen einfach untereinander zu schreiben und entsprechend zu annotieren, da dadurch ein ungültiges FJ-Programm entsteht. Trotzdem hat man durch Overriding die Möglichkeit, alternative Methoden zu realisieren, ohne die CFJ-Syntax und das CFJ-Typsystem anpassen zu müssen.

Soll eine Klasse `SomeClass`  $n$  alternative Implementierungen (bei  $n$  sich gegenseitig ausschließenden Features) der Methode `someMethod` haben, so kann man dies durch die Vererbungskette in Abbildung 4.1 erreichen.

Zu jeder Alternative wird eine eigene Superklasse angelegt, wobei die darin enthaltene Methode `someMethod` den Features der Alternative entsprechend annotiert ist. Die sich in dieser Vererbungskette am weitesten unten befindliche Klasse erhält den Namen derjenigen Klasse, die der Benutzer instanziierten soll (in unserem Beispiel `SomeClass`)<sup>2</sup>. Sind die Methoden tatsächlich mit alternativen Features annotiert, so existiert in jeder Variante der SPL höchstens eine der Implementierungen. Die Vererbungshierarchie ist in jeder Variante die gleiche, da die Klassen gleich annotiert sind.

Die Typregel zur Überprüfung eines Methodenaufrufs ist bereits so implementiert, dass sie entlang der Vererbungshierarchie überprüft, ob in jeder Variante, in der der Aufruf vorhanden ist, auch eine Variante der Methode vorhanden ist, die

---

<sup>2</sup> Die anderen Klassen könnte man in Java als *abstrakt* deklarieren, um eine Instanziierung zu verbieten.

```

1 class SomeClass extends SomeClass1 {
2     SomeClass() { super(); }
3     Object someMethod() { return new ThisClass(); }
4 }
5
6 class SomeClass1 extends SomeClass2 {
7     SomeClass1() { super(); }
8     Object someMethod() { return new ThatClass(); }
9 }
10
11 // ...
12
13 class SomeClass_n extends Object {
14     SomeClass_n() { super(); }
15     Object someMethod() { return new OtherClass(); }
16 }

```

Abbildung 4.1: Alternative Methoden durch Overriding

aufgerufen werden kann. Jedoch kann nicht gewährleistet werden, dass sich die Methodendeklarationen tatsächlich gegenseitig ausschließen, da sie nicht als solche gekennzeichnet werden können. Beabsichtigt man also tatsächlich, per Overriding Alternativen anzugeben, erhält man von der Entwicklungsumgebung keine Unterstützung bei der Überprüfung, ob die verschiedenen Varianten der Methode tatsächlich mit sich gegenseitig ausschließenden Features annotiert sind.

Ein weiterer Nachteil ist die Tatsache, dass die Vererbungskette lediglich aus technischen Gründen existiert und somit den Quellcode unnötig unüberschaubar macht. Auch die Entwicklung der benötigten Klassen stellt einen Aufwand dar. Zur Abmilderung dieser Nachteile könnte ein Mechanismus entwickelt werden, der die Erzeugung der Vererbungshierarchie automatisiert und vor dem Benutzer versteckt. Es bleibt jedoch der Nachteil, dass mit diesem Workaround lediglich alternative Methodendeklarationen ermöglicht werden.

### 4.2.2 Erweiterung des Sprachumfangs

Grundsätzlich besteht die Möglichkeit, die Syntax und Semantik einer Programmiersprache so zu verändern, dass alternative Codefragmente ermöglicht werden. Ein Beispiel dafür ist das bereits mehrfach erwähnte Nebeneinander- oder Untereinanderschreiben der Alternativen. Bei optionalen Codefragmenten (z. B. Methodendeklarationen) müsste dazu nur das Typsystem angepasst werden. Aus syntaktischer Sicht wäre dieses Beispiel möglich, da untereinander geschriebene Methoden aus Sicht des Parsers zwei Methodendeklarationen sind. Bei nicht-optionalen Codefragmenten (z. B. Rückgabetypen von Methoden) müsste neben dem Typsystem sogar die Syntax der Programmiersprache erweitert werden. In beiden Fällen wäre unsere Grundsatzentscheidung verletzt.

Eine Erweiterung des Sprachumfangs bietet viele weitere Möglichkeiten, wie z. B. in *Feature Featherweight Java (FFJ)* [AKL08], eine Erweiterung von FJ um *Refinements*, untersucht wurde. Eine genauere Betrachtung solcher Möglichkeiten würde den Rahmen dieser Arbeit jedoch sprengen.

### 4.2.3 Exkurs: Alternative Rückgabe-Statements in Java

In FJ hat man nicht die Möglichkeit, alternative **return**-Statements durch Unter-einanderschreiben der Alternativen zu realisieren, da bereits die Syntax vorgibt, dass ein Methodenrumpf aus *genau einem* **return**-Statement besteht. In Java ist dies nicht der Fall. Dort ist es syntaktisch erlaubt, das Beispiel aus Listing 4.1 zu implementieren.

```
public Object someMethod() {
    return new ThisClass();
    return new ThatClass();
}
```

Listing 4.1: Zwei **return**-Statements in Java

Jedoch bekommt man vom Typsystem den Fehler, dass es sich bei dem zweiten **return**-Statement um *unerreichbaren Code* (engl. *unreachable code*) handelt. Diese Fehlermeldung kann man aber durch einen bemerkenswert einfachen „Hack“ vermeiden, indem man vor das erste **return**-Statement ein **if (true)** schreibt. Dadurch, dass der Java-Compiler die Bedingung des **if** zur Kompilierungszeit nicht auswertet, weiß er nicht, dass das zweite **return**-Statement tatsächlich un-erreichbar ist. Zusammen mit einer geeigneten Färbung mit alternativen Features ist es in CIDE somit möglich, alternative **return**-Statements in Java zu implementieren (siehe Abbildung 4.2).

```
3 public Object someMethod() {
4     if (true) return new ThisClass();
5
6     return new ThatClass();
7 }
```

Abbildung 4.2: Alternative Rückgabe-Statements in Java

Kästner et al. schildern in einer Erweiterung und Überarbeitung von [KA08b], wie man alternative Methodendeklarationen und sogar alternative Klassen auf solche alternative **return**-Statements zurückführen kann. Dabei sind aber eine Reihe von Voraussetzungen notwendig, so dass das dort angegebene Verfahren nur in Spezialfällen angewandt werden kann.

Neben der Tatsache, dass dieser „Hack“ in FJ nicht verwendet werden kann, da es keine **if**-Anweisungen gibt, ist es ein Nachteil, dass es nur ein „Trick“ ist, um das Typsystem zu „überlisten“. Dafür spricht hingegen, dass dieser Workaround einfach erkannt werden kann, denn ein **if (true)** im Quellcode fällt dadurch auf, dass es eigentlich an keiner anderen Stelle verwendet wird. Möchte man mehr als zwei Alternativen angeben, könnte man weitere Alternativen zusätzlich mit **else if (true)** kennzeichnen. Hinzu kommt, dass keine Anpassungen an Syntax und Typsystem vorgenommen werden müssen. Eine Überprüfung, ob sich die Alternativen tatsächlich gegenseitig ausschließen, ist jedoch auch hier nicht einfach möglich, da Alternativen nicht als solche gekennzeichnet werden.<sup>3</sup> Außer-

<sup>3</sup> Es gibt jedoch die Möglichkeit, die Codebasis nach **if (true)**-Ausdrücken zu durchsuchen und dann die entsprechenden Annotationen zu überprüfen.

dem können Optimierungen von Compilern oder Entwicklungsumgebungen, mit denen solche eigentlich unnötigen Codefragmente erkannt werden, dazu führen, dass dieser Mechanismus nicht mehr verwendet werden kann.

Zusammenfassend stellen wir fest, dass – abgesehen von Ansätzen, die den Sprachumfang erweitern – in den hier angesprochenen Ansätzen nicht zu jeder Art von AST-Knoten Alternativen angegeben werden können. So funktioniert das Overriding nur bei alternativen Methodendeklarationen und der `if (true)`-Trick nur bei alternativen `return`-Statements in Java. Daher werden wir nun einen Mechanismus vorstellen, der die bestehende Syntax der Programmiersprache nicht verändert und der mit bestehenden Typsystemen in der Hinsicht kompatibel ist, dass Regeln nicht verändert, sondern nur neue Regeln hinzugefügt werden müssen. Damit können bestehende Typsysteme und Entwicklungsumgebungen weiterhin verwendet werden, da unser Mechanismus zur Verwaltung alternativer Features nur Erweiterungen vornimmt.

## 4.3 Prinzip der Meta-Expressions

Der in der vorliegenden Arbeit entwickelte Mechanismus zur Verwaltung alternativer Features wurde durch Arbeiten von Czarnecki und Antkiewicz [CA05] inspiriert. Dort wurden sog. *Meta-Expressions* vorgeschlagen, um alternative Werte innerhalb von UML-Modellen zu realisieren. Meta-Expressions werden außerhalb des ursprünglichen Dokuments separat gespeichert, so dass die Syntax der zugrunde liegenden Sprache nicht verändert werden muss. Gerade dies kommt unserer Grundsatzentscheidung entgegen. Die Entwicklungsumgebung unterstützt dabei die Verwendung dieser Meta-Expressions. So wird beispielsweise die Syntax der einzelnen Meta-Expressions geprüft. Außerdem werden Meta-Expressions bei der Erzeugung von Varianten mitberücksichtigt.

Wir erweitern die Idee der Meta-Expressions auf unsere Bedürfnisse. Dabei berücksichtigen wir sie insbesondere bei der Typprüfung. Außerdem wird es möglich sein, Meta-Expressions zu verschachteln, d. h. Meta-Expressions innerhalb von Meta-Expressions anzugeben. Grundsätzlich kann man – analog zur Färbung – nur Alternativen zu Knoten des AST und nicht zu jedem beliebigen Codefragment angeben.

## 4.4 Realisierung in CIDE

Dieser Abschnitt beschreibt die Realisierung der *Meta-Expressions* in CIDE, wobei wir im weiteren Verlauf nicht mehr von Meta-Expressions sprechen werden. Wir beginnen mit der Erklärung des Featuremodells, das wir für den Rest dieser Arbeit als Beispiel zugrunde legen werden. Es folgen Ausführungen zum grundlegenden Ablauf der Arbeit mit Alternativen in CIDE.

