Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien*

Sven Apel†, Christian Kästner‡, and Christian Lengauer†

† Fakultät für Informatik und Mathematik, Universität Passau
{apel,lengauer}@uni-passau.de
‡ Fakultät für Informatik, Universität Magdeburg
kaestner@iti.cs.uni-magdeburg.de

Abstract: Es gibt eine Vielzahl sehr unterschiedlicher Techniken, Sprachen und Werkzeuge zur Entwicklung von Softwareproduktlinien. Trotzdem liegen gemeinsame Mechanismen zu Grunde, die eine Klassifikation in Kompositions- und Annotationsansatz erlauben. Während der Kompositionsansatz in der Forschung große Beachtung findet, kommt im industriellen Umfeld hauptsächlich der Annotationsansatz zur Anwendung. Wir analysieren und vergleichen beide Ansätze anhand von drei repräsentativen Vertretern und identifizieren anhand von sechs Kriterien individuelle Stärken und Schwächen. Wir stellen fest, dass die jeweiligen Stärken und Schwächen komplementär sind. Aus diesem Grund schlagen wir die Integration des Kompositions- und Annotationsansatzes vor, um so die Vorteile beider zu vereinen, dem Entwickler eine breiteres Spektrum an Implementierungsmechanismen zu Verfügung zu stellen und die Einführung von Produktlinientechnologie in bestehende Softwareprojekte zu erleichtern.

1 Einleitung

In den letzten Jahren hat die Bedeutung von Produktlinien in der Softwareentwicklung stetig zugenommen [BCK98, PBvdL05]. Als Alternative zur individuellen Neuimplementierung eines jeden Programms, erlauben Softwareproduktlinien die systematische Wiederverwendung von Code innerhalb einer Domäne, d. h. innerhalb einer Familie von Programmen [CE00]. Programme, die zu einer Familie bzw. Softwareproduktlinie gehören und aus einer gemeinsamen Codebasis generiert werden, heißen auch *Varianten*.

Für Produktlinien hat das Konzept des Features besondere Bedeutung. *Features* repräsentieren die Unterschiede und Gemeinsamkeiten von Varianten einer Produktlinie [CE00]. In der Domäne der Datenbanksysteme sind beispielsweise die Transaktionsverwaltung und die Speicherverwaltung Features. Während eine grundlegende Speicherverwaltung in allen Varianten von Datenbanksystemen unabdingbar ist, wird nicht in allen Anwendungsfällen eine Transaktionsverwaltung benötigt [KAB07].

Typischerweise werden Features durch das Hinzufügen neuer und das Erweitern bestehender

*Diese Arbeit wurde durch die Deutsche Forschungsgemeinschaft unterstützt, Projektnummer AP 206/2-1.

Strukturen in der Codebasis einer Produktlinie implementiert, d. h. wir beschränken uns hier auf solche Features, die Inkremente in der Funktionalität eines Softwaresystems darstellen, und betrachten nicht etwa Features wie Ausführungszeit oder Sicherheit.

Unternehmen, die Produktlinientechnologie in ihren Entwicklungsprozess einführen wollen, können aus einer Vielzahl von Methoden, Sprachen und Werkzeugen wählen: von Versionsverwaltungssystemen, über die Verwendung einfacher *#ifdef*s, über Frameworks und Komponenten [BCK98] bis hin zu diversen spezialisierten Programmiersprachen und -werkzeugen [Pre97, BSR04, KHH+01, Gri00]. All diese haben verschiedene Stärken und Schwächen, welche in einschlägigen Foren umfassend diskutiert werden [SvGB05, LHBC05, MO04, KAK08, MP03, ALS08].

In unseren bisherigen Arbeiten haben wir uns mit verschiedenen Techniken zur Entwicklung von Softwareproduktlinien näher beschäftigt. Wir teilen Methoden, Werkzeuge und Sprachen zur Entwicklung von Produktlinien in zwei grundlegende Ansätze ein: dem Kompositionsansatz und dem Annotationsansatz [KAK08]. Wir wählen diese Einteilung, da sie uns erlaubt, eine signifikante Anzahl von Techniken zur Implementierung von Features systematisch zu vergleichen. Techniken, die sich nicht in eine der beiden Kategorien einteilen lassen, wie z. B. prozessorientierte Ansätze, werden nicht weiter betrachtet.

Im *Kompositionsansatz* werden die Produkte einer Produktlinie durch das Zusammensetzen von Codeeinheiten generiert. Wir haben in früheren Arbeiten einige Vertreter dieses Ansatzes, z. B. AHEAD [BSR04] und AspectJ [KHH+01], untersucht [ALS08, ALMK08, AKL08, KAB07, AL08] und sind auf einige prinzipielle Einschränkungen gestoßen [KAB07, KAK08], insbesondere hinsichtlich der Einführung von Produktlinientechnologie in bestehende Softwareprojekte. Aus diesem Grund wendeten wir uns gleichzeitig dem Annotationsansatz zu, dem auch Standardtechniken wie z. B. Präprozessoren und Frameprozessoren angehören. Im *Annotationsansatz* werden Produkte auf Basis von Annotationen einer gemeinsamen Codebasis generiert. Wir haben verschiedene Vertreter analysiert und Erweiterungen und Verbesserungen vorgeschlagenen [KAK08, KA08].

