Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Synchronisierung von Software-Varianten mit VariantSync

Autor:

Lei Luo

20. Dezember 2012

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake

Institut für Technische und Betriebliche Informationssysteme

Dipl.-Inform. Thomas Thüm

Institut für Technische und Betriebliche Informationssysteme

Luo, Lei:
Synchronisierung von Software-Varianten mit VariantSync
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2012.

Inhaltsangabe

Software-Varianten sind sinnvoll für die Softwareentwicklung. Man versucht immer, Varianten schnell und effizient zu entwickeln. Mit den Produktlinientechnologien kann man schnell und effizient Varianten entwickeln. Aber es ist nur ab einer gewissen Anzahl von Varianten rentabel. Eine neue Methode für die Entwicklung von Varianten wird in dieser Arbeit dargestellt. Wir versuchen die Änderungen von Varianten synchronisieren. Bei der Synchronisierung benutzen wir Domänenwissen für Varianten. Zum Schluss wird der Prototype *VariantSync* für diese Methode evaluiert. Aus der Evaluierung haben wir gefunden, dass wir ungefähr 55.56% der Unterchiede von Varianten durch *VariantSync* automatisch synchronisieren können. Dabei werden ungefähr 70.05% der Operationen von Quelltext durch *VariantSync* automatisch geführt. Damit ist unser Ansatz wesentlich effizienter als Versionsverwaltungssysteme für die Entwicklung von Varianten. Außerdem für jede Variante gibt es eigene Quelltexte. Somit ist es einfach für Entwickler den Quelltext zu lesen und warten. Überdies ist diese Methode unabhängig von Programmiersprache.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

AHEAD	Algebraic Hierarchical Equations for Application Design (Werkzeug für FOP)
DOP	Delta-orientierte Programmierung
FOP	Feature-orientierte Programmierung
IDE	Integrated Development Environment
SPL	Software-Produktlinien

1. Einleitung

1.1 Motivation

In Anbetracht der rasanten Computerentwicklung wird die Rechenkapazität immer weiter zunehmen. Infolgedessen können immer komplexere Probleme mit Hilfe von Computern gelöst werden, folglich steigt die Komplexität der für die Probleme entwickelten Software. Um die Entwicklung der Software besser zu gewährleisten, sind neue Programmiermethodik und Konzepte notwendig. Dadurch können viele Softwareentwicklungsprojekte nicht mehr innerhalb des geplanten Zeit- und Kostenrahmens abgeschlossen werden. Außerdem ist die Qualität der Software mehr als ungenügend. „Als es noch keine Maschinen gab, war Programmierung kein Problem, als es dann ein paar leistungsschwache Computer gab, war Programmierung ein kleines Problem, und jetzt haben wir gigantische Computer, ist Programmierung ebenso ein gigantisches Problem. “ [Dij72]. Im Jahr 1968 auf einer NATO-Tagung in Deutschland wurden diese Probleme diskutiert und der Begriff des Software Engineering geprägt [Kah01]. In dem Bereich der Softwareentwicklung werden große Anstrengungen unternommen, Werkzeuge und Prozesse zu verbessern, um Qualität und Produktivität der Softwareentwicklung zu steigern.

Die Wiederverwendung von Software und Softwarekomponenten ist eine Methode des Software Engineering. Bei der Entwicklung einer neuen Software werden bestehende Software-Artefakte wiederverwendet. Im Jahr 1968 hat Douglas McIlroy von Bell Labor vorgeschlagen, die Software-Industrie mit wiederverwendbaren Softwarekomponenten zu unterstützen. Danach ist die Wiederverwendung von Software zu einem Forschungsbereich im Software Engineering geworden und wird zunehmend wichtiger bei der Softwareentwicklung [WOZ91]. Wayne C. Lim führte eine Studie in zwei Unternehmen durch und kam zu folgenden Ergebnissen [Lim94]: Durch

die Wiederverwendung können Entwicklungszeit und Markteinführungszeit um 42%, und Entwicklungskosten um 12% gesenkt werden. Die Fehlerrate der Software wird um bis zu 51% reduziert. Die Produktivität kann zwischen 40% und 57% gesteigert werden. Die Entwicklungszeit wird verkürzt, die Entwicklungskosten werden gesenkt, und die Produktivität wird gesteigert, weil für den gleichen Funktionsteil keine Neuentwicklung benötigt wird. Wegen der mehrfachen Wiederverwendung von Software treten anfänglich behobene Fehler nicht mehr auf. Dadurch verbessert sich die Qualität der Software.

Die Produktlinienentwicklung ist ein Ansatz zur Softwareentwicklung mit organisierter Wiederverwendung und organisierter Variabilität auf Basis einer gemeinsamen Plattform [BKPS04]. Das Ziel von Software-Produktlinie (SPL) ist es, die gemeinsamen Teile von vielen ähnlichen Software-Produkten wiederzuverwenden, um die Entwicklungskosten zu senken. Eine SPL wird normalerweise für einen bestimmten Markt, Anwendungsbereich oder eine bestimmte Kundengruppe entwickelt, um die gemeinsamen Teile wiederzuverwenden. SPL etablierte die Idee der individualisierte Massenfertigung für Software, die in der Automobilindustrie und vielen anderen Industrien sehr bekannt ist [Ros09]. Statt für einzelne Kunden jedes Produkt von Grund auf neu zu entwickeln, werden viele ähnliche Varianten in einer koordinierten Art und Weise entwickelt [Käs10]. Wie in der Automobilindustrie, wählt der Hersteller die passenden Teile nach Wünschen des individuellen Kunden, um ein neues Auto zusammenzubauen. Die Teile werden vorher entworfen und hergestellt. Mit einer SPL können Softwarehersteller sehr schnell für ihre Kunden maßgeschneiderte Software mit unterschiedlichen Funktionalitäten anbieten.

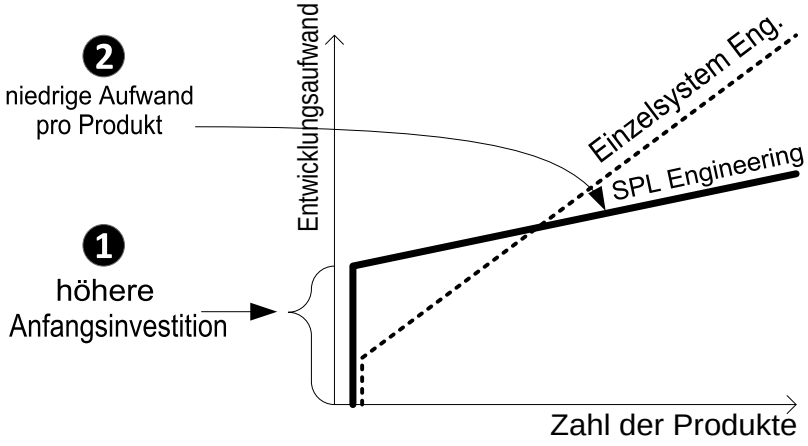


Abbildung 1.1: Aufwand der Entwicklung mit Software-Produktlinientechniken [JKB08]

Die Motivation zur Anwendung von SPL, anstelle einer normalen Einzelsystementwicklung, liegt darin, den Aufwand je Software-Produkt zu senken. Aber bei wenigen

Software-Produkten funktioniert es nicht. Wie die Abbildung 1.1 auf der vorherigen Seite jedoch zeigt, hat SPL höhere Anfangsinvestitionen als die normale Entwicklung. Demnach ist ein hoher Aufwand für die Initialisierung einer Produktlinie zu erwarten. Bei wenigen Varianten hat die normale Entwicklung einen niedrigeren Aufwand als die SPL. Das bedeutet, dass die Entwicklung von ein paar Varianten mit SPL wegen der hohen Investitionskosten nicht rentabel ist.

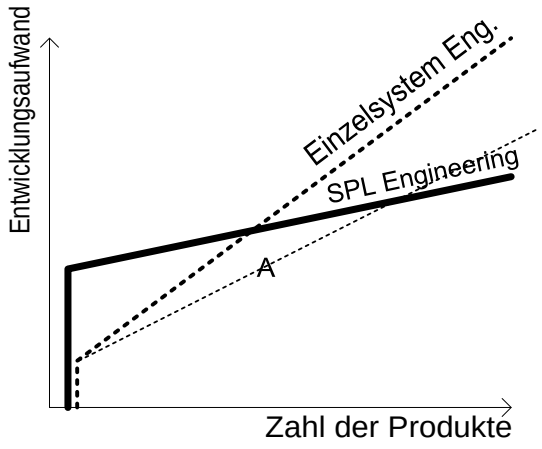


Abbildung 1.2: Gewünschter Aufwand der Entwicklung (vgl. [JKB08])

1.2 Zielsetzung

Das Ziel dieser Arbeit ist, eine Methode zu finden, mit der wenige Produkte effizienter entwickelt werden können. Die Abbildung 1.2 zeigt die Linie A, den gewünschten Aufwand. Weil SPL eine höhere Anfangsinvestition hat, ist die Entwicklung am Anfang mit SPL nicht effizient. Oder bei einer kleinen Firma, die nur wenige Produkte hat, ist die Entwicklung mit SPL auch nicht rentabel. In solchen Situationen wird die Entwicklung von unserer Methode profitieren. Wie in Abbildung 1.2 gezeigt, sofern später wir mehr Produkten hätten, wird die Methode nicht mehr effizienter als SPL sein. Aber das fortwährende Einsetzen unserer Methode zur Entwicklung von Software-Varianten ermöglicht eine einfache Migration zu SPL, wenn die Einzelentwicklung nicht mehr rentabel ist.

1.3 Gliederung

Diese Arbeit wird wie folgt gegliedert. Im Kapitel 2 werden die Grundlagen so wie Software-Produktlinien, Feature-Modelle, Versionverwaltungssoftware usw. eingeführt, die für das Verständnis der Arbeit genötigt werden. Was Variante ist, und warum Softwareprodukte Varianten benötigen, werden durch ein paar Beispiele im Kapitel 3 erklärt. Außerdem werden noch ein paar Methode zur Entwicklung von

Varianten im Kapitel 3 dargestellt. Danach wird eine kurze Diskussion über die Methoden geführt. Am Ende des Kapitel 3 wird die Methode Synchronisierung von Software-Varianten für die Entwicklung von wenigen Varianten beschrieben. Die prototypische Implementierung VariantSync für diese Methode wird im Kapitel 4 beschrieben. Im Kapitel 5 wird eine Evaluierung für die prototypische Implementierung dargestellt. Die Verwendete Programme, Vorgehensweise und Ergebnisse werden zuerst beschrieben. Danach wird eine Diskussion über die Ergebnisse geführt. Im Kapitel 6 werden ein paar Verwandte Arbeiten vermittelt. Zum Schluss kommen die Zusammenfassung und Zukünfige Arbeiten im Kapitel 7 und 8.

2. Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für das Verständnis der Arbeit benötigt werden. Zuerst werden Software-Produktlinien als ein effektives Mittel dargestellt, um bestehende Software-Artefakte wieder zu verwenden und schnell ähnliche Varianten zu generieren. Der zweite Teil beschäftigt sich mit Versionsverwaltungssystemen, sie typischerweise in der Softwareentwicklung verwendet werden.

2.1 Software-Produktlinien

Clements und Northrop legen für eine SPL folgende Definition fest:

„A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ [CN02]

Eine *Software-Produktlinie* beschreibt eine Menge von Software-Anwendungen, die bestimmte Features gemeinsam haben. Ein *Feature* repräsentiert eine Funktionalität der Programme. Die Gemeinsamkeiten und Unterschiede der Varianten von einer SPL werden mit den Features definiert. Alle Anwendungen einer SPL zielen auf ein bestimmtes Marktsegment oder die Anforderungen einer festgelegten Domäne. Software Engineering konzentriert sich traditionell auf die Einzelsystementwicklung. Für jedes Einzelsystem wird eine Entwicklung gebraucht. Ein typischer Entwicklungsprozess beginnt zunächst mit der Analyse der Anforderungen eines Kunden. Nach mehreren Entwicklungsschritten, in der Regel einiger Prozess der Spezifikation, Design, Implementierung, Test und Einsetzung, ist ein Einzelsystemsoftware-Produkt das Ergebnis [Käs10]. Im Gegensatz dazu konzentriert sich die SPL-Entwicklung auf mehrere ähnliche Software-Systeme in einer Domäne mit einer gemeinsamen

Quelltext-Basis [BCK03][PBvdL05]. Die mit SPL generierten Software-Produkte sind ähnlich und können viele verschiedene Anforderungen von Kunden erfüllen. Ein Produkt von den Software-Produkten wird Variante genannt. Im Vergleich mit der Einzelsystementwicklung bietet eine SPL eine effektive Möglichkeit bestehende Software-Artefakte wiederzuverwenden und unterschiedliche Varianten schnell zu generieren, dadurch hat die Bedeutung von SPL in den letzten Jahren stark zugenommen [BCK03][PBvdL05].

Die Entwicklung einer SPL gliedert sich in zwei parallel ablaufende Entwicklungsprozesse, die in Abbildung 2.1 dargestellt sind. Sie sind die Domänenentwicklung (engl. Domain Engineering) und die Anwendungsentwicklung (engl. Application Engineering).

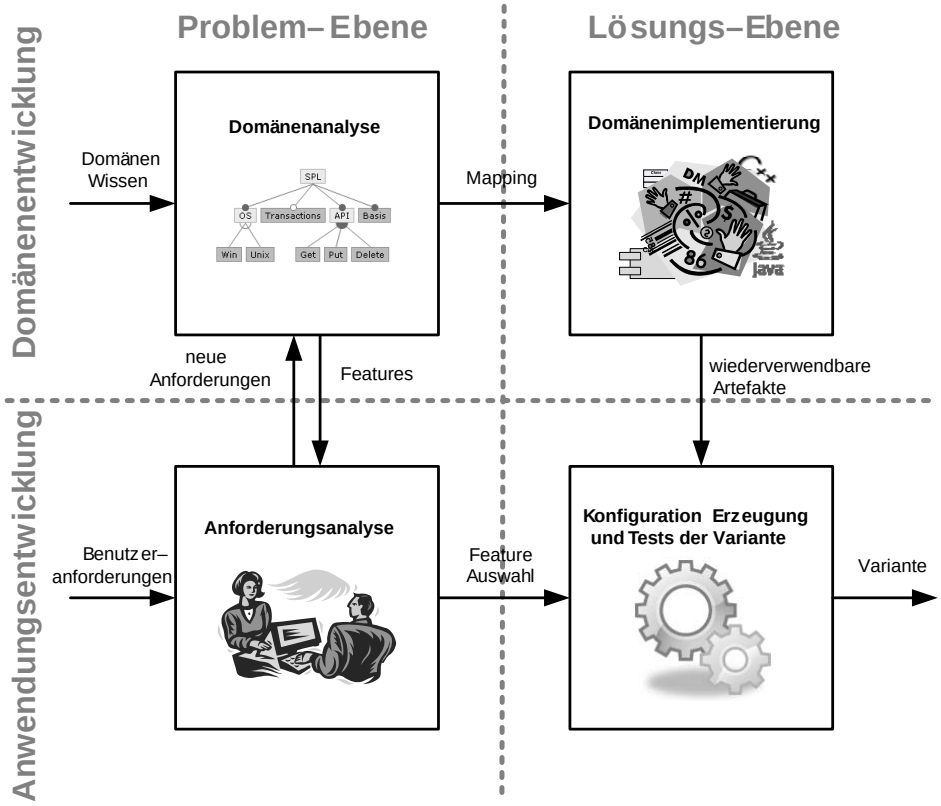


Abbildung 2.1: Ein Überblick über die Domänenentwicklung und Anwendungsentwicklung, adaptiert von [Käs10]

Der Prozess zur Entwicklung einer ganzen SPL, anstelle einer einzigen Anwendung, heißt Domänenentwicklung [Käs10]. Sie beinhaltet Analyse-, Design- und Implementierungsschritte. Es ist ähnlich wie bei der Entwicklung eines Einzelsystems. Eine SPL wird für mehrere Kunden erstellt, nicht nur aktuelle Kunden, sondern auch potenzielle, daher werden sowohl die ganze Domäne, als auch die potenziellen Anforderungen analysiert. In der Analysephase werden die identifizierenden Gemein-

samkeiten und Unterschiede zwischen den Varianten modelliert. Gemeinsamkeiten sind Bestandteile oder Funktionalitäten, die sich in allen Varianten befinden. Unterschiede sind die spezielle Funktionalität der Varianten. Anwendungsentwicklung bezeichnet den Vorgang, in dem ein spezieller Kunde geforderte Variante mit den Ergebnissen der Domänenentwicklung generiert wird. Im Idealfall werden alle Anforderungen vom Kunden in der Domänenentwicklung als potenzielle Anforderungen berücksichtigt, dann können die gewünschte Varianten mit den gemeinsamen und unterschiedlichen Teilen, die in der Domänenentwicklung entwickelt werden, schnell generiert werden. Falls nicht, werden die neuen Anforderungen in der Domänenentwicklung noch mal modelliert und implementiert. Damit kann sich die SPL später für diese Anforderung eignen. Nach der Generierung und bevor der Auslieferung der Variante, werden die Variante noch mal getestet.

2.1.1 Feature-Modelle

In der Domänenentwicklung werden die Gemeinsamkeiten und Unterschiede zwischen den Varianten modelliert. Eine in Forschung und Praxis gut bekannte und verbreitete Darstellungsform sind Feature-Modelle. Im Jahr 1990 wurden Feature-Modelle in der Feature-orientierten Domänenanalyse eingeführt [KCH+90].

Ein Feature-Modell beschreibt eine Menge von Features in einer Domäne und die Beziehungen dazwischen. Mit dem Modell können wir wissen, welche Kombinationen von Features wir bekommen können, bzw. welche Varianten eine SPL anbieten kann. Die grafische Repräsentation eines Feature-Modells ist das Feature-Diagramm. Das Diagramm wird aus einer Menge von Feature und gerichteten Kanten aufgebaut. Es ist eine Baumstruktur. Jedes Feature verfügt über ein übergeordnetes Feature, mit Ausnahme von einem Feature, das Wurzel-Feature genannt wird.

Die Beziehungen zwischen einem Elternfeature und dessen Kindfeature werden wie folgt kategorisiert [Bat05]:(1) und: Alle Kindfeatures müssen ausgewählt werden. (2) alternativ: Es kann nur ein der vorhandenen Kindfeatures ausgewählt werden. (3) oder: Mindestens ein Feature muss ausgewählt werden. (4) verbindlich: Falls das Elternfeature ausgewählt ist, muss dieses Feature ausgewählt werden. (5) optional: Wenn das Elternfeature ausgewählt ist, ist dieses Feature optional auszuwählen. Sie werden in der Abbildung 2.2 auf der nächsten Seite grafischen dargestellt.

Die Abbildung 2.3 auf der nächsten Seite zeigt ein Beispiel. Es ist ein Teil eines Feature-Modells einer TankWar-SPL. Diese demonstriert die Implementierung der SPL für ein Spiel (TankWar), das vom Autor und zwei Kollegen im Rahmen eines Laborpraktikums entwickelt wird. Das Wurzel-Feature *TankWar* wird immer ausgewählt. Das Spiel wurde für zwei Plattformen entwickelt. Deshalb wird das Feature *Plattform* immer ausgewählt, und wird nur eine von *PC* und *Handy* ausgewählt,

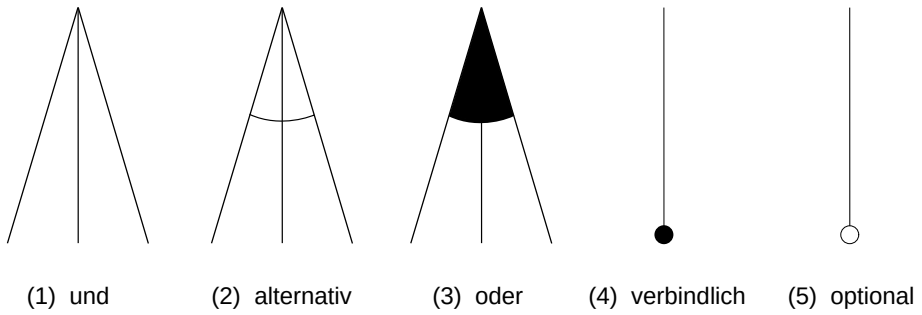


Abbildung 2.2: Notation der Beziehungen im Feature-Diagramm

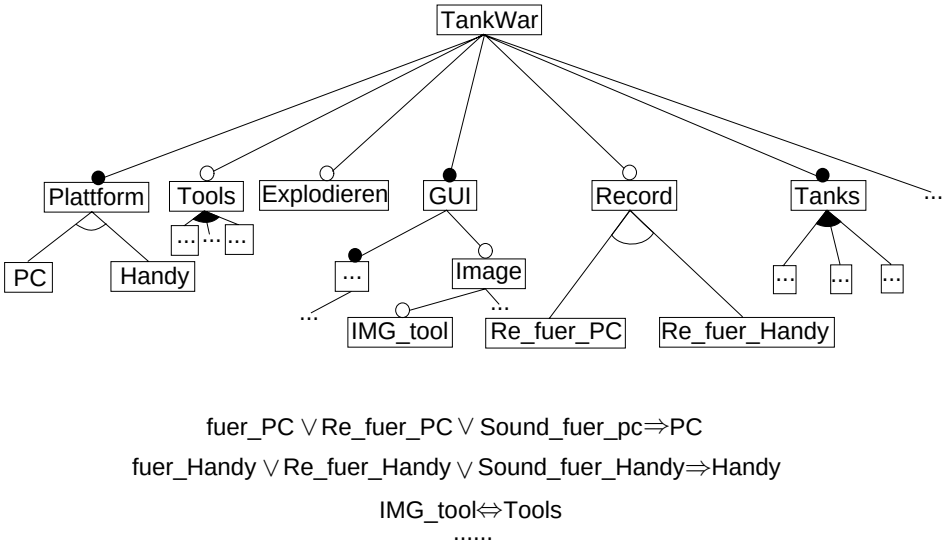


Abbildung 2.3: Feature-Modell der Produktlinie TankWar

weil es keine Variante gibt, die auf beiden Plattformen laufen kann. Das Feature *Explodieren* ist optional. Für manche Hardware, insbesondere Handy, ist es sinnvoll dieses Feature nicht auszuwählen, da es einen höheren Hardware Aufwand benötigt. Die zwei Kindfeatures von *Record* sind alternativ. Der Grund ist der gleiche, wie bei der Plattform.

Bei diesen zwei Features *Record* und *Plattform* können wir bemerken, dass nicht alle Kombination von Features möglich sind. In diesem gezeigten Beispiel können wir nicht *Re_fuer_PC* und *Handy* gleichzeitig auswählen. Zudem sind *IMG_tool* und *Tools* voneinander abhängig. Unter dem Diagramm werden solche Beschränkungen beschrieben. Sie werden als aussagenlogische Formel ausgedrückt. Für Feature *Tanks* wird mindestens ein Kindfeature ausgewählt, um zu bestimmen, wie viele Panzer für die Spieler verfügbar sind.

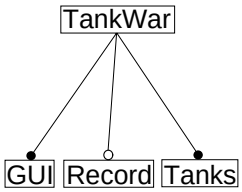
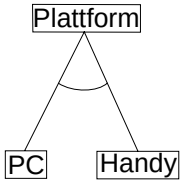
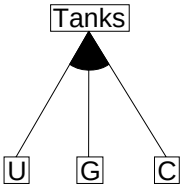
Beziehung	Grammatik	aussagenlogische Formel
	$TankWar : GUI[Record]Tanks$	$(TankWar \Rightarrow GUI \wedge Tanks) \wedge (GUI \vee Record \vee Tanks \Rightarrow TankWar)$
	$Plattform : PC\|Handy$	$(Plattform \Leftrightarrow PC \vee Handy) \wedge choose1(PC, Handy)$
	$Tanks : (U\|G\|C)+$	$(Tanks \Leftrightarrow U \vee G \vee C)$

Tabelle 2.1: Grammatiken und aussagenlogische Formel für Feature-Modelle

Es gibt noch Grammatiken für Feature-Modelle, die als eine textuelle Repräsentation für Feature-Modelle von Jong und Visser eingeführt wurden [BLHM02]. Außerdem können wir auch ein Feature-Diagramm in einer aussagenlogischen Formel transformieren [Bat05]. Diese wird erstmals von Mannion für Produktlinien verwendet [Man02]. Ein aussagenlogischer Ausdruck ist eine Menge von Variablen, die durch logische Operation verknüpft werden. Für jedes Feature haben wir eine boolesche Variable (in der Regel mit dem gleichen Namen). Falls eine Variable der Wert **True** hat, ist das entsprechende Feature ausgewählt. Der Wert der Variable wird auf **False** festgelegt, dann ist das Feature nicht enthalten. Für Wurzel-Feature ist der Wert immer **True**. In Tabelle 2.1 werden ein paar Beispiele für Grammatiken und aussagenlogische Formeln gezeigt.

2.1.2 Variantengenerierung

Bei der Programmierung können wir von Software-Produktlinien profitieren, weil Software von einer gemeinsamen Quelltext-Basis generiert wird [CE00]. Für den neuen Kunden müssen wir zuerst die Anforderungen analysieren. Im Idealfall, oder in den meisten Fällen, können wir mit vorhandenem Feature-Modell eine gültige Konfiguration erzeugen. In der Praxis ist es normal, dass nicht alle Kombinationen von Features gültig sind. Gültige Kombination und Quelltext-Basis werden als Eingabe dem Software-Generator eingegeben, um eine Variante zu generieren. Je nach der Implementierung von SPL unterscheidet sich der Generator. Von der Quelltext-Basis kann der Generator entsprechende Quelltext-Fragmente für Features bekommen, um

eine Variante mit der Kombination von der Quelltexte der ausgewählten Features zu generieren.

Von dem Feature-Modell kann man mehrere verschiedenen Konfigurationen bekommen, und jeweils mit dem Generator eine verschiedene Variante generieren. Eine Konfiguration legt die Werte der Variable von ausgewählten Features auf **True** fest. Falls mit einer Konfiguration der Wert der aussagenlogischen Formel von einem Feature-Modell **True** ist, ist die Konfiguration für dieses Feature-Modell gültig. Das Feature-Modell für TankWar-SPL (Abbildung 2.3 auf Seite 8) hat insgesamt 37 Features, und man kann über 2000 gültige Konfigurationen bzw. über 2000 verschiedene Varianten bekommen. Die Anzahl der möglichen Varianten kann exponentiell mit der Anzahl der Features wachsen. In der Theorie kann man mit einer SPL, die n unabhängige Features hat, 2^n Varianten generieren. In der Praxis wird die Anzahl der gültigen Konfigurationen wegen vieler Abhängigkeiten zwischen den Features reduziert, dennoch kann man mit den meisten SPL Millionen oder Milliarden Varianten generieren.

2.2 Versionsverwaltungssoftware

Heute ist es üblich, dass mehrere Entwickler an einem Softwareprojekt beteiligt sind. Dabei ist diese Situation oft so, dass mehrere Leute gleichzeitig an einer Datei arbeiten. Stellen Sie sich vor, ein Entwickler hat vom Server eine Datei heruntergeladen und arbeitet. Danach wird diese Datei wieder auf den Server hochgeladen und gespeichert. Aber falls inzwischen ein anderer Entwickler auch Änderungen an dieser Datei hochgeladen hat, dann werden die Änderungen des anderen Entwicklers überschrieben. Dieses Problem wird in der Abbildung 2.4 auf der nächsten Seite veranschaulicht. In der Softwareentwicklung kann es sehr oft auftreten. Die Lösung dafür nennt sich *Versionsverwaltung*.

Versionsverwaltung ist ein System, das typischerweise in der Softwareentwicklung zur Versionisierung und für den kontrollierten gemeinsamen Zugriff auf Quelltexte eingesetzt wird [Hof08]. Mit diesem System werden die Dateien zentral auf einen Server gespeichert, alle laufenden Änderungen erfasst und alle Versionsstände der Dateien in einem Archiv mit Zeitstempel und Benutzerkennung gesichert. Außerdem können alle Versionen später wiederhergestellt werden. Weil jeder Benutzer jederzeit auf Wunsch auf die archivierten Stände zugreifen kann, ist Versionsverwaltung nicht nur für die Entwickler in großen Teams, sondern auch für allein entwickelnde Entwickler sehr nützlich. Zum Beispiel hat ein Entwickler ein paar Änderungen gemacht um eine Funktion für ein Programm zu implementieren, aber es hat sich gezeigt, dass es nicht gut oder gar nicht funktioniert. Vielleicht ist die Idee für die Funktion nicht richtig, dann müsste man die Funktion nicht komplett neu programmieren. In diesem Fall ist es günstig mit der Versionsverwaltungssoftware eine ältere

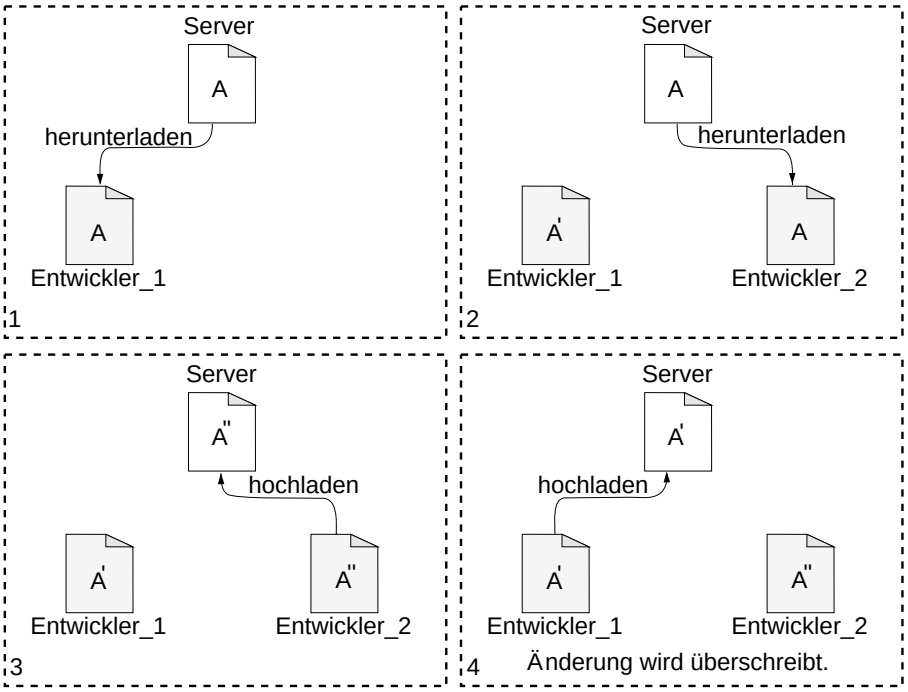


Abbildung 2.4: Das Filesharing Problem

Version aufzurufen. Für Versionsverwaltungssysteme ist die Abkürzung VCS (Version Control System) gebräuchlich.

Die Versionsverwaltungssoftware ist häufig als Client-Server-System aufgebaut (Z.B. SVN,CVS)[1], sodass der Zugriff auch über Netzwerk erfolgen kann, weil mehrere Entwickler räumlich getrennt arbeiten müssen. Das zentrale Archiv wird im Englischen als *Repository* bezeichnet. Es ist die Grundlage der Datenorganisation. Für das Archiv wird ein eigenes Dateiformat oder eine Datenbank verwendet. Dann kann eine Rechteverwaltung für mehrere Entwickler erfolgen. Nur die berechtigten Personen können neue Versionen in das Archiv legen. Die Versionsverwaltungssoftware speichert dabei üblicherweise nur die Unterschiede zwischen zwei Versionen, um Speicherplatz zu sparen [Was06].