### 4.4.1 Featuremodell

Für die folgenden Beispiele in dieser Arbeit legen wir das Featuremodell aus Abbildung 4.3 zugrunde.

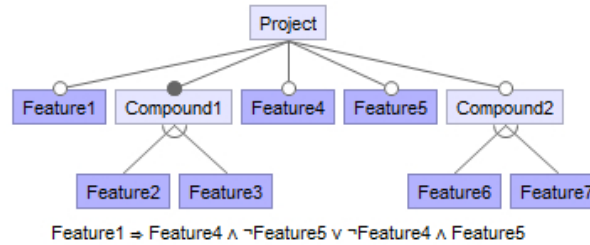


Abbildung 4.3: Featuremodell mit alternativen Features

Die Features *Feature2* und *Feature3* schließen sich gegenseitig aus. Dies erkennt man daran, dass sie mit ihrem gemeinsamen Vaterknoten *Compound1* durch einen nicht ausgefüllten Bogen verbunden sind. Der Vaterknoten selbst ist obligatorisch, da er mit einem ausgefüllten Kreis gekennzeichnet ist. Zusammen bedeutet dies, dass *Feature2* und *Feature3* in einer *XOR*-Beziehung stehen, d. h. in jeder Variante muss *genau eines* der beiden Features vorhanden sein.

Bei den Features *Feature6* und *Feature7* ist der Sachverhalt etwas anders. Sie schließen sich wohl auch gegenseitig aus (nicht ausgefüllter Bogen), jedoch ist ihr gemeinsamer Vaterknoten *Compound2* optional (nicht ausgefüllter Kreis), so dass in jeder Variante nur *höchstens eines* der beiden Features vorhanden sein muss. Hier ist es also erlaubt, dass keines der beiden Features für eine Variante ausgewählt ist.

Die Features *Feature1*, *Feature4* und *Feature5* sind zunächst nur drei voneinander unabhängige, optionale Features. Es ist jedoch noch eine Bedingung angegeben, die die Features *Feature4* und *Feature5* in eine *XOR*-Beziehung stellt, falls *Feature1* vorhanden ist<sup>4</sup>.

### 4.4.2 Entwicklungsprozess

Wir erklären nun anhand eines Beispiels, wie man mit dem in dieser Arbeit entwickelten Mechanismus in CIDE Alternativen anlegen und editieren kann. Grundlage soll zunächst eine Methode `someMethod` sein, die in zwei Alternativen vorliegen soll. Die erste Alternative entwickelt man einfach innerhalb des Editors und annotiert sie mit dem entsprechenden Feature der Alternative<sup>5</sup>. Um nun eine zweite Alternative zu dieser Methodendeklaration angeben zu können, markiert man sie (analog zum Färben) und wählt im Kontextmenü nun kein Feature aus, sondern den Menüpunkt *Create alternative* (siehe Abbildung 4.4).

<sup>4</sup> Die Formel  $(F_4 \wedge \neg F_5) \vee (\neg F_4 \wedge F_5)$  ist logisch äquivalent zu  $F_4 \text{ XOR } F_5$

<sup>5</sup> Theoretisch können auch zwei Konjunktionen von Features alternativ sein  $((F_1 \wedge \dots \wedge F_n) \text{ XOR } (G_1 \wedge \dots \wedge G_m))$ . Der Einfachheit halber betrachten wir hier aber nur Fälle, in denen eine Alternative durch *genau ein* Feature gegeben ist.

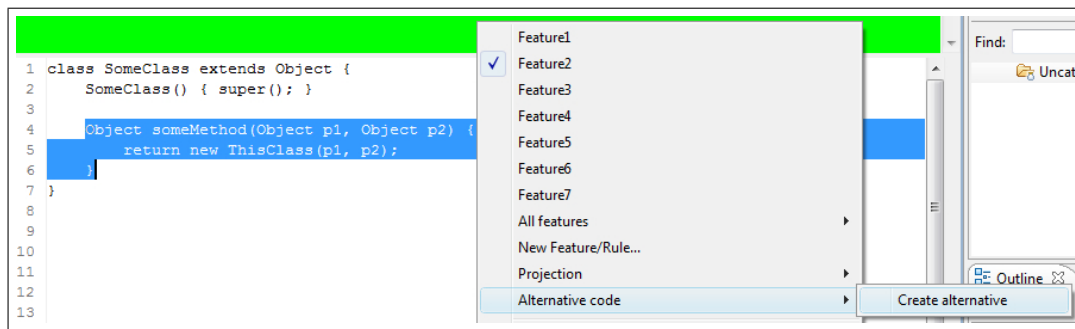


Abbildung 4.4: Anlegen einer Alternative zu einer Methodendeklaration

In einem Dialog wird dann der gewünschte Name der Alternative abgefragt (siehe Abbildung 4.5). Er soll dazu dienen, eine Alternative im weiteren Verlauf der Entwicklung einfacher identifizieren zu können. Er ist jedoch rein deklaratorisch und dient auf technischer Seite nicht dazu, eine Alternative eindeutig zu bestimmen. Für dieses Beispiel wählen wir den Namen *Alternative Feature3*.

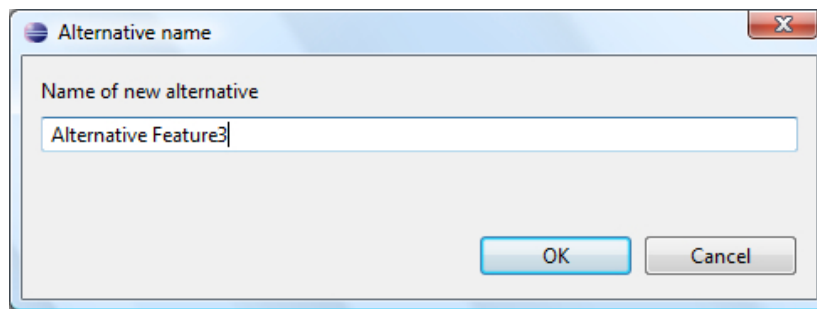


Abbildung 4.5: Eingabe eines Namens der Alternative

Nach der Eingabe eines Namens wird die bestehende Alternative aus dem Editor innerhalb einer XML-Datei als Alternative zu dem markierten AST-Knoten gespeichert. Außerdem wird eine neue Alternative angelegt und *aktiv* gesetzt sowie die alte Alternative deaktiviert. Die *aktive Alternative* ist immer diejenige Alternative, an der im Editor momentan gearbeitet wird. Somit wirken sich Editierungen und Annotationen im Editor immer nur auf die aktive Alternative aus. Der Quellcode der neu angelegten und aktiven Alternative entspricht anfänglich dem der zuletzt aktiven Alternative des AST-Knotens. Lediglich die Annotationen werden nicht übernommen.

Am linken Rand des Editors befindet sich nun ein Hinweis darauf, dass es Alternativen zu AST-Knoten in dieser Zeile gibt (siehe Abbildung 4.6). In einem Tooltip wird angezeigt, um welche Knoten des AST es sich genau handelt, denn eine Zeile kann aus mehreren AST-Knoten bestehen. Das Fehlersymbol weist darauf hin, dass die aktuelle Annotation (nämlich keine) dazu führt, dass sich beide Alternativen (noch) nicht gegenseitig ausschließen. Im Abschnitt 4.5 auf Seite 48 werden wir detaillierter auf diesen Sachverhalt eingehen.

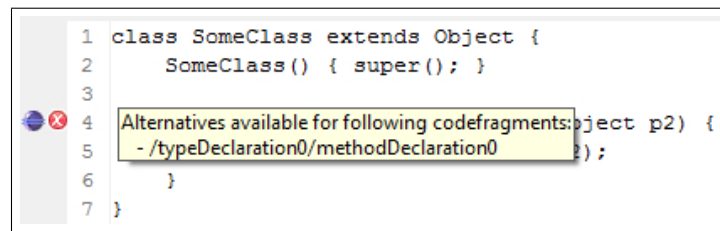


Abbildung 4.6: Die neu angelegte Alternative

Die neu angelegte Alternative zur Methode kann nun editiert werden. Außerdem muss sie noch mit dem entsprechenden Feature der Alternative annotiert werden, damit sich beide Alternativen tatsächlich gegenseitig ausschließen. Erst dann verschwindet auch das Fehlersymbol (siehe Abbildung 4.7).

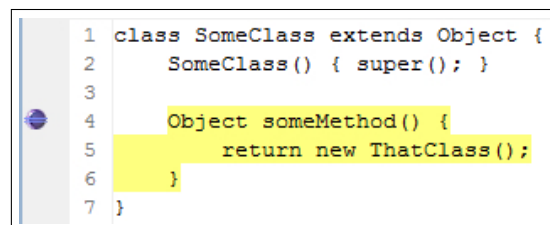


Abbildung 4.7: Die neu angelegte Alternative – editiert und annotiert

Es sind nun beide Alternativen zur Methode `someMethod` angelegt und korrekt annotiert. Ein Icon auf der linken Seite des Editors weist den Benutzer darauf hin, dass es Alternativen zu AST-Knoten in dieser Zeile gibt. Nun soll es auch die Möglichkeit geben, bestehende Alternativen zu editieren. Veränderungen im Quellcode und an den Annotationen können jedoch nur innerhalb des Editors an der aktiven Alternative vorgenommen werden. Daher gibt es die Möglichkeit, die aktive Alternative zu *wechseln*. Dazu markiert man wieder das entsprechende Codefragment und wählt nun im Kontextmenü den Menüpunkt *Switch to alternative* und im Anschluss die gewünschte Alternative aus, zu der gewechselt werden soll (siehe Abbildung 4.8). Dabei werden in diesem Menüpunkt gerade die Namen der Alternativen angezeigt, die der Benutzer angegeben hat. Einen Sonderfall stellt die Alternative *First alternative* dar. Dieser Name wurde von CIDE automatisch vergeben, als die Methode zum ersten Mal annotiert wurde<sup>6</sup>. Im weiteren Verlauf des Beispiels gehen wir davon aus, dass zur Alternative *First alternative* gewechselt wurde.