Während wir die spezifischen Probleme beider Ansätze (Komposition und Annotation) untersuchten, z. B. hinsichtlich Granularität, Sprachunabhängigkeit, Ausdruckskraft oder Typsicherheit [KAK08, AL08, ALS08, KAB07, KA08, AKL08], stellten wir nach und nach fest, dass beide komplementär in ihren Stärken sind, d. h. in Punkten, in denen der eine Ansatz Stärken hat, zeigt der andere Schwächen und umgekehrt. Zum Beispiel spielt der Kompositionsansatz seine Stärken bei der Modularisierung von großen, wiederverwendbaren Codeeinheiten aus. Der Annotationsansatz ist vorteilhaft auf kleineren Skalen, z. B. wenn es darum geht, Variabilität im Kontrollfluss von Funktionen auszudrücken.

Dieser Beitrag gibt einen Überblick über Erfahrungen mit beiden Ansätzen anhand von den drei repräsentativen Werkzeugen AHEAD [BSR04], FeatureHouse [AKL09] und CIDE [KAK08], diskutiert Unterschiede und Gemeinsamkeiten, und zeigt auf, wie beide Ansätze gewinnbringend kombiniert werden können. Unser Ziel ist, beide Ansätze so zu kombinieren, dass die Stärken des einen die Schwächen des anderen aufwiegen. Besonderes Augenmerk legen wir auf die Einführung von Produktlinientechnologie in großen Softwareprojekten. Generell ist die Einführung von Produktlinientechnologien schwierig, insbesondere wenn ein bestehendes Softwareprojekt nicht bereits als Produktlinie

entwickelt wurde. Dies bezeichnen wir im Weiteren als *Adoptionsbarriere*. In früheren Analysen wurde beobachtet, dass zwar leichtgewichtige Annotationsansätze wie Frames und Präprozessoren die Adoptionsbarriere verringern, aber auf lange Sicht die Softwarequalität und Architekturstabilität beeinträchtigen können [CK02]. Wir argumentieren, dass gerade die Verbindung von Komposition und Annotation die Adoptionsbarriere verringert und trotzdem Softwarequalität und Architekturstabilität nicht beeinträchtigt.

2 Ansätze zur Implementierung von Produktlinien

Es gibt viele verschiedene Technologien zur Implementierung von Softwareproduktlinien. Die meisten kann man in zwei Ansätze gliedern: den Kompositionsansatz und den Annotationsansatz [KAK08]. Technologien, die nicht in einer der beiden Kategorien fallen, werden nicht weiter betrachtet, z. B. prozessorientierte Ansätze, Versionsverwaltungssysteme oder Generatoren.

Kompositionsansatz. Im Kompositionsansatz werden die Features einer Produktlinie als eigenständige (physisch getrennte) Codeeinheiten[1] implementiert. Zur Generierung einer Variante einer Produktlinie, werden auf Grundlage einer Auswahl von Features die entsprechenden Codeeinheiten ermittelt und zusammengesetzt, üblicherweise zur Übersetzungs- oder Startzeit. Es gibt eine Vielzahl von Technologien zur Implementierung und Komposition von Codeeinheiten in Produktlinien, z. B. Komponenten und Frameworks [BCK98], Featuremodule [Pre97, BSR04, ALS08], Hypermodule [TOHS99], und Aspekte [Gri00]. Abhängig vom jeweiligen Ansatz variieren die verwendeten Kompositionsmechanismen von Plugins über Vererbung bis hin zu komplexen Codetransformationen. Die generelle Idee des Kompositionsansatzes ist in Abbildung 1 am Beispiel des featureorientierten Generators AHEAD [BSR04] illustriert.

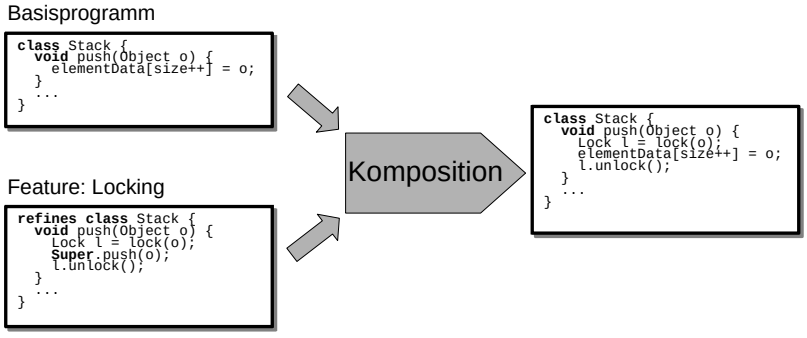


Abbildung 1: Generierung von Varianten mittels Komposition von Codeeinheiten.