2.2.1 Diff und Merge

Diff ist ein Unix-Programm, das die Unterschiede zwischen zwei Textdateien ausgibt. Die ersten Versionen des Programms wurden programmiert, um Textdateien zu vergleichen. Seit 1980 unterstützt Diff auch Binärdateien. Für die Softwareentwicklung und Versionsverwaltung wird es meistens für Textdateien, zum Beispiel Quelltexte von Programmen, benutzt. Es wird normalerweise verwendet, um die Änderungen zwischen einer Version einer Datei und einer früheren Version der gleichen Datei zu

[1] Während es früher nur zentrale Systeme gab, sind in den letzten Jahre viele dezentrale System wie Git erschienen

zeigen. Diff vergleicht zwei Dateien Zeile für Zeile, um die Gruppen von Zeilen zu finden, die unterschiedlich sind. Es kann über die unterschiedlichen Zeilen in verschiedenen Formaten berichten. Die Ausgabe von Diff geschieht in Textform und wird als *diff* oder *patch* bezeichnet. Denn die Ausgabe von Diff kann als Eingabe für das Unix-Programm *Patch* verwendet werden, um die Änderungen, die Diff festgestellt hat, an einer anderen Textdatei auszuführen.

```
1  public class A {
2      public static void main(String[] args) {
3          int a=1;
4          int b=2;
5          int c=a+b;
6          System.out.println("a+b="+c);
7          System.out.println("a-b="+(a-b));
8      }
9  }
```

Quelltext 2.1: Beispiel für einen Quelltext

```
1  public class A {
2      public static void main(String[] args) {
3          int a=3;
4          int b=2;
5          System.out.println("a+b="+(a+b));
6          System.out.println("a-b="+(a-b));
7          System.out.println("a*b="+a*b);
8          System.out.println("a/b="+a/b);
9      }
10 }
```

Quelltext 2.2: Beispiel für den geänderte Quelltext

Das Format der Ausgabe, die als Eingabe für das Patch verwendet wird, wird das vereinheitlichte Format (unified diff oder unidiff) genannt. Es wird sehr weit von Softwareentwickler benutzt. Ein *diff* mit diesem Format fängt mit zwei Zeilen an, die die originale Datei und die neue Datei beschreiben. Die erste Zeile fängt mit drei Minuszeichen an. Danach kommen der Ordnerpfad und der Name von der originalen Datei. In der zweiten Zeile stehen zuerst drei Pluszeichen, dann kommen der Ordnerpfad und der Name von der neuen Datei. Nach diesen zwei Zeilen stehen ein oder mehrere Blöcke, welche die unterschiedliche Zeilen in der Datei enthält. Jeder Block wird mit einer Zeile eingeleitet, die vom At-Zeichen(@) umgeben sind. Es zeigt an, wo sich der entsprechende Block in beiden Dateien befindet, und wie lang der Block in der jeweiligen Datei ist. Die mit Minuszeichen geleitete und durch ein Komma getrennte zwei Zahlen sind für die originale Datei. Die restlichen zwei Zahlen sind für

```
1  --- G:\Beispiel\original\A.java
2  +++ G:\Beispiel\neu\A.java
3  @@ -2,7 +2,8 @@
4      public static void main(String[] args) {
5  -       int a=1;
6  +       int a=3;
7          int b=2;
8  -       int c=a+b;
9  -       System.out.println("a+b="+c);
10 +       System.out.println("a+b="+(a+b));
11         System.out.println("a-b="+(a-b));
12 +       System.out.println("a*b="+a*b);
13 +       System.out.println("a/b="+a/b);
14     }
```

Quelltext 2.3: Das vereinheitlichte Format (unified diff oder unidiff)

die neue Datei. Danach kommt die Zeile in der Datei. Die nur in der originalen Datei entstehenden Zeilen werden mit einem Minuszeichen gekennzeichnet. Die nur in der neuen Datei entstehenden Zeile werden mit einem Pluszeichen gekennzeichnet. Die in der beiden Dateien entstehende gemeinsame Zeilen werden durch ein Leerzeichen gekennzeichnet. In den Quelltext 2.1 auf der vorherigen Seite und Quelltext 2.2 auf der vorherigen Seite werden zwei Quelltexte von Java als Beispiel gezeigt. In dem Quelltext 2.3 ist ein aus den zwei Quelltexten erzeugte *unidiff* zu sehen.

Die Funktionen, die die Änderungen zwischen zwei Textdateien auffinden und an einer Textdatei ausführen, sind wichtige Komponenten vieler Versionsverwaltungssystemen. Diff ist auch die Grundlage für Merge.

Merge ist ein Vorgang, der die Änderungen verschiedener Versionen derselben Datei zusammenführt. Es ist auch eine wichtige Komponente von den meisten Versionsverwaltungssystemen. Um das in der Abbildung 2.4 auf Seite 11 veranschaulichte Problem zu lösen, spielt Merge eine wichtige Rolle. Vor dem Hochladen von Entwickler_1 kann man die Version vom Server noch mal kontrollieren. Falls die Version vom Server schon von anderem Entwicklern geändert wird, dann wird zuerst ein Merge durchgeführt. Dadurch wird die Version vom Server die Änderungen von beiden enthalten. In den meisten Fällen wird der Merge-Vorgang automatisch ausgeführt. Es braucht keine menschliche Interaktion. Falls die verschiedenen Änderungen an den gleichen Zeilen oder den nebeneinanderstehenden Zeilen vorgenommen wurden, kommt es zu einem Konflikt für Merge. In diesem Fall kann es nur manuell aufgelöst werden. Viele Versionsverwaltungssoftware werden für Merge mit grafischen Hilfsprogrammen ausgeliefert.

2.2.2 Branch

Eine Hauptaufgabe des Versionsverwaltungssystems ist es, die gleichzeitige Entwicklung mehrerer Entwicklungszweige eines Projektes zu ermöglichen [Was06]. Ein Branch (Zweig) ist eine Abspaltung von einer Version. Damit können unterschiedliche Versionen parallel weiterentwickelt werden. Die Änderungen von Branch können später noch mal in der Hauptlinie zusammenführen. Das wird durch Merge realisiert. Die Hauptlinie oder der Hauptentwicklungszweig wird als *Trunk* bezeichnet. Branches können zum Beispiel für eine neue Hauptversion einer Software, für Entwicklungszweige für unterschiedliche Hardware, für experimentelle Versionen erstellt werden. In der Softwareentwicklung werden Branches viel genutzt.

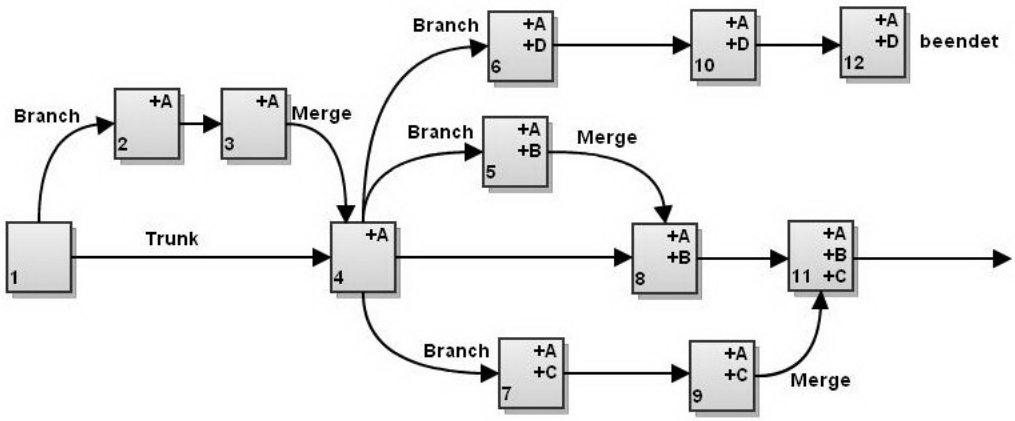


Abbildung 2.5: Ein Beispiel für Branches

Die Abbildung 2.5 zeigt ein Beispiel wie Branches für die Entwicklung eingesetzt werden können. Um eine neue Funktion A zu implementieren, wird ein Branch erstellt. Nach ein paar Weiterentwicklungen ist es einwandfrei geworden. Die Änderungen werden in der Hauptlinie zusammengeführt. Später fangen drei Entwicklungsgruppen an. Die Funktion B und Funktion C werden später implementiert und noch mal in der Hauptlinie zusammengeführt. Die Entwicklungsgruppe, die Funktion D implementiert hat, hat die experimentelle Entwicklung beendet.

3. Effiziente Entwicklung von wenigen Varianten

In diesem Kapitel wird zuerst durch ein paar Beispiele das erklären, was Variante ist, und warum Softwareprodukte Varianten benötigen. Danach werden ein paar Möglichkeit zur Entwicklung von Varianten dargestellt.Dann werden die Nachteile und Vorteile der Möglichkeiten diskutiert. Zum Schluss wird unser Ansatz für Entwicklung von wenigen Varianten beschrieben.

3.1 Ansätze zur Entwicklung von Varianten

Varianten sind keine neue Sache. Im Automobilbereich werden sie schon lange Zeit verwendet werden, um Produktfamilien zu produzieren. Vor dem Kauf des Autos kann der Kunde selbst nach dem individuellen Geschmack ein spezielles Auto konfigurieren. Welcher Kraftstoff wird von dem Motor gebraucht? Wie groß ist der Hubraum? Mit oder ohne Navigationssystem? Welche Farbe hat der Lack? Es gibt zahlreiche Möglichkeiten. Mit jeder Wahl können wir ein unterschiedliches Auto konfigurieren. Hersteller bieten unzählige Varianten des Autos, um die Wünsche von Kunden möglichst gut zu erfüllen. Effiziente Entwicklung von Varianten und die Zufriedenheit der Kunden sind zur Sicherung der Wettbewerbsfähigkeit des Herstellers erforderlich.

In der Softwareentwicklung können viele Varianten von Software durch die Änderung von Quellcodes erzeugt werden. Für verschiedene Chipsätze wird eine Software konfiguriert, um sie für die spezielle Plattform anzupassen. Darüber hinaus können Software-Varianten verschiedene Funktionalität bieten. Es gibt noch ein Wort, das meistens mit Variante verwechselt wird: Version. Versionen werden auch durch die Änderung von Quellcode erzeugt. Sie sind jedoch zeitlich nacheinander entstehende

Erzeugnisse. Das Ziel einer neueren Version ist eine ältere Version abzulösen. Das geht durch Veränderung oder Weiterentwicklung aus dieser hervor und stellt in der Regel eine Verbesserung dar. Im Vergleich Versionen sind Varianten zeitlich parallel existierende, vergleichbare Erzeugnisse. Die Abbildung 3.1 zeigt ein Beispiel für Versionen und Varianten von den Betriebssystemen.

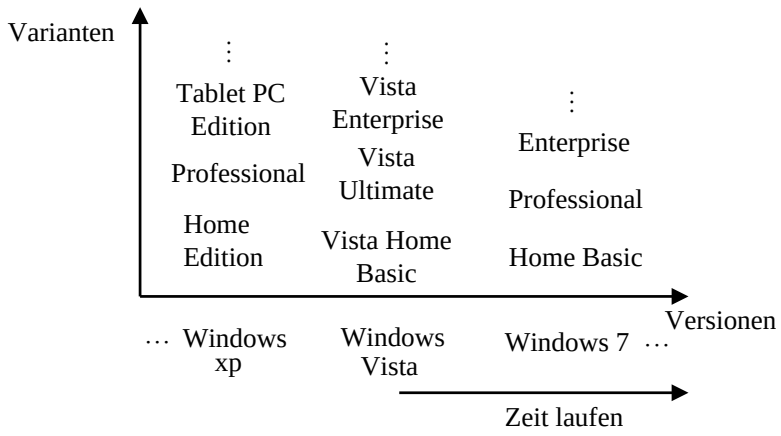


Abbildung 3.1: Varianten und Versionen

Die Verwendung der Alternativen von Varianten hängt vom konkreten Anwendungsfall ab. Zum Beispiel werden für unterschiedliche Hardware-Plattformen und unterschiedlichen Anforderungen an die Systemleistung passende Varianten ausgewählt. Microsoft Windows ist ein Markenname für die sehr bekannten Betriebsysteme des Unternehmens Microsoft. Die Version Microsoft Windows 7 wurde im Jahr 2000 angekündigt und sollte die Version Windows XP ablösen. 2001 hat Microsoft zwischen Windows XP und Windows 7 eine weitere Version von Windows veröffentlicht, die Windows Vista genannt wird. Seit dem 22. Oktober 2009 wird Windows 7 auf den Markt gebracht. Windows 7 ist bisher in sechs Varianten erschienen[1]: Windows 7 Home Premium, Windows 7 Professional, Windows 7 Ultimate, Windows 7 Enterprise, Windows 7 Starter, und Windows 7 Home Basic. Die Varianten werden zu unterschiedlichen Preisen verkauft. Dabei bieten Sie auch verschiedene Funktionen. Zum Beispiel ist Ultimate die teuerste Variante von Windows 7. Es ist die Einzige, bei der Windows auf verschiedene Sprachen umgeschaltet werden kann. Von Windows XP sind insgesamt sechs Varianten erschienen. Von Windows Vista sind Zehn Varianten erschienen.

Es gibt noch ein Beispiel für Varianten und Versionen. Android ist ein Betriebssystem für mobile Geräte wie Smartphones, Mobiltelefone, Netbooks und Tablets.

[1] http://windows.microsoft.com/en-PH/windows7/products/compare

Die derzeit aktuelle Version ist Version 4.1[2]. Nach der Veröffentlichung der neuen Version werden viele Varianten von mehreren Herstellern auf Basis dieser Version entwickelt. Die meisten Varianten werden entwickelt, um die Bedienoberfläche durch die eigene von Hersteller zu ersetzen, um die eigene Programme zu hinzufügen, oder um System nach der eigenen Hardware zu optimieren. Dadurch können Kunden verschiedene User Experience von verschiedenen Herstellern bekommen.

Ein erfolgreiches Softwareprodukt soll in einer Reihe von Varianten entwickelt werden. Damit können Kunden sich die Varianten aussuchen, die ihnen am ehesten zusagen oder die sie sich leisten können. Zum Beispiel bietet Windows 7 ein XP-Modus. Damit können viele alte Programme, die für Windows XP entwickelt werden, problemlos auch unter Windows 7 laufen. Aber der XP-Modus ist nur bei den teuren Windows 7 Varianten (Professional und Ultimate) verfügbar. Die Kunden, die keine alten XP-Programme benutzen müssen, können Geld sparen. Die teuersten Windows 7 Variante (Ultimate) bietet das Feature Bitlocker. Mit Bitlocker lassen sich komplette Datenträger automatisch verschlüsselt. Dieses Feature ist insbesondere für Notebook-Anwender interessant, wenn sie ihr Gerät verloren haben. Ein Softwareprodukt, das alle erdenklichen Funktionen beinhaltet, ist oft ungünstig. Die Möglichkeit, viele unnötige Fehlerquellen zu beinhalten, ist dann offensichtlich höher. Zugleich bedeuten mehr Funktionen für Kunden nicht immer mehr Zufriedenheit. Die vielen Funktionen führen oftmals zu einer Verwirrung der Kunden. Auch deshalb ist die effiziente Entwicklung von Varianten in der Softwareentwicklung sehr sinnvoll.

Varianten sind für ein Softwareprodukt sinnvoll und werden früher in der Softwareentwicklung für ein Softwareprodukt entwickelt. In den letzten Jahren erscheinen ein paar Programmierparadigmen, mit denen man schnell Varianten entwickeln kann. Beispielsweise gibt es Feature-orientierte Programmierung [AK09], Delta-orientierte Programmierung [SBB+10] und Aspekt-orientierte Programmierung [FECA04]. Aber sie sind meist sprachabhängig und brauchen spezielle Compiler. Programmierer müssen noch extra neue Syntax lernen. Jetzt schauen wir mal ein paar traditionelle Verfahren.

3.1.1 Versionsverwaltung zur Entwicklung von Varianten

Wegen des Entwicklungsaufwands werden neue Software-Varianten nicht von Grund auf neu entwickelt. Außerdem haben die Software-Varianten von einem Produkt meistens zu großen Teilen identischen Quellcode. Eine neue Variante wird durch ein paar Änderungen von einer vorhandenen Variante entwickelt. Dies ist auch logisch, da die meisten Funktionalität von Varianten auch gleich ist. Eine übliche Technik,

[2] http://www.android.com/

um Software-Varianten zu entwickeln, ist die Verwendung von Versionsverwaltungssoftware (z.B. CVS, SVN). Versionsverwaltungssystemen haben eine lange Tradition in der Software-Entwicklung und sind ein wichtiges Mittel zur Verwaltung von Varianten der heutigen Software-Systeme [CW98][ALB+11]. Dabei spielen Branches eine große Rolle.

Die Aufspaltungen in der Entwicklung werden *Branches* genannt. Wenn die Bearbeitung der Dateien eines Projektes für einen bestimmten Zeitraum in getrennten Linien parallel nebeneinander fortgeführt werden soll, ist einen Branch anzulegen sinnvoll (siehe Abschnitt 2.2.2). Für die Entwicklung von Varianten wird jede Variante häufig in einem eigenen Branch entwickelt [ALB+11].

Um eine neue Variante zu erstellen, wird oft der gesamte Quelltext einer existierenden Variante in einen neuen Branch kopiert. Danach werden durch Änderungen oder Erweiterungen an Quelltexte eine neue Variante erzeugt. Die Funktionalität von den Varianten ist oft ähnlich. Die Varianten unterscheiden sich lediglich in ein paar Funktionen (Features), die eine jeweilige Variante beschreibt. Manchmal werden einige Teile von der Software-Architektur durch neue entwickelte Teile ersetzt. Damit können die Varianten auf verschiedenen Hardwareplattformen laufen.

Nach der Aufspaltung kann die Variante vollständig unabhängig von anderen Varianten entwickelt werden. Aus der Sicht der Entwickler handelt es sich bei den Branches um alleinstehende Projekte. Ein solches Vorgehen verursacht jedoch auch einige Nachteile. Angenommen, die gemeinsame Funktionalität von verschiedenen Varianten wird vor der Aufspaltung (bzw. die Varianten durch Kopierungen erstellen) verbessert oder um einen Fehler korrigiert, dann werden die Änderungen durch das Kopieren in anderen Varianten übernommen. Durch die unabhängige Entwicklung kann es jedoch vorkommen, dass ein Fehler in einer Variante entfernt wird, aber in allen anderen Varianten (Branches bzw Projekte) weiterhin existiert.

Die Abbildung 3.2 auf der nächsten Seite bietet eine Darstellung der Entwicklung von Varianten in mehreren Branches. Wie die Abbildung gezeigt, dass es am Anfang nur eine Variante (*Variante 1*) gibt. Nach ein paar Zeit werden zwei Branches erstellt. In den zwei Branches werden ein paar Quelltexte geändert, um *Variante 2* und *Variante 3* zu erstellen. In den Quelltexte von *Variante 1* gab es zwei Fehler *B1* und *B2*. Vor der Erstellung der Branches für *Variante 2* und *Variante 3* wurde der Fehler *B1* behoben. In der Erstellung der Branches für *Variante 2* und *Variante 3* wurde die Quelltexte von *Variante 1* in neuen Branches kopiert. Deshalb der Fehler *B1* existiert nicht in *Variante 2* und *Variante 3*. Aber der Fehler B2 existiert noch in alle anderen Varianten. Später werden noch zwei Branches für *Variante 4* und *Variante 5* erstellt. Die alle 5 Variante haben ein paar gleiche Quelltexte oder Fragmente von Quelltexten. Sie werden in der Abbildung als *Gleiche Teile* gezeigt. Der Fehler *B2*

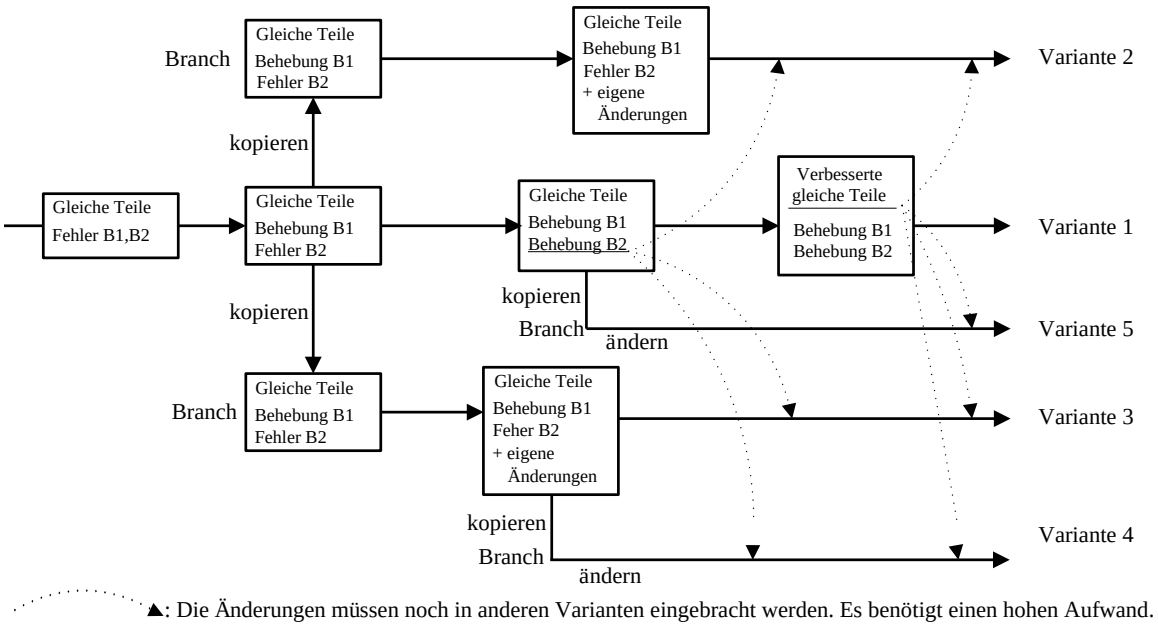


Abbildung 3.2: Entwicklung von Varianten in mehreren Branches

wurde in der *Variante 1* behoben. Nach der Behebung von *B2* wurde ein Branch für *Variante 5* erstellt. Deshalb der Fehler *B2* existiert noch in *Variante 2*, *Variante 3* und *Variante 4*. Die Änderung (um Fehler *B2* zu beheben) von *Variante 1* soll in anderen 3 Varianten übernommen werden. Die gleiche Quelltexte oder Fragmente von Quelltexten werden in *Variante 1* geändert. Zum Beispiel haben die Entwicklern der Algorithmus einer Funktion verbessert. Die Änderung soll in alle anderen 4 Varianten übernommen werden.

Verbesserungen oder Korrekturen in anderen Varianten einzubringen, verursacht meistens hohen Arbeitsaufwand. Änderungen müssen für jeden Branch einzeln „gemerged“ werden. Denn obwohl die Varianten die identischen Quelltexte für die gemeinsame Funktionalität haben, müssen meistens die Verbesserungen oder Korrekturen an verschiedenen Orte des Quelltextes je Variante durchgeführt werden. Überdies ist die Erstellung einer Variante komplex und unökonomisch. Zugleich zieht es oft lange Entwicklungszeit nach sich, denn um eine Variante zu entwickeln, muss man das komplette Basis-Programm kopieren und weiterentwickeln. Zum Beispiel möchte man eine Variante entwickeln, welche die in anderen Varianten schon implementierte Funktionen haben soll. Es ist aufwendig die in andere Variante existierende Implementierung wiederzuverwenden.

3.1.2 Präprozessor zur Entwicklung von Varianten

Die Anwendung von Präprozessoren ist eine einfache Methode, um Varianten zu entwickeln. Anstatt für jede Variante eine Kopie von Quelltexten anzufertigen (sie-

he Abschnitt 3.1), werden die verschiedene Varianten als ein gemeinsamer Quelltext entwickelt. In dem Quelltext werden bestimmte Fragmente mit *#ifdef X* und *#endif* oder ähnlichen Konstrukte kommentiert. Dabei ist *X* ein Parameter für den Präprozessor. Mit verschiedenen Parametern können verschiedene Varianten erstellt werden, welche die kommentierten Quelltext-Fragmente enthalten oder nicht enthalten. Präprozessoren transformieren Quelltext vor dem Compileraufruf. Sie wählen die passenden Quelltext-Fragmente entsprechend der Kommentare um den wünschten Quelltext zu erzeugen.

Die Varianten unterscheiden sich nur in den kommentierten Quelltext-Fragmenten. Die in den gemeinsamen Quelltext stehenden anderen Fragmente werden von den Varianten geteilt. Deshalb gibt es keinen Arbeitsaufwand für Synchronisierung zwischen Varianten. Die Varianten haben immer identische Änderungen bei den gemeinsamen Quelltext-Fragmenten.

Es gibt viele Präprozessoren, die ähnliche Funktionalität bieten. Beispielsweise wird für JavaME oft der Präprozessor *Antenna*[3] verwendet. *Antenna* ist geeignet für die Entwicklung von Wireless-Java-Anwendungen auf dem Mobile Information Device Profile. Es bietet Funktion von Präprozessor. Für Java gibt es noch den Präprozessor *Munge*[4]. *Munge* wird für die Entwicklung von Swing benutzt. Für die Programmiersprache C und C++ ist *cpp*[5] sehr bekannt. Präprozessoren führen textuelle Manipulationen am Quellcode vor dem Kompilieren durch. In Listing 3.1 auf der nächsten Seite zeigt ein Fragment von Java, das von *Munge* bearbeitet werden kann. Es gehört zu einem kleinen Programm, mit dem Benutzer miteinander per Netzwerk Nachrichten schicken können. Um die Nachrichten austauschen zu können gibt es Server-Programme und Client-Programme. Dieses gezeigte Fragment kommt aus dem Client-Programm. Es arbeitet für die grafische Benutzeroberfläche. Diese Methode *newChatLine(TextMessage Msg)* wird aufgerufen, wenn eine neue Nachricht vom Server empfangen wird. Diese Methode *handleEvent(Event e)* wird aufgerufen, wenn der Benutzer eine Nachricht eingetippt hat.

In Listing 3.2 auf der nächsten Seite ist ein Fragment von dem mit *Munge* bearbeiteten Quelltext. Dabei wird *Farbe* als Parameter für die Bearbeitung *Munge* gegeben. Dadurch wird der Quelltext für eine Variante erzeugt. Mit der Variante können Benutzer eine beliebige Textfarbe für die Nachrichten auswählen.

In Listing 3.3 auf Seite 22 ist ein Fragment von dem bearbeiteten Quelltext. *Historie* wird als Parameter für die Bearbeitung *Munge* gegeben. Mit der Variante können

[3] http://antenna.sourceforge.net/
[4] http://weblogs.java.net/blog/tball/archive/2006/09/munge_swings_se.html
[5] http://en.wikipedia.org/wiki/C_preprocessor

```
1  public void newChatLine(TextMessage msg) {
2          msg.verschluesseln(verS);
3          /*if[Historie]*/
4          msg.log("Client.dat");
5          /*end[Historie]*/
6          SimpleAttributeSet aset = new SimpleAttributeSet();
7          /*if[Farbe]*/
8          StyleConstants.setForeground(aset, msg.getFarbe());
9          /*end[Farbe]*/
10         try {
11         ....
12 }
13 public boolean handleEvent(Event e) {
14         if ((e.target == inputField) &&
15         (e.id == Event.ACTION_EVENT)) {
16             /*if[Farbe]
17             chatClient.send(new TextMessage(((String) e.arg),
18             colorButton.getBackground()));
19             else[Farbe]*/
20             chatClient.send(new TextMessage((String) e.arg));
21             /*end[Farbe]*/
22             inputField.setText("");
23             ....
24 }
```

Quelltext 3.1: Quelltext des Chatprogramms (mit Munge unterstützt)

```
1  public void newChatLine(TextMessage msg) {
2          msg.verschluesseln(verS);
3
4
5
6          SimpleAttributeSet aset = new SimpleAttributeSet();
7
8          StyleConstants.setForeground(aset, msg.getFarbe());
9
10         try {
11         ....
12 }
13 public boolean handleEvent(Event e) {
14         if ((e.target == inputField) &&
15         (e.id == Event.ACTION_EVENT)) {
16
17             chatClient.send(new TextMessage(((String) e.arg),
18             colorButton.getBackground()));
19
20
21
22             inputField.setText("");
23             ....
24 }
```

Quelltext 3.2: Der mit Munge bearbeitete Quelltext (mit Feature Farbe)

```
1  public void newChatLine(TextMessage msg) {
2          msg.verschluesseln(verS);
3
4          msg.log("Client.dat");
5
6          SimpleAttributeSet aset = new SimpleAttributeSet();
7
8
9
10         try {
11         ....
12 }
13 public boolean handleEvent(Event e) {
14         if ((e.target == inputField) &&
15         (e.id == Event.ACTION_EVENT)) {
16
17
18
19
20             chatClient.send(new TextMessage((String) e.arg));
21
22             inputField.setText("");
23             ....
24 }
```

Quelltext 3.3: Der mit Munge bearbeitete Quelltext (mit Feature Historie)

Benutzer die Nachrichten, die mit dem Programm geschickt und bekommen werden, in einer lokalen Datei *Client.dat* speichern.

Präprozessor werden häufig eingesetzt, um die Varianten bei der Kompilierung zu implementieren. In der Praxis werden sie oft benutzt [Käs10]. Zum Beispiel hat der Linux-Kern über 5000 Features [TSSpL09][SLB+10]. Die Benutzer können selbst ein passenden Linux-Kern kompilieren. Die Programme des Druckers von HP werden mit C-Präprozessor Varianten erzeugt [PO97][Ref09].

Präprozessor ist einfach zu benutzen. Aber es gibt auch einige Nachteile. Die von Präprozessor unterstützten Quelltexte sind meistens mit *#ifdef* und *#endif* oder ähnlichen Konstrukte (für Munge *#if[ ]* und *#end[ ]*) kommentiert. In der Praxis sind solche Fragmente in dem Quelltext oft zahlreich und miteinander verschachtelt. Im schlimmsten Fall werden die Parameter von einer Funktion kommentiert [Käs10]. Es bricht die regelmäßige Struktur von dem Quelltext. Für die Entwickler ist es sehr schwer zu lesen und verstehen. Es erschwert auch die Code-Wartung. Außerdem können Entwickler mit Präprozessor Syntaxfehler haben. In diesem Fall ist der mit Präprozessor bearbeitete Code offensichtlich nicht richtig.

Es gibt noch negative Behauptungen für Präprozessor in Literaturen: „#ifdef considered harmful“ [SC92] und „#ifdef hell“ [LST+06]. Außerdem gibt es auch zahlreiche Studien um die negativen Effekt von Präprozessor bei Code-Qualität und Wartbar-

keit zu diskutieren. Die Verwendung von #ifdef und ähnlichen Konstrukten macht Quellcode schwer zu lesen und neigt zu Fehlern zu führen.

3.1.3 FOP zur Entwicklung von Varianten

Im Vergleich mit der Versionsverwaltung und dem Präprozessor ist Feature-orientierte Programmierung (FOP) ein relativ neuer Ansatz für die Entwicklung von Varianten. FOP ist ein Programmierparadigma, mit dem kann man schnell viele Varianten entwickeln. Es ist für die Implementierung einer SPL geeignet. FOP basiert auf Features, die eine Eigenschaft oder eine Forderung der Kunde von Software dargestellt. Bei der FOP wird Software in Feature-Modell modularisiert [Pre97, BSR03]. Durch die Auswahl und Zusammensetzung von Features werden verschiedene Varianten mit einer gemeinsamen Quelltext-Basis synthetisiert.

Die Grundlage der FOP ist verwandt mit der Vererbung von den Objekt-orientierten Programmiersprachen. Ein Feature-Modul enthält Klassen oder Klassen-Fragmente. Die Klassen werden in die Klasse-Fragmente unterteilt, die verschiedene Features zugeordnet werden können. Dafür gibt es verschiedene Ansätze um die Klasse-Fragmente umzusetzen. Sie verlängern meistens die Syntax für Klassendeklarationen mit neuen Schlüsselwörter: *feature* [Pre97], *refines* [BSR03], *partial* (für C#). Nach der Feature-Auswahl werden die Klasse-Fragmente der ausgewählten Features mit einem Werkzeug nach einer bestimmte Reihenfolge komponiert, um eine Variante zu erzeugen. Zum Beispiel gibt es AHEAD [Bat06], FeatureC++, FeatureHouse usw. AHEAD wird für Java implementiert. FeatureC++ist für C++. FeatureHouse untstützt gleichzeitig viele Sprachen sowie Java, C#, C, Hashkell usw.