Beim Wechseln der aktiven Alternative wird der Quellcode und die Annotation ausgetauscht, wobei die gerade noch aktive Alternative in der XML-Datei gespeichert wird. Die somit aktivierte Alternative kann nun editiert werden. Außerdem kann man auch weitere Alternativen zur Methodendeklaration angeben.

<sup>6</sup> Damit Annotationen und Alternativen einfacher in einer gemeinsamen XML-Datei gespeichert werden können, wurde diese technische Vereinfachung gewählt. In einem nicht prototypischen SPL-Programm ließe sich dies selbstverständlich vermeiden.

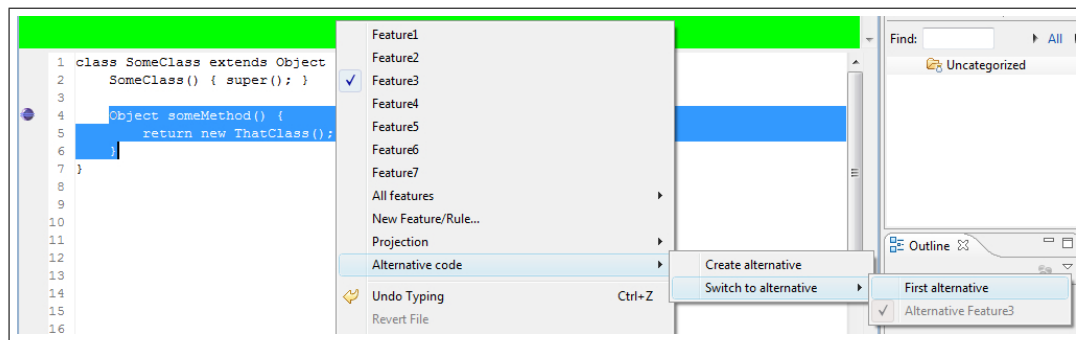


Abbildung 4.8: Wechseln der aktiven Alternative

### Verschachtelte Alternativen

Neben einer Editierung des Quellcodes und der Annotationen gibt es auch die Möglichkeit, *innerhalb* der aktiven Alternative eine Alternative zu einem Kindknoten des AST-Knotens der aktiven Alternative anzugeben. So könnte man nun zwei Alternativen zum ersten formalen Parameter der aktiven Alternative der Methode `someMethod` anlegen. Die Vorgehensweise ist die gleiche wie soeben beschrieben. Man hat also die Möglichkeit, Alternativen ineinander zu schachteln. Würde man nun wieder zur *Alternative Feature3* der Methode wechseln, würden keine Alternativen zum ersten formalen Parameter angeboten werden, da diese nur innerhalb der *First alternative* bestehen.

Schematisch ergibt sich eine Baumstruktur. Abbildung 4.9 zeigt die Struktur des bis jetzt entwickelten Beispiels. Hier erkennt man, dass es nur innerhalb der *First alternative* der Methode Alternativen zum ersten formalen Parameter gibt.

Innerhalb der Alternativen *First alternative* und *Alternative Feature6* des formalen Parameters gibt es im Schema das Feld *wasActive*. Es spielt eine Rolle, wenn die Vater-Alternative (in diesem Beispiel die *First alternative* der Methodendeklaration) zunächst nicht aktiv war und dann aktiviert wird. Der Quellcode einer Alternative wird nämlich als Text abgespeichert. Dieser Text enthält auch den Quellcode aller Kindknoten im AST, in unserem Beispiel insbesondere auch den Quellcode des ersten formalen Parameters. Es stellt sich also die Frage, welche Alternative dieses formalen Parameters in den Text der Alternative der Methode miteingeflossen ist. Diese Information ist wichtig, da Editierungen und Annotationen an diesem formalen Parameter ja auch in die korrekte Alternative einfließen muss. Ist der Wert des Feldes *wasActive* gleich *true*, bedeutet dies, dass der Text dieser Alternative in den Text der Vater-Alternative eingeflossen ist. Beim Aktivieren der *First alternative* der Methode wird in unserem Beispiel also auch zugleich die *First alternative* des ersten formalen Parameters aktiviert.

### Nicht-optionale Codefragmente

Das Anlegen von Alternativen zu nicht-optionalen Codefragmenten (z. B. Rückgabetypen von Methoden) erfolgt analog. Erster Schritt ist also, die erste Alternative innerhalb der Codebasis zu implementieren und entsprechend zu annotie-



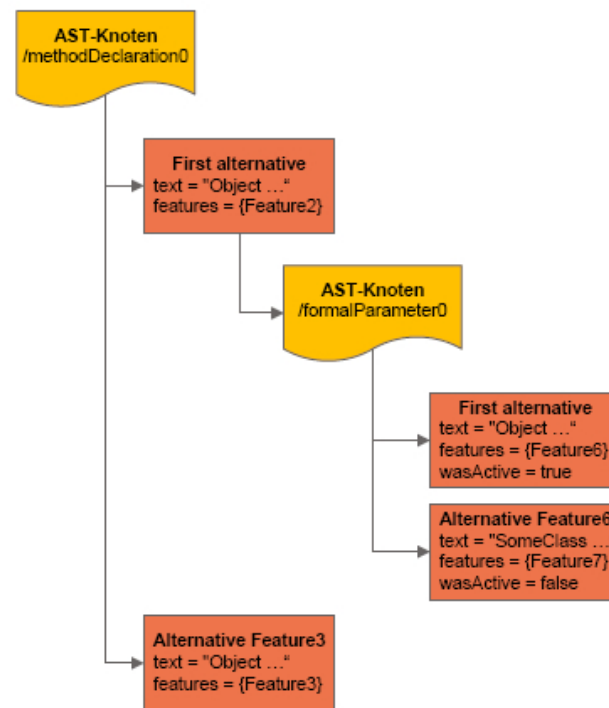


Abbildung 4.9: Alternative Features als Baumstruktur

ren. In CIDE war es bisher jedoch nicht möglich, nicht-optionale Codefragmente zu annotieren. Somit wurde gewährleistet, dass jede mögliche Variante der SPL syntaktisch korrekt ist (siehe Abschnitt 2.3.3 auf Seite 13). Im Zusammenhang mit unserem Mechanismus musste es aber erlaubt werden, auch nicht-optionale Codefragmente zu annotieren. Um weiterhin die syntaktische Korrektheit jeder Variante zu gewährleisten, wird überprüft, ob es gültige Varianten gibt, in der keine der angegebenen Alternativen vorhanden ist. Für das Anlegen einer Alternative bedeutet dies, dass direkt nach der Färbung des AST-Knotens der Fehler ausgegeben wird, dass es Varianten gibt, in denen der Knoten nicht vorhanden ist. Dieser Fehler verschwindet erst dann, wenn alle Alternativen korrekt angegeben wurden. Somit bleibt die syntaktische Korrektheit aller Varianten weiterhin gewährleistet. Weitere Ausführungen zur syntaktischen Korrektheit folgen in Abschnitt 4.5 auf Seite 48.

An dieser Stelle kommt der Unterschied zwischen den Features *Feature2* und *Feature3* sowie den Features *Feature6* und *Feature7* zum Tragen (siehe Abschnitt 4.4.1 auf Seite 43). Annotiert man beispielsweise zwei Alternativen einer *Methode* mit den Features *Feature6* bzw. *Feature7*, so ist jede Variante syntaktisch korrekt, da Methodendeklarationen optionale Sprachkonstrukte sind. Annotiert man so hingegen z. B. die Alternativen zu einem *Rückgabety*p einer Methode, wird weiterhin der Fehler ausgegeben, dass es syntaktisch inkorrekte Varianten gibt. Grund dafür ist, dass es laut Featuremodell auch Varianten geben kann, in denen beide Features nicht vorhanden sind. In diesen Varianten hätte die Methode kei-

nen Rückgabebetyp. Eine Annotation der Alternativen mit den Features *Feature2* bzw. *Feature3* wäre dagegen korrekt.

### Diskussion

Das Konzept der *aktiven Alternative* bringt den Vorteil mit sich, dass bei der Editierung und Annotation einer Alternative auf die gesamte Unterstützung durch die Entwicklungsumgebung zurückgegriffen werden kann. So stehen dem Entwickler bei der Entwicklung einer Alternative in Java z. B. Code-Vervollständigung und Syntax-Highlighting zur Verfügung.

Ein Nachteil ist, dass man den Überblick über die vorhandenen Alternativen nur schwierig behalten kann. Gerade bei verschachtelten Alternativen kann es passieren, dass sie übersehen werden. In unserem Beispiel sieht man die alternativen formalen Parameter der Methode z. B. gar nicht, wenn *Alternative Feature3* aktiv ist.

Dieser Nachteil lässt sich jedoch mit Hilfe der Entwicklungsumgebung umgehen. So wäre es einfach möglich, die Menge *aller* vorhandener Alternativen in einer speziellen View in Eclipse anzuzeigen. Verschachtelte Alternativen lassen sich beispielsweise als baumartige Struktur darstellen. Diese Baumstruktur verträgt sich auch besonders mit der XML-Struktur, in der Annotationen und Alternativen in CIDE abgespeichert werden. Im Rahmen der prototypischen Entwicklung des Mechanismus wurde jedoch eine solche View in CIDE vorerst nicht implementiert.

## 4.5 Überprüfung des gegenseitigen Ausschlusses aller Alternativen

Beim Anlegen einer Alternative mit dem Menüpunkt *Create alternative* wird nicht abgefragt, mit welchen Features diese Alternative annotiert sein soll. Vielmehr müssen diese Features in einem nächsten Schritt annotiert werden. Vorteil dieser Vorgehensweise ist, dass der in CIDE bereits existierende Mechanismus des *Färbens* wiederverwendet werden kann. Nachteil ist jedoch, dass *während* des Anlegens einer Alternative keine vollständigen Überprüfungen gemacht werden können, ob sich die Alternativen tatsächlich gegenseitig ausschließen. Diese Überprüfungen müssen nachträglich durchgeführt werden. Das bedeutet, dass Fehler vorliegen können, obwohl bereits Alternativen angelegt wurden.