[1] Der Kompositionsansatz ist auch auf andere Arten von Dokumenten, z. B. HTML-Dokumentationen oder Übersetzungsskripte, anwendbar.

```
1 class Stack {
2   void push(Object o) { ... }
3   Object pop() { ... }
4 }
```

```
5 refines class Stack {
6   void backup() { ... }
7   void restore() { ... }
8   void push(Object o) { backup(); Super.push(o); }
9 }
```

Abbildung 2: Eine Stackimplementierung in AHEAD (Kompositionsansatz).

Im Weiteren verwenden wir AHEAD sowie das kompatible und sprachunabhängige FeatureHouse [AKL09] als repräsentative Vertreter des Kompositionsansatzes. Wenn die Bewertung für andere Vertreter des Kompositionsansatzes abweicht, diskutieren wir das explizit. In AHEAD und FeatureHouse werden Features als Module implementiert, die, wenn einem Basisprogramm hinzugefügt, diesem neue Pakete und Klassen beigeben und bestehende Pakete, Klassen und Methoden erweitern. Abbildung 2 gibt ein einfaches Codebeispiel. Das Basisprogramm implementiert einen Stack, und ein separates Modul implementiert das Feature *UnDo*. Das Schlüsselwort *refines* zeigt an, dass ein Feature eine bereits vorhandene Klasse verfeinert. Die Verfeinerung in unserem Beispiel fügt zwei neue Methoden ein (*backup* und *restore*) und erweitert die Methode *push* durch Überschreiben. Das Schlüsselwort *Super* verweist auf die verfeinerte Klasse. Das Basisprogramm und das Featuremodul werden mit AHEAD bzw. FeatureHouse komponiert. Dabei werden die gewählten Module überlagert, d. h. im Beispiel entsteht eine zusammengesetzte Klasse *Stack* mit den Methoden *push*, *pop*, *backup* und *restore*, wobei die Methode *push* um den zusätzlichen Aufruf der Methode *backup* erweitert ist.

Abhängig von den im Kompositionsprozess ausgewählten Features, werden unterschiedliche Varianten der Produktlinie generiert. In unserem Beispiel gibt es nur zwei Varianten: einen Stack mit und einen ohne *UnDo*-Feature. Andere Kompositionswerkzeuge funktionieren sehr ähnlich, z. B. AspectJ [KHH+01].

Annotationsansatz. Im Annotationsansatz werden Features nicht als separate Codeeinheiten implementiert, sondern liegen in einer gemeinsamen Codebasis vor. Mittels Annotationen (z. B. textuellen Markierungen) werden die Codefragmente, die zu einem Feature gehören, entsprechend ausgezeichnet.[2] Eine Standardtechnik, die oft in industriellen Softwareproduktlinien Verwendung findet, ist die "Umrahmung" von Code mittels *#ifdef*- und *#endif*-Anweisungen. Alternativ dazu können auch andere Technologien wie Annotationen, Frames [JBZZ03] oder Softwarepläne [CPR07] verwendet werden. Diese Technik ist in kommerziellen Produktlinienwerkzeugen, wie z. B. *pure::variants*[3] oder *Gears*[4], verbreitet.

Wir verwenden unser Werkzeug CIDE als Vertreter des Annotationsansatzes [KAK08]. CIDE ähnelt Präprozessoren in der Hinsicht, dass es ermöglicht Codefragmente individuellen Features zuzuordnen, und dass Codefragmente nicht ausgewählter Features vor der Übersetzung entfernt werden. CIDE eliminiert einige Schwächen von präprozessorbasier-

[2] Auch der Annotationsansatz ist auf andere Arten von Dokumenten, z. B. HTML-Dokumentationen oder Übersetzungsskripte, anwendbar.

[3] http://www.pure-systems.com/

[4] http://www.biglever.com/

Abbildung 3: Bedingte Übersetzung mit CIDE (Annotationsansatz).

ten Techniken [KAK08]: (a) Annotationen werden als Hintergrundfarben präsentiert und "verunstalten" damit nicht den Quellcode; (b) Annotationen basieren auf der syntaktischen Struktur des Quellcodes, so dass nur optionale Elemente einer Sprache annotiert werden können (z. B. ist es nicht möglich, nur eine öffnende Klammer oder nur den Namen eine Klasse zu annotieren); (3) Annotationen werden von einer Werkzeuginfrastruktur verwaltet, die eine Navigationsfunktion und unterschiedliche Sichten auf den Quellcode anbietet (z. B. kann aller Code, der zu einem Feature gehört, ein- und ausgeblendet werden).