Die Abbildung 3.3 auf der nächsten Seite Zeigt ein Beispiel für FOP. Es ist ein Beispiel-Projekt von FeatureIDE und wird mit AHEAD entwickelt. Teil *(a)* von der Abbuildung 3.3 zeigt das Feature-Modell. Für dieses Feature-Modell gibt es insgesamt drei mögliche Konfigurationen. Mit den werden drei verschiedene Varianten erzeugt können. Teil *(c), (d), und (e)* zeigen die generierte Quelltexte für die drei Varianten. Die von dem Teil *(c)* gezeigte Variante enthält Feature {Hello, World}. Die von dem Teil *(d)* enthält Feature {Hello, Beautiful, World}. Und die von dem Teil *(e)* enthält Feature {Hello, Wonderful, World}. Die Fragmente von den Quelltexte, die für ein bestimmtes Feature implementiert wird, werden mit farbigen Rechtecke gezeigt. Teil *(b)* zeigt die gemeinsame Quelltext-Basis. Für AHEAD ist es ein paar Dateien, die mit der Namenserweiterung „*.jak*“ sind. Für jedes Feature werden ein Ordner erstellt. Alle Quelltext, die für das Feature implementiert wird, werden in diesem Ordner gelegt. Bei der Generierung von Varianten werden diese passende Dateien (je nach der Feature-Auswahl) zusammen komponiert und werden in Java-Datei (*.java) umgewandelt. Die Reihenfolge der Dateien für Features spielt

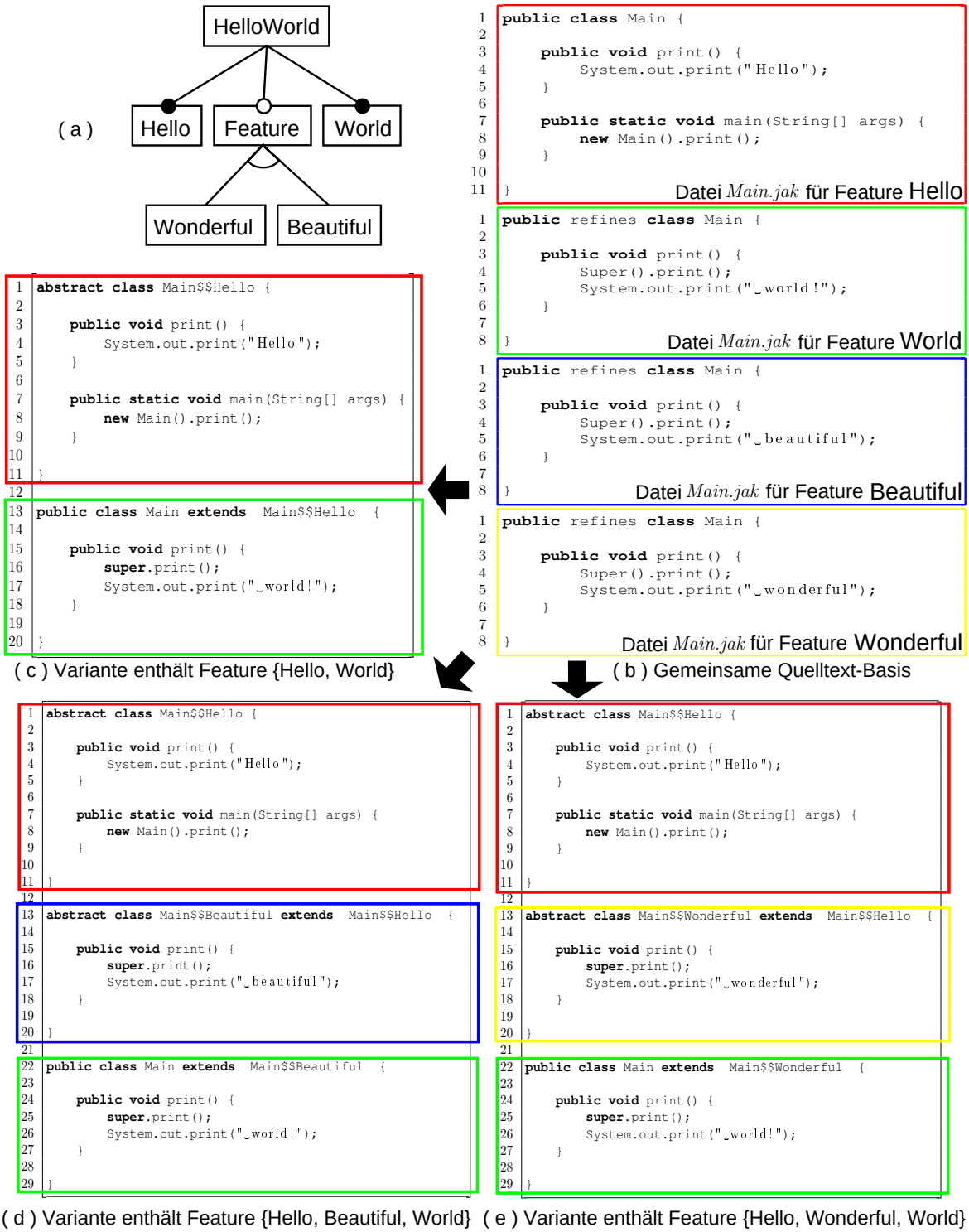


Abbildung 3.3: Beispiel (von FeatureIDE mit AHEAD) für Feature-orientierte Programmierung

bei der Generierung eine große Rolle. Es kann das Ergebnis beeinflussen. Weil die Grundlage der FOP verwandt mit der Vererbung von den Objekt-orientierten Programmiersprachen ist, ist die Reihenfolge der komponierte Features sehr wichtig. Wenn eine Klasse von einem Feature verfeinert wird, muss eine Datei für diesem Feature erstellt werden. Der Name der Datei ist wie den Name der Klasse und mit der Namenserweiterung „*.jak*“. Die Syntax ist gleich wie Java aber plus mit ein paar neuen Schlüsselwörter. In der Datei wird die Klasse mit Schlüsselwörter *refines* deklariert. In einer Methode wird die von der originale Klasse erstellte Methode mit Schlüsselwörter *Super()* aufgeruft. In der umgewandelte Java-Datei werden ein paar abstrakte Klasse für die Features erstellt, von den Features die Klasse verfeinert wird. Der Name von der abstrakte Klasse hat eine Form wie „*Klasse-Name$$ Feature-Name*“. Für das letzte Feature von der Reihenfolge der Generierung ist es ein Ausnahme.

Mit der FOP kann man schnell und zahlreich Varianten entwickeln. Zum Beispiel für das in dem Kapitel 2 gesprochene Programm TankWar können über 2000 verschiedene Varianten mit 37 Features erzeugt werden. Aber mit diesem Ansatz gibt es auch Nachteile. Die meisten Implementierungen für FOP sind Erweiterungen für bestehende Programmiersprachen. Die dafür erstellte Werkzeug sind nicht so genügend und nicht so neu wie die ursprüngliche Programmiersprachen. Zum Beispiel AHEAD unterstützt nur Java 1.4. Das bedeutet, dass Entwickler nicht neuen Sprachkonstrukte benutzen können.

3.1.4 Probleme der Ansätze

Es gibt viele Möglichkeit um Varianten zu entwickeln. Sie haben sowohl eigene Vorteile als auch eigene Nachteile. Mit Branches bei der Versionsverwaltung um Varianten zu entwickeln ist traditionell und sprachunabhängig. Mit Versionsverwaltung können wir eine neue Variante unabhängig von anderen Varianten entwickeln. Dabei gegen Entwicklung mit Präprozessor haben die Entwickler eine klar Sicht von Quelltext. Es ist einfach der Quelltext zu lesen und warten. Aber es fehlt die identische Änderungen bei gemeinsamen Quelltexte. Und es ist nicht so flexibel wie Präprozessor. Mit Präprozessor können wir zahlreiche Kombination von Funktionalität von verschiedenen Varianten bekommen. Im Vergleich zum Präprozessor haben die neue Programmierparadigmen keine zahlreiche Kommentare sowie *#ifdef* für bestimmte Fragmente. Aber Sie sind meist sprachabhängig und werden von bestimmte Programmsprach erweitert (z.B AspectJ, AHEAD, DeltaJ). Außerdem durch die Spracherweiterung müssen neue Entwicklungswerkzeuge sowie Editoren, Debugger, Metriken, Testwerkzeuge usw. erstellt werden. Diese Werkzeuge werden immer nicht schnell genug wie die normale Entwicklungswerkzeuge erstellt. Deshalb lohnt sich die Entwicklung von SPLs für wenige Varianten nicht.

3.2 Synchronisierung von Varianten

Wie können wir effizient wenige Varianten entwickeln? Wie können wir die Vorteile von Branches und Präprozessor gleichzeitig haben? Der Lösungsansatz ist Synchronisierung von Varianten. Wir Entwickeln die Varianten mit eigenen Quelltexten. Dabei haben wir eine klar Sicht von Quelltexten. Durch die Synchronisierung von Varianten können wir identische Änderungen am gemeinsamen Quellcode herstellen. Zugleich sind die Varianten bei der Entwicklung voneinander unabhängig. Außerdem können wir die Funktionalität, die nur in einigen Varianten existiert, nach anderen Varianten synchronisieren. Damit können die anderen Varianten auch die Funktionalität haben. Bei diesem Vorgehen können wir eine neue Variante erzeugen. Zum Beispiel haben wir eine Variante (Die Variante enthält Features {Historie, Verschlüsselung}) vom Chat-Programm geändert, damit können wir mit dieser Variante die Nachrichten verschlüsseln. Dabei werden alle die Änderungen für Erstellung der Verschlüsselung gespeichert. Später können wir die Änderungen nach einer Kopie einer Variante (Die Variante enthält Features {Historie, Farbe}) synchronisieren. Dann können wir die Funktionalität mischen. Damit werden neue Varianten (Die Variante enthält Features {Historie, Farbe, Verschlüsselung}) erzeugt.

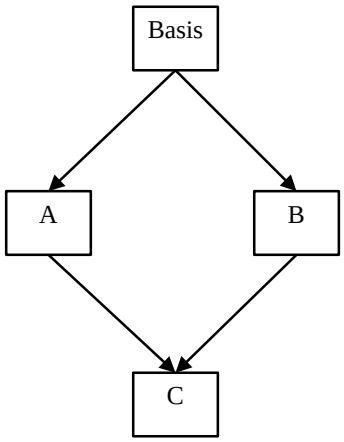


Abbildung 3.4: 3-Wege-Merge

3.2.1 Synchronisierung

Um die Varianten zu synchronisieren müssen wir die Varianten voneinander vergleichen und die Änderungen von Varianten speichern. Das Loggen von den Änderungen ist sehr wichtig, denn es wird später bearbeitet und nach anderen Varianten synchronisiert. Nach jedem Speichern von Quelltext wird ein Vergleich durchgeführt. Die die zwischen diese mal und letzte mal Speichern entstehende Änderung für jeden Quelltext wird mit dem vereinheitlichten Format (unidiff) gespeichert.

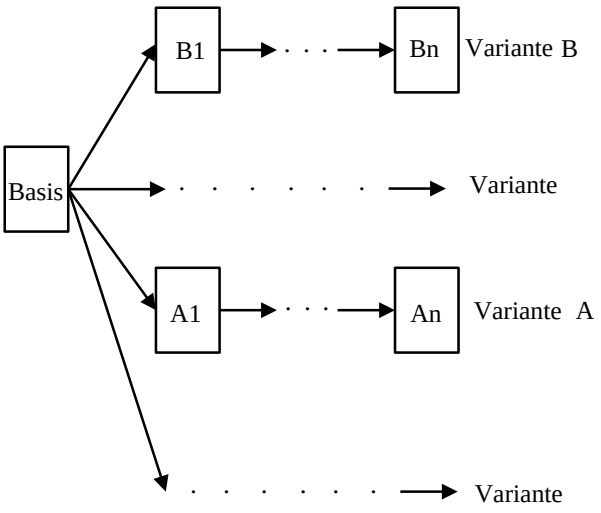


Abbildung 3.5: Entwicklung von Varianten

Um die Änderung nach anderen Varianten zu synchronisieren ist es ähnlich wie Drei-Wege-Merge. Die Abbildung 3.4 auf der vorherigen Seite zeigt die Darstellung von Drei-Wege-Merge. A und B sind die Ableitungen von Basis. C hat beide Änderungen von Basis zu A und B. Die Abbildung 3.5 zeigt die Darstellung von der Entwicklung von Varianten. Die Varianten haben eine gleiche Basis. Um die Änderung zwischen B_{n-1} und B_n nach Variante A zu synchroniesieren, können wir ein Drei -Wege-Merge machen. Dabei wird B_{n-1} als Basis behandelt. Weil wir durch ein paar Änderungen an B_{n-1} die A_n bekommen können. Wie viel Änderungen zwischen B_{n-1} und A_n spielt hier keine Rolle (Abbildung 3.6).

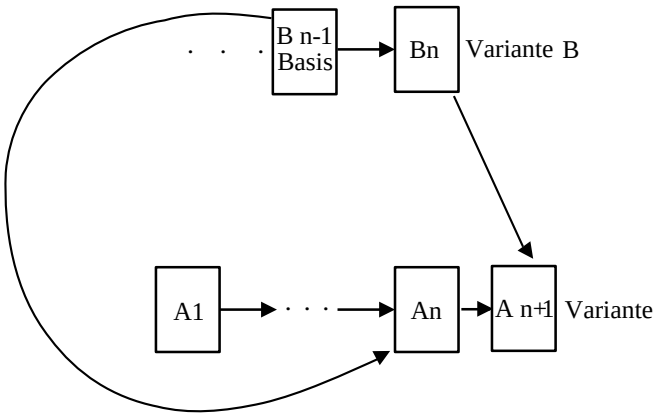


Abbildung 3.6: Synchronisierung in andere Varianten

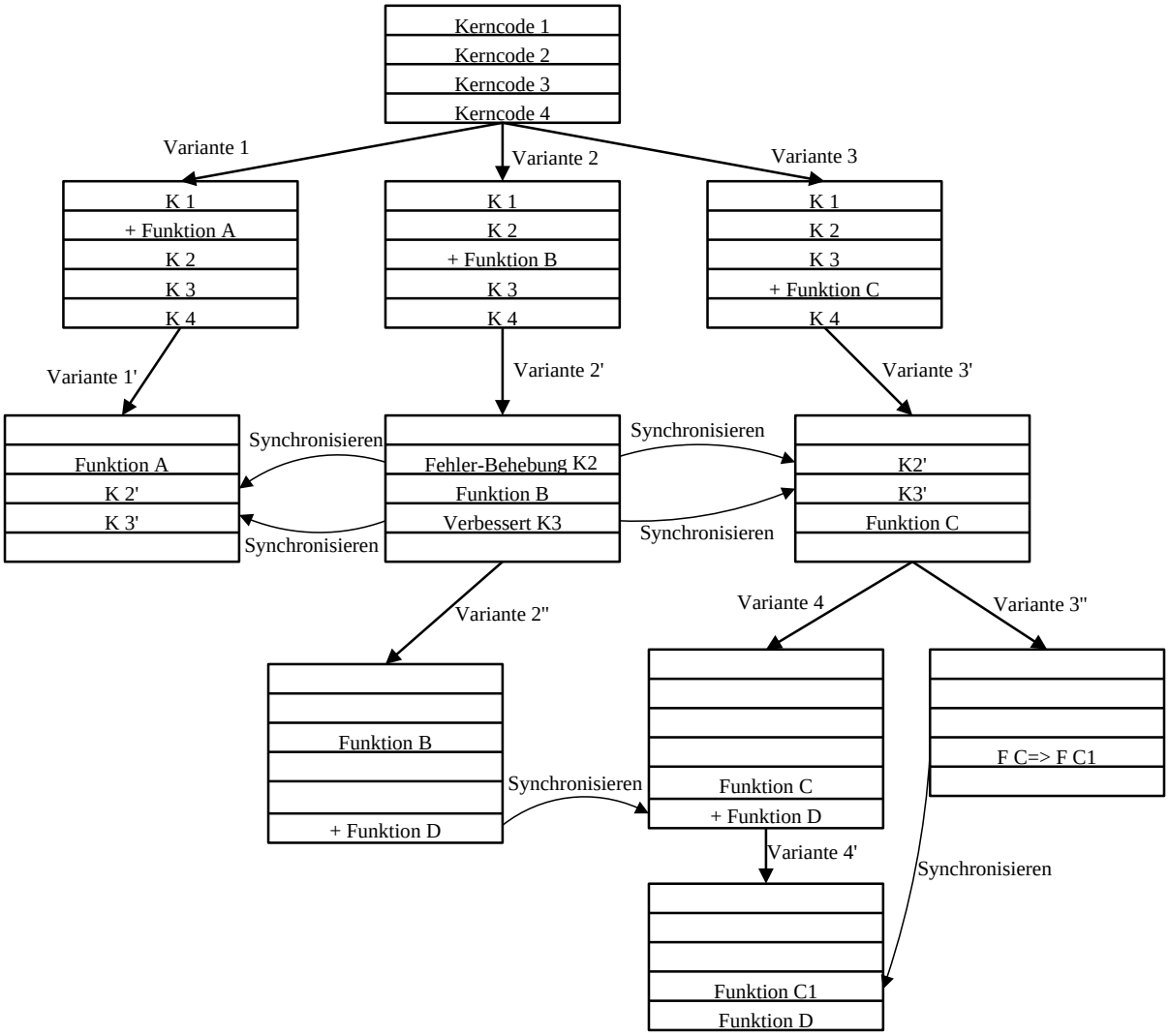


Abbildung 3.7: Synchronisierungen von Varianten

Die Abbildung 3.7 bietet eine theoretische Darstellung der Synchronisierungen von Varianten. Die Behebung des Fehlers bei Kerncode 2 und die Änderungen von Kerncode 3 werden durch Synchronisierungen in Variante1 und Variante3 eingebracht. Die Änderung von Variante 2′ bis Variante 2″ wird speichert. Darinnen wird eine neue Funktion D hinzugefügt. Um die Änderung nach Variante 3 zu synchronisieren, können wir eine neue Variante 4 erstellen. Später wird die Funktion C in der Variante 3 geändert. Durch Synchronisierung kann Variante 4 auch aktualisiert werden.

3.2.2 Konflikte

Die Synchronisierungen können meistens automatische durchgeführt werden. Zum Beispiel wird ein Element umbenannt, ein Zeile in dem Quelltext hinzugefügt. In Abbildung 3.8 auf der nächsten Seite werden zwei Beispiel gezeigt. Aber bei Konflikten ist Benutzereingriff erforderlich. Es gibt mehrere Ursachen für Konflikte:

```
public class TextMessage implements Serializable {
        ...

        private String head;
        private String content;

        public TextMessage(String content) {
        ...
        }
```

```
public class TextMessage implements Serializable {
        ...

        private String msgHead;
        private String content;

        public TextMessage(String content) {
        ...
        }
```

```
public void send(TextMessage msg) {
        try {
                outputStream.writeObject(msg);
                outputStream.flush();
        } catch (IOException ex) {
        ...
        }
}
```

```
public void send(TextMessage msg) {
        try {
                msg.verschluesseln(verS);
                outputStream.writeObject(msg);
                outputStream.flush();
        } catch (IOException ex) {
        ...
        }
}
```

Abbildung 3.8: Beispiele für automatische Synchronisierungen

(1) Reihenfolgekonflikt. Es steht in verschiedenen Varianten. In den Varianten werden in gleicher Position ein paar Zeilen vom Quelltext hinzugefügt. Die Reihenfolge von den hinzugefügten Codes kann wahrscheinlich das Ergebnis des Programms beeinflussen. Deshalb ist Benutzereingriff erforderlich, um die Änderungen zu synchronisieren. Die Abbildung 3.9 auf der nächsten Seite zeigt der Basiscode (Fragment A) von $Variante_1$ (Fragment B) und $Variante_2$ (Fragment C). Um die Änderung von einer Variante nach anderer Variante zu synchronisieren, ist Benutzereingriff erforderlich. Die Reihenfolge der zwei neuen Zeilen müssen Benutzer selbst bestimmen. In disem Fall, wird gewünscht, dass „Hello world!“ ausgedruckt wird.

(2) Abhängigkeitskonflikt. Es kann in verschiedenen Varianten und derselben Variante stehen. Für den benachbarten Quellcodes gibt es dazwischen wahrscheinlich Abhängigkeit. Für dieselbe Variante kann eine Änderung an einer vorher durchgeführten Änderung stehen. Die Änderungen, die auf anderen Änderungen beruhen, können auch nicht automatisch synchronisieren. Um die Änderungen zu synchronisieren, müssen die vorher durchgeführten Änderungen zuerst synchronisiert werden.

Die Abbildung 3.10 auf Seite 31 zeigt einen Abhängigkeitskonflikt zwischen zwei Varianten. In $Variante_1$ (Fragment B) wird Zeile 4 geändert. In $Variante_2$ (Fragment C) wird Zeile 5 geändert. Die zwei Änderungen sind Nachbarn. Die Änderung von $Variante_1$ kann nicht automatisch nach $Variante_2$ synchronisiert werden, und die Änderung von $Variante_2$ kann auch nicht automatisch nach $Variante_1$ synchronisiert werden. Diese Konflikt muss von Entwicklern per Hand gelöscht werden.

Falls solch ein Konflikt in einer Variante steht, ist es wahrscheinlich kein richtiger Konflikt. Das bedeutet, dass eine Variante wird zuerst an Zeile 4 und danach an Zeile 5 geändert wird. Die zwei Änderungen können nach zeitlicher Reihenfolge nach anderen Varianten synchronisiert werden. Es braucht auch Benutzereingriff,

```
1 public class Main {
2     public static void main(String[] args){
3         System.out.print("Information: ");
4         System.out.print(" wird ausgedruckt!");
5         ....
6     }
7 }
```

Fragment A

Basiscode

```
1 public class Main {
2     public static void main(String[] args){
3         System.out.print("Information: ");
4         System.out.print("Hello");      ← Änderung    +1 Zeile
5         System.out.print(" wird ausgedruckt!");
6         ....
7     }
8 }
```

Fragment B

$Variante_1$

```
1 public class Main {
2     public static void main(String[] args){
3         System.out.print("Information: ");
4         System.out.print(" world!");     ← Änderung    +1 Zeile
5         System.out.print(" wird ausgedruckt!");
6         ....
7     }
8 }
```

Fragment C

$Variante_2$

Abbildung 3.9: Beispiel für einen Reihenfolgekonflikt

```
1 public class Main {
2     public static void main(String[] args){
3         ....
4         int a=c*b;
5         for(int i=b;i<a;i++){
6         ....
7         }
8     }
9 }
```

Fragment A

Basiscode

```
1 public class Main {
2     public static void main(String[] args){
3         ....
4         int a=c*b++;          ← Änderung Zeile 4
5         for(int i=b;i<a;i++){
6         ....
7         }
8     }
9 }
```

Fragment B

$Variante_1$

```
1 public class Main {
2     public static void main(String[] args){
3         ....
4         int a=c*b;
5         for(int i=++b;i<a;i++){   ← Änderung Zeile 5
6         ....
7         }
8     }
9 }
```

Fragment C

$Variante_2$

Abbildung 3.10: Beispiel für einen Abhängigkeitskonflikt in zwei Varianten

weil manchmal es keine Abhängigkeit zwischen zwei Änderungen gibt und nur eine Änderung nach anderen Varianten synchronisiert werden soll.

```
1  public class Main {
2      public static void main(String[] args){
3          System.out.println("Hello world!");
4      }
5  }
```

Fragment A

$Revision_n$

```
1  public class Main {
2      public static void main(String[] args){
3          System.out.println("Hello world!");
4          printInfo();                    Änderung  +Zeile
5      }
6      public static void printInfo(){     Änderung  +Zeile
7      }                                   Änderung  +Zeile
8  }
```

Fragment B

$Revision_{n+1}$

```
1  public class Main {
2      public static void main(String[] args){
3          System.out.println("Hello world!");
4          printInfo();                                Änderung  +Zeile
5      }
6      public static void printInfo(){
7          System.out.println(Thread.currentThread().getName());
8      }
9  }
```

Fragment C

$Revision_{n+2}$

```
1  public class Main {
2      public static void main(String[] args){                Änderung  +Zeile
3          System.out.println("Hello world!");
4          printInfo();
5          System.out.println(System.currentTimeMillis()+"ms");
6      }
7      public static void printInfo(){
8          System.out.println(Thread.currentThread().getName());
9      }
10 }
```

Fragment D

$Revision_{n+3}$

Abbildung 3.11: Beispiel für einen Abhängigkeitskonflikt in derselber Variante

Die Abbildung 3.11 zeigt die Revisionen von einer Variante. Die Änderung zwischen $Revision_n$ (Fragment A) und $Revision_{n+1}$ (Fragment B) $\Delta_{n,n+1}$ hat einen Abhängigkeitskonflikt mit $\Delta_{n+1,n+2}$. $\Delta_{n+1,n+2}$ hat beruht auf $\Delta_{n,n+1}$. Um $\Delta_{n+1,n+2}$ nach anderen Varianten zu synchronisieren müssen wir zuerst $\Delta_{n,n+1}$ synchronisieren. $\Delta_{n+2,n+3}$ hat auch einen Konflikt mit $\Delta_{n,n+1}$. Bei Zeile 4 und 5 sind sie Nachbarn. Zeile 5 hat vielleicht eine Abhängigkeit mit Zeile 4. Es kann aber auch wahrscheinlich einzeln synchronisiert werden. Das soll der Benutzer selbst entscheiden.

(3) Überdeckungskonflikt. Es steht auch in verschiedenen Varianten. In den Varianten werden gleiche Zeilen von Quellcodes gelöscht und verschiedene Zeilen hinzugefügt.

```
1  ....
2  public static void printInfo(){
3          System.out.println("Hello world!");
4  }
5  ....
```
Fragment A

Basiscode

```
1  ....
2  public static void printInfo(){
3          System.out.println("Hola mundo!");  ← Änderung Zeile 3
4  }
5  ....
```
Fragment B

$Variante_1$

```
1  ....
2  public static void printInfo(){
3          System.out.println("Hallo Welt!");  ← Änderung Zeile 3
4  }
5  ....
```
Fragment C

$Variante_2$

Abbildung 3.12: Beispiel für einen Überdeckungskonflikt

Die Abbildung 3.12 zeigt einen Überdeckungskonflikt. $Variante_1$ wird einen Text auf Spanisch ausdrucken. $Variante_2$ wird einen Text auf Deutsch ausdrucken. Die Änderungen machen die Varianten sich unterschieden. Es soll nicht synchronisiert werden. Aber falls die Änderungen sind um eine Funktion zu verbessern, können wir eine Änderung auswählen, die mit effizienteren Algorithmus ist, und nach anderen Varianten synchronisieren. Das muss auch von Benutzer entschieden werden.

Alle Änderungen des Quelltextes, die keine obengenannte Konflikte haben, können automatisch synchronisiert werden. Hinzufügungen und Löschungen von Dateien können auch Konflikte haben. Aber es ist einfacher zu erkennen. Wenn eine Datei in einer Variante hinzugefügt wird, und die synchronisierte Variante auch eine Datei, die gleichen Name hat, gibt es ein Konflikt. Wenn eine Datei in einer Variante gelöscht wird, und die synchronisierte Variante keine Datei, die gleichen Name hat, gibt es auch ein Konflikt. Wenn die Hinzufügungen und Löschungen keine Konflikte haben, können auch automatisch synchronisiert werden. Eine Umbenennung kann als eine Löschung und eine Hinzufügung zusammen gehandelt werden.

3.2.3 Domänenwissen

Änderungen, die Konflikte haben, können nicht automatisch synchronisiert werden. Für die Synchronisierung ist Benutzereingriff erforderlich. Änderungen, die automa-

tisch synchronisiert werden können, sind in mancher Situation nicht für alle anderen Varianten anwendbar. Ob eine Änderung in einer Variante für andere Varianten anwendbar ist, muss erst erkannt werden. Um dies besser zu verstehen, schauen wir mal ein Beispiel an.

```
1  ....
2  public class TextMessage implements Serializable {
3      ...
4
5      private String msgHead;
6      private String content;
7
8      public TextMessage(String content) {
9      ...
10     }
11 ....
```

Fragment A

Feature-Auswahl {Chat_basis,UI,Gui}

```
1  ....
2  public class TextMessage implements Serializable {
3      ...
4
5      private String msgHead;
6      private String content;
7      private Color farbe;                    ← Änderung +Zeile
8
9      public TextMessage(String content) {
10     ...
11     }
12 ....
```

Fragment B

Feature-Auswahl{Chat_basis,UI,Gui,Farbe}

```
1  ....
2  public class TextMessage implements Serializable {
3      ...
4
5      private String msgInfoHead;             ← Änderung Umbenennung
6      private String content;
7
8      public TextMessage(String content) {
9      ...
10     }
11 ....
```

Fragment C

Feature-Auswahl{Chat_basis,UI,Gui,Verschluesselung,ROT13}

Abbildung 3.13: Vereinfachte Synchronisierung durch Domänenwissen

Die Abbildung 3.13 zeigt die Fragmente der Quelltexte von Varianten von einem Chatprogramm (siehe Abschnitt 3.2). Mit Feature *Farbe* können Benutzer die Nachrichten eine Farbe geben, die später für die Empfänger gezeigt wird. Um die Farbe für jede Nachricht zu speichern, wird eine Variable für Klass TextMessage hinzuge-

fügt (sehe Fragment B). Die Änderung hat keinen vorher gesagten Konflikt. Es kann automatisch nach alle anderen Varianten synchronisieren. Aber es soll das nicht. Die Änderung soll nur nach den Varianten synchronisieren, die Feature *Farbe* haben. Die Umbenennung von der Variable *msgHead* (sehe Fragment C) hat auch keinen Konflikt. Es kann auch automatisch nach alle anderen Varianten synchronisieren. Und es soll auch nach alle anderen Varianten synchronisieren. Denn es gehört Feature *Chat_basis* und *UI* zu. Jede Variante von Chatprogramm muss Feature *Chat_basis* und *UI* haben. Es ist offensichtlich, dass die Änderungen nicht nach alle anderen Varianten synchronisieren sollen. Die Varianten sollen sich unterschiedlich bleiben.

Wir möchten die Änderungen von einer Variante nach anderen Varianten synchronisieren. Es kann die Entwicklungsaufwand von Varianten reduzieren. Aber für die Synchronisierung müssen wir wissen, ob eine Änderung nach einer Variante synchronisieren soll. Die Änderungen sind nicht immer nach alle anderen Varianten zu synchronisieren gewünscht. Das Beispiel aus der Abbildung 3.13 auf der vorherigen Seite hat es schon erklärt. Wir können Domänenwissen anwenden, um die Synchronisierung zu entscheiden. Dafür wird eine Domäneanalyse für den Bereich der entwickelte Varianten gemacht. Eine Feature-Modell wird für den Bereich entwickelt. Für jede Variante können wir eine Konfiguration-File erstellen. Mit Konfiguration und Feature-Modell können wir für jede Variante wissen, dass welches Feature diese Variante haben. Die Änderungen werden auch nach Feature sotieren. Bei einer Synchroniesierung muss der Benutzer wissen, dass welchem Feature die Änderung zugehört. Wir synchronisieren eine Änderung nach alle Varianten oder ein paar Varianten, die der Feature der Änderung enthalten.

Weil die Änderungen werden nach Features sortiert, ist die Frequenz des Speichern von Quelltexten sehr wichtig. Falls ein Benutzer hat ein Quelltext für mehrere Features geändert, ist die Änderung meistens zu groß für Synchronisierung. Es kann nur nach den Varianten synchronisieren, die die betroffene Features enthalten. Wir können noch mal Beispiel aus der Abbildung 3.13 auf der vorherigen Seite anschauen. Falls eine Änderung bei der Variante, die Feature *Farbe* hat, hat die Variable *msgHead* umbenannt und gleich die Variable für Feature *Farbe* hinzugefügt. Dann kann die Umbenennung nur nach den Varianten synchronisieren, die Feature *Farbe* haben. Denn außerdem Umbenennung in der Änderung handelt es auch Feature *Farbe*. Aber die Umbenennung soll nach alle anderen Varianten synchronisieren. Deshalb ist diese Änderung zu groß. Manchmal muss eine Änderung für mehrere Features sein. Weil manche Features haben Abhängigkeit oder Konflikt von anderen Features. Zum Beispiel muss eine Variante von Chatprogramm, die Feature *Verschluesselung* und *Historie* gleichzeitig hat, einen Vorgang für Entschlüsselung der Nachricht in der Funktion für Historie haben. Die Synchronisierung der Änderungen für die Entschlüsselung ist nur für die Varianten möglich, die Feature *Verschlues-*

selung und *Historie* gleichzeitig enthalten. Bei der Synchronisierung muss Benutzer genau wissen, welche Feature wird eine Änderung behandelt. Sonst können wir nicht entscheiden, ob eine Änderung nach einer Variante synchronisiert werden soll.