Beim Anlegen einer Alternative mit dem Menüpunkt *Create alternative* gehen wir davon aus, dass die aktuell aktive Alternative bereits korrekt annotiert wurde. Wir nehmen also an, dass das markierte Codefragment mit mindestens einem *eigenen*, d. h. nicht vom Vaterknoten geerbten Feature, annotiert ist. Wäre dies nicht der Fall, so würde das Codefragment in jeder Variante vorhanden sein, in der der Vaterknoten vorhanden ist, so dass es im Kontext des Vaterknotens gar keine Alternativen geben kann. Somit wird der Menüpunkt *Create alternative* nur dann angeboten, wenn das markierte Codefragment mit mindestens einem eigenen Feature gefärbt ist. Dies ist jedoch offensichtlich keine vollständige Überprüfung

auf gegenseitigen Ausschluss der Alternativen, sondern nur eine Unterstützung des Entwicklers durch die Entwicklungsumgebung.

In CIDE war es bisher nicht notwendig, die syntaktische Korrektheit aller Varianten explizit zu überprüfen, da sie bereits dadurch gewährleistet war, dass nur optionale Codefragmente gefärbt werden können. Die syntaktische Korrektheit der Codebasis reichte also aus, damit alle Varianten syntaktisch korrekt sind. Wie bereits erwähnt, wurde es im Zusammenhang mit unserem Mechanismus notwendig, auch nicht-optionale Codefragmente annotieren zu können. Die syntaktische Korrektheit aller Varianten muss damit jedoch explizit überprüft werden.

Das Anlegen und die Editierung alternativer Codefragmente erfolgt im Kontext der Codebasis innerhalb des Editors. Daher werden alle Alternativen beim Anlegen oder Editieren stets durch den Parser der Programmiersprache überprüft. Dies ist ein weiterer Vorteil des Konzepts der *aktiven Alternative*. In CIDE werden für jede unterstützte Programmiersprache  $LL(k)$  Parser – Parser für spezielle *kontextfreie* Grammatiken – erzeugt [KAT<sup>+</sup>09]. Daher reicht das Parsen jeder Alternative beim Anlegen oder Editieren aus, um die syntaktische Korrektheit jeder gültigen Variante zu gewährleisten – bei nicht-optionalen Codefragmenten vorausgesetzt, dass sich die Alternativen tatsächlich gegenseitig ausschließen.

Die sowieso notwendige Überprüfung, ob sich die zu einem AST-Knoten angegebenen Alternativen gegenseitig ausschließen, gewährleistet also zusätzlich, dass jede gültige Variante syntaktisch korrekt ist. Bei optionalen Codefragmenten  $c$  muss dazu überprüft werden, ob in allen gültigen Varianten, in denen der Vaterknoten vorhanden ist, *höchstens eine* der Alternativen vorhanden ist. Formal muss überprüft werden, ob  $F \Rightarrow (annot(parent(c)) \Rightarrow mutExcl(alt(c)))$  eine Tautologie ist. Dabei bezeichne  $F$  die aussagenlogische Formel des Featuremodells,  $parent(c)$  den Vaterknoten des Codefragments  $c$  und  $alt(c)$  die Liste der Annotationen aller Alternativen von  $c$ . Außerdem sei  $mutExcl$  auf einer Liste von Wahrheitswerten definiert und gebe *true* zurück, falls die Anzahl der *true*-Werte in der Liste höchstens 1 ist, sonst *false*. Die Funktion  $mutExcl$  ähnelt also der Funktion  $XOR$ , mit dem Unterschied, dass bei der Eingabe  $[false, \dots, false]$  auch *true* zurückgegeben wird. Tabelle 4.1 veranschaulicht die Funktionen  $mutExcl$  und  $XOR$  anhand einer Wahrheitstafel für Eingabelisten  $l = [l_1, l_2, l_3]$  der Länge 3.

$l_1$	$l_2$	$l_3$	$mutExcl(l)$	$XOR(l)$
T	T	T	F	F
T	T	F	F	F
T	F	T	F	F
T	F	F	T	T
F	T	T	F	F
F	T	F	T	T
F	F	T	T	T
F	F	F	<b>T</b>	<b>F</b>

Tabelle 4.1: Wahrheitstafel für  $mutExcl$  und  $XOR$

Wie bereits erwähnt, muss bei nicht-optionalen Knoten überprüft werden, dass in allen gültigen Varianten, in denen der Vaterknoten vorhanden ist, *genau eine* der Alternativen vorhanden ist, formal, ob die aussagenlogische Formel  $F \Rightarrow (annot(parent(c)) \Rightarrow XOR(alt(c)))$  eine Tautologie ist. Die Überprüfung auf die Allgemeingültigkeit dieser und obiger Formel kann analog zu Abschnitt 3.3.2 auf Seite 23 mit einem SAT-Löser erfolgen.

Wir geben nun zwei Beispiele für solche Überprüfungen an. Dabei legen wir weiterhin das Featuremodell aus Abschnitt 4.4.1 auf Seite 43 zugrunde.

### Beispiel 1: Alternative Methoden

Zwei alternative Methodendeklarationen seien mit den Features *Feature6* bzw. *Feature7* annotiert. Wir gehen davon aus, dass die Klasse als ihr Vaterknoten nicht annotiert ist. Nun ist es bei Betrachtung des Featuremodells offensichtlich, dass in jeder gültigen Variante höchstens eine der beiden Alternativen vorhanden ist.

Setzt man  $c$  auf eine der beiden alternativen Methodendeklarationen, so ist  $parent(c)$  die Klasse, in der die Methode deklariert wurde. Ist außerdem  $alt(c) = [Feature6, Feature7]$ , so kann die Aussage auch mit oben angegebener Formel überprüft werden.

### Beispiel 2: Alternative Rückgabetypen

Dieses Beispiel gestalten wir etwas komplizierter. Zwei alternative Rückgabetypen  $c$  einer Methode seien mit den Features *Feature4* bzw. *Feature5* annotiert, d. h.  $alt(c) = [Feature4, Feature5]$ . Ohne weitere Annahmen folgt aus dem Featuremodell, dass es gültige Varianten geben kann, in denen beide Alternativen vorhanden sind, da die beiden Features zunächst als voneinander unabhängige, optionale Features definiert sind. Nehmen wir aber zusätzlich an, dass die Methode  $parent(c)$  mit *Feature1* annotiert ist (d. h.  $annot(parent(c)) = Feature1$  und  $alt(c) = [Feature1 \wedge Feature6, Feature1 \wedge Feature7]$ ), so folgt aus dem Featuremodell, dass die Features *Feature4* und *Feature5* in einer *XOR*-Beziehung stehen. Da die Rückgabetypen nur dann existieren, wenn auch die Methode existiert, und der gegenseitige Ausschluss der Alternativen nur dann gefordert ist, wenn die Methode vorhanden ist, folgt somit, dass die Überprüfung positiv verläuft.

Würde im Featuremodell anstatt  $Feature1 \Rightarrow Feature4 \text{ XOR } Feature5$  die Formel  $Feature1 \Rightarrow Feature4 \text{ mutExcl } Feature5$  stehen, so würde die Überprüfung einen Fehler ausgeben, da die Variante, in der weder *Feature4* noch *Feature5* vorhanden ist, gültig wäre. Dies ist jedoch nur bei nicht-optionalen Codefragmenten ein Fehler. Wären die Methoden aus Beispiel 1 so annotiert, wäre die Überprüfung erfolgreich.

## 4.6 Typsystem

Das CFJ-Typsystem aus Abschnitt 3.3.5 auf Seite 26 müssen wir nun so erweitern, dass es zusätzlich auch Alternativen berücksichtigt. In Abschnitt 4.1 auf Seite 37 haben wir bereits angesprochen, dass es ohne Fallstudie schwierig ist, zu bewerten, bei welcher Art von AST-Knoten Alternativen sinnvoll sind. Daher haben wir uns anfänglich auf den Standpunkt gestellt, dass es grundsätzlich möglich sein soll, zu jeder Art von AST-Knoten Alternativen angeben zu können. Viele Arten von AST-Knoten werden wir bei der Erweiterung des CFJ-Typsystems um Alternativen jedoch bereits sehr früh ausschließen können, so z. B. alternative Bezeichner von Klassen, Methoden usw. Andere AST-Knoten werden ausgeschlossen, da wir feststellen werden, dass durch entsprechende Berücksichtigung das Typsystem zu komplex und kompliziert werden würde, was die Verwendung alternativer Features für den Programmierer stark erschweren würde.

In einem ersten Schritt werden wir also diskutieren, zu welchen AST-Knoten Alternativen angegeben werden können sollen. Dabei werden wir insbesondere den Einfluss auf das zu definierende Typsystem berücksichtigen und letztendlich feststellen, dass es bei den meisten Arten von AST-Knoten zu Problemen führt, falls sie in mehreren Alternativen vorliegen können. Die hauptsächlichen Probleme sind

- hohe Komplexität des Typsystems, d. h. laufzeitintensive Typprüfungen,
- daraus resultierende umfangreiche und unübersichtliche Fehlermeldungen für den Programmierer und
- Schwierigkeiten für den Programmierer, einen Überblick über alle Alternativen und deren Zusammenhänge zu behalten.

Nachdem wir die AST-Knoten festgelegt haben, zu denen es Alternativen geben kann, werden wir das CFJ-Typsystem entsprechend erweitern, so dass weiterhin gewährleistet ist, dass jede Variante ein gültiges FJ-Programm ist.

### 4.6.1 Ausschluss von AST-Knoten für Alternativen

Wir betrachten nun einzelne Arten von AST-Knoten und diskutieren, ob es sinnvoll ist, Alternativen zu ermöglichen, und welchen Einfluss dies auf das Typsystem hat. Dabei wird insbesondere berücksichtigt, dass der Programmierer einen Überblick behalten kann und dass das Typsystem nicht zu komplex wird.