In Abbildung 3 ist unser Stackbeispiel zu sehen, diesmal mittels Annotationen implementiert. Die Codestücke des Basisprogramms sowie aller Features sind in einer Codebasis verschmolzen (also nicht physisch separiert). Codefragmente, die zum Feature *UnDo* gehören, sind farbig unterlegt und unterstrichen. Zur Generierung einer Variante ohne das *UnDo*-Feature, wird der markierte Code vor der Übersetzung entfernt (bedingte Übersetzung). Erneut sei betont, dass wir uns hier nicht speziell mit CIDE, sondern mit dem Annotationsansatz im Allgemeinen beschäftigen.

3 Vergleich von Kompositionsansatz und Annotationsansatz

Um die individuellen Stärken und Schwächen von Komposition und Annotation gegenüber zu stellen, analysieren wir beide Ansätze anhand von sechs Kriterien. Die Kriterien sind mit dem Ziel gewählt, Gemeinsamkeiten und Unterschiede aufzudecken, und basieren auf Erfahrungen mit Fallstudien zur Produktlinienentwicklung [TBKC07, AKL09, KAB07, KAK08, R+08]. Das Ergebnis unseres Vergleichs ist durch qualitative Bewertung hinsichtlich einzelner Kriterien in Tabelle 1 zusammengefasst. Wir betonen, dass die Bewertungen und Begründungen sich auf unsere Erfahrung stützen und somit keinesfalls als definitiv anzusehen sind. Vielmehr verstehen wir Tabelle 1 und die Ausführungen dieses Abschnitts als Anregung zur Diskussion.

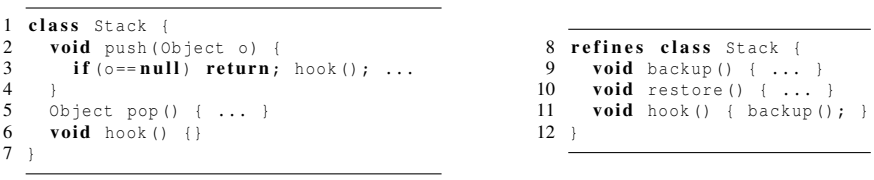
Auffindbarkeit von Featurecode. Features sind zuerst einmal dem Geist der Entwickler und Domänenexperten entsprungene abstrakte Konstrukte. Zur Erleichterung des Verständnisses und der Wartung des Quelltexts sollte die Abbildung von Features auf Implementierungseinheiten möglichst einfach strukturiert sein [CE00], d. h. der Code, der zu einem Feature gehört, sollte leicht in der Codebasis aufzufinden sein. Dies ist besonders wichtig für die Fehlersuche, um vom fehlerhaften Verhalten (Feature) auf den entsprechenden

Code schließen zu können. Im Kompositionsansatz ist es leicht, die Verbindung zwischen Feature und Code herzustellen, da jeweils ein Feature durch eine separate Codeeinheit implementiert wird. Zum Beispiel ist der Code, der das *UnDo*-Feature implementiert, im entsprechenden Featuremodul zu finden (siehe Abbildung 2, Zeilen 5–9). Im Gegensatz dazu kann das Auffinden des zu einem Feature gehörenden Codes im Annotationsansatz kompliziert sein, besonders dann, wenn der Code über weite Teile der Codebasis verstreut ist [SC92] (siehe auch Abbildung 3). Mit speziellen Werkzeugen wie CIDE kann dieses Problem gelindert werden, indem eine Navigationsfunktion und Sichten zur Verfügung gestellt werden. So kann man sich in CIDE z. B. nur den Code des Features *UnDo* anzeigen lassen. Nichtsdestotrotz ist immer ein extra Werkzeug von Nöten.

Modularität. Modularität von Featurecode wird im Kompositionsansatz naturgemäß besser unterstützt als im Annotationsansatz. Beispielsweise erlauben Komponenten [BCK98] oder Hypermodule [TOHS99] die separate Übersetzung von Code, der zu unterschiedlichen Features gehört. Es gibt allerdings auch einige populäre Sprachen und Werkzeuge, die dem Kompositionsansatz folgen, aber nicht einen ähnlich hohen Grad an Modularität ermöglichen. Zum Beispiel können Featuremodule in AHEAD, FeatureHouse oder AspectJ nicht separat übersetzt werden und verletzen das Geheimnisprinzip. Diese Probleme sowie mögliche Lösungen wurden insbesondere im Kontext von aspektorientierter Programmierung diskutiert [GSS+06]. Im Annotationsansatz werden keine Mechanismen zur Modularität von Featurecode verwendet. Werkzeugunterstützung kann über dieses Problem zum Teil hinweghelfen, aber eine separate Übersetzung und die isolierte Analyse von Features werden unmöglich.