Bei einer Langzeitnutzung von VariantSync können wir die Sortierung von Änderungen aufzeichnen. Weil für jede Synchronisierung muss der Benutzer Features für die Änderung bestimmen. Wir können wissen, welche Änderungen für eine bestimmte Feature sind. Falls wir alle Änderungen für die Entwicklung eines Features aufgezeichnet haben, können wir theoretisch ein Feature nach einer Variante synchronisieren. Das bedeutet, dass wir Features nach unserem Wunsch mischen können. Nach den Synchronisierungen für bestimmte Features können wir neue Variante erstellen. Außerdem können wir noch Zeilen von Quelltexte Features zuordnen. Bei der Synchronisierung sollen wir die Zeilen in den Quelltexte ändern und hinzufügen, dabei können wir die Information von Features für die Zeilen extra speichern. Denn um SPL einzusetzen ist immer noch eine große Herausforderung und das Risiko für ein Unternehmen [KDO11], können wir mit den Informationen bei der Einsetzung einer SPL profitieren. Zum Beispiel können wir mit der Information der Features die Extraktion der Features vereinfachen.

3.3 Zusammenfassung

Am Anfang dieses Kapitels werden durch ein paar Beispiele dargestellt, was Variante ist, warum sie so sinnvoll für die Softwareentwicklung. Ein paar Ansätze für die Entwicklung der Varianten werden durch Beispiele dargestellt. Daraus haben wir bemerkt, dass die zurzeit bestehende Ansätze ungenügend sind. Aus diesem Grund wird die neue Methode „Synchronisierung von Software-Varianten“ dargestellt. Nach der Idee werden die Änderungen für alle Quelltexte bei der Entwicklung gespeichert. Für jede Änderung vergleichen wir alle Varianten um die gleiche Fragment von den Varianten zu suchen. Und synchronisieren wir die Änderung in den mögliche Varianten. Die Synchronisierung ist nicht für alle Varianten gültig. Dafür benutzen wir noch Domänewissen für die Varianten um es zu entscheiden, ob die Änderung in einer Varianten synchronisiert werden soll. Außerdem kontrollieren wir noch die Änderungen, ob ein Änderung ein Konflikt für die Synchronisierung hat. Zur zeit synchronisieren wir nur die Änderungen, die keine Konflikt haben. Diese Idee wird für das Projekt entworfen, das wenige Varianten haben. Für das Projekt, das eine gewissen Zahl von Varianten hat, ist diese Methode nicht rentabel. Aber mit dem fortwährende Einsetzen der Methode für die Entwicklung eines Projektes ermöglicht eine einfache Migration zu SPL. Weil bei den Synchronisierungen wird Domänewissen benutzt. Wir können bei den Synchronisierungen ein paar Informationen für Features des Quelltextes speichern. Später können Entwickler bei Migration zu SPL eines Projektes von den Informationen profitieren.

4. Prototypische Implementierung VariantSync

In diesem Kapitel wird die prototypische Implementierung VariantSync beschrieben. Dafür werden zunächst die erforderlichen Werkzeuge vorgestellt. Zuerst wird die Entwicklungsumgebung Eclipse vorgestellt. Für die Unterstützung des Featuremodells wird FeatureIDE eingeführt. Danach wird eine Programmbibliothek vorgestellt, die bei der Implementierung für VariantSync benutzt wird. Anschließend wird die prototypische Implementierung ausführlich erklärt.

4.1 Benutzte Werkzeuge

4.1.1 Eclipse

Eclipse[1] ist ein quelloffenes Programmierwerkzeug. Ursprünglich wurde Eclipse von IBM als Entwicklungsumgebung für die Programmiersprache Java entwickelt. Ziel war es, IBM Visual Age for Java (eine Integrierte Entwicklungsumgebung von IBM) abzulösen.

Eclipse hat eine sehr gute Erweiterbarkeit. Durch das Laden von Plug-ins können viele verschiedene Funktionalitäten zur Verfügung gestellt werden. Eclipse ist eine Plattform für viele Möglichkeiten. Derzeit gibt es ca. 2.9 Mill. Plug-ins für Eclipse. Die Entwicklung für Java-Programme ist nur eine spezielle Anwendungsmöglichkeit dieser Plattform. Der Kern von Eclipse ist sehr klein. Es ist nur so groß und kompliziert, dass es die Plug-ins laden und lassen kann. Ein Plug-in ist ein Satz von Software-Komponenten, die bestimmte Funktionalität zu einer großen Softwareanwendung hinzufügt. Sowohl Eclipse als auch die Plug-ins sind vollständig mit Java

[1] http://www.eclipse.org/

entwickelt werden. Ursprünglich wird Eclipse als Java-IDE (Abkürzung IDE, von engl. integrated development evironment) verwendet, aber mit weiteres Plug-ins für andere Programmiersprachen kann Eclipse auch als eine IDE für viele Programmiersprachen (z.B. C, C++, Python, C#, PHP) sein.

Als IDE bietet Eclipse Texteditor, Compiler bzw. Interpreter und Debugger für Programmiersprachen. Damit können Entwickler die Aufgaben der Softwareentwicklung ohne Unterbrechungen mit einem Werkzeug arbeiten. Außerdem bietet Eclipse auch viele nützliche Funktionen. Zum Beispiel bietet der Java-Editor Syntaxhervorhebung, Quelltextvervollständigung, Quelltextformatierung, Quelltextassistenz und mehrere Ansichten, um die Struktur des Quelltextes zu zeigen. Es zeigt auch Fehlerhinweise an fehlerhaften Zeilen. Manchmal können Entwicklern ein paar Vorschlägen von Eclipse bekommen, um die Fehler zu beheben. Mit der Quelltextformatierung können Entwicklern leicht Strukturfehler entdecken. Mit der Quelltextvervollständigung müssen Entwicklern weniger Quelltext eintippen. Zum Beispiel für lange Variablennammen brauchen Entwickler nur ein Teil davon eintippen. Der Rest wird automatisch ergänzt. Mit Eclipse können Entwickler effizient entwickeln. Denn Eclipse bietet viele nützliche Funktionen und automatisiert banale und zeitfressende Operationen.

Eclipse wird für viele verschiedene Systeme und Architekturen bereitgestellt. Eclipse stellt SWT- und JFace-Bibliotheken Alternativen zu Suns AWT- und Swing-Bibliotheken zur Verfügung. Mit SWT- und JFace-Bibliotheken implementierte Benutzeroberflächen verhalten sich wie native Anwendungen. Denn SWT- und JFace-Bibliotheken sind auf Basis von nativer Implementierung entwickelt. Für verschiedene Systeme läuft Eclipse mit unterschiedlichen Aussehen.

Für die prototypische Implementierung von VariantSync wählen wir Eclipse als Plattform und programmieren mit Java. Das ist nicht notwendig. Für die Wahl gibt es folgende Gründe: 1. Mit Plug-ins kann Eclipse viele Programmiersprachen unterstützen. Die Idee für VariantSync ist kompatibel zu allen Programmiersprachen, weil es nur auf Textdokumenten operiert. Wir implementieren ein Plug-in von Eclipse für VariantSync, danach können wir mit VariantSync leicht mit vielen Programmiersprachen arbeiten. 2. Eclipse ist eine hervorragende Entwicklungsumgebung. Wir können mit Eclipse effizient VariantSync implementieren. Eclipse wird für viele verschiedene Systeme und Architekturen bereitgestellt. Die Implementierung von VariantSync mit Eclipse kann auch leicht auf anderen Systeme und Architekturen laufen. 3. Es gibt ein paar Projekt für Software-Produktlinien, die als Plug-in für Eclipse entwickelt wurden. Später können sie dadurch leicht kombiniert werden. Zum Beispiel wird die Funktionalität des Featuremodells von FeatureIDE in der Implementierung von VariantSync verwendet (Es wird in 4.3 im Detail beschrieben).

4.1.2 FeatureIDE

FeatureIDE ist ein quelloffenes Werkzeug für Feature-orientierten Softwareentwicklung und wurde als Plug-in für Eclipse entwickelt [TKB+12]. FeatureIDE unterstützt alle Phasen des Feature-orientierten Softwareentwicklung und hat alle Entwicklungsprozess und benötigte Werkzeugen in der Entwicklungsumgebung integriert. Die Architektur von FeatureIDE erleichtert die Entwicklung von Unterstützung für bestehende und neue Programmsprachen für Feature-orientierten Softwareentwicklung und somit reduziert sich der Aufwand um neue Sprachen und Konzepte zu unterstützen. Jetzt konzentriert sich die Entwicklung von FeatureIDE in akademischen Umfeld.

Der erste Portotype von FeatureIDE wurde im Jahr 2005 erstellt. Es wird ständig erweitert und verbessert. Zurzeit werden viele Produktlinientechnologien sowie Feature-orientierte Programmierung (FOP), Aspekt-orientierte Programmierung (AOP), Delta-orientierte Programmierung (DOP), Päprozessor mit der Unterstützung des Feature-Modells in FeatureIDE intergriert. Für Feature-Modell wird ein graphischer Editor von FeatureIDE geboten. Damit ist es einfach um ein Feature-Modell zu entwickeln. Es gibt noch ein graphischen Editor für Konfiguration von Feamtur-Modell, mit dem können wir eine Konfiguration erstellen und validieren. Für *VariantSync* wird die Funktionalität des Feature-Modells von FeatureIDE verwendet. FeatureIDE hat viele Funktion und wird von viele Plug-ins zusammen gebaut. Für *VariantSync* brauchen wir vier Plug-ins davon: *de.ovgu.featureide.core*, *de.ovgu.featureide.ui*, *de.ovgu.featureide.fm.core* und *de.ovgu.featureide.fm.ui*.

4.1.3 Diff Utils

Um die Varianten zu synchronisieren müssen wir die Varianten vergleichen. Außerdem sollen wir auch die neue Dateien und alte Dateien vergleichen, damit können wir wissen, wie die Dateien geändert werden. Dadurch können wir die Änderungen speichern. Änderungen zwischen zwei Textdateien herauszufinden ist in der Praxis schon lange bekannt. Ursprünglich ist es ein Unix-Programm Diff. Es wird mit Programmiersprache C implementiert. Jetzt können wir viele Implementierungen auf verschiedene Programmiersprachen finden. Für VariantSync implementieren wir nicht noch mal Diff. Wir benutzten die schon existierende Programmbibliothek: „ java-diff-utils“.

Die Programmbibliothek „ java-diff-utils“ ist eine quelloffene Programmbibliothek. Es ist unter der Lizenz „Apache License 2.0“ verfügbar. Es wurde auf der *Google Code* veröffentlicht.[2] Mit der statische Methoden von der Klasse *DiffUtils* können wir einfach folgende Operationen durchführen:

[2] http://code.google.com/p/java-diff-utils/

Diff erzeugen

Methode: DiffUtils.diff(List<?> original,List<?>revised):Patch patch

DiffUtils.generateUnifiedDiff(String fname1, String fname2,List<String> originalLines,Patch patch, int contextSize):List<String> unifiedDiff

Mit der Methode *diff* können wir ein Objekt *Patch* bekommen. Das Objekt *Patch* beschreibt, wie der originale Quelltext nach dem revidierten Quelltext geändert werden kann. Der Inhalt von zwei Quelltext wird in zwei *Listen* gepackt und als Parameter der Methode gegeben. Mit der Methode *generateUnifiedDiff* können wir von einem *Patch* ein *unified Diff* (siehe Abschnitt 2.2.1) bekommen. Für Parameter brauchen wir die Namen von zwei Quelltext, der in *Liste* gepackte originale Quelltext, der *Patch* und die Zahl der Zeilen, die vor und hinter die Änderungen stehend sind.

Patch

Methode: DiffUtils.patch(List<?> original, Patch patch): List<?> revised

Mit der Methode *patch* können wir den Quelltext nach der Änderungen ändern, die von dem gegebenen Objekt *patch* beschrieben sind. Die Eingabe der Methode sind ein Objekt *List*, der originaler Quelltext gepackt hat, und ein Objekt *patch*. Das Ergebnis der Methode ist ein Objekt *List*, der geänderte Quelltext gepackt hat.

Unpatch

Methode: DiffUtils.unpatch(List<?> revised, Patch patch): List<?> original

Mit der Methode *unpatch* können wir gegen die Methode *patch* die Änderungen von einem Quelltext rückgängig machen. Die Eingabe der Methode sind ein Objekt *List*, der geänderte Quelltext gepackt hat, und ein Objekt *patch*. Das Ergebnis der Methode ist ein Objekt *List*, der originaler Quelltext gepackt hat.

Diff analysieren

Methode: DiffUtils.parseUnifiedDiff(List<String> diff): Patch patch

Mit der Methode *parseUnifiedDiff* können wir ein *unified Diff* analysieren und ein Objekt *patch* bekommen. Nach dem Vergleich von zwei Quelltext mit der Methode *diff* bekommen wir ein Objekt *patch*, der der Unterschied zwischen zwei Quelltext beschrieben hat. Wenn wir der Unterschied speichern möchten, sollen wir der Unterschied nach eine Textform transformieren. Wir speichern nicht den Objekt ptach sondern eine Textform (*unified Diff*) in eine Datei. Später wenn wir der Unterschied noch mal handeln möchten, analysieren wir ein *unified Diff* und erzeugen einen passende Objekt *patch*. Die Eingabe der Methode sind ein Objekt *List*, der der Inhalt von einem *unified Diff* gepackt hat.

Mit oben beschriebenen Methoden können wir Quelltext vergleichen und ändern. Für die Synchronisierung sollen wir uns noch die Klasse *Delta* beschäftigen. Die Klasse wird von der Bibliothek geboten und die einzelne Änderung beschrieben. Bei Synchronisierung benutzen wir es um Konflikte zu erkennen. Das wird genauer im nächsten Abschnitt beschrieben.

4.2 VariantSync

Die prototypische Implementierung VariantSync wird als ein Plug-in für Eclipse entwickelt. Dank der Plug-in-Architektur von Eclipse ist VariantSync für viele Programmiersprachen verfügbar. Die ganze Entwicklung wird mit Eclipse durchgeführt. Die Grundkenntnisse für Benutzung von Eclipse, Erstellung des Projektes, Konfiguration usw. werden hier nicht berücksichtigt.

4.2.1 Änderung des Quelltextes überwachen

Wie in Abschnitt 3.3 geschrieben, sollen wir die Änderungen von allen Varianten speichern, um die Varianten später synchronisieren zu können. Die Hinzufügungen, Löschungen und Änderungen des Quelltextes sollen für jede von VariantSync unterstützte Variante (hier mit Eclipse entwickelnde Projekte) speichern. Dafür müssen wir die Dateien von den Projekte überwachen, um die Änderungen zu bemerken und speichern.

Für Eclipse-Plug-ins ist es einfach die Änderungen der Dateien zu überwachen. Wir können einen Listener benutzen. Mit dem Listener können wir Nachrichten von dem Eclipse bekommen, wenn die Ordner und Dateien geändert werden. Der Listener muss das Interface *IResourceChangeListener* implementieren. Außerdem müssen wir noch mit der Methode *addResourceChangeListener* von *IWorkspace* der Listener bei Eclipse registrieren.

```
addResourceChangeListener(IResourceChangeListener listener, int eventMask)
```

Damit können wir die Nachrichten der Änderungen von Eclipse bekommen. Bei der Registrierung können wir mit den zweiten Parameter von der Methode *addResourceChangeListener* entscheiden, welche Nachrichten der Listener bekommen kann. Da Ressourcenänderungen im Eclipse relativ häufig sind, sollen wir die Nachrichten filtern. Es kann Nachrichten von Eclipse für Listener erzeugen, wenn ein Projekt öffnen und aktualisieren, und vor der Kompilierung, nach der Änderung usw. Für VariantSync kümmern wir uns nur um die Nachrichten, die für gemachte Änderungen von Dateien erzeugt werden. *IResourceChangeEvent.POST_CHANGE* wird als zweiter Parameter der Methode *addResourceChangeListener* ausgewählt.

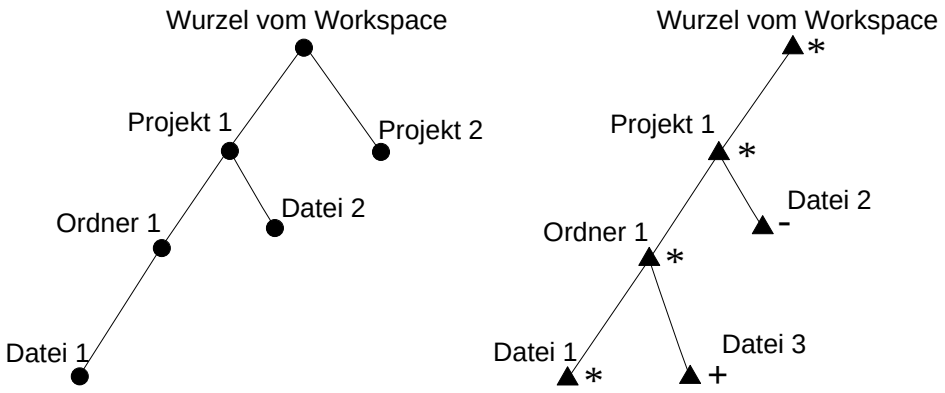


Abbildung 4.1: Dateien und deren Änderungen in Eclipse

Jetzt haben wir den Listener mit der Methode *addResourceChangeListener* bei Eclipse registriert. Wir müssen noch die Methode *resourceChanged* von Interface *IResourceChangeListener* für unseren Listener implementieren. Damit können wir die Nachrichten behandeln. Die Methode *resourceChanged* wird von Eclipse aufgerufen, wenn eine Änderung gemacht wird. Ein Objekt, der Interface *IResourceChangeEvent* implementiert, wird als Eingabe der Methode gegeben. Dieses Objekt beschreibt eine Änderung mit mehreren Objekte, die Interface *IResourceDelta* implementieren. Die *ResourceDelta-Objekte* haben eine Baum-Struktur, die eine Änderung beschrieben. Der Baum von *ResourceDelta-Objekte* ist ähnlich wie die Struktur von *IResource-Objekte*, aus denen der Workspace von Eclipse strukturiert und die im Workspace stehende Ordner und Dateien beschrieben werden.

Die Abbildung 4.1 zeigt ein Beispiel für Baum-Struktur von den Dateien und den ResourceDelta-Objekte. Der linke Baum beschreibt die Dateien von dem Workspace. Wir haben zwei Projekte. In dem *Projekt 1* haben wir einen Ordner und eine Datei. In dem *Ordner 1* gibt es noch eine Datei. Jetzt machen wir folgende Änderungen: Wir löschen *Datei 2*, änderen *Datei 1*, und fügen eine neue Datei *Datei 3* in dem *Ordner 1* hinzu. Die *ResourceDelta-Objekte*, die die Änderungen geschrieben haben, haben eine Baum-Struktur wie der rechte Baum. Die Figur ▲ beschreibt *ResourceDelta-Objekte*. Neben ▲ stehendes Symbol + bedeutet, dass dieses *ResourceDelta-Objekt* ein Ergebnis (IResourceDelta.ADDED) von den Methode *getKind* hat. Symbol * ist für IResourceDelta.CHANGED. Symbol - ist für IResourceDelta.REMOVED.

Mit der Methode *getKind* von einem *ResourceDelta-Objekt* können wir wissen, wie die Dateien geändert wurden. Wir handeln drei Typen : IResourceDelta.ADDED, IResourceDelta.REMOVED, und IResourceDelta.CHANGED. Sie beschreiben die Hinzufügung, Löschung und Änderung von Dateien. Es gibt noch eine wichtige Methode von *ResourceDelta-Objekte*: *getFlags*. Mit dem Ergebnis von *getFlags* können wir mehr Details von den Änderung bekommen. Zum Beispiel haben wir einen Quell-

text geändert und ein Mark für einen Quelltext hinzufügt. In Eclipse können wir ein Mark für einen Quelltext hinzufügen. Es kann für Lesezeichen, Task und Anhaltepunkt sein. Aber die Textinhalt wird nicht geändert. Trotzdem bekommen wir IResourceDelta.CHANGED von der Methode *getKind* von *ResourceDelta-Objekte*. Wir möchten nur die Änderung aufzeichnen, die Textinhalt geändert hat. Mit der Methode *getFlags* können wir das kontrollieren. Für die Änderungen, die Textinhalt geändert haben, soll „ResourceDelta.getFlags() & IResourceDelta.CONTENT“ nicht Null sein. Außerdem müssen wir noch Methode *isDerived* von *IResource-Objekte* benutzen. Von *ResourceDelta-Objekte* können wir ein Objekt mit Methode *getResource* bekommen. Dieses Objekt beschreibt die behandelte Datei. Falls diese Datei von Eclipse selbst erzeugt wird, ist das Ergebnis von *isDerived* TRUE. Für ein Projekt kann Eclipse automatisch den Quelltext compilieren und neue Dateien erzeugen. Die Änderungen von solche Dateien sollen nicht aufzeichnen werden.

Jetzt haben wir die Nachrichten feiner gemacht. Aber es ist noch nicht genug. Wir bekommen die Nachrichten von Änderungen für jeder Projekt in dem Workspace von Eclipse. Wir sollen es noch feiner machen. Wir handeln nur die Nachrichten von Änderungen für die Projekte, die von VariantSync unterstützt werden. Dafür können wir Project-Nature benutzen.

4.2.2 Project-Nature und Decorator

Mit Project-Nature können wir in Eclipse eine Assoziation zwischen einem Projekt und bestimmten Funktionen definieren. Wir können für bestimmte Projekte bestimmte Builder feststellen. Außerdem wird Project-Nature in vielen Fällen genutzt um bestimmte Funktionalitäten in Anhängigkeit vom jeweiligen Typ eines Projektes zu steuern. Im VariantSync machen wir eine Filterung für verschiedene Menüs, die je nach Project-Nature unterschiedlich sind. Für die Projekte, die von VariantSync unterstützt und nicht unterstützt werden, bekommt man verschiedenes Menü. Ein Projekt kann mehrere Natures haben. Ein Eclipse-Plug-In-Projekt hat gleichzeitig eine Java-Nature und eine Plug-In-Nature.

Die Information für Project-Nature wird in der Datei *.project* gespeichert. Jedes Projekt im Eclipse hat ein *.project* um ein paar Information für Projekt zu speichern. Unten zeigt es den Inhalt von einem Projekt. Dieses Projekt ist ein Java-Projekt und wird von VariantSync unterstützt. Deshalb hat dieses Projekt zwei Natures, eine Java-Nature und eine von VariantSync gegebene Nature (siehe Zeile 13 bis 16).

Project-Nature funktioniert als eine Markierung für eine Eigenschaft eines Projektes. Mit der Methode *hasNature(String natureId)* können wir wissen, ob ein Projekt eine bestimmte Eigenschaft hat. Für die Projekte, die von VariantSync unterstützt werden, werden ein Nature (*natureId*=„de.ovgu.variantsync.syncNature“) gegeben.

```
<projectDescription>
    <name>Test</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.jdt.core.javanature</nature>
        <nature>de.ovgu.variantsync.syncNature</nature>
    </natures>
</projectDescription>
```

Quelltext 4.1: Inhalt von der Datei .project

Wenn wir bei solche Projekte die Methode *hasNature* anrufen (obengenannte *natureId* wird als Eingabe gegeben), bekommen wir TRUE als Ergebnis. Diese Methode ist nützlich für die Überwachung von Änderungen, die in letzten Abschnitt geschrieben wird. Wenn wir eine Nachricht für Änderung bekommen, kontrollieren wir das betroffene Projekt mit der Methode *hasNature*, ob das Projekt von VariantSync unterstützt wird. Die Nachricht für nicht unterstützte Projekt wird nicht weiter handeln.

Um eine Nature zu implementieren müssen wir eine Deklaration in Form eines Extension-Points in der Datei *plugin.xml* machen. Es besteht aus den Element Id, Anzeige-Name, Extension-Point für Nature „org.eclipse.core.resources.natures“ und Klassendatei mit Implementierung der Nature-Funktionalität. In 4.2 zeigt ein Fragment für die Deklaration der VariantSync-Nature von der Datei *plugin.xml*.

```
   <extension
         id="vsyncSupportProjectNature"
         name="Variantsync Support Project Nature"
         point="org.eclipse.core.resources.natures">
      <runtime>
         <run
               class="de.ovgu.variantsync.SyncNature">
         </run>
      </runtime>
   </extension>
```

Quelltext 4.2: Deklaration für die VariantSync-Nature in der Datei plugin.xml

Die für VariantSync-Nature erstellte Klasse *SyncNature* muss das Interface *IProject-Nature* implementieren. In den Methode *configure* und *deconfigure* von dem Interface *IProjectNature* können wir ein paar Implementierungen für ein bestimmtes Verhalten erstellen. Wenn eine Nature für ein Projekt hinzugefügt und entfernt wird, werden die zwei Methode aufgerufen. Zum Beispiel können wir in die Methode *deconfigure* ein paar Implementierungen erstellen um die für VariantSync erzeugte Information (z.B gespeicherte Änderungen) zu löschen, wenn ein Projekt nicht mehr von VariantSync untertützt wird.

Um die Natures von einem Projekt zu konfigurieren brauchen wir noch die Methode *getDescription* und *setDescription*. Bei einem Projekt können wir die Methode *getDescription* anrufen. Das Ergebnis davon ist ein Objekt, das Interface *IProjectDescription* implementiert hat. Dieses Objekt enthält die Meta-Daten, die ein Projekt definieren. Bei dem Objekt rufen wir noch die Methode *getNatureIds* auf, dann können wir ein Array bekommen, das aus die NatureIds besteht. Wir können ein NatureId in dem Array hinzufügen oder ein NatureId von dem Array entfernen, danach rufen wir die Methode *setNatureIds* mit dem geänderte Array an, um das IProjectDescription-Objket zu konfigurieren. Zum Schluss rufen wir noch *setDescription* auf mit dem konfigurierte IProjectDescription-Objekt. Dadurch können wir die Natures von einem Projekt konfigurieren.

Mit Project-Nature können wir ein Mark ein Projekt geben, damit können wir kontrollieren, ob ein Projekt von VariantSync unterstützt wird. Aber man muss das durch das Ergebnis von der Methode *hasNature* erkennen. Oder kann man auch das durch das Inhalt von der Datei *.project* erkennen (siehe Listing 4.1, Zeile 13 bis 16). So ist es nicht nutzerfreundlich. Die von VariantSync unterstützte Projekte und normale Projekte sehen in der graphischen Benutzeroberfläche gleich aus. Wir können das mit Decorator verbessern.

Mit Decorator können wir visuelle Hinweise, nützliche Statusinformationen mit Objekten oder Ressourcen in Eclipse Views anzeigen. Abbildung 4.2 auf der nächsten Seite zeigt ein Beispiel für einen Decorator. Das Projekt 1 ist ein normales Java-Projekt. Mit Decorator wird ein Zeichen für „J“ bei rechts oben des Bildsymbols von dem Projekt hinzugefügt. Es hinweist darauf, dass dieses Projekt ein Java-Projekt ist. Das Projekt 2 ist ein FeatureIDE-Projekt, und es wird mit Java entwickelt. Es gibt bei links oben des Bildsymbols von dem Projekt ein ganz kleines Zeichen für „*f*“. Es beutet, dass das Projekt ein FeatureIDE-Projekt ist. Projekt 3 und 4 sind Java-Projekte, und sie werden von VariantSync unterstützt. Bei links unten des Bildsymbols wird ein Zeichen hinzugefügt. Mit diesem Zeichen kann man schnell erkennen, ob ein Projekt von VariantSync unterstützt wird.

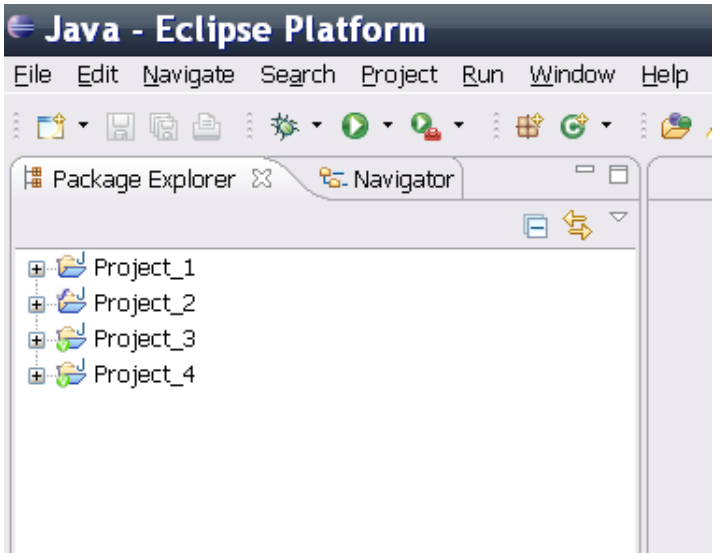


Abbildung 4.2: Der Project-Decorator von VariantSync

Für die Implementierung eines Decorators brauchen wir auch eine Deklaration in der Datei *plugin.xml*. Quelltext 4.3 zeigt ein Fragment für diese Deklaration.

```
1   <extension
2         point="org.eclipse.ui.decorators">
3      <decorator
4            adaptable="true"
5            class="de.ovgu.variantsync.VSyncSupportProjectDecorator"
6            id="de.ovgu.variantsync.VSyncSupportProjectDecorator"
7            label="VariantSync Support Project Decorator"
8            lightweight="true"
9            state="true">
10         <enablement>
11            <objectClass
12                  name="org.eclipse.core.resources.IProject">
13            </objectClass>
14         </enablement>
15      </decorator>
16   </extension>
```

Quelltext 4.3: Deklaration für den VariantSync-Decorator in der Datei plugin.xml

Extension-Point ist „org.eclipse.ui.decorators“. Der Parameter *objectClass* definiert, bei welche Objekte dieser Decorator funktionieren soll. Der Parameter *adaptable* wird mit *true* definiert. So kann dieser Decorator auch bei anderen Objekte (Hier ist die Objekte, die zu IProject-Objekte konvertieren können) funktioniert werden. Falls *adaptable* mit *false* definiert wird, wird dieser Decorator in Package-Explorer-View nicht funktionieren. Der Parameter *class* definiert die Klasse, die dieser Decorator implementiert. Der Parameter *lightweight* definiert, dass dieser Decorator ein

Lightweight-Decorator ist. Die Klasse, die dieser Decorator implementiert, soll das Interface *ILightweightLabelDecorator* implementieren. Der Parameter *state* definiert den Standard-Zustand des Decorators.

In die Klasse *VSyncSupportProjectDecorator*, die für unserer Decorator erstellt wird (es wurde definiert im Quelltext 4.3 auf der vorherigen Seite Zeile 5), sollen wir die Methode *decorate* überschreiben. Im Quelltext 4.4 zeigt diese Methode. Wir kontrollieren die Projekte mit der Methode *hasNature*. Wenn die Projekte von VariantSync unterstützt werden, können wir mit mit der Methode *addOverlay* ein Zeichen bei dem Bildsymbol hinzufügen. Der Parameter *OVERLAY* ist ein ImageDescriptor-Objekt, das ein Bild beschriebt. Der Parameter *IDecoration.BOTTOM_LEFT* definiert die Position des Zeichens. Es gibt auch andere Möglichkeiten wie *BOTTOM_RIGHT*, *TOP_LEFT* und *TOP_RIGHT*. Aber man muss das aufpassen, dass die Position am besten nicht mit der Position von anderen Zeichens gleich ist, sonst wird ein Zeichen bedeckt sein.

```
1  public void decorate(Object element, IDecoration decoration) {
2     IProject project = (IProject) element;
3     try {
4         if (project.isOpen()
5            && project.hasNature(SyncNature.NATURE_ID)) {
6          decoration.addOverlay(OVERLAY, IDecoration.BOTTOM_LEFT);
7         }
8     } catch (CoreException e) {
9         VariantSyncLog.logError(e);
10    }
11 }
```

Quelltext 4.4: Methode decorate() in der Klasse VSyncSupportProjectDecorator

4.2.3 Änderungen speichern

In Abschnitt 4.3.1 wurde das dargestellt, wie man die Änderungen von des Quelltextes überwachen kann. Wir können von Eclipse Nachrichten bekommen, wenn Quelltexte geändert werden. Jetzt müssen wir noch die Änderungen speichern, damit sie später noch benutzt werden kann.