#### Alternative Bezeichner

Bei Bezeichnen, also z. B. Namen von Klassen, Methoden, Feldern usw., macht es keinen Sinn, Alternativen zu ermöglichen, da Namen zur Identifikation des entsprechenden Sprachkonstrukts dienen. So sollte z. B. eine Methode in zwei Varianten nicht mit unterschiedlichen Namen vorliegen können. Auch erreicht man

dadurch keinen höheren Grad an Flexibilität, da die unterschiedliche Bezeichnung auch bei der Verwendung des Sprachkonstrukts, also z. B. beim Aufruf der Methode, zusätzlich berücksichtigt werden muss.

### Alternative Superklassen

Grundsätzlich könnte man erlauben, dass eine Klasse in verschiedenen Varianten unterschiedliche Superklassen hat, indem man Alternativen zum Codefragment nach dem Schlüsselwort **extends** ermöglicht (Nichtterminal `ExtendedType` in der Grammatik). Dies führt jedoch dazu, dass die Vererbungshierarchie von der Featureauswahl in einer Variante abhängig ist. Diese Tatsache beeinflusst viele Bereiche des CFJ-Typsystems. So müsste bei der Überprüfung eines Feldzugriffs möglicherweise nicht nur eine Vererbungskette nach diesem Feld durchsucht werden, sondern mehrere. Gleiches gilt für die Suche nach einer Methode bei der Überprüfung eines Methodenaufrufs. Dies führt dazu, dass eine Methode, auf die zugegriffen wird, je nach Annotation und je nach Featureauswahl an den verschiedensten Stellen im Quellcode stehen kann. Für einen Programmierer würde es sehr schwierig sein, all diese Stellen zu identifizieren und für all diese Vorkommen zu gewährleisten, dass der Aufruf auch korrekt ist.

Zusätzlich zu dieser Unübersichtlichkeit ist anzumerken, dass eine solche nicht-statische Vererbungshierarchie zu einer kombinatorischen Explosion der durchzuführenden Typprüfungen führen kann. Gibt es zu einer Klasse  $n$  alternative Superklassen, so kann es Fälle geben (z. B. wenn ein zu überprüfender Methodenaufruf gar nicht annotiert, also in allen Varianten vorhanden ist), in denen all diese  $n$  Superklassen durchsucht werden müssen. In diesen Superklassen kann es jedoch wiederum vorkommen, dass ihre Supertypen in mehreren Alternativen vorliegen, so dass auch hier wieder mehrere Äste des Vererbungsbaums durchsucht werden müssen.

Zusammenfassend werden wir keine alternativen Superklassen ermöglichen, so dass die Vererbungshierarchie, in der sich eine Klasse befindet, in jeder Variante gleich ist.

### Alternative Klassen

Liegt eine Klasse in mehreren Alternativen vor, so müssen bei jeder Verwendung (z. B. Instanziierung eines Objekts, Deklaration eines Feldes mit der Klasse als Typ) alle Alternativen, die gleichzeitig mit ihrer Verwendung vorhanden sein können, berücksichtigt werden. Dies stellt grundsätzlich kein Problem dar, da dies die inhärente Komplexität von Alternativen ist, die man prinzipiell in Kauf nehmen muss, wenn alternative Features vorliegen.

Jedoch gibt es viele verschiedene Stellen im Quellcode, an denen Klassen verwendet werden können. So wird eine Klasse z. B. bei der Deklaration von Feldern, bei der Definition formaler Methodenparameter oder bei der Instanziierung von Objekten verwendet. Können diese Stellen wiederum in mehreren Alternativen vorliegen, so droht prinzipiell eine kombinatorische Explosion. Grundsätzlich müssen bei jeder Verwendung dieser Klasse all ihre Alternativen berücksichtigt

werden. Liegen die Stellen, an denen die Klasse verwendet wird, zusätzlich in mehreren Alternativen vor, müssen theoretisch alle möglichen Kombinationen der Alternativen überprüft werden. Zur Veranschaulichung betrachte man Abbildung 4.10. Dort liegt ein Objekterzeugungsterm in mehreren Alternativen vor. Jede dieser Alternativen erzeugt unterschiedliche Instanzen. Die entsprechenden Klassen liegen auch jeweils in mehreren Alternativen vor. Somit müssen bei der Typprüfung des Objekterzeugungsterms (z. B. Überprüfung der übergebenen Parameter) viele verschiedene Kombinationen von Alternativen überprüft werden.

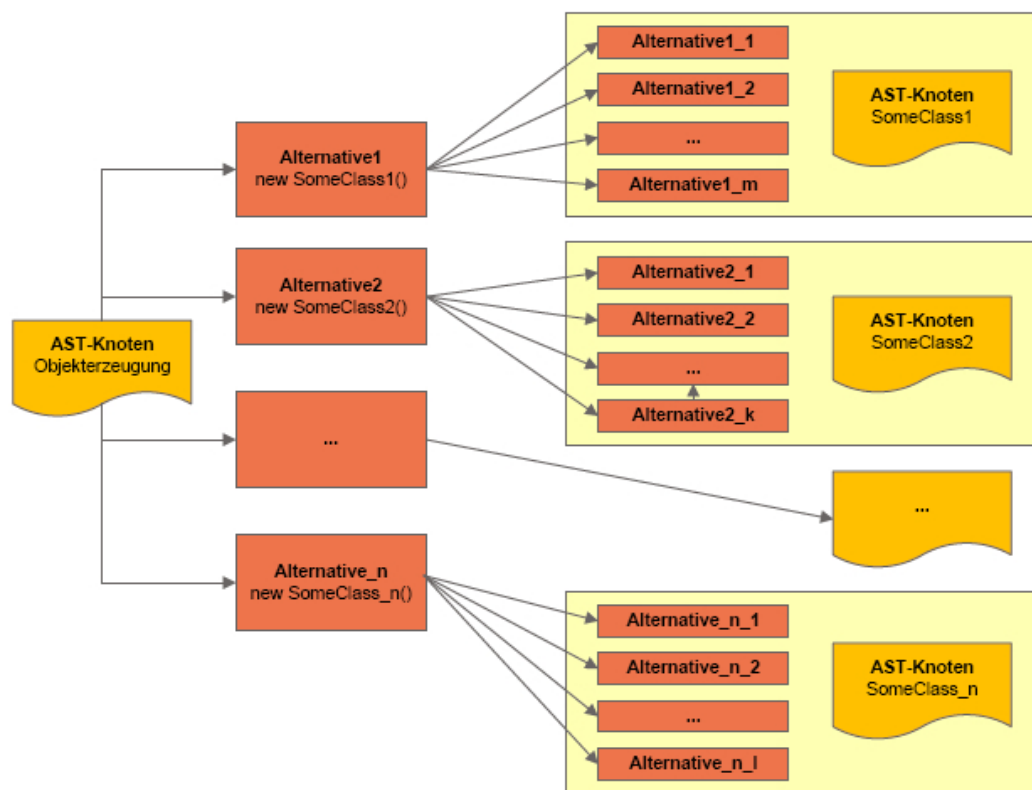


Abbildung 4.10: Beispiel für kombinatorische Explosion der Typprüfungen

Neben der Tatsache, dass all diese Kombinationen auch vom Programmierer überblickt werden müssen und dass die Typprüfungen sehr komplex werden können, können die Fehlermeldungen sehr lang und unübersichtlich werden, da der gesamte Pfad von Alternativen ausgegeben werden muss. Bei ersten Versuchen, alternative Klassen, Methoden und Felder trotzdem im Typsystem zu berücksichtigen, wurden z. B. Fehlermeldungen folgender Art erzeugt: „Alternative2 der Klasse SomeClass: Alternative3 der Methode someMethod: Typ von Alternative2 des Feldes someField in Alternative1 der Klasse AnotherClass ist kein Subtyp des Rückgabetyps.“

Aus diesen Gründen werden wir alternative Klassendeklarationen nicht ermöglichen.

### Alternative Methoden

Alternative Methoden bringen das Problem mit sich, dass die bereits in CFJ sehr komplizierte Suche nach einer aufgerufenen Methode und die Überprüfung auf gültiges Overriding zusätzlich erschwert wird. Alternative Methoden können nämlich unterschiedliche Rückgabetypen und unterschiedliche formale Parameter haben. In CFJ hatten wir das Problem, dass annotierte Methoden in manchen Varianten nicht vorhanden sind, so dass die Override-Relation abhängig von der Annotierung ist. Bei alternativen Methoden kommt zusätzlich das Problem hinzu, dass die Methode mit unterschiedlichen Signaturen vorhanden sein kann. Dies erscheint uns sowohl für den Programmierer, als auch für das Typsystem eine zu hohe Komplexität zu sein.

Hinzu kommt, dass sich unterschiedliche formale Parameter einer Methode z. B. in Java durch Overloading einfach realisieren lassen. Overloading wurde in FJ nur deshalb verboten, um die Suche nach einer aufgerufenen Methode zu erleichtern. Im Falle alternativer Methoden wird durch diese Restriktion die Überprüfung auf gültiges Overriding jedoch deutlich erschwert, so dass wir alternative Methoden nicht ermöglichen.

Alternative Methoden würden es ermöglichen, zu einer Methode verschiedene Rückgabetypen anzugeben. Dies ist z. B. in Java auch durch Overloading nicht möglich, wenn die formalen Parameter identisch sind. Diesen Sachverhalt werden wir später im Zusammenhang mit alternativen Rückgabetypen diskutieren.

### Einschub: Grundlegende Bestimmung der Typen von alternativen Termen

Für folgende Diskussionen müssen wir vorab betrachten, wie im Typsystem alternative Terme gehandhabt werden können. Wir diskutieren zwei Möglichkeiten, die Typen von alternativen Termen zu bestimmen:

1. Der Typ des Terms ist der *kleinste gemeinsame Supertyp*  $T$  der Typen  $T_1, \dots, T_n$  aller Alternativen. Dabei ist  $T$  der eindeutig bestimmte Typ mit folgenden Eigenschaften:
  - a)  $T <: T_1, \dots, T <: T_n$
  - b) Für alle Typen  $S$  mit  $S <: T_1, \dots, S <: T_n$  gilt  $T <: S$

Algorithmisch ermittelt man  $T$ , indem man ausgehend von  $T_1$ <sup>7</sup> so lange die Vererbungshierarchie nach oben steigt, bis der betrachtete Typ Supertyp aller  $T_i$  ist.