Granularität. Die Granularität eines Implementierungsmechanismus ist eng mit seiner Ausdruckskraft verknüpft. Sehr grobgranulare Mechanismen erlauben nur das Hinzufügen und Entfernen von Verzeichnissen und Dateien. Feingranulare Mechanismen ermöglichen das Hinzufügen, Weglassen und Erweitern von Klassen, Methoden, Parametern und einzelnen Anweisungen [KAK08]. Im Annotationsansatz kann jedes Codefragment einem Feature zugeordnet werden. Dieses Prinzip skaliert in Werkzeugen wie CIDE von ganzen Verzeichnissen bis hin zu Parametern und Ausdrücken (siehe Abbildung 3). Im Kompositionsansatz wird ein weniger breites Spektrum abgedeckt. Üblicherweise wird das Hinzufügen von gesamten Verzeichnissen und Dateien unterstützt. Daneben gibt es auch einige Werkzeuge und Sprachen, die das Hinzufügen und Erweitern von Klassen, Methoden, und Instanzvariablen erlauben (siehe Abbildung 2). Die Manipulation von Anweisungen innerhalb einer Methode oder von Parametern und Ausdrücken wird im Kompositionsansatz nicht unterstützt [KAK08].[5] Stattdessen kann der Programmierer nur auf Umwegen zum Ziel gelangen. Zum Beispiel macht eine nur minimale Änderung unseres Stackbeispiels die Implementierung in AHEAD ungleich komplizierter. Steht der Aufruf der Methode *backup* in Abbildung 2 nicht an erster Stelle im Rumpf der Methode *push*, dann muss der Programmierer eine Hookmethode wie in Abbildung 4 verwenden.

[5] Einige aspektorientierte Sprachen unterstützen die Erweiterung von Anweisungen zu einem gewissen Grad (mittels *'call && withincode'*). Dies birgt aber einige schwerwiegende Probleme [KAB07, KAK08].



```
1  class Stack {
2    void push(Object o) {
3      if(o==null) return; hook(); ...
4    }
5    Object pop() { ... }
6    void hook() {}
7  }
```

```
8  refines class Stack {
9    void backup() { ... }
10   void restore() { ... }
11   void hook() { backup(); }
12 }
```

Abbildung 4: Feingranulare Erweiterungen mit AHEAD.

Sicherheit. Für den Kompositionsansatz und den Annotationsansatz gibt es gleichermaßen Bestrebungen, sicherzustellen, dass alle erlaubten Varianten (potentiell mehrere Millionen) einer Produktlinie korrekt sind [TBKC07, AKL08, KA08]. Im Kompositionsansatz wird syntaktische Korrektheit durch die verwendeten Sprachmechanismen garantiert. Syntaktisch inkorrekte Kompositionen werden vom Übersetzer oder vom Kompositionswerkzeug zurückgewiesen. Im Annotationsansatz ist dies problematischer. Werkzeuge die auf Präprozessoren und Frames basieren, können leicht inkorrekte Varianten generieren [KAK08, KA08]. Man braucht nur fälschlicherweise eine Klammer zuviel zu annotieren, und der Fehler wird erst gefunden, wenn die Variante nach der Generierung übersetzt wird. Bei mehreren Millionen Varianten ist dies unter Umständen zu spät, da nicht alle Varianten getestet werden können und erst bei konkreten Kundenanfrage generiert werden. Modernere Werkzeuge wie CIDE, die disziplinierte Annotationen auf Basis der syntaktischen Struktur erzwingen, erreichen ein vergleichbares Maß an Sicherheit wie im Kompositionsansatz [KAK08]. Auch zur Typsicherheit gibt es erste Arbeiten für beide Ansätze, die vom Konzept her sehr ähnlich sind [TBKC07, AKL08, CP06, KA08, KKB08].

Sprachunabhängigkeit. Hinsichtlich Sprachunabhängigkeit liegen beide Ansätze gleich auf. Präprozessoren und Frames sind komplett sprachunabhängig. Selbst disziplinierte Ansätze wie CIDE benutzen die Grammatik einer beliebigen Sprache zum Annotieren von Code [KAK08]. Im Gegensatz dazu sind Sprachen und Werkzeuge, die dem Kompositionsansatz folgen, üblicher Weise auf eine Artefaktsprache festgelegt, z. B. ist AspectJ auf Java als Zielsprache festgelegt. AHEAD und FeatureHouse sind dabei erwähnenswerte Ausnahmen. Besonders in FeatureHouse wird ein sprachunabhängiger Mechanismus zur Komposition verwendet, der leicht um weitere Sprachen erweitert werden kann [AKL09].