Die Änderungen werden in Ordner *.variantsync* gespeichert. Wir stellen für jedes Projekt ein Ordner *.variantsync* direkt im Projekt-Ordner. Die Struktur von den Änderungen unter dem Ordner *.variantsync* ist gleich wie die Struktur vom Quelltext. Das bedeutet, dass der Ordnerpfad von den Quelltexte und von den Änderungen sich nur durch den Ordner *.variantsync* unterscheiden. Zum Beispiel wird eine Datei geändert. Der Ordnerpfad vom dieser Datei ist */Project_1/Ordner_1/Ordner_2/Test.java*. Dann wir speichern die Änderungen für diese Datei in Ordner */Project_1/.variants-*

ync/Ordner_1/Ordner_2/. Falls ein Ordner */Project_1/Ordner_1/Ordner_3* gelöscht oder erstellt wird, speichern wir die Änderungen in Ordner */Project_1/.variantsync/ Ordner_1/Ordner_3*. So können wir einfach die gespeichert Änderungen ordnen. Wenn wir es brauchen, können wir auch schnell es finden. Mit eine kleine Änderung bei dem Ordnerpfad von einem Quelltext können wir der Ordnerpfad der gespeicherte Änderungen für diesem Quelltext bekommen.

Für jedere Änderung, zum Beispiel die Löschung, Änderung (Für die Änderung wie Umbenennung wird es als eine Löschung und eine Erstellung behandelt), und Erstellung, erstellen wir eine Datei um die Änderung zu speichern. Für den Dateinamen stellen wir eine Regel auf. Der Dateiname wird aus zwei Teil gebaut. Der erste Teil ist der originale Name von der Datei oder dem Ordner, die die Änderung haben. Der zweite Teil enthält Informationen zur Änderung. Sie sind der Typ von der Änderung, ein Zeitstempel und ein Zeichen (Es wird als *1* oder *0* festgelegt). Mit dem Zeichen können wir wissen, ob die Änderung von Nutzer gemacht wird. Falls die Änderung von VariantSync durch Synchronisierung geführt wird, wird das Zeichen als *1* festgelegt, sonst wird es als *0* festgelegt. Alle Änderungen, die von Nutzer gemacht oder durch Synchronisierung geführt werden, werden gespeichert. Denn wir brauchen später sie noch für die Synchronisierung. Der Zeitstempel ist ein Wert, der mit der Mehtode *getLocalTimeStamp* von einem *IResource-Objekt* bekommt wird. Dieses *IResource-Objekt* beschreibt die geänderte Datei oder den geänderte Ordner. Mit dem Zeitstempel können wir die Änderung einen eindeutigen Zeitpunkt bekommen. Für eine Löschung von Datei oder Ordner können wir nicht mit der Mehtode *getLocalTimeStamp* einen sinnvollen Wert bekommen. Es ist immer *-1*. Bei diesem Fall benutzen wir die Methode *System.currentTimeMillis* anstelle von der der Mehtode *getLocalTimeStamp*. Für eine Änderung haben wir fünf Typen erstellt:

1. **FILE_CHANGED:** Es ist nur für die Dateien verfügbar, die der Inhalt geändert werden.
2. **FOLDER_REMOVED:** Wenn ein Ordner gelöscht wird, wird es benutzt.
3. **FILE_REMOVED:** Wenn eine Datei gelöscht wird, wird es benutzt.
4. **FOLDER_ADDED:** Wenn ein Ordner erstellt wird, wird es benutzt.
5. **FILE_ADDED:** Wenn eine Datei erstellt wird, wird es benutzt.

Der originale Name, der Typ von der Änderung, das Zeichen und der Zeitstempel werden mit „_“ verbunden. Zum Beispiel hat eine Datei, die um eine Änderung von dem Inhalt zu speichern ist, einen Name wie „Test.java_FILE_CHANGED_0_1234567 890687“.

Der Inhalt von den Dateien, die für die Änderungen mit Typ *FILE_CHANGED* erstellt werden, sind *unified Diff* (siehe Abschnitt 2.2.1). Die Dateien, die für andere Änderung erstellt werden, haben keinen Inhalt. Wir brauchen nur die Namen um Informationen zu bekommen. Um ein *unified Diff* zu erzeugen können wir die Methode *diff* und *generateUnifiedDiff* von der Programmbibliothek „ java-diff-utils“ benutzen (siehe Abschnitt 4.2). Für die Methode *diff* müssen wir die aktuelle Datei und die letzte geänderte Datei haben. Damit können wir vergleichen, wie der Inhalt geändert wird. Die in dem Ordner von Projekt stehende Datei ist die aktuelle Datei. Wie können wir die letzte geänderte Datei bekommen? Sie wird eigentlich von Nutzer überschrieben. Eclipse bietet eine Funktion, mit der wir die alte Versionen bekommen können. Mit der Methode *getHistory* von einem *IFile-Objekt* können wir ein Array bekommen, das aus den vergangenen Zustände einer Datei besteht. Ein Zustand wird von einem *IFileState-Objekt* beschrieben. Mit der Methode *getContents* von einem *IFileState-Objekt* können wir ein Eingabestrom für den Inhalt von eine Datei mit einem bestimmtem Zustand bekommen.

Für die Erstellung einer Datei oder eines Ordners für Änderung müssen wir noch aufpassen, dass wir die von Eclipse gebotene Funktion nicht benutzen sollen. Normalerweise können wir mit der Methode *create* von *IFile-Objekt* oder *IFolder-Objekt* eine Datei oder einen Ordner erstellen. Aber wir möchten die Dateien und Ordner für Änderungen während der Behandlung von den *ResourceDelta-Objekte* erstellen, dann müssen wir die Funktion von Paket „ java.io “ benutzen. Weil während der Behandlung von den *ResourceDelta-Objekte* bzw. der Behandlung von der Nachrichten für Änderungen ist der Workspace von Eclipse gesperrt. In dieser Zeit falls wir die Methode *create* benutzen, werden wir einen Fehler bekommen.

4.2.4 View für Änderungen

Wir können die Änderungen von Dateien überwachen und speichern, jetzt sollen wir die Informationen der Änderungen für Nutzer grafisch darstellen. Damit kann der Nutzer ganz klar sein, wo, wie und wann die Änderungen geführt wurden. Durch die View kann der Nutzer auch Informationen für Synchronisierung bekommen. Es zeigt auch, wohin die Änderungen synchronisiert werden können, und welche Projekte schon mit die Änderungen synchronisiert werden. Außerdem bietet die View eine Funktion, mit der der Nutzer die Änderungen nach anderen gewünschten Projekte synchronisieren kann. Die View soll nach jeder Änderung der von VariantSync unterstützten Projekte aktualisieren. Dadurch kann der Nutzer die Änderungen in Echtzeit sehen.

Um die Orte von den Änderungen deutlich zu zeigen, sollen wir am besten mit Baum darstellen. Die Wurzel beschreibt den Projekt-Ordner. Die Blätter beschreiben die

Änderungen. Die inneren Knoten beschreiben die Ordner und Dateien. Die Struktur ist fast wie die Struktur der Dateien von Projekt. Aber die Blätter beschreiben nicht mehr Dateien sondern Änderungen. Außerdem die Dateien, die nicht geändert haben, werden nicht die diesem Baum zeigen. Es ist deutlicher die Orte mit Baum als mit Textform zu darstellen. Für diese View brauchen wie eine Baumdarstellung für die Orte von den Änderungen und eine Textform wie eine Tabelle für die restliche Informationen. Wir können die Baumdarstellung und Tabelledarstellung in einer View erstellen. Es wird *TableTree* genannt. Früher gibt es ein Klasse *TableTree*, mit dem wird es implementiert. Jetzt wird empfohlen es mit Klasse *Tree* und *TreeColumn* zu implementieren. Mit *TreeColumn* können wir die Spalten für eine Tree-View erstellen. Für unsere View der Änderungen werden vier Spalten erstellt. Die erste Spalte ist für Ort der Änderungen, hier wird eine Baumdarstellung implementiert. In der zweiten Spalte werden Namen von Projekten gezeigt. Dadurch kann der Nutzer wissen, in welchem Projekt eine Änderung durchgeführt wurde. In der dritten Spalte werden die Projekte dargestellt, die mit der beschriebene Änderung synchronisieren können. Für diese Spalte sollen wir die Konflikte von Synchronisierung einer Änderung kontrollieren. Es wird in Abschnitt 4.3.6 dargestellt. Falls ein Projekt einen Konflikt mit der beschriebene Änderung, wird dieses Projekt nicht in der dritten Spalte gezeigt. In der vierten Spalte werden die Projekte dargestellt, die schon mit die beschriebene Änderung synchronisiert werden. Diese Information wird für jedes Projekt in Datei *.variantsyncInfo* im Ordner *.variantsync* gespeichert.

Die mit VariantSync gearbeitete Varianten haben meistens ähnliche Funktionen, und haben normalerweise ähnliche Strukturen von Dateien und Ordner. Deshalb können wir in der Änderung-View (wir nannen die View „Änderung-View“, die um die Informationen von Änderungen zu darstellen erstellt wird) nur eine Baumdarstellung für alle unterstützte Projekte erstellen.

Die Abbildung 4.3 auf der nächsten Seite zeigt ein Beispiel für eine Änderung-View. In diesem Beispiel gibt es fünf Varianten (Project 1-5). Die Spalte *Resource* Zeigt die Baumdarstellung und die Daten für Änderungen. Die erste Zeile steht *Project root*. Es ist für alle fünf Projekte um den Projekt-Ordner zu beschreiben. Unter dem Knoten *Test1.java* (Zeile 3) stehen zwei Blätter *FILE_ADDED* und *FILE_CHANGED*. Sie beschreiben zwei Änderungen für *Project1* und *Project4*. Zuerst wurde die Datei *Test1.java* in *Project1* hinzugefügt. Die Änderung wurde nach *Project2*, *Project3*, *Project4* und *Project5* synchronisiert. Deshalb stehen die Namen für diese vier Projekte in der Spalte *Targets*. Die Änderungen, die durch Synchronisierung gemacht werden, werden nicht gezeigt. Es gibt eine Schaltfläche in der oberen rechte Ecke. Mit der können die durch Synchronisierung gemachte Änderungen gezeigt werden. Damit können wir kontrollieren, welche Synchronisierung gemacht wurden. Die zweite Änderung für die Datei *Test1.java* wird in *Project4* gemacht. Das steht in zweiter

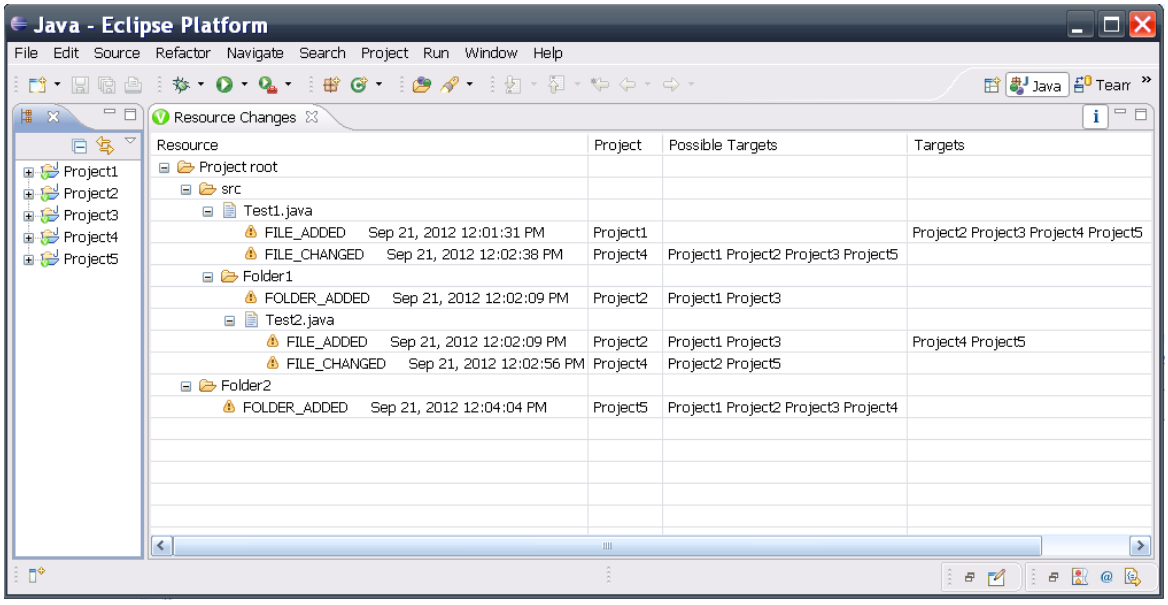


Abbildung 4.3: View für Änderungen in VariantSync

Spalte für dieses Blatt. Die Datei *Test1.java* wird in *Project4* geändert. In der dritten Spalte steht, dass diese Änderung in vier andere Projekte synchronisiert werden kann. Das in letzte Zeile stehte Blatt *FOLDER_ADDED* ist für den Ordner *Folder2*. In *Project5* wurde der Ordner *Folder2* hinzugefügt. Und es kann nach *Project1*, *Project2*, *Project3* und *Project4* synchronisiert werden.

```
1   <extension
2         point="org.eclipse.ui.views">
3      <category
4            name="Variant Sync"
5            id="de.ovgu.variantsync">
6      </category>
7      <view
8            name="Resource Changes"
9            icon="icons/VariantSyncSupport.png"
10           category="de.ovgu.variantsync"
11           class="de.ovgu.variantsync.views.ResourceChangesView"
12           id="de.ovgu.variantsync.views.ResourceChanges">
13     </view>
14  </extension>
```

Quelltext 4.5: Deklaration für die View „Resource Changes“ in der Datei plugin.xml

Um eine View zu implementieren brauchen wir eine Deklaration in der Datei *plugin.xml* und eine Klasse. Quelltext 4.5 zeigt das Fragment für diese Deklaration von unserer Änderung-View. Extension-Point ist „org.eclipse.ui.views“. Das Element *category* ist um die View in dem Dialog *Show View* zu einordnen (sehe Abbildung 4.4 auf der nächsten Seite). Mit dem Dialog können wir gewünschte View einschalten.

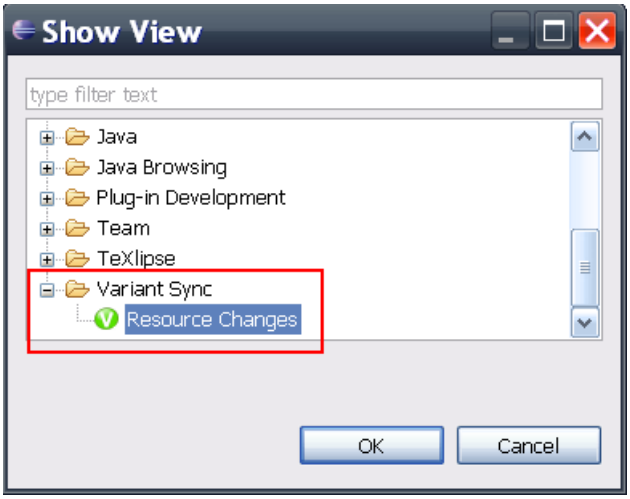


Abbildung 4.4: Eine Eclipse-Kategorie für VariantSync

Mit den Parameter *name* können wir die Änderung-View einen Titel einstellen. In der Abbildung 4.3 auf der vorherigen Seite wird die Änderung-View mit Titel „Resource Changes“ gezeigt. Der Parameter *icons* ist für das Bildsymbol, das am links oben von der View gezeigt wird (sehe Abbildung 4.3 auf der vorherigen Seite). Der Parameter *category* in dem Element *View* beweist, in welche Kategorie dieser View eingeordnet wird soll. Hier wird dieser Parameter mit dem ID von dem Element *category* definiert. Der Parameter *class* definiert die Klasse, die die Änderung-View implementiert.

Die Klasse *ResourceChangesView*, die für die Änderung-View erstellt wird, soll die Klasse *ViewPart* erweitern. Wir müssen die Methode *createPartControl* implementieren. Diese Methode wird aufgerufen, wenn diese View von Eclipse initialisiert wird. In dieser Methode sollen wir ein Objekt von der Klasse *TreeViewer* erstellen. Diese Objekt beschreibt die View. Mit der Methode *new TreeColumn* können wir eine Spalte für die View erstellen. Für die Eingabe der Methode brauchen wir ein Tree-Objekt, das wir mit Methode *getTree* von dem TreeViewer-Objekt bekommen können. Ändere Funktion so wie Schaltfläche sollen wir auch hier implementieren. Zum Beispiel die Schaltfläche für die durch Synchronisierung gemacht Änderungen, Doppelklick der Änderungen um die Änderungen zu synchronisieren. Um die Änderungen richtig zu darstellen, brauchen wir noch eine wichtige Methode *setContentProvider*. Wir sollen mit der Methode für den TreeViewer-Objekt ein Objekt konfigurieren, das Objekt das Interface *ITreeContentProvider* implementiert wird. Das ITreeContentProvider-Objekt bietet dem TreeViewer-Objekt die Informationen von Änderungen. Wenn die View aktualisiert wird, werden die Ordner *.variantsync* in allen unterstützten Projekte nach gespeicherten Änderungen durchsucht. Um die Informationen von Änderungen zu beschreiben haben wir drei Klassen und ein Interface erstellt. Die drei Klassen sind *ResourceChangesFolder*, *ResourceChangesFile*

und *ResourceChangesFilePatch*. Sie beschreiben die Ordner, die geänderte Dateien und die gespeicherte Änderungen. Sie implementieren das Interface *IResourceChangesViewItem*. Damit können die beschriebene Ordner, Dateien und Änderungen in der View gezeigt werden. Die in Ordner *.variantsync* in allen unterstützten Projekte gespeicherte Ordner, Dateien und Änderungen werden mit einer Baumstruktur in dem ITreeContentProvider-Objekt gespeichert. Damit können sie für die Baumdastellung in der View benutzt werden.

Die Konsolenansicht von VariantSync

Um den Status von VariantSync oder ein paar Information zu zeigen haben wir noch ein Konsole erstellt. Es wird in der Konsole-View gezeigt. Mit Konsole können wir einfach ein paar Text für Nutzer zeigen. Die Abbildung 4.5 zeigt ein Beispiel für die Konsole von VariantSync.

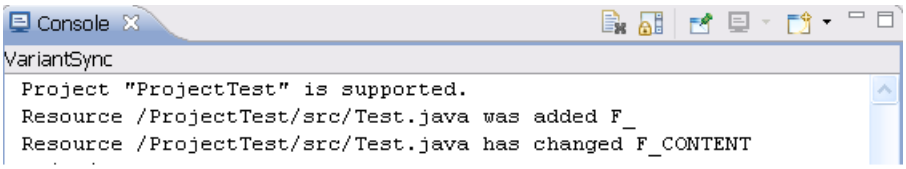


Abbildung 4.5: Die Konsolenansicht von VariantSync

Durch die Texte, die von dieser Konsole gezeigt wird (sehe Abbildung 4.5), können wir wissen, dass ein Projekt gerade mit VariantSync unterstützt wird. In diesem Projekt wird eine Datei *Test.java* in Ordner *src* hizugefügt. Danach wird diese Datei geändert. Am Ende der letzten Zeile steht Text „F_CONTENT“. Das bedeutet, dass die Änderung für den Inhalt von der Datei ist. Wenn ein Projekt von VariantSync unterstützt oder nicht unterstützt wird, und die Dateien oder Ordner geändert werden, werden ein paar Texte in der Konsole gezeigt. Mit solche Information in Konsole kann der Nutzer genau wissen, was mit VariantSync geführt werden. Außerdem können wir auch diese Konsole benutzen um die Informationen von dem Fehler auszudrucken.

Für die Implementierung dieser Konsole müssen wir auch in der Datei *plugin.xml* ein Extension-Point definieren und ein paar Klassen erstellen. Die Definition enthält Extension-Point, Klasse für Implementierung dieser Konsole, Bildsymbol und Titel für diese Konsole. Im Quelltext 4.6 auf der nächsten Seite ist die detaillierte Definition.

Die Parameter *point*, *class* und *icon* funktionieren wie in vorher gezeigter Definition von Extension-Point. Der Parameter *label* definiert den Text, der für den Name der Menü-Option gezeigt wird.

```
1   <extension
2         point="org.eclipse.ui.console.consoleFactories">
3      <consoleFactory
4            class="de.ovgu.variantsync.console.ConsoleFactory"
5            icon="icons/VariantSyncSupport.png"
6            label="VariantSync">
7      </consoleFactory>
8   </extension>
```

Quelltext 4.6: Deklaration für die Konsole in der Datei plugin.xml

Um die Konsole zu implementieren haben wir drei Klassen erstellt: *ChangeOutPutConsole*, *ConsoleDocument* und *ConsoleFactory*. Die Klasse *ConsoleDocument* wird für die Speicherung der Informationen erstellt. Falls diese Konsole nicht gezeigt wird, werden die Informationen, die später in diese Konsole gezeigt werden sollen, in einem ConsoleDocument-Objekt gespeichert. Wenn diese Konsole später wieder gezeigt wird, werden die Information zusammen gezeigt. Die Informationen werden in einem Array für Text gespeichert. Die Klasse *ConsoleFactory* soll das Interface *IConsoleFactory* implementieren. In diesem Interface gibt es nur eine Methode *openConsole*. Wir sollen es überschreiben. Wenn die in der Konsole-View für unsere Konsole erstellte Menü-Option ausgewählt wird, wird die Methode *openConsole* aufgerufen. In dieser Methode sollen wir unsere Konsole verwalten. In Eclipse gibt es einen Manager für alle Konsolen. Wir sollen zuerst die Methode *getConsoleManager* bei einem ConsolePlugin-Objekt anrufen um den Manager für alle Konsolen zu bekommen. Mit der Methode *getConsoles* bei diesem Manager können wir ein Array bekommen. Darin werden alle Konsolen gespeichert. Wir kontrollieren zuerst, ob ein Objekt für unsere Konsole existiert ist. Falls es keinen Objekt für unsere Konsole gibt, erstellen wir einen neuen ChangeOutPutConsole-Objekt, und speichern wir mit der Methode *addConsoles* den neuen Objekt in dem Manager. Zum Schluss zeigen wir unsere Konsole mit der Methode *showConsoleView*. Die Methode ist bei dem Manager verfügbar. Die Eingabe ist den neu erstellten ChangeOutPutConsole-Objekt oder den in dem Manager gefunden Objekt. Die Klasse *ChangeOutPutConsole* beschreibt unsere Konsole. Die Funktionen wie Erstellung des ConsoleDocument-Objektes, Speicherung und Auslesen der Informationen werden in diese Klasse geführt. Mit der Methode *write* bei einem IOConsoleOutputStream-Objekt können wir Text in der Konsole ausdrucken.

4.2.5 Konflikte erkennen und synchronisieren

Wir können nicht alle Änderungen automatisch synchronisieren. Falls eine Änderung einen Konflikt mit anderen Projekte hat, oder abhängig mit den Änderungen von dem selben Projekt ist, können wir nicht mit VariantSync einfach synchronisieren.

Es muss von Nutzer per Hand gemacht werden. In der Änderung-View (Es wird in Abschnitt 4.3.4 dargestellt. siehe Abbildung 4.3 auf Seite 51) werden die Projekte, die automatisch synchronisiert werden können, in Spalte *Possible Targets* für jede Änderung gezeigt. Wenn die Änderung-View aktualisiert wird, kontrollieren wir die Konflikte in jedem von VariantSync unterstützten Projekt für jede Änderung. Wenn eine Änderung keine Konflikt mit einem Projekt hat, wird dieses Projekt für die Änderung in Spalte *Possible Targets* gezeigt. Die Konflikte für die Änderungen (Typ: FILE_CHANGED, siehe Abschnitt 4.3.3) haben insgesamt drei Typen: Reihenfolgeskonflikt, Abhängigkeitskonflikt und Überdeckungskonflikt (siehe Abschnitt 3.4.2).

Wie können wir die Konflikte erkennen? Sie haben die gleichen Eigenschaften: Die Intervalle der Nummer von den geänderten Zeile sind benachbart oder überlappen sich. Um dies gut zu verstehen können wir noch mal die Beispiele für die Konflikte in Abschnitt 3.4.2 benutzen. In der Abbildung 3.9 auf Seite 30 wird eine Reihenfolgeskonflikt gezeigt. Die Änderung vom Basiscode (Fragment A) zu $Variante_1$ (Fragment B) hat eine Änderungspunkt zwischen Zeile 3 und 4. Eine neue Zeile wird zwischen Zeile 3 und 4 hinzugefügt. Die Änderung vom Basiscode (Fragment A) zu $Variante_2$ (Fragment C) hat auch eine Änderungspunkt zwischen Zeile 3 und 4. Eine andere neue Zeile wird hier hinzugefügt. Die Intervalle von zwei Änderungen sind gleich (3,4). Sie überlappen sich. In der Abbildung 3.12 auf Seite 33 wird ein Überdeckungskonflikt gezeigt. Die zwei Änderungen haben die gleiche Zeile geändert. Die Intervalle von zwei Änderungen sind auch gleich ([3,3]) und überlappen sich somit. In der Abbildung 3.10 auf Seite 31 wird ein Abhängigkeitskonflikt gezeigt. Die Änderung vom Basiscode (Fragment A) zu $Variante_1$ (Fragment B) hat Zeile 4 geändert. Die Änderung von Basiscode (Fragment A) zu $Variante_2$ (Fragment C) hat Zeile 5 geändert. Die Intervalle von zwei Änderungen sind [4,4] und [5,5]. Sie sind benachbart. In der Abbildung 3.11 auf Seite 32 werden zwei Änderungen von einem Projekt gezeigt. Sie haben einen Abhängigkeitskonflikt. Wenn wir die zweite Änderung (von $Revision_{n+1}$ zu $Revision_{n+2}$) synchronisieren möchten, müssen wir zuerst die erste Änderung ($Revision_n$ zu $Revision_{n+1}$) synchronisieren. Die erste Änderung (von $Revision_{n+1}$ zu $Revision_n$) hat Intervalle [4,4] und [6,7]. Die zweite Änderung hat Intervalle (6,7). Sie überlappen sich. Von solche Beispiele können wir bemerken, wenn die Intervalle der Nummer von den geänderten Zeile sind benachbart oder sich überlappen, haben die Änderungen Konflikte.

Um Konflikte für eine Änderung zu erkennen vergleichen wir die vor dieser Änderung stehenden Quelltext und den Quelltext, der in den Synchronisierenden Projekt steht. Durch diesen Vergleich können wir eine Änderung bekommen. Zum besseren Verständnis können wir noch mal Abbildung 3.6 auf Seite 27 und Abbildung 3.5 auf Seite 27 schauen. Durch Vergleich bekommen wir eine Änderung Δ_{B_{n-1},A_n}. Danach

kontrollieren wir, ob die Änderung Δ_{B_{n-1},A_n} und die Änderung Δ_{B_{n-1},B_n} ein überlapptes oder benachbartes Intervall haben. Falls es keines überlappte oder benachbarte Intervall zwischen Δ_{B_{n-1},A_n} und Δ_{B_{n-1},B_n} gibt, kann die Änderung Δ_{B_{n-1},B_n} nach Projekt (Variante) A synchronisiert werden.

Um die Quelltexte zu vergleichen, können wir vorher in Abschnitt 4.2 dargestellte Programmbibliothek benutzen. Mit Methode *diff* können wir einfach zwei Quelltexte vergleichen. Das Ergebnis ist einen Patch-Objekt. Dieses Objekt beschreibt die Änderung von erstem Eingabe-Quelltext zu zweitem Eingabe-Quelltext. Eine Änderung zwischen zwei Quelltexte wird normalerweise von eine oder mehr als eine kleine Änderung zusammengebaut. Zum Beispiel ändern wir einen Quelltext an der Zeile 3,4,5 und 10. Diese Änderung wird aus zwei kleine Änderungen zusammengebaut: Änderung bei den Zeilen 3,4 und 5 und Änderung bei der Zeile 10. Solche kleine Änderungen werden mit ChangeDelta-Objekt, DeleteDelta-Objekt oder InsertDelta-Objekt beschrieben und in einem Patch-Objekt gespeichert. ChangeDelta-Objekt, DeleteDelta-Objekt und InsertDelta-Objekt beschreibt die kleine Änderung sowie Änderung, Löschung und Hinzufügung bei den Zeilen. Mit Methode *getOriginal* und *getRevised* bei einem Delta-Objekt können wir zwei Chunk-Objekte bekommen. Chunk-Objekte werden immer paarweise in einem Delta-Objekt gespeichert. Sie beschreiben die alte Zeilen und die neue Zeilen für eine kleine Änderung. Mit Methode *getPosition* bei dem Chunk-Objekte, das mit der Methode *getOriginal* bekommt werden kann, können wir einen Index für den Anfang geänderter Zeilen bekommen. Mit Methode *getSize* können wir wissen, wie viel Zeilen gelöscht und geändert werden. Für Hinzufügung von Zeilen ist das Ergebnis von der Methode *getSize* immer Null. Mit Methode *getPosition* und *getSize* können wir die Intervalle von einer Änderung bekommen. Danach kontrollieren wir die Intervalle, dann können wir die Konflikte erkennen. Die Konflikte für die Änderungen, deren Typ *FOLDER_ADDED* oder *FILE_ADDED* sind, sind einfacher zu erkennen. Wir sollen nur kontrollieren, ob das Projekt, das synchronisiert wird, ein Quelltext oder ein Ordner mit gleichen Name und gleichen Ordnerpfad hat. Wenn das Projekt, das synchronisiert wird, ein Quelltext oder ein Ordner mit gleichem Namen und gleichem Ordnerpfad hat, ist hier ein Konflikt.

Wenn eine Änderung keinen Konflikt mit einem Projekt hat, können wir die Änderung in dieses Projekt synchronisieren. In der Änderung-View (siehe Abbildung 4.3 auf Seite 51) wird eine Funktion für Synchronisierung implementiert. Um eine Änderung zu synchronisieren können wir einfach in der Änderung-View die gewünschte Änderung doppleklicken. Dann wird ein Dialog für Nutzer dargestellt. In Abbildung 4.6 auf der nächsten Seite zeigt ein Beispiel für diesen Dialog. In diesem Dialog wird drei Informationen für Nutzer gezeigt: Die Features für die ausgewählte Änderung, die mögliche Projekte für Synchronisierung, und die Details für die aus-

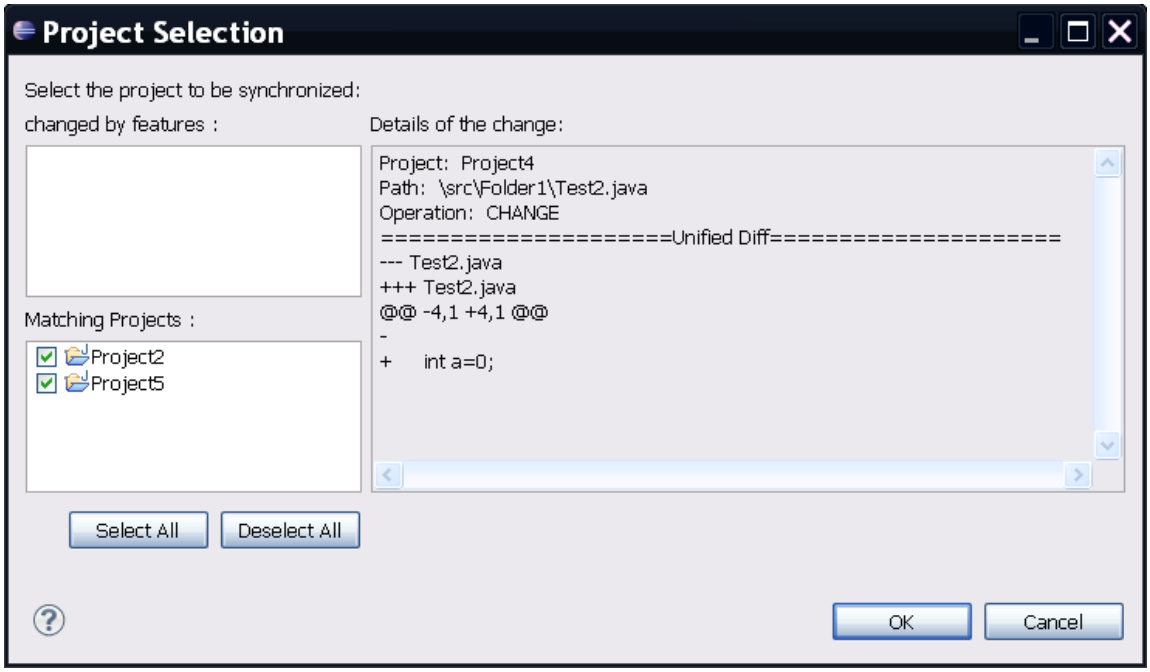


Abbildung 4.6: Dialog für Synchronisierung von Änderungen

gewählte Änderung. Die Information für die Features wird in Abschnitt 4.3.7 weiter gesprochen. Die mögliche Projekte sind alle in dem Workspace von VariantSync unterstützte Projekte außer die Projekte, die mit diese Änderung Konflikt haben. Die Details für eine Änderung ist :Der Name von dem Projekt, in dem diese Änderung gemacht wird, der Ordnerpfad für den geänderte Quelltext, und der Typ für die Änderung. Wenn es möglich ist (Wenn ein Quelltext oder ein Ordner gelöscht und hinzugefügt wird, gibt es keine *unified diff*), wird auch der *unified diff* für die Änderung gezeigt. Damit kann Nutzer genau wissen, wie der Quelltext geändert wird.