Vorteil dieser Vorgehensweise ist, dass sich bei alternativen Termen die Komplexität wieder auf einen Typ reduziert und nicht mit jedem einzelnen Typ der Alternativen weitergearbeitet werden muss. Von Nachteil ist die Tatsache, dass die Typen der einzelnen Alternativen verloren gehen. Wir gehen jedoch davon aus, dass alternative Terme in der Praxis primär

---

<sup>7</sup> Man kann mit einem beliebigen  $T_i$  anfangen.



dazu verwendet werden, auf unterschiedliche Methoden, Felder usw. zuzugreifen, und nicht dazu, Objekte unterschiedlichen Typs zurückzugeben<sup>8</sup>. Für weitere Diskussionen des Typsystems werden wir also diese Variante wählen.

2. Eine weitere Möglichkeit bestünde darin, für jede Alternative des Terms die Typregeln separat zu überprüfen. Der Typ des Terms muss also für alle Alternativen zum Kontext (z. B. ein Methodenaufruf, dem der Term als Parameter übergeben wird) passen. Mit dieser Vorgehensweise kann man z. B. folgenden Codeausschnitt realisieren. Man betrachte den Term `new C().f`, wobei `f` ein Feld der Klasse `C` sei. Sei nun `D` eine Klasse, die in keiner Subtypbeziehung zu `C` steht. Dies bedeutet, dass der kleinste gemeinsame Supertyp `Object` ist. Gibt man den Term `new D()` als Alternative zu `new C()` an und enthält `D` auch das Feld `f`, so wird dies durch das Typsystem akzeptiert.

Der erste Ansatz hingegen ermittelt als Typ des Objekterzeugungs-Terms den kleinsten gemeinsamen Supertyp von `new C()` und `new D()`, in diesem Beispiel also `Object`. Da `Object` das Feld `f` nicht enthält, würde ein Typfehler ausgegeben werden.

Der Vorteil dieser Vorgehensweise besteht also darin, dass Gemeinsamkeiten von eigentlich voneinander unabhängigen Klassen (in diesem Beispiel das Feld `f`) genutzt werden können. Davon abgesehen, dass wir diesen Nutzen in der Praxis nicht hoch einschätzen, bringt es den Nachteil mit sich, dass die Typprüfungen sehr komplex werden können. Analog zur bisherigen Diskussion droht hier die Gefahr einer kombinatorischen Explosion. Die Ermittlung des kleinsten gemeinsamen Supertyps umgeht diese Komplexität. Hinzu kommt, dass das Beispiel auch mit der ersten Vorgehensweise realisiert werden kann, indem zwei alternative Terme `new C().f` und `new D().f` angelegt werden.

### Alternative Felder und formale Parameter

Bei alternativen Feldern macht die Bedingung Sinn, dass die Bezeichnung des Feldes in jeder Alternative gleich sein muss. Die Begründung erfolgt analog zum Verbot alternativer Bezeichner. Alternative Felder unterscheiden sich also nur in ihrem Typ. Ein Feld einer Klasse wird nur in einem Feldzugriffs-Term verwendet, dessen Typ nach der ersten Vorgehensweise der kleinste gemeinsame Supertyp aller Alternativen des Feldes ist. Somit geht die Information über die Typen der einzelnen Alternativen verloren, so dass an dieser Stelle ein Feld ohne Alternativen, dessen Typ der kleinste gemeinsame Supertyp ist, verwendet werden könnte.

Zusammenfassend erreicht man durch Verwendung alternativer Felder keine erhöhte Flexibilität. Analog kann man mit alternativen formalen Methodenparametern argumentieren, die wir auch nicht ermöglichen. Auch hier liegt der Unterschied der Alternativen lediglich im Typ des Parameters.

---

<sup>8</sup> Dazu gäbe es in Java nämlich z. B. die Möglichkeit, Generics zu verwenden, worauf hier aber nicht im Detail eingegangen werden soll.

### Alternative Rückgabetypen

Liegt ein Rückgabetyt in mehreren Alternativen vor, so verwenden wir auch hier den kleinsten gemeinsamen Supertyp aller Alternativen. Rückgabetypen spielen im Typsystem bei der Überprüfung einer Methode und eines Methodenaufrufs eine Rolle. In einer Methode muss der Typ des Terms ein Subtyp des Rückgabetyps sein. Bei Verwenden des kleinsten gemeinsamen Supertyps aller Alternativen des Rückgabetyps gewinnen wir in diesem Fall nicht an Ausdrucksstärke, da ein einziger Rückgabetyt in Form des kleinsten gemeinsamen Supertyps aller Alternativen das gleiche Ergebnis liefert. Bei einem Methodenaufruf muss der Rückgabetyt zum Typ des Kontexts des Methodenaufrufs passen. Ist der Methodenaufruf z. B. ein Parameter eines Konstruktors, so muss der Rückgabetyt ein Subtyp des entsprechenden Feldes sein. Auch hier hat man durch Verwenden alternativer Rückgabetypen keinen Vorteil, wenn der kleinste gemeinsame Supertyp verwendet wird, so dass wir alternative Rückgabetypen nicht ermöglichen.

### Alternative Terme

Abschließend bleiben nur noch Terme übrig, die in mehreren Alternativen vorliegen können. Wir werden sehen, dass die Verwendung alternativer Terme zu keiner kombinatorischen Explosion führt.

#### 4.6.2 Erweiterung der CFJ-Typregeln

Dadurch, dass wir nur noch alternative Terme erlauben, müssen wir nur wenige Anpassungen im bestehenden CFJ-Typsystem vornehmen. Es ändern sich lediglich die Regeln zur Bestimmung des Typs eines Terms, die in der Regel (M.5) verwendet werden. Diese Veränderung können wir unabhängig von der konkreten Gestalt des Terms (Variablenzugriff, Feldzugriff, Methodenaufruf, Objekterzeugung oder Cast) formulieren. Zur Ermittlung des Typs eines Terms  $t$  müssen wir all diejenigen Alternativen  $t_1, \dots, t_n$  von  $t$  ermitteln, deren Eltern-Alternative mit der Eltern-Alternative von  $t$  übereinstimmt (verschachtelte Alternativen). Der Typ des Terms ist dann der kleinste gemeinsame Supertyp der Typen von  $t_1, \dots, t_n$ . Die Typen von  $t_1, \dots, t_n$  werden wie in den CFJ-Typregeln bestimmt, wobei die Typen von Subtermen (z. B. Terme, die einem Methodenaufruf als Parameter übergeben werden) analog zu diesen Ausführungen bestimmt werden, da sie wiederum in mehreren Alternativen, mit  $t$  als Eltern-Alternative, vorliegen können.

#### 4.6.3 Rechtfertigung des Mechanismus für alternative Features

Die bisherige Diskussion hat ergeben, dass die meisten Arten von AST-Knoten in FJ nicht dazu geeignet sind, in Alternativen vorzuliegen. Diese Erkenntnisse stammen zum Großteil nicht aus Erfahrungen in der Praxis, sondern aus der Tatsache, dass der Programmierer sehr schnell den Überblick verlieren kann und

dass die Typregeln sehr komplex werden können und grundsätzlich eine kombinatorische Explosion droht. Auch wenn die Komplexität prinzipiell alternativen Features inhärent ist, wird sie zum Problem, wenn sich Codefragmente, die jeweils in mehreren Alternativen vorliegen, gegenseitig referenzieren.

Letztendlich bleiben nur noch Terme übrig, zu denen Alternativen angegeben werden können. Zur Vermeidung einer kombinatorischen Explosion verwenden wir den kleinsten gemeinsamen Supertyp aller Alternativen als Typ des Terms. Somit schränken wir wohl die Möglichkeiten ein, die alternative Codefragmente grundsätzlich bieten, ermöglichen es aber dennoch, beispielsweise unterschiedliche Methoden oder Felder aufzurufen.

Wie in dieser Arbeit bereits angesprochen, gibt es in Java die Möglichkeit, alternative Rückgabe-Statements durch einen Trick mit `if (true)` Anweisungen zu realisieren. Diese Idee hatten wir zuvor u. a. mit dem Argument abgelehnt, dass wir auch zu anderen Arten von Codefragmenten Alternativen angeben können möchten. In unserem Fall von FJ ist dies nun doch nicht der Fall. Es liegt die Frage nahe, warum ein recht komplexer Mechanismus zur Verwaltung alternativer Features entwickelt wurde, um nur alternative Terme zu ermöglichen. Die Tatsache, dass der `if (true)` Trick in FJ nicht möglich ist, spielt keine große Rolle, da FJ keine für die Praxis relevante Programmiersprache ist. Für Java hätte man auf einfachere Art und Weise einen Mechanismus entwickeln können, der den `if (true)` Trick vor dem Benutzer versteckt und stattdessen einen Hinweis auf alternative Features anzeigt.

Zur Rechtfertigung unseres Mechanismus zur Verwaltung alternativer Features sei betont, dass er unabhängig von der zugrunde liegenden Programmiersprache ist. Man kann zu jeder in CIDE vorhandenen Sprache alternative Codefragmente verwalten. Die Sprache FJ wurde herangezogen, um zu untersuchen, welchen Einfluss CIDE-Annotationen und alternative Features auf das Typsystem haben. Dabei konnten Annotationen auf einfache Art und Weise zusätzlich berücksichtigt werden. Im Zusammenhang mit alternativen Features haben wir festgestellt, dass Alternativen beliebiger Codefragmente sehr schnell zu einer kombinatorischen Explosion führen können. Nichtsdestotrotz kann unser Mechanismus für alternative Terme verwendet werden.

Außerdem bleibt die Möglichkeit bestehen, andere Programmiersprachen hinsichtlich alternativer Features zu untersuchen und ein entsprechendes Typsystem zu implementieren. Auch Anpassungen oder Erweiterungen unseres CFJ-Typsensystems mit Alternativen können vorgenommen werden, z. B. wenn eine kombinatorische Explosion in Kauf genommen werden kann. Diese Arbeit soll grundsätzlich aufzeigen, wie im Typsystem mit alternativen Features umgegangen werden kann und welche Probleme sich dabei ergeben.