Einführung von Produktlinientechnologie. Unternehmen sind sehr vorsichtig bei der Einführung des Kompositionsansatzes zur Produktlinienentwicklung. Ein Grund ist, das bestehender Code sowie etablierte Prozesse zur sehr abgeändert werden müssten. Selbst in einer mittelgroßen akademischen Studie war dies langwierig und fehleranfällig [KAK08]. Derzeit werden allenfalls Komponenten und Frameworks eingeführt [BCK98, PBvdL05]. In Gegensatz dazu setzt sich der Annotationsansatz besser durch. Dies liegt im Wesentlichen an den unkomplizierten Werkzeugen sowie an dem Umstand, dass bestehender Code und etablierte Prozesse zunächst nicht abgewandelt werden müssen [CK02]. Der Annotationsansatz erleichtert dadurch die Einführung von Produktlinientechnologie enorm.

	Komposition	Annotation	Integration	Signifikanz im Entwicklungsproz.
Auffindbarkeit	++	+	+	alle Phasen
Modularität	+	−−	+	Wartung
Granularität	−	++	++	schnelle Adoption, Impl.
Sicherheit	+	+	+	alle Phasen
Sprachunabh.	+	+	+	alle Phasen
Adoption	−	++	++	Projektbeginn

++ sehr gut unterstützt; + gut unterstützt; − schlecht unterstützt; −− gar nicht unterstützt

Tabelle 1: Vergleich von Kompositionsansatz, Annotationsansatz und integriertem Ansatz.

```
13 class Stack {
14   void push(Object o) {
15     if(o==null) return; backup(); ...
16   }
17   Object pop() { ... }
18 }
```

```
19 refines class Stack {
20   void backup() { ... }
21   void restore() { ... }
22 }
```

Abbildung 5: Implementierung eines Stacks mit Featuremodulen und Annotationen.

4 Ein Integrierter Ansatz

Der Vergleich des Kompositions- und Annotationsansatzes zeigt, dass beide hinsichtlich unserer sechs Kriterien unterschiedliche Stärken und Schwächen aufweisen. Oft sind die Stärken und Schwächen komplementär. In anderen Fällen sind vergleichbare Fortschritte für beide Ansätze gemacht worden. Alles in allem scheint eine Kombination sinnvoll. In der aktuellen Forschung findet der Kompositionsansatz viel Beachtung. Der Annotationsansatz spielt eine untergeordnete Rolle. Im industriellen Umfeld setzen Produktlinienwerkzeuge wie Gears und pure::variants dagegen häufig auf Annotationen. Wie schlagen eine Integration zum beiderseitigen Vorteil vor.

Konzeptionell ist eine Integration beider Ansätze einfach. Dem Kompositionsansatz folgend werden grobgranulare Features in separate Codeeinheiten aufgeteilt. Für feingranulare Features, bei denen die Grenzen dieses Ansatzes erreicht werden, greift der Programmierer stattdessen zu Annotationen. Abbildung 5 illustriert an unserem Beispiel, wie beide Ansätze zur Implementierung eines Stacks kombiniert werden können. Die Methoden *backup* und *restore* werden in einem separaten Featuremodul als Erweiterungen des Basisprogramms implementiert. Der Aufruf der Methode *backup* in der Methode *push* wird auf Grund der in Abbildung 4 gezeigten Probleme nicht herausgelöst, sondern entsprechend annotiert. Zur Generierung einer Programmvariante werden die separaten Codeeinheiten komponiert und dann die Annotationen evaluiert, um Code gegebenenfalls zu entfernen.

Technisch ist die Integration ebenfalls nicht kompliziert. In unserem Beispiel müssen die Funktionalitäten von CIDE und AHEAD/FeatureHouse vereinigt werden. Zum Beispiel könnte CIDE so erweitert werden, dass es das Schlüsselwort *refines* versteht und eine Komposition mit bestehenden Kompositionswerkzeugen durchführt, nachdem Annotationen

ausgewertet werden. Darüber hinaus müssen die Funktionalitäten für Navigation und Sichten entsprechend angepasst werden. In diesem Zusammenhang haben wir bereits zwei Transformationswerkzeuge entwickelt, die annotierten Code in physisch separierten Code [KKB07] und umgekehrt überführen [KKB08].

Die Integration von Kompositionsansatz und Annotationsansatz vereint nicht automatisch nur die Stärken beider Ansätze. Zum Beispiel, wenn Code innerhalb von Modulen annotiert wird, ist eine separate Übersetzung ausgeschlossen. Der Vorteil der Integration ist aber, dass Entwickler jederzeit entscheiden können, welches Kriterium Priorität hat. Zusammengefasst ergibt sich für die sechs Kriterien folgendes Bild (siehe Tabelle 1).