Nach der Auswahl von Projekte wird die Synchronisierung von VariantSync automatisch gemacht. Die Synchronisierung für die Änderungen, deren Typ *FOLDER_REMOVED*, *FILE_REMOVED*, *FOLDER_ADDED* oder *FILE_ADDED* sind, kann einfach implementiert werden. Wir müssen nur die Quelltexte oder Ordner kopieren oder löschen. Für die Synchronisierung der Änderungen, deren Typ *FILE_CHANGED* sind, müssen wir noch die in Abschnitt 4.2 dargestellte Programmbibliothek benutzen. Um den Vorgang der Synchronisierung zu verstehen, können wir noch mal Abbildung 3.5 auf Seite 27 sehen. Angenommen, wir möchten die Änderung Δ_{B_3,B_4} nach Projekt (Variante) A synchronisieren. Mit Methode *parseUnifiedDiff* können wir von der gespeicherten Datei ein Patch-Objekt für die Änderung Δ_{B_3,B_4} bekommen. Wir vergleichen noch mal B_3 und A_n und bekommen ein Patch-Objekt für die Änderung Δ_{B_3,A_n}. B_3 ist für uns in Projekt (Variante) B nicht mehr verfügbar. Es wird von B_n ersetzt. Wir können B_n mit Methode *unpatch* und die

gespeicherte Dateien für die Änderungen, die von B_3 zu B_n sind, nach B_3 zurückkehren. Mit dem Patch-Objekt für die Änderung Δ_{B_3,A_n} ändern wir ein Patch-Objekt für die Änderung Δ_{B_3,B_4}, damit wir das geänderte Patch-Objekt für A_n benutzen können. Zum Beispiel hat die Änderung Δ_{B_3,A_n} an der Zeile 3 zwei neue Zeilen hinzugefügt, und an der Zeile 5 eine neue Zeile hinzugefügt. Die Änderung Δ_{B_3,B_4} hat die Zeile 10 geändert. Wenn wir die Änderung Δ_{B_3,B_4} nach Projekt (Variante) A synchronisieren möchten, sollen wir die Zeile 13 bei A_n ändern, denn die Änderung Δ_{B_3,A_n} hat insgesamt 3 Zeilen vor Zeile 10 hinzugefügt. Die Änderung Δ_{B_3,B_4} hat keinen Konflikt mit Änderung Δ_{B_3,A_n}. Die geänderte Zeilen werden nur nach hinten oder nach vorne verschoben. Das bedeutet, dass wir sollen nur die Position der kleine Änderungen (Sie werden mit den Delta-Objekte beschriebt) von der Änderung Δ_{B_3,B_4} ändern. Wir kontrollieren jedes Delta-Objekt, das in einem Patch-Objekt für die Änderung Δ_{B_3,B_4} gespeichert wird, ob die Position ändern soll. Mit Methode *setPosition* bei einem Delta-Objekte können wir die Position ändern. Mit dem für A_n geändert Patch-Objekt und der Methode *patch* können wir die Änderung Δ_{B_3,B_4} nach Projekt (Variante) A synchronisieren und A_{n+1} bekommen.

4.2.6 Einbeziehung von Domänenwissen

Eine Änderung können nach anderen Projekte (Varianten) synchronisieren werden, wenn es keinen Konflikt mit anderen Projekte hat. Aber manchmal möchten wir es nicht nach alle mögliche Projekte synchronisieren. Wir können Domänenwissen anwenden, um die Synchronisierung zu entscheiden. Das wird schon in Abschnitt 3.4.3 gesprochen. Für jede Synchronisierung wird eine Wahl für Features der Änderung für Nutzer geboten. Mit der Auswahl des Features können wir die Projekte filtern, die für eine Synchronisierung möglich sind.

Die Abbildung 4.7 auf der nächsten Seite zeigt ein Beispiel. Die Synchronisierung wird mit Feature-Auswahl unterstützt. Wir entscheiden, dass die Änderung für Feature *Feature4* ist, und wählen dieses Feature aus. Das Projekt *Project2* wird automatisch nicht ausgewählt. Weil *Project2* das Feature *Feature4* nicht enthält. Mit dem Feature-Modell für alle Variante erstellen wir für jede Variante eine Datei für die Konfiguration. Es beschreibt, welche Features eine Variante enthält.

Um Feature-Modell und Datei für eine Konfiguration zu erstellen brauchen wir nicht noch mal die Funktionalität zu implementieren. Wir können einfach die Funktionalität von FeatureIDE verwenden. FeatureIDE ist eine Eclipse-basierte IDE, die alle Phasen des Feature-orientierten Software-Entwicklung unterstützt. Mit dem von FeatureIDE gebotenen Editor für Feature-Modell und Konfiguration können wir einfach Feature-Modell entwerfen und die Konfigurationen erstellen und validieren. FeatureIDE wird von viele Plug-ins zusammen gebaut. Davon sind *de.ovgu.featureide.core*,

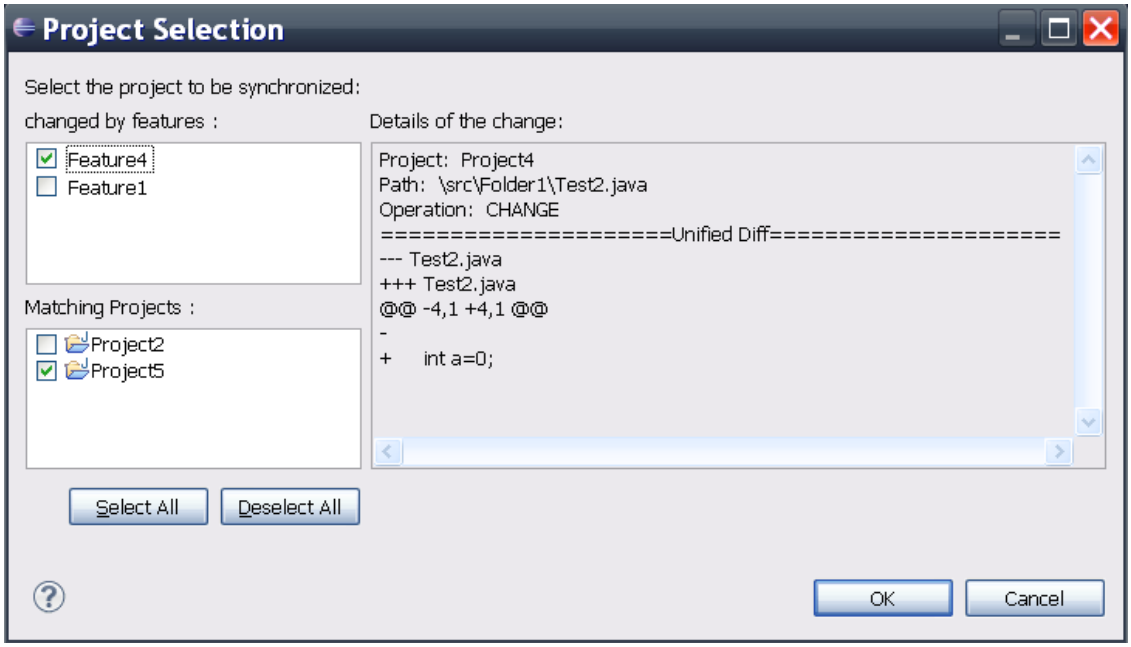


Abbildung 4.7: Dialog für Synchronisierung mit Feature-Auswahl

de.ovgu.featureide.ui, *de.ovgu.featureide.fm.core* und *de.ovgu.featureide.fm.ui* für uns erfordlich.

Wir erstellen zuerst ein FeatureIDE-Projekt mit Name *variantsyncFeatureInfo* für alle in dem Workspace von VariantSync unterstützte Projekte. Mit die von FeatureIDE gebotene Editoren erstellen wir eine Feature-Modell und für jede Variante eine Konfiguration, deren Name wie die Variante ist. Wenn wir auf eine Änderung doppelklicken, wird ein Dialog dargestellt (sehe Abbildung 4.7). Der Workspace von Eclipse wird durchgesucht. Wenn ein FeatureIDE-Projekt mit Name *variantsyncFeatureInfo* existiert, werden Konfigurationen für die mögliche Projekte und geändertes Projekt durchgesucht. Mit der Konfiguration können wir wissen, welche Features eine Projekt hat. Die Features für die geändertes Projekt werden in Dialog (Abbildung 4.7) in dem Bereich *changed by features* dargestellt. Wir können ein oder mehrere Features für die Änderung auswählen. Es wird geprüft, ob die mögliche Projekte die ausgewählte Features haben. Die mögliche Projekte werden für die Synchronisierung automatisch ausgewählt oder nicht ausgewählt.

4.3 Zusammenfassung

Die prototypische Implementierung VariantSync wird als ein Plug-in für Eclipse entwickelt. Dabei werden eine quelloffene Programmbibliothek „java-diff-utils“ und die Funktionalität des Feature-Modells von FeatureIDE benutzt. Die Änderungen der Quelltexte werden mit der von Eclipse gebotene Funktionalität überwacht. Und die Details der Änderungen werden in Dateien mit dem vereinheitlichten Format

(unidiff) gespeichert. Um die Änderungen zu zeigen, werden zwei View erstellt. Für diese prototypische Implementierung werden insgesamt 29 Klassen und ca. 4000 Quelltext-Zeilen (LOC) erstellt.

5. Evaluierung

Im vorherigen Kapitel wurde der Prototyp *VariantSync* beschrieben. In diesem Kapitel wird die Funktionsweise des Prototyps evaluiert. Die Evaluierung simuliert eine Entwicklung eines Projektes mit vielen Varianten. Dafür sollten wir zunächst ein geeignetes Projekt auswählen. Es wird in erste Abschnitt vorgestellt. Danach wird die Vorgehensweise der Evaluierung beschrieben. Dann wird das Ergebnis und die Auswertung beschrieben. Zum Schluss kommt die Diskussion und die Zusammenfassung.

5.1 Verwendete Programme zur Evaluierung

Das Ziel von *VariantSync* ist Varianten zu synchronisieren. Um die Funktionsweise *VariantSync* zu evaluieren müssen wir ein Projekt auswählen. Mit dem Projekt können wir verschiedene Varianten mit verschiedenen Revisionen bekommen, weil es eine SPL ist. Für diese Evaluierung wählen wir das Projekt *DesktopSearcher*[1] aus.

Das Projekt *DesktopSearcher* wurde mit FeatureIDE und AHEAD[2] entwickelt. In dem Projekt wurde eine Software-Produktlinie entwickelt. Es gibt insgesamt 22 Features. Mit dem Projekt können wir 462 verschiedene Varianten bekommen. Die Quelltexte wird in einem Repository[3] gespeichert. Es gibt also Versionen über die Zeit.

Das Projekt ist geeignet für diese Evaluierung, weil es eine SPL implementiert hat und Varianten für viele Versionen erzeugen kann. Wir brauchen nicht mehr die Varianten analysieren und ein passendes Feature-Modell erstellen. In dem Projekt gibt

[1] Es ist ein studentisches Projekt von Sebastian Bress, Alexander Grebhahn, Sönke Holthusen und Reimar Schröter und wurde im 2011 abgeschlossen.

[2] http://www.cs.utexas.edu/users/schwartz/ATS.html

[3] https://faracvs.cs.uni-magdeburg.de/svn/tthuem-FO_Search_Engine

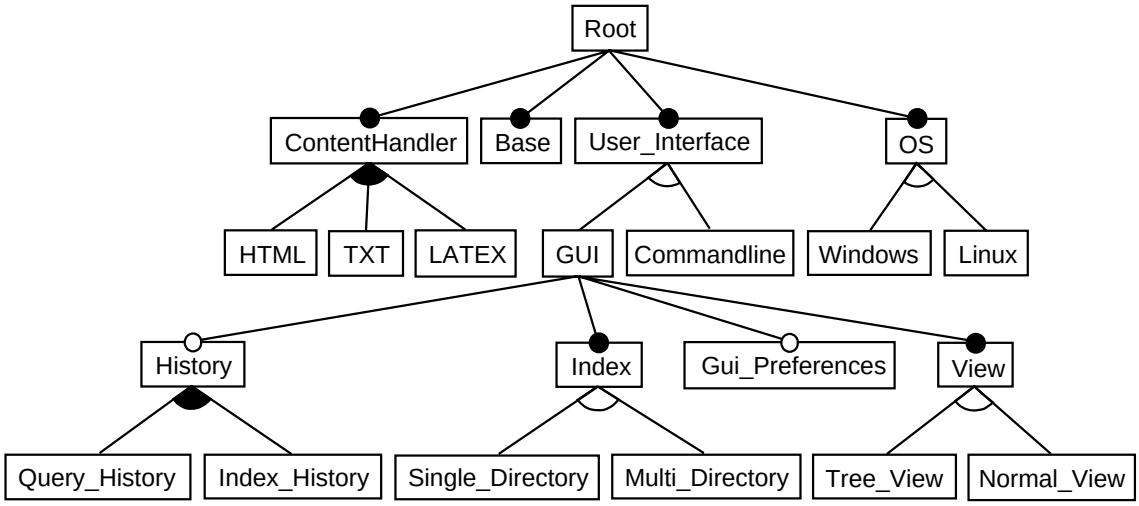


Abbildung 5.1: Feature-Modell von *Revision 91* des Projekts *DesktopSearcher*

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
Root	•	•	•	•	•	•	•	•	•	•
ContentHandler	•	•	•	•	•	•	•	•	•	•
HTML	•					•	•		•	•
TXT		•			•		•	•	•	•
LATEX			•	•	•	•	•		•	
Base	•	•	•	•	•	•	•	•	•	•
User_Interface	•	•	•	•	•	•	•	•	•	•
GUI		•		•		•	•	•	•	•
History				•			•	•	•	
Query_History				•			•			
Index_History							•	•	•	
Index		•		•		•	•	•	•	•
Single_Directory				•		•		•	•	
Multi_Directory		•					•			•
Gui_Preferences						•		•	•	
View		•		•		•	•	•	•	•
Tree_View								•	•	•
Normal_View		•		•		•	•			
Commandline	•		•		•					
OS	•	•	•	•	•	•	•	•	•	•
Windows	•	•		•		•	•	•		•
Linux			•		•				•	

Tabelle 5.1: Die Features der zehn evaluierten Varianten

es ein Feature-Modell. Wir können es für *VariantSync* benutzen. Außerdem ist es ein bisschen einfacher um eine Änderung nach Feature zu sortieren. Dieses Projekt wird mit AHEAD unterstützt. Der für ein bestimmtes Feature erstellte Quelltext wird in einem Ordner mit dem gleiche Namen wie das Feature gespeichert. Durch den Vergleich zwischen zwei Revision können wir wissen, wofür die Änderung ist. Zum Beispiel durch die Änderung zwischen *Revision 90* und *Revision 91* wurde eine Datei */features/User_Interface/MrPinkMain.jak* geändert. Durch den Namen des Ordners können wir wissen, dass die Änderung für Feature *User_Interface* ist.

Für diese Evaluierung erstellen wir zufällig zehn Varianten von dem Projekt *DesktopSearcher*. Dabei stellten wir sicher, dass jedes Feature von dem Feature-Modell mindest einmal ausgewählt ist. Wir sollen jedes Feature berücksichtigen. Weil manche Änderungen zwischen Versionen vielleicht nur für bestimmte Features sind. Falls solche Features nicht in den zehn Varianten enthalten werden, sind die zehn Varianten zwischen die Versionen gleich. Dann haben manche Versionen keine Bedeutung mehr für diese Evaluierung. Tabelle 5.1 auf der vorherigen Seite zeigt die Features für den zehn Varianten. Abbildung 5.1 auf der vorherigen Seite zeigt das Feature-Modell. Mit dem wird diese Tabelle erstellt.

Das in der Abbildung 5.1 auf der vorherigen Seite gezeigte Feature-Modell kommt aus *Revision 91* (letzte Revision für Quelltexte). Das Feature-Modell von dem Projekt *DesktopSearcher* wurde insgesamt 5 mal geändert. Dabei wurden manche Features umbenannt. Für eine Revision, können wir zehn Konfigurationen erstellen und damit zehn Varianten generieren. Für die Revisionen, die gleiche Feature-Modell haben, brauchen wir nicht neue Konfigurationen erstellen. Für die Revisionen, die verschiedenes Feature-Modell haben, müssen wir noch mal zehn Konfigurationen erstellen, damit die zehn Konfigurationen das Feature-Modell angepasst werden können.

5.2 Vorgehensweise

Die Evaluierung simuliert eine Entwicklung eines Projektes mit vielen Varianten. Dafür erstellen wir die Quelltexte für jede Revision für die in Abschnitt 5.1 ausgewählten Varianten. Die Änderungen, die zwischen zwei Revision sind, werden teilweise per Hand gemacht. Wir nehmen zuerst die Änderung einer Variante aus dem Repository und versuchen mit *VariantSync* die Änderungen an anderen Varianten automatisch zu erhalten. Die detaillierten Schritte sind wie folgt:

1. **Quelltexte für Varianten für jede Revision erzeugen**

Zunächst laden wir die erste Revision herunter, mit der wir Quelltexte für zehn Varianten erzeugen können. Das Projekt *DesktopSearcher* hat eine SPL mit FeatureIDE (unterstützt AHEAD) implementiert. Um die Quelltexte für die Varianten zu erzeugen, müssen wir zuerst für die zehn Varianten (siehe Tabelle 5.1 auf Seite 62) zehn Konfigurationen erstellen. Die Konfiguration für jede Variante muss zu der Tabelle 5.1 gepasst werden. Das bedeutet, dass die Varianten wie in Tabelle 5.1 beschrieben, bestimmte Features haben und nicht haben müssen. Vor Revision 61 heißt Feature „ContentHandler“ „Parser“. Feature „HTML“ heißt „HTML_Parser“. Feature „TXT“ heißt „TXT_Parser“, und Feature „LATEX“ heißt „LATEX_Parser“. Nach der Erstellung von Konfigurationen können wir die Funktion *„Build All Current Configurations“* von FeatureIDE benutzen um die Qelltexte alle für den zehn Varianten zu generieren. Die erste Revision, mit der wir Quelltexte für zehn Varianten generieren können, ist *Revision 4*. Danach generieren wir weiter die Quelltexte für zehn Varianten mit nächster Revision.

Bei der Generierung von Quelltexte müssen wir noch aufpassen, dass eine Datei („.order“) in *Revision 15* in dem Projekt hinzugefügt wurde. Diese Datei legt die Reihenfolge fest, in dem die Quelltexte generiert werden. Die Quelltexte für bestimmte Feature werden nach diese Reihenfolge zusammen gebaut. Mit der gleichen Revision und gleichen Konfiguration können wir für eine Variante mit verschiedene Datei *.order* verschiedene Quelltexte bekommen. Die Datei *.order* ist für alte Versionen von FeatureIDE. Bei der aktuellen Version von FeatureIDE wird die Reihenfolge in der Datei *model.xml* gespeichert. Wenn wir die neue Version von FeatureIDE benutzen, sollen wir die Reihenfolge kontrollieren. Damit ist die Reihenfolge in der Datei *model.xml* gleich wie in der Datei *.order*.

In manchen Revisionen sind die Änderungen nicht für die Quelltexte, sondern für die Dokumentationen für das Projekt. In solchen Revisionen sind die generierte Quelltexte für die zehn Varianten gleich wie die vorherige Revision. Zum Beispiel sind die Änderungen von *Revision 65* bis zu *Revision 80* nur für die Dokumentationen. Bei der Generierung von Quelltexte werden solche Revisionen ignoriert.

2. **Umgebung für *VariantSync* erstellen**

Nach dem ersten Schritt haben wir Revisionen für zehn Varianten. Es ist wie eine parallele Entwicklung von zehn Projekten. Die zahlreiche Quelltexte (für Projekt *DesktopSearcher* sind Dateien in Java) von verschiedene Revisionen für zehn Varianten wurden auf der Festplatte gespeichert. Jetzt erstellen wir

die Umgebung für *VariantSync* um es zu evaluieren. Zunächst erstellen wir zehn Projekte in Eclipse mit den Namen *Variante1*, ..., *Variante10*. Danach kopieren wir die Quelltexte von *Revision 4* zum passenden Projekt in Eclipse für alle zehn Varianten. Und hinzufügen wir die Unterstützung von *VariantSync* mit der Funktion „Add VariantSync Support“ für jedes Projekt. Zum Schluss erstellen wir ein FeatureIDE-Projekt mit Name *variantsyncFeatureInfo* für das Feature-Modell und Konfigurationen für die zehn Varianten. Das Feature-Modell ist von *Revision 91*. Die Konfigurationen für die zehn Varianten sind mit dem Name wie die zehn Projekte (z.B *Variante1.config*).

3. **Varianten vergleichen, eine Variante zufällig auswählen und per Hand ändern**

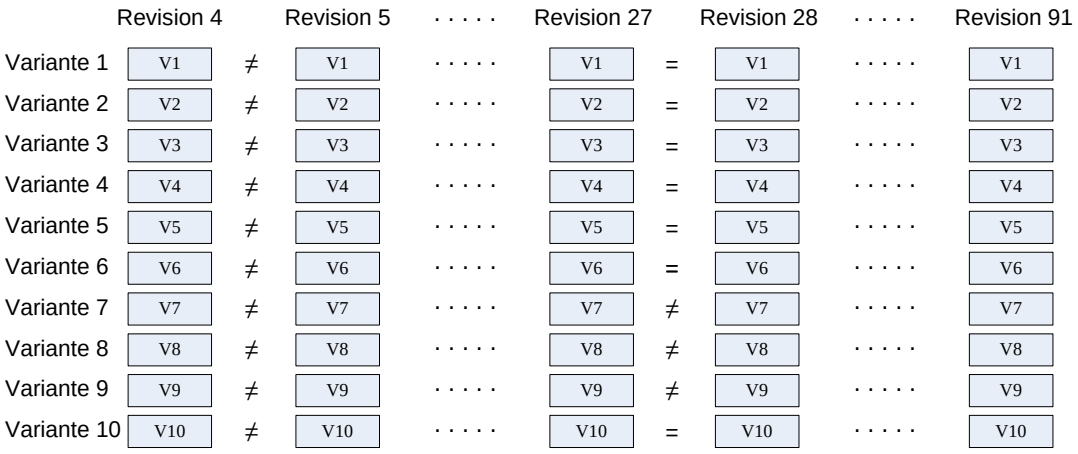


Abbildung 5.2: Vergleich der Varianten zwischen den betrachteten Revisionen

Wir vergleichen die Quelltexte für jede Varianten. Abbildung 5.2 zeigt ein Beispiel für den Vergleich. Zunächst vergleichen wir die Quelltexte (in Eclipse stehende) von *Revision 4* und *Revision 5*. Alle zehn Varianten werden von *Revision 4* zu *Revision 5* geändert (Bei manchen Revisionen werden nur ein paar Varianten geändert. Zum Beispiel von *Revision 27* bis *Revision 28* werden nur drei Varianten geändert. Siehe Abbildung 5.2). Wir wählen zufällig eine Variante von den geänderten Varianten aus. Danach ändern wir per Hand die Quelltexte von *Revision 4* im Eclipse für die ausgewählte Variante, damit die Quelltexte gleich wie in *Revision 5* sind. Dieser Vorgang wird mit Eclipse gemacht und mit *VariantSync* unterstützt. Die per Hand gemacht Änderungen werden von *VariantSync* gespeichert.

4. **Synchronisierung in anderen Varianten**

Zunächst öffnen wir den Änderung-View von *VariantSync*. Mit den Änderung-View können wir klar sehen, wie die ausgewählte Variante geändert wird. Abbildung 5.3 zeigt ein Beispiel. Für die Evaluierung haben wir für *Revision 4* zu *Revision 5* zufällig *Variante7* ausgewählt und per Hand geändert. Es gibt insgesamt acht Änderungen. Sechs Dateien wurden hinzugefügt und zwei Dateien wurden geändert. Wir können auf die Zeile doppleklicken, die eine Änderung beschrieben hat. Danach wird ein Dialog dargestellt (siehe Abbildung 5.4). Nach der Auswahl von Features für die Änderung werden die Projekte für Synchronisierung automatisch ausgewählt. Wenn wir die Projekte für Synchronisierung festgelegt haben, können wir ausgewählte Projekte durch *VariantSync* automatisch synchronisieren lassen.

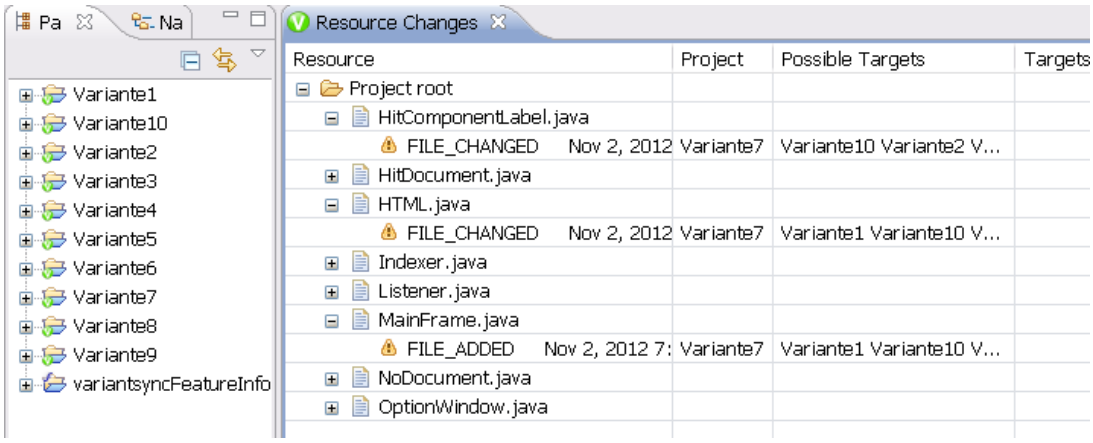


Abbildung 5.3: Per Hand nachvollzogene Änderungen für Variante7

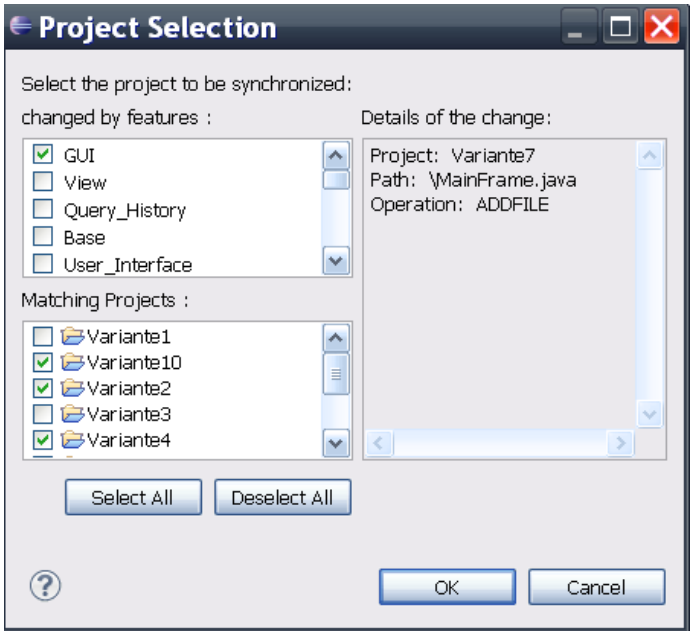


Abbildung 5.4: Auswahl von Features für Änderung

Für die in Abbildung 5.4 auf der vorherigen Seite gezeigt Änderung haben wir Feature *GUI* ausgewählt. Nach der Auswahl werden *Variante1*, *Variante3* und *Variante5* automatisch deselektiert, weil die drei Varianten kein Feature *GUI* haben (siehe Tabelle 5.1 auf Seite 62). Um Features für eine Änderung zu entscheiden ist die Vergleich von Revisionen des Projektes *DesktopSearcher* sehr hilfreich. Durch die Vergleich von *Revision 4* und *Revision 5* des Projektes *DesktopSearcher* haben wir gefunden, dass die Datei *MainFrame.java* nur unter Ordner *GUI* geändert wird. Deshalb wählen wir Feature *GUI* für die Änderung der Datei *MainFrame.java* in *Variante7*. Dieser Fall ist einfach für die Entscheidung von Features.

Es ist manchmal schwierig, die Features für eine Änderung zu bestimmen. Und manchmal ist eine Änderung gleichzeitig für mehrere Features. Zum Beispiel von *Revision 7* bis *Revision 8* wurde Quelltext *Indexer.java* in *Variante10* geändert. Diese Änderung beeinflusst gleichzeitig drei Features : HTML, TXT, Base. Von *Revision 14* bis *Revision 15* wurde eine Datei *.order* in dem Projekt hinzugefügt. Es gibt keine Änderungen für die Quelltexte. Aber sind die von dem Projekt generierte Quelltexte für Varianten geändert. In diesem Fall ist es schwer die Features für eine Änderung zu bestimmen.

5. **Synchronisierung kontrollieren**

   Nach dem Schritt 4 kontrollieren wir die zehn Varianten, ob die Quelltexte von jeder Varianten wie in nächster Revision sind. Falls es noch Varianten gibt, die nicht gleich wie in nächster Revision sind. Dann wählen wir noch mal zufällig eine Variante aus und per Hand ändern, damit die Quelltexte gleich wie in nächster Revision sind. Das heißt, dass Schritt 3 und 4 werden noch mal ausgeführt. Falls die Quelltexte gleich wie in nächster Revision sind, dann gehe auch zu Schritt 3. Aber vergleichen wir die in Eclipse stehende Quelltexte mit übernächster Revision. Zum Beispiel haben wir zuerst die Quelltexte für Revision 4 für zehn Varianten in Eclipse kopiert, durch per Hand Änderung und Synchronisierungen sind die in Eclipse stehende Quelltexte gleich wie die Quelltexte für Revision 5. Danach gehen wir zu dem Schritt 3 und vergleichen wir die in Eclipse stehende Quelltexte mit den Quelltexte für Revision 6. Die Reihenfolge ist *Revision 5* bis *Revision 91* (letzte Revision für Quelltexte).