---

---

# KAPITEL 5

---

## Zusammenfassung und Ausblick

Abschließend sollen die Ergebnisse dieser Diplomarbeit zusammengefasst werden. Außerdem wird diskutiert, wie man auf diese Arbeit aufsetzen und die Erkenntnisse auf praxisrelevante Programmiersprachen wie Java erweitern kann. Erste Schritte in diese Richtung wurden im Rahmen dieser Arbeit bereits gegangen, wobei sich dabei eine technische Hürde ergeben hat, die hier auch angesprochen werden soll.

### 5.1 Zusammenfassung

Zur Entwicklung von Software-Produktlinien wurden bereits viele Ansätze und Technologien erforscht. Grundlage dieser Diplomarbeit war das Eclipse-Plugin CIDE, das einen Annotationsansatz implementiert, der am ehesten `#ifdef` Präprozessor-Direktiven ähnelt. Bisherige Arbeiten zu CIDE behandelten die Gewährleistung der syntaktischen Korrektheit aller Varianten, Granularität der Annotationen und Sprachunabhängigkeit. Bislang wurde jedoch nicht gewährleistet, dass jede mögliche Variante auch ein sprachsemantisch korrektes Programm der zugrunde liegenden Programmiersprache ist. Diese Diplomarbeit hat daher ein Typsystem entwickelt, das für ein CFJ-Programm – ein Programm in der Sprache FJ, das mit CIDE-Annotationen versehen ist – gewährleistet, dass jede gültige Variante ein korrektes FJ-Programm im Sinne des FJ-Typsystems ist. Dabei wurde eine Grundsatzentscheidung berücksichtigt, die fordert, dass jedes gültige CFJ-Programm unter Vernachlässigung der Annotationen auch ein gültiges FJ-Programm sein muss. Hintergrund dieser Forderung ist das Bestreben, bestehende Typsysteme für die zugrunde liegenden Programmiersprachen (z. B. FJ oder Java) wiederverwenden zu können, indem man sie nur echt erweitern muss, um Annotationen zusätzlich zu berücksichtigen.

Im Kontext des so entwickelten CFJ-Typsystems ist es nicht ohne Workaround möglich, alternative Features zu implementieren. Unter alternativen Features versteht man Features, die sich gegenseitig ausschließen. Dies bedeutet, dass in jeder Variante höchstens oder genau eines dieser Features vorhanden sein darf. Ein einfaches Nebeneinander- oder Untereinanderschreiben alternativer Codefragmente ist entweder syntaktisch (bei nicht-optionalen Codefragmenten) oder sprachsemantisch (bei optionalen Codefragmenten) nicht möglich. Aus diesem Grund wurde ein Mechanismus entwickelt, mit dem alternative Codefragmente angelegt und editiert werden können. Dieser Mechanismus speichert alternative Codefragmente

außerhalb des Basis-Quellcodes in einer XML-Struktur zusammen mit den Annotationen ab. Mit Hilfe dieser Baumstruktur ist es somit sogar möglich, Alternativen ineinander zu schachteln. Das Anlegen und die Editierung von Alternativen erfolgt weiterhin innerhalb des Editors im Kontext der Codebasis, so dass eine bestehende Unterstützung durch die Entwicklungsumgebung weiterhin genutzt werden kann. Dies erfolgt mit Hilfe des Konzepts der *aktiven Alternative*, das den vorteilhaften Nebeneffekt hat, dass die syntaktische Korrektheit einer Alternative automatisch überprüft wird.

Der in dieser Diplomarbeit entwickelte Mechanismus zur Verwaltung alternativer Features ist unabhängig von der zugrunde liegenden Sprache, da er – analog zu CIDE-Annotationen – auf Knoten des abstrakten Syntaxbaums (AST) basiert. Es ist jedoch nicht offensichtlich, zu welchen Arten von AST-Knoten Alternativen tatsächlich ermöglicht werden sollen. So macht es gewiss keinen Sinn, zu dem Namen einer Klasse Alternativen angeben zu können. Nichtsdestotrotz kann ohne geeignete Fallstudien keine genaue Aussage darüber gemacht werden, welche AST-Knoten tatsächlich in der Praxis sinnvoll sind.

Das CFJ-Typsystem musste nun so erweitert werden, dass es zusätzlich alternative Codefragmente berücksichtigt. Dabei wurde zunächst davon ausgegangen, dass man – abgesehen von den offensichtlich unsinnigen Fällen wie Bezeichnen – zu jeder Art von AST-Knoten Alternativen angeben kann. Bei der Erweiterung des Typsystems haben sich jedoch schnell Probleme ergeben, die v. a. darin bestanden, dass die Komplexität der Typprüfungen und Fehlermeldungen extrem anstieg. Die Problematik besteht nicht direkt in der inhärenten Komplexität von Alternativen, sondern darin, dass bei gegenseitiger Referenzierung alternativer Codefragmente eine kombinatorische Explosion droht. Neben der Tatsache, dass dies die Entwicklung des Typsystems stark erschwert, ist es dem Programmierer nicht mehr einfach möglich, den Überblick zu behalten und Fehlermeldungen einfach zu interpretieren.

Eine genauere Analyse der verschiedenen Arten von AST-Knoten in FJ hat ergeben, dass es lediglich bei Termen sinnvoll ist, Alternativen zu ermöglichen. Bei einem Term, der in mehreren Alternativen vorliegt, hat man zusätzlich die Möglichkeit, als Typ dieses Terms den *kleinsten gemeinsamen Supertyp* aller Alternativen zu verwenden. Dadurch wird die Komplexität der Alternativen auf einen Typ reduziert. Eine Erweiterung des CFJ-Typsystems war unter diesen Voraussetzungen einfach möglich.

## 5.2 Ausblick

Die vorliegenden Implementierungen innerhalb von CIDE haben sich primär auf den Mechanismus an sich konzentriert. Daher gibt es noch viele Möglichkeiten, die Benutzung des Plugins zu verbessern. So wäre eine Übersicht aller vorhandener Alternativen zusammen mit deren Verschachtelung und Annotierungen sinnvoll. Diese könnte innerhalb der in Eclipse vorhandenen *Outline-View* – eine baumartige Darstellung des Quelltextes – erfolgen, da die Alternativen selbst auch eine

baumartige Struktur darstellen. Auch gibt es zur Zeit noch keine Möglichkeit, angelegte Alternativen umzubenennen oder zu löschen.

FJ ist als rein funktionale Teilmenge keine für die Praxis relevante Programmiersprache. Sie diene in dieser Arbeit lediglich zur einfacheren Analyse der Sachverhalte, die untersucht wurden. Nachdem erste Erkenntnisse zur Berücksichtigung von Annotationen und Alternativen in Typsystemen gesammelt werden konnten, bietet sich als nächster Schritt die Erweiterung auf Java an. Somit wären auch erste Fallstudien möglich, mit denen man die Möglichkeiten, den Bedarf sowie Vor- und Nachteile alternativer Features untersuchen könnte.

Die in dieser Arbeit aufgestellte Grundsatzentscheidung ermöglicht, dass bestehende Java-Typsysteme, wie sie z. B. die Eclipse-Bibliothek JDT<sup>1</sup> anbietet, wiederverwendet werden können. Um Annotationen und Alternativen zusätzlich zu berücksichtigen, muss ein bestehendes Typsystem echt erweitert werden, ohne bereits existierende Regeln abändern zu müssen. Nichtsdestotrotz muss es die Möglichkeit geben, nicht nur eine komplette Klasse, sondern auch einzelne Codefragmente vom bestehenden Typsystem überprüfen zu lassen. Möchte man beispielsweise in Java alternative **return**-Statements ermöglichen, so möchte man z. B. die bestehende Auflösung von Methodenzugriffen wiederverwenden können, da sie durch Overriding und Overloading recht komplex ist. Hierfür stehen in JDT sog. *Bindings* zur Verfügung, die einen Methodenaufruf und eine Methodendeklaration miteinander *verbinden*. Solche Bindings werden von JDT erstellt, wenn man eine ganze Klasse vom Typsystem analysieren lässt. Bei alternativen **return**-Statements besteht jedoch die Notwendigkeit, einzelne **return**-Statements durch das Typsystem analysieren zu lassen. Diese Analyse muss wohl im Kontext der ganzen Klasse erfolgen, jedoch sollten beispielsweise Methoden der Klasse, die mit dem **return**-Statement in keinerlei Zusammenhang stehen, nicht überprüft werden, da sonst eine Analyse jeder Alternative zu einer erneuten Analyse der gesamten Klasse führt. Es ist in JDT wohl möglich, ein einzelnes Codefragment (z. B. ein einzelnes **return**-Statement) syntaktisch zu überprüfen. Jedoch gibt es bislang keine Möglichkeit, die benötigten Bindings erstellen zu lassen, ohne die gesamte Klasse typzuprüfen.

Auf die Erfahrungen dieser Diplomarbeit können weitere Arbeiten aufsetzen. Die Tatsache, dass JDT nach aktuellen Erkenntnissen nicht dazu geeignet ist, das Java-Typsystem zu erweitern, bedeutet nicht, dass dies prinzipiell unmöglich ist. Möglicherweise gibt es Wege, auf die Erstellung der Bindings innerhalb des Bibliothek-Quellcodes Einfluss zu nehmen. Auch ist man nicht unbedingt an Eclipse-Bibliotheken gebunden, so dass die Suche nach anderen Java-Typsystemen sinnvoll sein kann.

Der in dieser Diplomarbeit entworfene Mechanismus zur Verwaltung alternativer Features kann für Fallstudien verwendet werden, um zu erforschen, bei welchen Arten von AST-Knoten Alternativen in der Praxis Anwendung finden. Solche Fallstudien können auch zur Analyse der Relevanz und Vorkommen alternativer Features in einer SPL herangezogen werden.

---

<sup>1</sup> [www.eclipse.org/jdt](http://www.eclipse.org/jdt)

Zusammenfassend wurden mit dem in dieser Arbeit entwickelten Mechanismus zur Verwaltung alternativer Codefragmente die Grundlagen für die Verwendung alternativer Features in CIDE geschaffen. Eine Erweiterung bestehender Typsysteme zur Berücksichtigung von Annotationen und Alternativen ist nicht nur in technischer Hinsicht eine anspruchsvolle Aufgabe. Die Komplexität der Typprüfungen kann sehr schnell ansteigen, falls man dem Programmierer zu viel Flexibilität ermöglicht. Diese Flexibilität kann durch eine bedachte Auswahl der AST-Knoten, die in Alternativen vorliegen können, gesteuert werden.