Die Abbildung von Features auf Code wird durch Annotationen komplizierter. Ein Feature ist nicht nur durch ein Modul implementiert, sondern möglicherweise durch weitere verstreute, annotierte Codefragmente (siehe Abbildung 5). Wie schon beim Annotationsansatz erwähnt, helfen hier die Navigationsfunktion und Sichten von Werkzeugen wie CIDE. Das heißt, im schlechtesten Fall (es gibt nur ein Modul, dass Code aller Features enthält) ist die Integration vergleichbar mit dem Annotationsansatz und im besten Fall (pro Feature wird genau ein Featuremodul implementiert) mit dem Kompositionsansatz. In der Praxis wird die Lösung irgendwo dazwischen liegen und somit besser sein als der Annotationsansatz allein und trotzdem von dessen Vorteilen auf kleinen Skalen profitieren.

Bei der Integration von Kompositions- und Annotationsansatz kann der Entwickler ein Feature auf zwei Arten implementieren: entweder modular mittels Klassen, Featuremodulen, Aspekten, etc., oder nicht-modular mittels Annotationen. Ähnlich dem Kriterium der Auffindbarkeit, muss der Entwickler eine Entscheidung treffen, welcher Grad an Modularität gewünscht ist.

Der integrierte Ansatz profitiert von der feinen Granularität des Annotationsansatzes. Grobgranulare Features werden immer noch als Module implementiert und für die Implementierung von feingranularen Features kommen dann Annotationen hinzu. Annotationen können natürlich auch für die Implementierung von grobgranularen Feaures verwendet werden, verschlechtern aber die Modularität.

Da sich beide Ansätze hinsichtlich der Sicherheit und Sprachunabhängigkeit sehr ähneln, ändern sich diese Eigenschaften nicht im Gesamtansatz. Bestehende Methoden sind sehr ähnlich und können integriert werden [AKL08, KA08], was im Wesentlichen ein Implementierungsproblem ist.

Der vielleicht interessanteste Vorteil einer Integration liegt im Prozess der Einführung von Produktlinientechnologie in ein Unternehmen. In den frühen Phasen einer Einführung verwenden die Entwickler hauptsächlich Annotationen. Dies ermöglicht die schnelle Definition neuer Varianten, ohne die Codebasis oder den Entwicklungsprozess verändern zu müssen. In späteren Phasen, nachdem die Idee von Produktlinien im Projektmanagement und im Entwicklungsprozess verankert ist, können annotierte Codefragmente *schrittweise* in separate Featuremodule überführt werden. Transformationswerkzeuge helfen diesen Prozess zu automatisieren [KKB07, KKB08]. Die Entwickler können zu jedem Zeitpunkt entscheiden, inwiefern sie weiter Annotationen in Module überführen. Unsere hier vorgestellten Kriterien weisen den Weg und beleuchten die Effekte einer solchen Entscheidung. Eine Evaluierung in der Praxis steht allerdings noch aus.

5 Verwandte Arbeiten

In der Literatur finden sich einige vergleichende Studien, die verschiedene Implementierungsansätze für Produktlinien vergleichen. Lopez-Herrejon et al. [LHBC05] und Mezini und Ostermann [MO04] vergleichen Sprachen, die dem Kompositionsansatz folgen. Beide untersuchen im Wesentlichen Modularität. Der Annotationsansatz wird nicht betrachtet.

Anastasopoulos und Gacek [AG01] und Muthig und Patzke [MP03] vergleichen elf bzw. sechs Implementierungsansätze für Produktlinien. Beide Studien beziehen bedingte Übersetzung und Frames (Annotationsansätze) mit ein. In beiden Fällen stehen die Ausdruckskraft und verschiedene Details der verwendeten Sprachen im Vordergrund. Eine Integration des Kompositionsansatzes und des Annotationsansatzes wird nicht in Betracht gezogen. Des Weiteren vergleichen wir die generellen Ansätze von Komposition und Annotation und nicht spezielle Programmiersprachen und –werkzeuge.

Svahnberg et al. [SvGB05] schlagen eine Taxonomie für Variabilitätsmechanismen vor. Unsere Gliederung in Kompositionsansatz und Annotationsansatz ist orthogonal zu ihrer Klassifizierung.

Die Idee der schrittweisen Einführung von Produktlinientechnologie auf Grund von Komposition und Annotation erwuchs aus einer Diskussion im Dagstuhl-Seminar 08281 'Software Engineering for Tailor-made Data Management'. Die Migration von Produktlinientechnologie in ein Unternehmen sollte von 'in Produktlinien denken', über die Nutzung von Präprozessoren und Objektorientierung, in Richtung moderner Kompositionsansätze wie Aspektorientierung oder Featureorientierung verlaufen, bis hin zu modellgetriebener Entwicklung. Wir unterstützen diesen Migrationspfad mit unserem Ansatz.