5.3 Ergebnisse

Nach dem Schritt 1 (siehe Abschnitt 5.2) haben wir insgesamt 46 Revisionen für den 10 Varianten erzeugt. Ein paar Revisionen von dem Projekt *DesktopSearcher*

werden ignoriert. Tabelle 5.2 zeigt die ignorierte Revisionen und die Gründe. Zum Beispiel werden *Revision 35* bis *Revision 42* und *Revision 49* bis *Revision 53* ignoriert. Diese Revisionen sind nur für die Dokumentationen des Projekts. Von den Revision erzeugte Quelltexte für zehn Varianten haben keine Änderungen. *Revision 46* bis *Revision 47* werden auch ignoriert. In den Revisionen werden nur ein paar Ordner (keine Dateien) gelöscht. Es beeinflusst nicht die Generierung der Quelltexte für Varianten. *Revision 44* wird auch ignoriert. Die Revision hat die Datei

Ignorierte Revisionen	Grund
R35-R42, R49-R53 R56, R59, R60, R65-R80 R82, R88, R92	Nur Änderungen an den Dokumentationen
R44	„.order“ passt nicht zu Feature-Modell
R46, R47	Löschen leerer Ordner
R58	Die Änderungen sind nur an dem Feature-Modell. Bis R61 gibt es keine Änderungen an den Quelltexte.
R83-R86	Verschieben von Quelltextdateien

Tabelle 5.2: Ignorierte Revisionen und die Gründe

.order geändert, und es passt nicht zu dem Feature-Modell in *Revision 44*. Bei *Revision 44* können wir nicht richtig Quelltexte für Varianten generieren. Bei *Revision 45* wird das Feature-Modell geändert, dann können wir Quelltexte für die 10 Varianten generieren. *Revision 58* bis *Revision 60* werden ignoriert. *Revison 59* und *Revison 60* haben nur die Dokumentateionen geändert. *Revison 58* und *Revision 61* haben zwei mal Änderungen bei dem Feature-Modell. Dabei generieren wir ein mal Quelltexte für Varianten. Nach *Revison 81* wurden die Quelltexte des Projektes *DesktopSearcher* von Ordner *trunk/Search_Engine_Feature_oriented* nach Ordner *trunk/Search_Engine_Feature_oriented/DesktopSearcher-AHEAD* verschoben. Wenn wir Quelltexte für Varianten generieren, sollen wir die Dateien in Ordner *trunk/Search_Engine_Feature_oriented/DesktopSearcher-AHEAD* benutzen.

Wir haben mit *VariantSync* 10 Varianten 43 Revisionen synchronisiert. Tabelle 5.3 auf der nächsten Seite zeigt die Details für Synchronisierungen von *Revision 4* bis *Revision 91*. Die ignorierte Revisionen werden mit „–“ gezeigt. Die zweite Spalte R_{PHGV} ist die Reihenfolge der per Hand geänderte Varianten. Die Bedeutungen der Zahlen in Tabelle 5.3 auf der nächsten Seite sind folgende:

- **Zahl der geänderten Varianten (Abk. Z_{GV})**
  Die Zahl der geänderten Varianten bedeutet, wieviele Varianten zwischen Revisionen geändert wurden. Zum Beispiel von *Revision 4* bis *Revision 5* werden

alle 10 Varianten geändert. Von *Revision 6* bis *Revision 7* werden nur 7 Varianten geändert.

- **Zahl der per Hand geänderten Varianten (Abk. Z_{PHGV})**
  Die Zahl der per Hand geänderte Varianten bedeutet, wieviele Varianten in dem Schritt 3. (siehe Abschnitt 5.2) geändert wurden. Zum Beispiel von *Revision 4* bis *Revision 5* wird nur eine Variante per Hand geändert.

- **Zahl der Operation per Hand (Abk. ZO_{PH})**
  Die Zahl der Operation per Hand bedeutet, wieviele Quelltexte per Hand geändert wurden. Jede Hinzufügung, Löschung und Änderung für einen Quelltext wird als eine Operation gerechnet.

- **Zahl der Operation per VariantSync (Abk. ZO_{PVS})**
  Die Zahl der Operation per VariantSync bedeutet, wieviele Quelltexte durch VariantSync automatisch geändert wurden.

- **Zahl der Unterschiede (Abk. Z_U)**
  Die Zahl der Unterschiede bedeutet, wieviele Quelltexte hinzugefügt, gelöscht und geändert werden müssen, wenn die Quelltexte für alle zehn Varianten von einer Revision gleich wie folgende Revision geworden sind. Zum Beispiel von *Revision 4* bis *Revision 5* müssen wir für alle zehn Varianten 45 Quelltexte hinzufügen und 12 Quelltexte ändern. Es gibt insgesamt 57 Unterschiede.

Tabelle 5.3: Synchronisierung von *Revision 4* bis *91*

	R_{PHGV}	Z_{GV}	Z_{PHGV}	ZO_{PH}	ZO_{PVS}	Z_U
R4-R5	V7	10	1	8	49	57
R5-R6	V2/1	10	2	9	44	53
R6-R7	V6	7	1	3	18	21
R7-R8	V1/2/3/10	10	4	8	36	41
R8-R9	V4/8	7	2	9	26	35
R9-R10	V1/10/4/8/9/2/6	10	7	13	35	34
R10-R11	V2	3	1	1	2	3
R11-R12	V1/3/10/2	10	4	4	13	10
R12-R13	V4/10/6/3/8	10	5	9	27	24
R13-R14	V7/8	3	2	4	2	6
R14-R15	V6/10/4/2/7/8	7	6	13	8	21
R15-R16	V3/7/4/8	10	4	6	31	37
R16-R17	V8/10	3	2	4	5	9
R17-R18	V10/9	10	2	6	16	19

folgt Tabelle 5.3

	R_{PHGV}	Z_{GV}	Z_{PHGV}	ZO_{PH}	ZO_{PVS}	Z_U
R18-R19	V1	7	1	1	6	7
R19-R20	V8	3	1	2	4	6
R20-R21	V3/6/5/7/4	6	5	6	6	12
R21-R22	V1/5	3	2	4	5	9
R22-R23	V7/6/10	7	3	3	5	7
R23-R24	V8/10	3	2	4	5	9
R24-R25	V8	3	1	1	2	3
R25-R26	V9/2/4/7	10	4	8	23	30
R26-R27	V7	3	1	1	2	3
R27-R28	V8	3	1	2	4	6
R28-R29	V9	3	1	1	2	3
R29-R30	V7	3	1	1	2	3
R30-R31	V9	3	1	1	2	3
R31-R32	V2/8	7	2	2	5	7
R32-R33	0	0	0	0	0	0
R33-R34	V3	6	1	1	5	6
R35 bis R42	–	–	–	–	–	–
R34-R43	V1/3	10	2	2	8	10
R44	–	–	–	–	–	–
R43-R45	V10/3/4/7	10	4	5	9	14
R46,R47	–	–	–	–	–	–
R45-R48	V8	3	1	1	2	3
R49 bis R53	–	–	–	–	–	–
R48-R54	V6/10/7/9	4	4	4	0	4
R54-R55	V3	6	1	1	5	6
R56	–	–	–	–	–	–
R55-R57	V4/6	5	2	2	3	5
R58 bis R60	–	–	–	–	–	–
R57-R61	V3/5/1/2/6/9/10/4/7/8	10	10	15	15	30
R61-R62	V2/5/8	6	3	4	8	12
R62-R63	V3/1/4/7/5/10/6/9	8	8	10	10	19
R63-R64	0	0	0	0	0	0
R65 bis R80	–	–	–	–	–	–
R64-R81	V3	6	1	1	5	6
R82 bis R86	–	–	–	–	–	–
R81-R87	V1/6/9/7/10/5/2/4	10	8	41	103	143
R88	–	–	–	–	–	–

folgt Tabelle 5.3

	R_{PHGV}	Z_{GV}	Z_{PHGV}	ZO_{PH}	ZO_{PVS}	Z_U
R87-R89	V7/1/4/10/2/5/6/9/3/8	10	10	31	69	93
R89-R90	V1/10/6	10	3	10	31	39
R90-R91	V3	10	1	1	9	10
R92	–	–	–	–	–	–

Zwischen der Revisionen haben *Revision 32* bis *Revision 33* und *Revision 63* bis *Revision 64* keine Änderungen (siehe Tabelle 5.3 auf Seite 69). Bei der *Revision 33* werden zwei Quelltexte in dem Projekt *DesktopSearcher* hinzugefügt. Aber sind die von dem Projekt generierte Quelltexte für Varianten gleich wie in *Revision 32*. Die Ursache ist, dass der Name von dem Ordner nicht zu dem Feature-Modell passt. Bei der Generierung von Quelltext für Varianten werden neu hinzugefügte Ordner von dem Generator ignoriert.

Mit den 46 Revisionen für zehn Varianten haben wir mit *VariantSync* erfolgreich die Varianten synchronisiert. Abbildung 5.5 zeigt die Statistik für die Synchronisierungen.

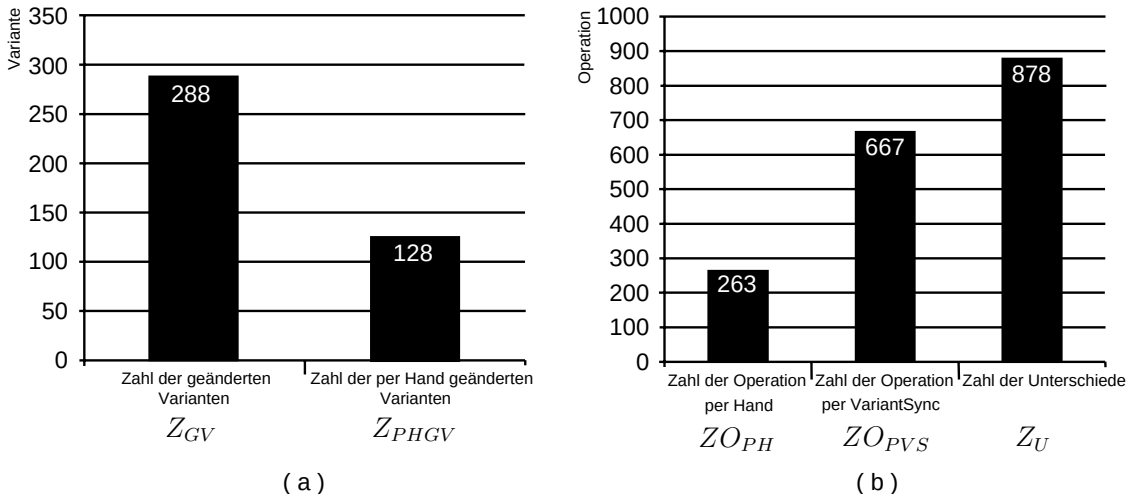


Abbildung 5.5: Statistik für Synchronisierungen

Der Teil *(a)* der Abbildung 5.5 zeigt die Summe der Zahl von geänderten Varianten (Z_{GV}) und die Summe der Zahl von per Hand geänderten Varianten (Z_{PHGV}) für *Revision 4* bis *Revision 91*. Es gibt insgesamt 288 Änderungen für Varianten. Und wurden insgesamt 128 Änderungen für Varianten per Hand gemacht. Der Teil *(b)* der Abbildung 5.5 zeigt die Summe der Zahl der per Hand ausgeführte Operationen (ZO_{PH}) für Quelltexte, die Summe der Zahl der per *VariantSync* automatisch ausgeführte Operationen (ZO_{PVS}) für Quelltexte, und die Summe der Zahl der Unterschiede (Z_U) für Quelltexte. Von *Revision 4* bis *Revision 91* wurden insgesamt

263 Operationen (Hinzufügung, Löschung und Änderung) für Quelltexte per Hand ausgeführt, und wurden insgesamt 667 Operationen für Quelltexte per *VariantSync* automatisch ausgeführt. Wenn es alles per Hand ändern, sollen insgesamt 878 Operationen ausgeführt werden.

Aus der Statistik von Abbildung 5.5 auf der vorherigen Seite geht hervor, dass in dieser Evaluierung ungefähr 55.56%[4] der Unterschiede von Varianten durch *VariantSync* automatisch synchronisiert werden. Und ungefähr 70.05%[5] der Operationen (Hinzufügungen, Löschungen und Änderungen) von Quelltexte werden durch *VariantSync* automatisch ausgeführt.

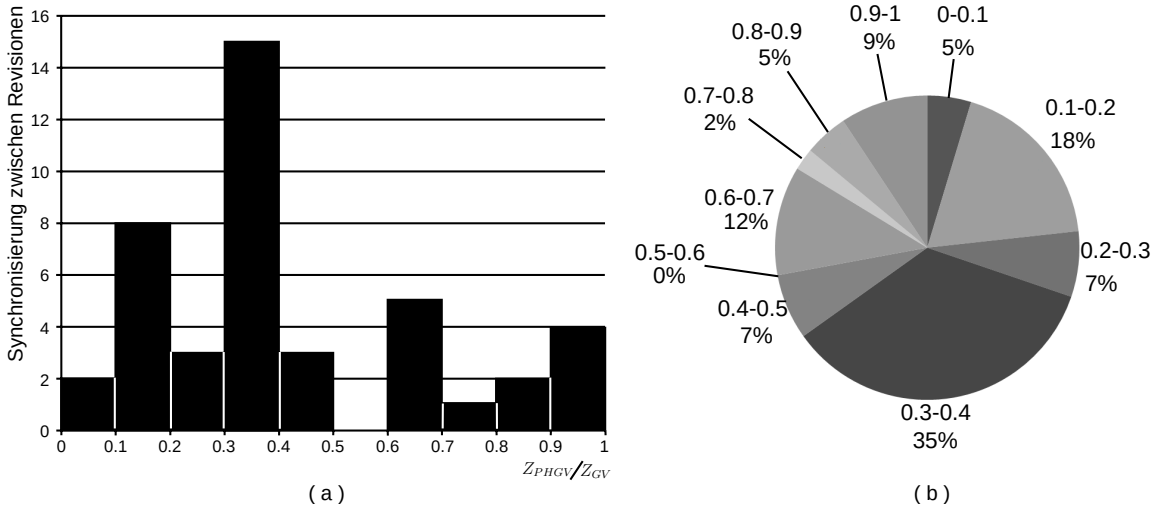


Abbildung 5.6: Aufwand und Häufigkeit von den Synchronisierungen

Die Abbildung 5.6 zeigt die Aufwand und Häufigkeit für die Synchronisierungen für Revisionen. Der Teil *(a)* der Abbildung 5.6 stellt die horizontale Achse die Aufwand dar. Die Aufwand rechnen wir mit Z_{PHGV} durch Z_{GV}. Die vertikale Achse stellt die Zahl der Synchronisierungen. Zum Beispiel gibt es zwei Synchronisierung, die die Aufwand großer als 0 und gleich oder kleiner als 0.1 sind. Die Aufwand der Synchronisierung von *Revision 4* bis *Revision 5* und die Aufwand der Synchronisierung von *Revision 90* bis *Revision 91* sind 0.1[6]. Der Teil *(b)* der der Abbildung 5.6 zeigt, dass die häufigste Aufwand der Synchronisierung zwischen 0.3 und 0.4 liegen. Es beträgt auf 35% von alle.

Bei der Evaluierung haben wir noch ein Test gemacht. Die Quelltexte von zehn Varianten werden ein Kopie erstellt. Bei jeder Synchronisierung werden die geänderte Zeilen von Quelltexte für jede Synchronisierte Variante mit den ausgewählten Features markiert. Für die Änderungen, die nicht synchronisiert werden können, werden

[4] $(Z_{GV} - Z_{PHGV})/Z_{GV} \approx 0.555556$
[5] $(Z_U - ZO_{PH})/Z_U \approx 0.700456$
[6] $Z_{PHGV}/Z_{GV} = 1/10 = 0.1$

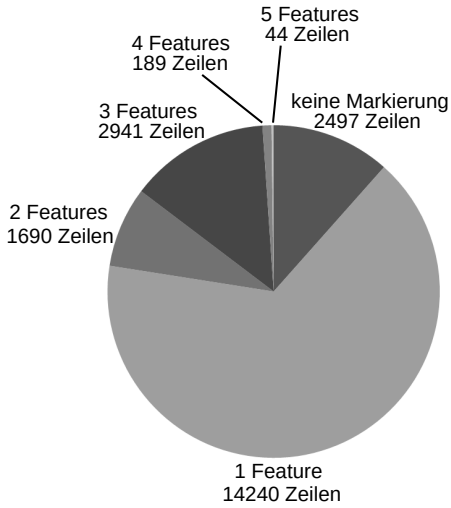


Abbildung 5.7: Feature-Bestimmung für Quelltexte

nur die geänderte Zeilen von geänderte Variante markiert. Die Abbildung 5.7 zeigt die Feature-Bestimmung für die Quelltexte. Bis *Revision 91* haben die Quelltexte von zehn Varianten insgesamt 21601 Zeilen. Davon werden 14240 Zeilen für ein bestimmtes Feature markiert. 1690 Zeilen werden für zwei Features markiert. Und 2941 Zeilen werden für drei Features markiert. Es gibt nur 2497 Zeilen, die keine Markierung haben. Das bedeutet, dass wir ungefähr 65.9% der Quelltexte für ein Feature feststellen können. Am Anfang bei wenigen Varianten können wir mit mit *VariantSync* die Varianten entwickeln. Wenn wir später eine SPL für mehrere Varianten erstellen, können wir mit den Informationen der Feature-Bestimmung profitieren. Es kann die Migration zu SPL vereinfachen.

5.4 Interpretation des Ergebnisses

Aus der Abbildung 5.5 auf Seite 71 erfahren wir noch, dass ZO_{PH} *plus* ZO_{PVS} größer als Z_U ist. Die Grund ist die inhaltliche Änderungen von Quelltexte. Manche Quelltexte werden durch mehr als eine Operation geführt, damit sie gleich wie in andere Revision sind. Zum Beispiel von *Revision 7* bis *Revision 8* wird der Quelltext *Indexer.java* in *Variante7* und *Variante9* von *VariantSync* zwei mal synchronisiert. Wir ändern zuerst die Quelltexte in *Variante1*. Die Änderung vom Quelltext *Indexer.java* wurde von *Variante1* nach *Variante7* und *Variante9* synchronisiert. Nach dieser Synchronisierung sind Quelltext *Indexer.java* noch nicht gleich wie in *Revision 8*. Das ist logisch, weil *Variante1* keine Feature {TXT} enthält. Die Änderungen vom Quelltext *Indexer.java* in *Variante7* und *Variante9* beeinflussen jedoch Features {HTML, TXT, Base}. Die per Hand ausgeführte Änderung vom Quelltext *Indexer.java* von *Variante10* wurde auch nach *Variante7* und *Variante9* syn-

chronisiert. Nach diesen zwei Synchronisierungen wurde Quelltext *Indexer.java* in *Variante7* und *Variante9* gleich wie in *Revision 8*.

Die Aufwand der Synchronisierung ist abhängig von der Änderungen zwischen Revisionen. Es gibt zwei Voraussetzungen. Wenn die zwei Voraussetzungen erfüllt werden, können die Änderungen von einer Varianten nach anderen Varianten synchronisiert werden. Und damit die Quelltexte von allen Varianten gleich wie in der nächste Revision sind. Die zwei Voraussetzungen sind:

1. Die Änderungen von einer Variante zwischen Revisionen die alle betroffene Features enthalten, die die zwischen Revisionen beeinflusst werden.
2. Die von einer Änderung eines Quelltextes beeinflusste Features werden in dem gleich Quelltext in anderten Varianten enthalten.

Die Änderung bei *Variante7* von *Revision 4* bis *Revision 5* hat die zwei Voraussetzungen erfüllt. Deshalb können wir nur *Variante7* per Hand ändern und die Änderung nach anderen Varianten synchronisieren. Damit die Quelltexte von allen Varianten wie in nächster Revision sind. Die von *Revision 4* bis *Revision 5* beeinflusste Features sind {GUI, HTML, Base}. Von *Revision 4* bis *Revision 5* hat *Variante7* insgesamt Sieben Quelltexte geändert. Die Quelltexte beeinflussen jeweils ein Feature. Die Änderung bei *Variante2* von *Revision 5* bis *Revision 6* hat nur die erste Voraussetzung erfüllt. Deshalb nach der Synchronisierung der Änderung von *Variante2* müssen wir noch Variante per Hand ändern. Die Änderung vom Quelltext *MrpinkMain.java* von *Variante2* beeinflusst Features {Base, Interface, GUI}. Aber die Änderung vom Quelltext *MrpinkMain.java* von *Variante1* beeinflusst nur Features {Base, Interface}. Deshalb kann die Änderung beim Quelltext *MrpinkMain.java* von *Variante2* nicht nach *Variante1* synchronisiert werden.

Die von uns gerechnete Aufwand ist noch abhängig von den beeinflussten Features. Die Änderung bei *Variante2* von *Revision 10* bis *Revision 11* hat die zwei Voraussetzungen erfüllt. Von *Revision 10* bis *Revision 11* haben wir nur eine Variante per Hand geändert. Durch die Synchronisierungen werden die Quelltexte in allen anderen Varianten gleich wie in der nächste Revision geändert. Aber von *Revision 10* bis *Revision 11* werden nur drei Varianten geändert. Weil die von *Revision 10* bis *Revision 11* beeinflusste Features {Multi_Directory} sind. Und es gibt nur drei Varianten, die das Feature enthalten. Die Aufwand in dem Fall ist 0.333 und liegt zwischen 0.3 und 0.4.

5.5 Diskussion

In dieser Evaluierung haben wir gefunden, dass es ein paar Faktoren gibt, die die Effizienz von *VariantSync* beeinflussen können. Die Faktoren sind:

Die Reihenfolge der per Hand geänderte Varianten

Mit verschiedener Reihenfolge der per Hand geänderte Varianten ist die Effizienz von *VariantSync* auch unterschiedlich. Von *Revision 4* bis *Revision 5* haben wir nur eine Variante per Hand geändert (siehe Tabelle 5.3 auf Seite 69). Die andere Varianten werden mit *VariantSync* synchronisiert. Aber es gibt auch andere Möglichkeit. Die Reihenfolge der per Hand geänderte Varianten kann als *V3/1/2* laufen. Dabei müssen wir 3 Varianten per Hand änderen. Die Grund ist, dass die Änderungen von *Variante3* sind für Feature {Base}. Die Änderungen von *Variante1* sind für Feature {Base, HTML}. Die Änderungen von *Variante2* sind für Feature {Base, GUI}. Die von *Revision 4* bis *Revision 5* beeinflusste Features sind {GUI, HTML, Base}. Wir müssen *Variante3*, *Variante1* und *Variante2* per Hand ändern, dann können wir alle die Features {GUI, HTML, Base} berücksichtigen. Die Änderungen von *Variante7* enthalten alle die von *Revision 4* bis *Revision 5* beeinflusste Features:{Base, HTML, GUI}. Und die Änderungen von *Variante7* kann nach anderen Varianten synchronisiert werden. Deshalb wenn wir zuerst *Variante7* per Hand ändern und danach die Änderungen nach anderen Varianten synchronisieren, brauchen wir nur eine Variante per Hand ändern.

Die Größe der Änderung

Die Größe der Änderung kann manchmal auch die Effizienz von *VariantSync* beeinflussen. Die Größe bedeutet hier nicht die Anzahl der unterschiedlichen Zeilen von dem Quelltext sondern die Zahl der Features, die von der Änderung beeinflusst werden. Wenn eine Änderung bei einem Quelltext von einer Variante viele Features beeinflusst hat, und sie darinnen Verbund haben, kann diese Änderung nur nach wenige Varianten synchronisiert werden, die auch die beeinflusste Features haben. Deshalb sollen wir für eine Änderung eines Quelltextes möglichst wenige Features beeinflussen. Damit können die Änderungen möglichst nach anderen Varianten synchronisiert werden.

Bestimmung der Feature für eine Änderung

Die Features für eine Änderung zu bestimmen ist auch wichtig für die Effizienz von *VariantSync*. Bei der Synchronisierung sollen wir Features für eine Änderung auswählen (siehe Abbildung 5.4 auf Seite 66). Bei einer falschen Auswahl von Features kann eine Änderung falsch synchronisiert werden. In dieser Evaluierung ist es manchmal schwer die Features für eine Änderung zu bestimmen. Mann muss die Änderung lesen und verstehen. Allerdings hat ein Programmierer wahrscheinlich mehr Einblick in den Quelltext. Dennoch ist unsere Zuordnung nicht unrealistisch, da auch Programmierer bei der Zuordnung zu Features Fehler machen können.

5.6 Zusammenfassung

In dieser Evaluierung mit dem Projekt *DesktopSearcher* haben wir mit *VariantSync* 10 Varianten 43 Revisionen synchronisiert. Dabei werden ungefähr 55.56% der Unterschiede von Varianten durch *VariantSync* vollautomatisch synchronisiert und werden ungefähr 70.05% der Änderungen von Quelltexte durch *VariantSync* vollautomatisch ausgeführt. Daraus können wir sehen, dass die Idee von *VariantSync* funktioniert. Es ist wesentlich effizienter als Versionsverwaltungssysteme für die Entwicklung von Varianten. Mit Versionsverwaltungssysteme um die Varianten zu entwickeln müssen Entwickler die Änderungen von einer Variante manuell durch Merge in allen anderen möglichen Varianten einbringen. Mit *VariantSync* können wir einmal die Änderungen von einer Variante in allen anderen möglichen Varianten einbringen. Außerdem müssen Entwickler entscheiden, ob eine Änderung in einer Variante durch Merge eingebracht werden soll. Mit *VariantSync* können wir von Domänenwissen profitieren. Damit ist diese Entscheindung vereinfacht.

6. Verwandte Arbeiten

Semistrukturierter Merge

Bei Versionsverwaltungssystemen ist Merge eine übliche Operation, um die Änderungen des Quelltexts zu vermischen. Dabei gibt es oft Konflikte, die man nicht mit dem Programm automatisch lösen kann. Es gibt drei verschiedene Arten vom Merge. Unstrukturierter Merge wird wegen der Sprachunabhängigkeit in der Praxis oft verwendet. Strukturierter Merge ist meistens in der Forschung der Wissenschaft mit dem Ziel die Konflikte von unstrukturiertem Merge zu lösen [Buf95, Wes91]. Semistrukturierter Merge wird in 2011 in [ALB+11] vorgestellt. Unstrukturierter Merge arbeit nur auf Textzeilen. Falls in zwei Änderungen zwei verschiedene Zeilen bei der gleichen Position hinzugefügt wurden, wird ein Konflikt von unstrukturierter Merge ausgegeben. Solche Reihenfolgekonflikte können durch strukturierte und semistrukturierter Merge gelöst werden. Strukturierter Merge ist für eine bestimmte Sprache entwickelt und benutzt sprachspezifische Wissen für die Konfliktlösung. Semistrukturierter Merge erbt die Eigenschaften von strukturiertem und unstrukturierterm Merge. Mit semistrukturierter Merge werden Software-Artefakte als Bäume dargestellt. Dadurch wird die Informationen erzeugt, wie die Knoten eines bestimmten Typs (z.B. Methoden oder Klassen) und deren Teilbäume zusammengeführt werden. Die Reihenfolge von Methoden hat keine Auswirkung. Falls ein Konflikt in einer Methode auftritt, ist die Reihenfolge sehr wichtig. In diesem Fall arbeitet semistrukturierter Merge als unstrukturierter Merge. Bei VariantSync werden die Änderungen zwischen die Varianten synchronisiert. Derzeit wird ein unstrukturierter Merge benutzt. Das bedeutet, dass die Synchronisierungen in manchem Fall nicht automatisch durchgeführt werden können. Um die Synchronisierung zu optimieren, kann semistrukturierter Merge für VariantSync benutzt werden. Im Vergleich zum dem semistrukturierten Merge macht VariantSync für eine Änderung Vergleichun-

gen für alle anderen Varianten, und synchronisiert die Änderung automatisch nach anderen Varianten. Außerdem benutzt VariantSync Domänewissen um die synchronisierungen zu entscheiden. Weil falls eine Änderung mit anderen Varianten keinen Konflikt hat, soll die Änderung manchmal nicht in alle anderen Varianten synchronisieren. Vor der Synchronisierung einer Änderung können wir ein oder mehrere Features für die Änderung auswählen, um das zu entscheiden, ob die Änderung nach einer bestimmte Variante synchronisieren soll.

Eine Fallstudie des Verison Editor

David Atkins, Thomas Ball, Todd Graves und Audris Mockus haben im Jahr 2000 eine Methode für Evaluierung eines Werkzeugs präsentiert, damit kann man die Bemühungen der Entwickler einschätzen. Sie haben eine Fallstudie mit Version Editor (VE) gemacht [ABGM02], und heraus gefunden, dass die Entwickler ungefähr 40% produktiver waren bei Verwendung von VE als bei Standard-Text-Editoren. VE ist ein Text-Editor, mit dem man verschiedene Varianten mit einer gemeinsamen Quelltextbasis entwickeln kann. Die Theorie ist ähnlich wie beim Präprozessor. Es benutzt auch Kommentare für die Unterschiede zwischen Varianten. Aber mit VE können die Entwickler nicht die zahlreiche Kommentare sehen. VE zeigt die extrahierten Quelltexte für die Entwickler. Im Vergleich zum VariantSync entwickeln die Entwickler verschiedene Varianten mit einer gemeinsamen Quelltextbasis. Dieses Verfahren hat ein paar Nachteile: Man kann nicht gleichzeitig zwei Varianten entwickeln. Weil man kann nicht einen Quelltext für zwei Varianten ändern. Außerdem ist die Quelltext nicht für normalen Compiler verfügbar. Die Quelltexte müssen zunächst mit VE gearbeitet werden. Weil VE die Kommentare nur für verschiedene Varianten oder Versionen benutzt, kann man später schwer die Quelltext für Features zuordnen. Außerdem es ist schwer die Änderungen von einer Variante nach anderen Varianten zu bringen.

Werkzeugeunabhängiges Varianten-Management mit dem Leviathan Filesystem

Mit Präprozessor kann man Varianten entwickeln. Aber es gibt jedoch Nachteile. Die zahlreiche Kommentare machen die Quelltexte schwer zu lesen und warten. Mit Werkzeuge kann man dieses Problem lösen. Manche Werkzeuge bieten Entwicklern die schon gearbeitete Quelltexte. Damit können die Entwickler eine klar Sicht von Quelltexten haben. Aber dieses Verfahren ist Werkzeugabhängig. Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat und Daniel Lohmann haben im Jahr 2010 eine neue Methode ausprägt [HEB+10]. Sie haben ein Dateisystem entwickelt. Es heißt *Leviathan*. Mit dem System können die Entwickler auch eine klar Sicht von Quelltexten haben. Außerdem ist es Werkzeugeunabhängig. Es wird auf der Ebene des Dateisystems implementiert. Die Eingaben für dieses System sind

Konfigurationen und die kommentierte Quelltexte. Die Entwickler bekommen Quelltexte für bestimmte Varianten, und können mit beliebigen Werkzeugen die Quelltexte arbeiten. Nach den Arbeiten werden die Änderungen zurück in den kommentierten Quelltext gespeichert. Dieses Verfahren ist Werkzeugeunabhängig bei der Entwicklung. Aber es gibt auch einige Nachteile. Für das System sind die Eingabe auch kommentierte Quelltexte. Es unterstützte jetzt C-Präprozessor. Aber es gibt noch viele Quelltexte, die mit anderen Präprozessor unterstützt. Außerdem wie können wir mit dem System mit wenigen vorhandenen Varianten, die nicht mit Präprozessor unterstützt werden, weiter entwickeln? Bei den Arbeiten mit dem System können die Entwickler nicht neue Kommentare hinzufügen. Die Entwickler können nur die vorhandene kommentierte Fragmente ändern. Wenn die Entwickler neue Fragmente hinzufügen möchten, müssen sie die Eingabe des System ändern. Mit VariantSync haben wir keine solche Problem. VariantSync synchronisiert Änderungen zwischen Varianten. Damit die gleiche Fragmente zwischen Varianten immer gleich sind. Bei der Synchronisierung sollen wir Features auswählen, um das zu entscheiden, ob eine Änderung in einer Variante synchronisiert werden soll.

Variabilität-Mining mit LEADT

Es ist eine große Herausforderung und erhebliche Risiken um eine SPL für die existierende Software zu erstellen. Dafür haben Christian Kästner, Alexander Dreiling und Klaus Ostermann ein Werkzeug LEADT entwickelt [KDO11]. Das Ziel des Werkzeug ist für die Entwickler bei der Lokalisierung von Quelltext, Dokumentation und der Extraktion der Implmentierung eines Features von existierenden Software (Variabilität-Mining) eine Unterstützung zu bieten. Bei der Variabilität-Mining suchen die Entwickler mit ein paar Quelltext-Fragmente (In [KDO11] wurde sie *seeds* gannant.), die definitiv zu eim Feature gehören, weiter Fragmente für ein Feature. Bei der Lokalisierung der Quelltext für Features bietet LEADT ein paar Vorschlage für die Entwickler. Dabei ist das Typsystem der Schlüssel für Vorschlage-Mechanismus. VariantSync kann für LEADT Untertützungen bieten. Wenn die Entwickler lange Zeit mit VariantSync Varianten entwickeln und Informationen für Features speichern, können die Entwickler zahlreiche Fragmente für ein Feature bestimmen. Diese Fragmente können bei der Variabilität-Mining als *seeds* für LEADT benutzt werden. Die Informationen von VariantSync für Fragmente, die nicht für ein Feature bestimmt aber mit mehr als ein Feature markiert sind, können als Ergänzung für den Vorschlage-Mechanismus von LEADT benutzt werden.