---

## Literaturverzeichnis

---

- [AK09] APEL, Sven ; KÄSTNER, Christian: An Overview of Feature-Oriented Software Development. In: *Journal of Object Technology (JOT)* 8 (2009), Juli/August, Nr. 5, S. 49–84
- [AKGL09] APEL, Sven ; KÄSTNER, Christian ; GRÖSSLINGER, Armin ; LENGAUER, Christian: Feature (De)composition in Functional Programming. In: *Proceedings of the International Conference on Software Composition (SC)* Bd. 5634, Springer, Juli 2009 (Lecture Notes in Computer Science), S. 9–26
- [AKL08] APEL, Sven ; KÄSTNER, Christian ; LENGAUER, Christian: 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)*, ACM Press, Oktober 2008, S. 101–112
- [AKL09a] APEL, Sven ; KÄSTNER, Christian ; LENGAUER, Christian: Feature-House: Language-Independent, Automated Software Composition. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, Mai 2009, S. 221–231
- [AKL09b] APEL, Sven ; KÄSTNER, Christian ; LENGAUER, Christian: Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien. In: *Software Engineering 2009 – Fachtagung des GI-Fachbereichs Softwaretechnik* Bd. P-143, Gesellschaft für Informatik (GI), März 2009 (Lecture Notes in Informatics), S. 101–112
- [B<sup>+</sup>02] BRYANT, Avi u. a.: Explicit Programming. In: *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, ACM Press, 2002, S. 10–18
- [Bal96] BALZERT, Helmut: *Lehrbuch der Software-Technik – Software-Entwicklung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 1996
- [Bat05] BATORY, Don S.: Feature Models, Grammars, and Propositional Formulas. In: *Proceedings of the International Software Product Line Conference (SPLC)* Bd. 3714, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 7–20
- [BPSP04] BEUCHE, D. ; PAPAJEWSKI, H. ; SCHRÖDER-PREIKSCHAT, W.: Variability Management with Feature Models. In: *Science of Computer Programming* 53 (2004), Nr. 3, S. 333–352

- [BSR04] BATORY, Don S. ; SARVELA, Jacob N. ; RAUSCHMAYER, Axel: Scaling Step-Wise Refinement. In: *IEEE Transactions on Software Engineering* 30 (2004), Nr. 6, S. 355–371
- [CA05] CZARNECKI, Krzysztof ; ANTKIEWICZ, Michael: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)* Bd. 3676, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 422–437
- [CK02] CLEMENTS, Paul ; KRUEGER, Charles: Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. In: *IEEE Software* 19 (2002), Nr. 4, S. 28–31
- [CN02] CLEMENTS, P. ; NORTHROP, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002
- [CP06] CZARNECKI, Krzysztof ; PIETROSZEK, Krzysztof: Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ACM Press, 2006, S. 211–220
- [CPR07] COPPIT, David ; PAINTER, Robert ; REVELLE, Meghan: Spotlight: A Prototype Tool for Software Plans. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2007, S. 754–757
- [EFT96] EBBINGHAUS, Heinz-Dieter ; FLUM, Jörg ; THOMAS, Wolfgang: *Einführung in die mathematische Logik*. 4. Heidelberg, Berlin : Spektrum Akademischer Verlag, 1996
- [GJSB05] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *Java<sup>TM</sup> Language Specification*. 3. Addison-Wesley Professional, 2005 (The Java<sup>TM</sup>Series)
- [IP02] IGARASHI, Atsushi ; PIERCE, Benjamin: On inner classes. In: *Information and Computation* 177 (2002), Nr. 1, S. 56–89
- [IPW01] IGARASHI, Atsushi ; PIERCE, Benjamin ; WADLER, Philip: Feather-weight Java: a Minimal Core Calculus for Java and GJ. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23 (2001), Nr. 3, S. 396–450
- [JBZZ03] JARZABEK, Stan ; BASSETT, Paul ; ZHANG, Hongyu ; ZHANG, Weishan: XVCL: XML-based Variant Configuration Language. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2003, S. 810–811

- [JF88] JOHNSON, Ralph E. ; FOOTE, Brian: Designing Reusable Classes. In: *Journal of Object-Oriented Programming* 1 (1988), Nr. 2, S. 22–35
- [KA08a] KÄSTNER, Christian ; APEL, Sven: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLe)*, 2008, S. 35–40
- [KA08b] KÄSTNER, Christian ; APEL, Sven: Type-checking Software Product Lines - A Formal Approach. In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, September 2008, S. 258–267
- [KAB07] KÄSTNER, Christian ; APEL, Sven ; BATORY, Don S.: A Case Study Implementing Features Using AspectJ. In: *Proceedings of the International Software Product Line Conference (SPLC)*, IEEE Computer Society, September 2007, S. 223–232
- [KAK08] KÄSTNER, Christian ; APEL, Sven ; KUHLEMANN, Martin: Granularity in Software Product Lines. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, Mai 2008, S. 311–320
- [Käs07] KÄSTNER, Christian: *Aspect-Oriented Refactoring of Berkeley DB*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, März 2007
- [KAT<sup>+</sup>08] KÄSTNER, Christian ; APEL, Sven ; TRUJILLO, Salvador ; KUHLEMANN, Martin ; BATORY, Don: Language-Independent Safe Decomposition of Legacy Applications into Features / School of Computer Science, University of Magdeburg. Germany, März 2008 (2/08). – Forschungsbericht
- [KAT<sup>+</sup>09] KÄSTNER, Christian ; APEL, Sven ; TRUJILLO, Salvador ; KUHLEMANN, Martin ; BATORY, Don: Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In: *Proceedings of TOOLS EUROPE*, Springer, Juni 2009, S. 175–194
- [KCH<sup>+</sup>90] KANG, Kyo C. ; COHEN, Sholom G. ; HESS, James A. ; NOVAK, William E. ; PETERSON, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University, Software Engineering Institute. 1990 (CMU/SEI-90-TR-21). – Forschungsbericht
- [KHH<sup>+</sup>01] KICZALES, G. ; HILSDALE, E. ; HUGUNIN, J. ; KERSTEN, M. ; PALM, J. ; GRISWOLD, W.: An Overview of AspectJ. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 2072, Springer, 2001 (Lecture Notes in Computer Science), S. 327–353

- [KKB07] KÄSTNER, Christian ; KUHLEMANN, Martin ; BATORY, Don: Automating Feature-Oriented Refactoring of Legacy Applications. In: *Poster presented at European Conference on Object-Oriented Programming (ECOOP)*, 2007
- [KKB08] KIM, Chang Hwan P. ; KÄSTNER, Christian ; BATORY, Don: On the Modularity of Feature Interactions. In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ACM Press, Oktober 2008, S. 23–34
- [KLM<sup>+</sup>97] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 1241, Springer-Verlag, 1997 (Lecture Notes in Computer Science), S. 220–242
- [Kru02] KRUEGER, Charles: Easing the Transition to Software Mass Customization. In: *Proceedings of the International Workshop on Software Product-Family Engineering (PFE)*. London, UK : Springer-Verlag, 2002, S. 282–293
- [KTA08] KÄSTNER, Christian ; TRUJILLO, Salvador ; APEL, Sven: Visualizing Software Product Line Variabilities in Source Code. In: *Proceedings of the SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, 2008, S. 303–313
- [KTS<sup>+</sup>09] KÄSTNER, Christian ; THÜM, Thomas ; SAAKE, Gunter ; FEIGENSPAN, Janet ; LEICH, Thomas ; WIELGORZ, Fabian ; APEL, Sven: FeatureIDE: Tool Framework for Feature-Oriented Software Development. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, Mai 2009, S. 611–614
- [LAMS05] LEICH, T. ; APEL, S. ; MARNITZ, L. ; SAAKE, G.: Tool Support for Feature-Oriented Software Development – FeatureIDE: An Eclipse-Based Approach. In: *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, ACM Press, Oktober 2005, S. 55–59
- [LBL06] LIU, J. ; BATORY, D. ; LENGAUER, C.: Feature-Oriented Refactoring of Legacy Applications. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, 2006, S. 112–121
- [LHB01] LOPEZ-HERREJON, R. ; BATORY, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)* Bd. 2186, Springer-Verlag, 2001 (Lecture Notes in Computer Science), S. 10–24

- [Mey97] MEYER, Bertrand: *Object-oriented software construction*. 2. Prentice-Hall, 1997
- [Par72] PARNAS, David L.: On the Criteria to be used in Decomposing Systems into Modules. In: *Communications of the ACM (CACM)* 15 (1972), Nr. 12, S. 1053–1058
- [Paw06] PAWLAK, Renaud: Spoon: Compile-time Annotation Processing for Middleware. In: *IEEE Distributed Systems Online* 7 (2006), Nr. 11, S. 1
- [PBv05] POHL, Klaus ; BÖCKLE, Günter ; VAN DER LINDEN, Frank J.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005
- [Pie02] PIERCE, Benjamin C.: *Types and programming languages*. Cambridge, MA, USA : MIT Press, 2002
- [SB02] SMARAGDAKIS, Yannis ; BATORY, Don S.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), Nr. 2, S. 215–255
- [SGM02] SZYPERSKI, C. ; GRUNTZ, D. ; MURER, S.: *Component Software – Beyond Object-Oriented Programming*. 2nd. Addison-Wesley / ACM Press, 2002
- [TBK09] THÜM, Thomas ; BATORY, Don ; KÄSTNER, Christian: Reasoning about Edits to Feature Models. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, Mai 2009, S. 254–264
- [TOHS99] TARR, P. ; OSSHER, H. ; HARRISON, W. ; SUTTON, JR., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 1999, S. 107–119
- [TR07] TURON, Aaron ; REPPY, John: Metaprogramming with Traits. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 4609, Springer, 2007 (Lecture Notes in Computer Science), S. 373–398



---

---

## Eidesstattliche Erklärung

---

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 23.07.2009

Malte Rosenthal