FeatureC++ [ALRS05], eine C++-Erweiterung für featureorientierte und aspektorientierte Programmierung, erlaubt derzeit schon die direkte Kombination von Annotationen in Form von *#ifdef*s und Featuremodulen. Damit kann FeatureC++ als erster Vertreter des hier vorgeschlagenen integrierten Ansatzes verstanden werden. Allerdings fallen *#ifdef*s mit all ihren Problemen hinter moderneren Ansätzen wie CIDE zurück [KAK08].

6 Zusammenfassung

Es gibt eine Vielzahl von Techniken, Sprachen und Werkzeugen zur Entwicklung von Softwareproduktlinien. Wir haben diese in den Kompositionsansatz und den Annotationsansatz gegliedert, und haben beide Ansätze anhand von drei Werkzeugen und sechs Kriterien analysiert und verglichen. Wir sind zu dem Schluss gekommen, dass die Stärken und Schwächen beider komplementär sind, und wir schlagen eine Integration beider Ansätze vor.

Anstatt den Entwickler zu zwingen, sich für einen Ansatz zu entscheiden, erlaubt die hier vorgeschlagene Integration, mit dem Annotationsansatz zu beginnen und schrittweise Teile eines Softwareprojekts mittels des Kompositionsansatzes zu transformieren. Für jedes Feature können Entwickler entscheiden, welcher Ansatz am besten geeigneten ist.

Unsere Kriterien bilden dafür eine Analyse- und Diskussionsgrundlage. Anstatt eines Alles-Oder-Nichts-Ansatzes mit hohem Risiko schlagen wir eine schrittweise Einführung mit abschätzbarem Risiko vor.

Literatur

[AG01] M. Anastasopoules und C. Gacek. Implementing Product Line Variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[AKL08] S. Apel, C. Kästner und C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering*. ACM Press, 2008.

[AKL09] S. Apel, C. Kästner und C. Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l Conf. on Software Engineering*. IEEE CS Press, 2009.

[AL08] S. Apel und C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l Symposium on Software Composition*, Jgg. 4954 of *LNCS*, Seiten 20–35. Springer-Verlag, 2008.

[ALMK08] S. Apel, C. Lengauer, B. Möller und C. Kästner. An Algebra for Features and Feature Composition. In *Proc. Int'l. Conf. Algebraic Methodology and Software Technology*, Jgg. 5140 of *LNCS*, Seiten 36–50. Springer-Verlag, 2008.

[ALRS05] S. Apel, T. Leich, M. Rosenmüller und G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Jgg. 3676 of *LNCS*, Seiten 125–140. Springer-Verlag, 2005.

[ALS08] S. Apel, T. Leich und G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[BCK98] L. Bass, P. Clements und R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BSR04] D. Batory, J. Sarvela und A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[CE00] K. Czarnecki und U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CK02] P. Clements und C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[CP06] K. Czarnecki und K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Seiten 211–220. ACM Press, 2006.

[CPR07] D. Coppit, R. Painter und M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. on Software Engineering*, Seiten 754–757. IEEE CS Press, 2007.

[Gri00] M. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*, Seiten 271–288. Kluwer Academic Publishers, 2000.

[GSS+06] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai und H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[JBZZ03] S. Jarzabek, P. Bassett, H. Zhang und W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. on Software Engineering*, Seiten 810–811. IEEE CS Press, 2003.

[KA08] C. Kästner und S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering*. IEEE CS Press, 2008.

[KAB07] C. Kästner, S. Apel und D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference*, Seiten 223–232. IEEE CS Press, 2007.

[KAK08] C. Kästner, S. Apel und M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering*, Seiten 311–320. ACM Press, 2008.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm und W. Griswold. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 2072 of *LNCS*, Seiten 327–353. Springer-Verlag, 2001.

[KKB07] C. Kästner, M. Kuhlemann und D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Poster at Europ. Conf. Object-Oriented Programming*, 2007.

[KKB08] C. Kim, C. Kästner und D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering*. ACM Press, 2008.

[LHBC05] R. Lopez-Herrejon, D. Batory und W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 3586 of *LNCS*, Seiten 169–194. Springer-Verlag, 2005.

[MO04] M. Mezini und K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[MP03] D. Muthig und T. Patzke. Generic Implementation of Product Line Components. In *Proc. Net.ObjectDays*, Seiten 313–329. Springer-Verlag, 2003.

[PBvdL05] K. Pohl, G. Böckle und F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 1241 of *LNCS*, Seiten 419–443. Springer-Verlag, 1997.

[R+08] M. Rosenmüller et al. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, Seiten 1–6. ACM Press, 2008.

[SC92] H. Spencer und G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, Seiten 185–198, 1992.

[SvGB05] M. Svahnberg, J. van Gurp und J. Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[TBKC07] S. Thaker, D. Batory, D. Kitchin und W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Seiten 95–104. ACM Press, 2007.

[TOHS99] P. Tarr, H. Ossher, W. Harrison und S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. on Software Engineering*, Seiten 107–119. IEEE CS Press, 1999.