Kombinieren Ähnliche Produkte in Produktlinien

Es gibt noch eine Arbeit für Erstellung einer SPL für die existierende Softwares. Diese Arbeit werden von Julia Rubin und Marsha Chechik gemacht [RC12]. Die existierende Ähnliche Softwares werden in Produktlinien kombiniert. Für die Kom-

binierung werden ein Modell erstellt. Das Modell darstellt die Ähnliche Softwares. Mit Vergleich und Vermischung von den Modelle für die Ähnliche Softwares werden ein Modell für die alle Ähnliche Softwares erstellt. Um die Varianten (Ähnliche Softwares) in eine Produktlinie zu kombinieren ist notwendig, wenn es mehr und mehr Varianten werden. VariantSync ermöglicht eine effiziente Entwicklung von Varianten. Deshalb ist diese Kombinierung vielleicht nicht notwendig. Falls es später sehr viele Varianten gibt, kann VariantSync ein Unterstützung für die Kombinierung bieten, da VariantSync ein Mapping zwischen Features und Quelltext protokolliert. Im Vergleich zu dem vorgestellten Ansatz hat VariantSync echte Features und nicht nur Produktfeatures. Es basiert auf das Feature-Modell. Damit ist es einfach, um die Unterschiede zwischen Produkten zu beschreiben.

7. Zusammenfassung

Software-Varianten sind wichtig in der Softwareentwicklung. Ein erfolgreiches Softwareprodukt soll in einer Reihe von Varianten entwickelt werden. Damit können Softwarehersteller möglichst gut die Anforderungen von Kunden erfüllen. Aber wie man effizient Varianten entwickeln kann, ist eine Herausforderung für Softwarehersteller. Es gibt traditionelle Methoden. Zum Beispiel kann man mit Branches vom Versionsverwaltungssystem und mit Präprozessoren Varianten entwickeln. Es gibt noch moderne Methode. Zum Beispiel es gibt ein paar Programmierparadigmen so wie FOP, DOP und AOP. Aber die Methoden haben eigene Nachteile. Zum Beispiel ist mit FOP, DOP und AOP erstellte SPL ist für die Entwicklung nur ab einer gewissen Anzahl von Varianten geeignet, weil die Produktlinientechnologien neu sind, damit weniger ausgereifte Werkzeuge verfügbar sind.

In dieser Arbeit wird eine neue Methode für die Entwicklung bei den wenige Varianten dargestellt, und wird eine prototypische Implementierung für diese Methode entwickelt. Wir nennen die Implementierung *VariantSync*. Wir Entwickeln die Varianten mit eigenen Quelltexten, dabei synchronisieren wir die Änderungen von einer Variante nach anderen Varianten. Zum Beispiel eine neue Funktionalität oder eine Beseitigung eines Fehlers von einer Variante werden auf andere Varianten übernommen. Bei der Synchronisierung ist es offensichtlich, dass nicht alle Änderungen von einer Variante auf alle andere Varianten übernommen werden sollen. Um zu entscheiden, in welche anderen Varianten eine Änderung übernommen werden soll, benutzen wir die enthaltene Features von jeder Variante. Bei jede Synchronisierung von einer Änderung entscheiden wir zunächst, welche Features von dieser Änderung beeinflusst werden. Danach kontrollieren wir, ob die andere Varianten die Features enthalten. Wenn eine Variante die entscheidenden Features enthält und es dabei keine Konflikte gibt, synchronisieren wir diese Änderung nach der Variante.

Um eine Änderung von einer Variante nach anderer Variante zu synchronisieren, verwenden wir die sehr bekannte Diff- und Patch-Operation. Die Operationen werden schon in der Programmbibliothek *java-diff-utils* implementiert. Wir vergleichen eine Variante und die vor eine Änderung stehende Variante mit Diff-Operation, damit können wir wissen, welche Zeile in den zwei Varianten gleich sind. Danach erzeugen wir einen passenden Patch, der die Änderungen enthält, für die synchronisierte Variante. Zum Schluss führen wir die Patch-Operation aus, um die Änderung auf die synchronisierte Variante zu übernehmen. Für die Änderung wie Hinzufügung, Löschung einer Datei von einer Variante wird bei einer Synchronisierung direkt die Operation auf die passende Datei ausgeführt, wenn es keine Konflikte gibt.

Die prototypische Implementierung *VariantSync* wird als ein Plug-in für Eclipse entwickelt. Wir verwenden die von Eclipse gebotene Funktionalität um die Quelltexte von allen Varianten zu überwachen. Wenn die Quelltexte von Entwickler geändert werden, vergleichen wir die Revision für den Quelltext, und speichern wir ein Datei für die Änderung. Um alle Änderungen von allen Varianten für Entwicklern klar darzustellen, haben wir eine View mit Baumdarstellung und Tabelledarstellung implementiert. Mit den View kann Entwickler wissen, in welcher Variante, für welchen Quelltext, wann und wie eine Änderung gemacht wurde.

Um Features zu berücksichtigen, haben wir die Funktionalität für Feature-Modell von FeatureIDE verwendet. Mit FeatureIDE erstellen wir ein Feature-Modell-Projekt für alle Varianten. In dem Projekt erstellen wir ein Feature-Modell und für jede Variante eine Konfigurationsdatei. Von FeatureIDE können wir profitieren. Wir können schnell eine Konfigurationsdatei für eine Variante validieren. Bei der Synchronisierung einer Änderung benutzen wir die Konfigurationsdatei um die Features von der geänderten Variante für Entwickler zu zeigen. Und kontrollieren wir mit der Konfigurationsdatei von der synchronisierten Variante, ob die Variante die bestimmte Features enthalten.

Zum Schluss der Arbeit haben wir eine Evaluierung gemacht. Für die Evaluierung wählen wir ein Projekt aus, das mit AHEAD entwickelt wurde. Mit dem Projekt kann man viele Varianten erzeugen. Wir erstellen zufällig zehn Konfigurationsdatei und erzeugen für 46 Revisionen jeweils zehn Varianten. Für jede Revision versuchen wir ein paar Varianten per Hand zu ändern und anderen Varianten durch *VariantSync* synchronisieren. Aus der Evaluierung haben wir gefunden, dass ungefähr 55.56% der Unterschiede von Varianten durch *VariantSync* automatisch synchronisiert werden, und ungefähr 70.05% der Operationen von Quelltexte durch *VariantSync* automatisch geführt werden.

8. Zukünftige Arbeiten

In die Evaluierung haben wir gefunden, dass ungefähr 55.56% der Unterschiede von Varianten durch *VariantSync* automatisch synchronisiert werden, und ungefähr 70.05% der Operationen von Quelltexte durch *VariantSync* automatisch geführt werden. In der Zukunft können wir die zwei Zahlen erhöhen, *VariantSync* effizienter machen.

Die Änderungen können durch *VariantSync* nach anderen Varianten synchronisiert werden, wenn es keine Konflikte gibt. Die Änderungen, die mit anderen Varianten Konflikte haben, müssen wir derzeit nur per Hand auf andere Varianten übernehmen. Wir können manche Konflikte schnell mit der Entscheidung von Entwicklern löschen. Zum Beispiel in den zwei Varianten werden in gleiche Position verschiedene Zeile in Quelltext hinzugefügt. Es ist möglich, dass die beide hinzugefügte Zeile in dem Quelltext bleiben sollen. Die Reihenfolge muss von Entwickler entschieden werden. Wenn die Reihenfolge festgelegt ist, kann man auch mit *VariantSync* die Änderung synchronisieren. Die Konflikt wird in dem dritten Kapitel als Reihenfolgekonflikt benannt. Außerdem gibt es noch Abhängigkeitskonflikt. Solche Konflikte werden auch nicht von *VariantSync* synchronisiert. Falls Entwickler entscheiden, dass es kein Konflikt ist, kann man auch mit *VariantSync* die Änderung synchronisieren. Für die zwei Arten von Änderungen können wir für *VariantSync* ein Benutzeroberfläche erstellen, um Entwicklern zu fragen. Dann kann man ein paar Änderungen halbautomatisch synchronisieren.

Die Größe der Änderung beeinflusst auch die Effizienz von *VariantSync*. Manche Änderungen sind zu groß, und beeinflussen mehr als ein Feature. Die Änderungen können nur mit *VariantSync* nach die Varianten synchronisiert werden, die alle von der Änderung beeinflusste Features enthalten. Falls die von der Änderung beeinflusste Features zwischen einander keine Beziehung haben, können wir die Änderungen

in ein paar kleine Änderungen zerlegen. Damit können wir die Änderungen nach mehr Varianten automatisch synchronisieren. Zum Beispiel haben wir drei Variante: V1, V2 und V3. V1 hat eine Änderung. Die Änderung beeinflusst Features *Fa, Fb*. V2 enthält kein *Fa*. V3 enthält kein *Fb*. Die Änderung von V1 kann nicht nach V2 und V3 synchronisiert werden. Wenn wir die Änderung in zwei Änderungen zerlegt haben, können wir die kleine Änderung synchronisieren.

In der Zukunft können wir *VariantSync* effizienter machen. Außerdem können wir noch ein paar Studie für *VariantSync* weiter machen. Zum Beispiel falls wir in der Praxis mit *VariantSync* ein paar Varianten lange Zeit entwickeln, können wir ausreichende Änderungen für ein Feature speichern. Wenn wir für ein paar Feature ausreichende Änderungen gespeichert haben, können wir ein mal probieren, ob wir mit den gespeicherte Änderungen neue Variante erstellen können.

Außerdem bei der lange Zeit Entwicklung mit *VariantSync* können wir meisten Zeilen von Quelltexten bestimmte Feature zuordnen. Weil wir bei jeder Synchronisierung Features für die Änderung ausgewählt haben. Dadurch können wir entscheiden, welche Feature die geändert Zeile gehört. Wenn wir die meisten Zeilen bestimmte Feature zuordnen können, und später zu viel Varianten haben, können wir ein Verfahren finden. Damit können wir versuchen, Features zu extrahieren und eine SPL zu erstellen.

Für den Prototyp *VariantSync* haben wir eine Evaluierung gemacht. Es simuliert eine Entwicklung eines Projektes mit vielen Varianten. Dafür haben wir Projekt *DekstopSerarcher* ausgewählt. Mit dem Projekt können wir viele Varianten für Revisionen bekommen. Aber hat die mit dem Projekt gemachte Evaluierung eine Abweichung von der tatsächlicher Nutzung. Die Änderung für eine Variante ist ein bisschen groß und manchmal beeinflusst mehr als ein Feature. Das hat eine negative Wirkung für die Effizienz von *VariantSync*. Die Änderung für unsere Evaluierung wurde für Revisionen erstellt. In einer tatsächlicher Nutzung sind die Änderungen von Varianten kleiner als in unserer Evaluierung. *VariantSync* speichert bei der Entwicklung eine Änderung des Quelltextes für jede Speicheroperation der Entwickler. Deshalb in der Zukunft sind ein paar Evaluierungen in den tatsächlichen Nutzungen erwartet. Außerdem wird das von uns ausgewählte Projekt *DekstopSerarcher* mit AHEAD entwickelt. Es ist Feature-orientieren-programmiert. Die struktur der Quelltexte von Varianten ist nicht so üblich. In der Zukunft sind die Evaluierungen auch erwartet, die mit anderen Projekte gemacht, die mit anderen Ansätze sowie Präprozessor, AOP, DOP oder Versionsverwaltungssysteme für viele Varianten entwickelt werden.

A. Details der Synchronisierungen

Die Tabelle A.1 zeigt die per Hand geänderte Varianten, Dateien und für die Änderungen aus gewählte Features für die Evaluierung im Kapitel 5. Die zweite Spalte zeigt die per Hand geänderte Varianten. Die Reihenfolge ist von oben nach unten. Die dritte Spalte zeigt die Nummer der synchronisierte Varianten. Für die zweite und dritte Spalte bedeutet Symbol „## “, dass es keine Änderung oder Synchronisierung gibt. Die Spalte *Dateien* zeigt die Namen von den geänderte Dateien. Die Namenserweiterung von den Dateien werden nicht gezeigt. Sie sind alles „.java “. Die Dateien werden meistens geändert. Für die Hinzufügungen und Löschungen werden mit Symbol „+ “ und „− “ gezeigt.

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R4-R5	V7	2,4,6,8,9,10	HitcomponentLabel	GUI
		2,4,6,8,9,10	HitDocument(+)	GUI
		1,6,9,10	HTML	HTML
		1,2,3,4,5,6,8,9,10	Indexer(+)	Base
		2,4,6,8,9,10	Listener(+)	GUI
		2,4,6,8,9,10	MainFrame(+)	GUI
		2,4,6,8,9,10	NoDocument(+)	GUI
		2,4,6,8,9,10	OptionWindow(+)	GUI
R5-R6	V2	4,6,7,8,9,10	MainFrame	GUI
		4,6,7,8,9,10	MrPinkMain	Base,GUI User_Interface
		4,6,7,8,9,10	OptionStorage	Base,GUI

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R5-R6	V2	4,6,7,8,9,10	OptionWindow	Base,GUI
		5,7,8,9,10	Plain	TXT
		1,3,4,5,6,7,8,9,10	UI(+)	User_Interface
	V1	3,5	MrPinkMain	Base,User_Interface
		3,5	OptionStorage(+)	Base
		3,5	OptionWindow(+)	Base
R6-R7	V6	2,4,7,8,9,10	ButtonListener(+)	GUI
		2,4,7,8,9,10	Listener(-)	GUI
		2,4,7,8,9,10	MainFrame	GUI
R7-R8	V1	2,3,4,5,6,7,8,9,10	ContentHandler	Base
		6,7,9,10	HTML	HTML
		6,7,9,10	Indexer	Base,HTML
		2,3,4,5,6,7,8,9,10	Parser(+)	ContentHandler
	V2	5,8	Indexer	Base,TXT
		5,7,8,9,10	Plain	TXT
	V3	4	Indexer	Base
	V10	7,9	Indexer	Base,HTML,TXT
R8-R9	V4	2,6,7	HitComponentLabel	Normal_View,GUI
		2,6,7	HitDocument	Normal_View,GUI
		2,6,7	MainFrame	Normal_View,GUI
		2,6,7,8,9,10	MrPinkMain	GUI
		2,6,7	NoDocument	Normal_View,GUI
	V8	9,10	HitcomponentLabel(-)	GUI
		9,10	HitDocument(-)	GUI
		9,10	MainFrame	GUI
		9,10	NoDocument(-)	GUI
R9-R10	V1	2,3,4,5,6,7,8,9,10	Indexer	Base
		2,3,4,5,6,7,8,9,10	MrPinkMain	Base
	V10	2,4,6,7,8,9	ButtonListener	GUI
		7	Indexer	Base,HTML Multi_Directory
		8,9	MainFrame	GUI
		2,7	MrPinkMain	Base,User_Interface Multi_Directory
	V4	##	Indexer	Base,Single_Directory

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R9-R10	V4	2,6,7	MainFrame	GUI
		6,8,9	MrPinkMain	Base,User_Interface Single_Directory
	V8	##	Indexer	Base,TXT Single_Directory
	V9	##	Indexer	Base,HTML Single_Directory
	V2	##	Indexer	Base,TXT Multi_Directory
	V6	##	Indexer	Base,HTML Single_Directory
R10-R11	V2	7,10	Indexer	Multi_Directory
R11-R12	V1	2,3,4,5,6,7,8,9,10	Indexer	Base
	V3	##	Indexer	Base
	V10	7,9	Indexer	Base
	V2	6,8	Indexer	Base
R12-R13	V4	2,6,7,8,9,10	ButtonListener	GUI
		1,2,3,5,6,7,8,9,10	Indexer	Base
		2,6,7,8,9,10	MainFrame	GUI
	V10	7,9	Indexer	HTML,TXT
	V6	8,9	ButtonListener	GUI,GUI_Preferences
		2,8	Indexer	Base
		##	MainFrame	GUI,GUI_Preferences
	V3	##	Indexer	Base
	V8	9	MainFrame	GUI,GUI_Preferences
R13-R14	V7	##	ButtonListener	Index_History
		##	MainFrame	Index_History
	V8	9	ButtonListener	Index_History
		9	MainFrame	Index_History
R14-R15	V6	##	ButtonListener	GUI,GUI_Preferences Single_Directory
		##	MainFrame	GUI,GUI_Preferences Single_Directory
		4,8,9	MrPinkMain	GUI,Single_Directory User_Interface

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R14-R15	V10	2	ButtonListener	GUI,Multi_Directory
		##	MainFrame	GUI,Multi_Directory
		2,7	MrPinkMain	GUI,Multi_Directory User_Interface
	V4	##	ButtonListener	GUI,Single_Directory
		##	MainFrame	GUI,Normal_View Single_Directory
	V2	##	MainFrame	GUI,Normal_View Multi_Directory
	V7	##	ButtonListener	GUI,Index_History Multi_Directory
		##	MainFrame	GUI,Index_History Multi_Directory
	V8	9	ButtonListener	GUI,Index_History Single_Directory
		9	MainFrame	GUI,Index_History Single_Directory
R15-R16	V3	1,2,4,5,6,7,8,9,10	Indexer	Base
		1,2,4,5,6,7,8,9,10	OptionWindow	Base
		1,2,4,5,6,7,8,9,10	Parser(-)	ContentHandler
	V7	2	MainFrame	GUI,Normal_View
	V4	6	MainFrame	GUI,Normal_View
	V8	9,10	MainFrame	GUI
R16-R17	V8	9,10	HitcomponentLabel(+)	Tree_View
		9,10	HitDocument(+)	Tree_View
		9	MainFrame	Tree_View GUI_Preferences
	V10	##	MainFrame	Tree_View Multi_Directory
R17-R18	V10	8,9	HitDocument	Tree_View
		##	MainFrame	Tree_View
		1,2,3,4,5,6,7,8,9	OS(+)	OS
		8,9	SearchResultTree(+)	Tree_View
	V9	3,5	OS	Linux
		8	MainFrame	Tree_View

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R18-R19	V1	2,4,6,7,8,10	OS	Windows
R19-R20	V8	9,10	MainFrame	Tree_View
		9,10	SearchResultTree	Tree_View
R20-R21	V3	##	Indexer	Base,Latex
		4,5,6,7,9	Latex(+)	Latex
	V6	##	Indexer	HTML,Latex Single_Directory
	V5	##	Indexer	Base,TXT,Latex
	V7	9	Indexer	HTML,TXT,Latex
	V4	##	Indexer	Base,Latex Single_Directory
R21-R22	V1	3,5	Commandline(+)	Commandline
		3	Indexer	Base,Commandline
		3,5	MrPinkMain	Base,Commandline User_Interface
	V5	##	Indexer	Base,TXT Commandline
R22-R23	V7	2	MainFrame	GUI
	V6	4,8,9,10	MainFrame	GUI
	V10	##	MainFrame	GUI
R23-R24	V8	9,10	HitDocument	Tree_View
		9	MainFrame	Tree_View
		9,10	SearchResultTree	Tree_View
	V10	##	MainFrame	Tree_View
R24-R25	V8	9,10	SearchResultTree	Tree_View
R25-R26	V9	8	ButtonListener	GUI,Index_History Single_Directory
		7,8	History_Indexer(+)	Index_History
		1,2,3,4,5,6,7,8,10	Indexer	Base
		7,8	Index_History_Selector(+)	Index_History
		2,4,6,7,8,10	MainFrame	GUI
	V2	7,10	ButtonListener	GUI,Multi_Directory
	V4	6	ButtonListener	GUI,Single_Directory
	V7	##	ButtonListener	Index_History
R26-R27	V7	8,9	Index_History_Selector	Index_History

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R27-R28	V8	7,9	History_Indexer	Index_History
		7,9	Index_History_Selector	Index_History
R28-R29	V9	7,8	Index_History_Selector	Index_History
R29-R30	V7	8,9	Index_History_Selector	Index_History
R30-R31	V9	7,8	Index_History_Selector	Index_History
R31-R32	V2	4,6,7	MainFrame	Normal_View,GUI
	V8	9,10	MainFrame	GUI,Tree_View
R32-R33	##			
R33-R34	V3	4,5,6,7,9	Latex	Latex
R34-R43	V1	2,4,6,7,8,10	OS	Windows
	V3	5,9	OS	Linux
R43-R45	V10	1,2,4,6,7,8	OS	Windows
	V3	5,9	OS	Linux
	V4	7	History_Query(+)	Query_History
		##	MainFrame	Single_Directory Query_History
	V7	##	MainFrame	Index_History Query_History Multi_Directory
R45-R48	V8	9,10	SearchResultTree	Tree_View
R48-R54	V6	##	Indexer	HTML Single_Directory
	V10	##	Indexer	HTML Multi_Directory
	V7	##	Indexer	HTML,TXT Multi_Directory
	V9	##	Indexer	HTML,TXT Single_Directory
R54-R55	V3	4,5,6,7,9	Latex	Latex
R55-R57	V4	7	History_Query	Query_History
	V6	8,9	ButtonListener	GUI_Preferences
R57-R61	V3	##	Indexer	Base,Latex Commandline
		4,5,6,7,9	Latex(-)	Latex
	V5	##	Indexer	Base,TXT,Latex

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R57-R61	V5			Commandline
		2,7,8,9,10	Plain(-)	TXT
	V1	##	Indexer	Base,HTML Commandline
		6,7,9,10	HTML(-)	HTML
	V2	##	Indexer	Base,TXT Multi_Directory
	V6	##	Indexer	Base,HTML,Latex Single_Directory
		##	MainFrame	Normal_View Single_Directory GUI_preferences
	V9	##	Indexer	Base,HTML TXT,Latex Single_Directory
		8	MainFrame	Tree_View Index_History GUI_preferences
	V10	##	Indexer	Base,HTML,TXT Multi_Directory
	V4	##	Indexer	Base,Latex Single_Directory
	V7	##	Indexer	Base,HTML TXT,Latex Multi_Directory
	V8	##	Indexer	Base,TXT Single_Directory
R61-R62	V2	7,10	Indexer	Multi_Directory,TXT
		5,7,8,9,10	Plain(+)	TXT
	V5	##	Indexer	Commandline,TXT
	V8	9	Indexer	Single_Directory,TXT
R62-R63	V3	##	Indexer	Commandline,Latex
		4,5,6,7,9	Latex(+)	Latex
	V1	##	Indexer	Commandline,HTML
		6,7,9,10	HTML(+)	HTML

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R62-R63	V4	6	Indexer	Single_Directory,Latex
	V7	##	Indexer	HTML,TXT,Latex Multi_Directory,
	V5	##	Indexer	Comandline TXT,Latex
	V10	##	Indexer	HTML,TXT Multi_Directory
	V6	##	Indexer	HTML,Latex Single_Directory
	V9	##	Indexer	HTML,TXT,Latex Single_Directory
R63-R64	##	##		
R64-R81	V3	4,5,6,7,9	Latex	Latex
R81-R87	V1	3,5	Commandline	Commandline
		2,3,4,5,6,7,8,9,10	ContentHandler	Base
		6,7,9,10	HTML	HTML
		3	Indexer	Base
		2,3,4,5,6,7,8,9,10	IndexerException	Base
		3,5	MrPinkMain	Base
		2,3,4,5,6,7,8,9,10	OptionStorage	Base
		2,3,4,5,6,7,8,9,10	OptionWindow	Base
		2,4,6,7,8,10	OS	Windows
		2,3,4,5,6,7,8,9,10	UI	User_Interface
	V6	##	ButtonListener	GUI,Gui_Preferences Single_Directory
		2,4,7	HitcomponentLabel	Normal_View
		2,4,7	HitDocument	Normal_View
		##	Indexer	Base,HTML,Latex Single_Directory
		3,4,5,7,9	Latex	Latex
		##	MainFrame	GUI,Gui_Preferences Normal_View
		2,4,7,8,9,10	MrPinkMain	Base,User_Interface
		2,4,7	NoDocument	Normal_View
	V9	8	ButtonListener	GUI,Gui_preferences

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R81-R87	V9			Index_History Single_Directory
		7,8	History_Indexer	Index_History
		8,10	HitcomponentLabel	Tree_View
		8,10	HitDocument	Tree_View
		##	Indexer	Base,HTML TXT,Latex Single_Directory
		7,8	Index_History_Selector	Index_History
		8	MainFrame	GUI,Gui_Preferences Index_History
		3,5	OS	Linux
		2,5,7,8,10	Plain	TXT
		8,10	SearchResultTree	Tree_View
	V7	##	ButtonListener	GUI,Index_History Multi_Directory
		4	History_Query	Query_History
		##	Indexer	Base,HTML TXT,Latex Multi_Directory
		##	MainFrame	GUI,Normal_View Query_History Index_History
	V10	2,4	ButtonListener	GUI
		##	Indexer	Base,HTML,TXT Multi_Directory
		##	MainFrame	GUI
	V5	##	Indexer	Base,TXT,Latex Commandline
	V2	##	Indexer	Base,TXT Multi_Directory
		##	MainFrame	GUI,Normal_View
	V4	##	ButtonListener	Single_Directory
		8	Indexer	Base,Single_Directory
		##	MainFrame	GUI,Normal_View

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R87-R89	V7	2,10	ButtonListener	GUI,Multi_Directory
		1,2,3,4,5,6,8,9,10	ContentHandler	ContentHandler
		8,9	History_Indexer	Index_History
		4	History_Query	Query_History
		2,4,6	HitcomponentLabel	Normal_View
		2,4,6	HitDocument	Normal_View
		##	Indexer	Base,HTML TXT,Latex Multi_Directory
		1,2,3,4,5,6,8,9,10	IndexerException	Base
		8,9	Index_History_Selector	Index_History
		##	MainFrame	GUI,Query_History Normal_View Multi_Directory
		2,4,6,8,9,10	MrPinkMain	Base,User_Interface
		1,2,3,4,5,6,8,9,10	OptionStorage	Base
		1,2,3,4,5,6,8,9,10	OptionWindow	Base
	V1	3,5	Commandline	Commandline
		##	Indexer	Base,HTML Commandline
		3,5	MrPinkMain	Base,Commandline User_Interface
	V4	6,8,9	ButtonListener	GUI
		6,9	Idexer	Base,Latex Single_Directory
		##	MainFrame	GUI,Normal_View Query_History
	V10	##	Indexer	Base,,HTML,TXT Multi_Directory
		##	MainFrame	GUI,Multi_Directory
	V2	##	Indexer	Base,TXT Multi_Directory
		##	MainFrame	GUI,Normal_View Multi_Directory
	V5	##	Indexer	Base,TXT,Latex

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite

Rev.	V_H	V_S	Dateien	$Features_{Auswahl}$
R87-R89	V5			Commandline
	V6	8,9	ButtonListener	Gui_Preferences
		9	Indexer	HTML
		##	MainFrame	GUI,Normal_View
	V9	8	Indexer	TXT
		8	MainFrame	GUI
	V3	##	Indexer	Base,Latex Commandline
	V8	##	Indexer	Base,Single_Directory
R89-R90	V1	6,7,9,10	HTML	HTML
	V10	8,9	HitcomponentLabel	Tree_View
		8,9	HitDocument	Tree_View
		8,9	MainFrame	Tree_View
		2,5,7,8,9	Plain	TXT
		8,9	SearchResultTree	Tree_View
	V6	4,8,9	ButtonListener	Single_Directory
		4,8,9	Indexer	Single_Directory
		3,4,5,7,9	Latex	Latex
		4,8,9	MainFrame	Single_Directory
R90-R91	V3	1,2,4,5,6,7,8,9,10	MrPinkMain	User_Interface

Tabelle A.1: Details der Synchronisierungen in der Evaluierung

Literaturverzeichnis

[ABGM02] David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28:324–333, 2002. (zitiert auf Seite 78)

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Refereed Column. (zitiert auf Seite 17)

[ALB+11] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *SIGSOFT FSE*, 2011. (zitiert auf Seite 18 und 77)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Software product lines: 9th international conference, SPLC 2005, Rennes, France, September 26-29, 2005 : proceedings*, pages 7–20. Springer, 2005. (zitiert auf Seite 7 und 9)

[Bat06] Don S. Batory. A tutorial on feature oriented programming and the ahead tool suite. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2006. (zitiert auf Seite 23)

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (zitiert auf Seite 6)

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid, editors. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt, Heidelberg, 2004. (zitiert auf Seite 2)

[BLHM02] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *ASE '02: Proceedings of Automated Software Engineering Conference*, pages 81–92, 2002. (zitiert auf Seite 9)

[BSR03] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 187–197. IEEE Computer Society, 2003. (zitiert auf Seite 23)

[Buf95] Jim Buffenbarger. Syntactic software merging. In *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, pages 153–172, London, UK, UK, 1995. Springer-Verlag. (zitiert auf Seite 77)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000. (zitiert auf Seite 9)

[CN02] P. Clements and L. Northrop. *Software product lines: practices and patterns*. The SEI series in software engineering. Addison-Wesley, 2002. (zitiert auf Seite 5)

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, June 1998. (zitiert auf Seite 18)

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. (zitiert auf Seite 1)

[FECA04] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Ak?it. *Aspect-oriented software development*. Addison-Wesley Professional, first edition, 2004. (zitiert auf Seite 17)

[HEB+10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Toolchain-independent variant management with the leviathan filesystem. In Sven Apel, Don S. Batory, Krzysztof Czarnecki, Florian Heidenreich, Christian Kästner, and Oscar Nierstrasz, editors, *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*, pages 18–24. ACM, 2010. (zitiert auf Seite 78)

[Hof08] D.W. Hoffmann. *Software-Qualität*. Springer, 2008. (zitiert auf Seite 10)

[JKB08] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. *Technical Report Lero-TR-SPL-2008-02*, 2008. (zitiert auf Seite vii, 2 und 3)

[Kah01] B. Kahlbrandt. *Software-Engineering Mit Der Unified Modeling Language*. Springer, 2001. (zitiert auf Seite 1)

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, 2010. (zitiert auf Seite vii, 2, 5, 6 und 22)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (zitiert auf Seite 7)

[KDO11] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability mining with LEADT. Technical Report 01/2011, Department of Mathematics and Computer Science, Philipps University Marburg, September 2011. (zitiert auf Seite 36 und 79)

[Lim94] W.C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994. (zitiert auf Seite 1)

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel, 2006. (zitiert auf Seite 22)

[Man02] Mike Mannion. Using first-order logic for product line model validation. In Gary Chastek, editor, *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 149–202. Springer Berlin / Heidelberg, 2002. (zitiert auf Seite 9)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering - foundations, principles, and techniques.* Springer, 2005. (zitiert auf Seite 6)

[PO97] T. Troy Pearse and Paul W. Oman. Experiences developing and maintaining software in a multi-platform environment. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 270–, Washington, DC, USA, 1997. IEEE Computer Society. (zitiert auf Seite 22)

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP 97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997. (zitiert auf Seite 23)

[RC12] Julia Rubin and Marsha Chechik. Combining related products into product lines. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2012. (zitiert auf Seite 79)

[Ref09] J. G. Refstrup. Adapting to change: Architecture, processes and tools: A closer look at hp's experience in evolving the owen software product line. 2009. (zitiert auf Seite 22)

[Ros09] S. Rosensteiner. *Cost Estimation in Software Product Line Engineering*. Diplomica Verlag Gmbh, 2009. (zitiert auf Seite 2)

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proc. of Software Product Line Conference (SPLC 2010)*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010. (zitiert auf Seite 17)

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with c news, 1992. (zitiert auf Seite 22)

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *VaMoS'10*, pages 45–51, 2010. (zitiert auf Seite 22)

[TKB+12] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012. To appear; accepted 2012-06-07. (zitiert auf Seite 39)

[TSSpL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-preikschat, and Daniel Lohmann. Dead or alive: finding zombie features in the linux kernel. In *In Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09*, pages 81–86, 2009. (zitiert auf Seite 22)

[Was06] T. Wassermann. *Wassermann, Versionsmanagement mit Subversion; PR*. mitp-Verlag, 2006. (zitiert auf Seite 11 und 14)

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, SCM '91, pages 68–79, New York, NY, USA, 1991. ACM. (zitiert auf Seite 77)

[WOZ91] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. Reusable software components. *Advances in Computers*, 33:1–65, 1991. (zitiert auf Seite 1)

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

Magdeburg, den