Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Feature-orientiertes Refactoring zur Migration von Produktvarianten

Autor:

Steffen Schulze

08. Dezember 2015

Betreuer:

Prof. Gunter Saake
Wolfram Fenske
Jens Meinicke

Universität Magdeburg - Datenbanken und Software Engineering

Dr.-Ing. Sandro Schulze

TU Hamburg - Software Technology Systems

Schulze, Steffen:
Feature-orientiertes Refactoring zur Migration von Produktvarianten
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2015.

Danksagung

Hiermit möchte ich mich bei all jenen ganz herzlich bedanken, die mich im Rahmen dieser Arbeit unterstützt haben. Zunächst gilt mein Dank Prof. Gunter Saake, welcher mir die Möglichkeit gab, innerhalb seiner Arbeitsgruppe meine Masterarbeit zu schreiben. Ganz besonders möchte ich Wolfram Fenske und Jens Meinicke für die ausgezeichnete Betreuung danken. Ihre wertvollen Hinweise zur Strukturierung und konstruktiven Kritiken zu den einzelnen Kapiteln haben wesentlich zum Fortschritt dieser Arbeit beigetragen. Weiterhin danke ich Sandro Schulze, der sich als weiterer Betreuer zur Verfügung stellte und mich ebenfalls mit Anmerkungen unterstützte. Abschließend geht mein Dank an meine Familie und Freunde, die mich während der Masterarbeit moralisch unterstützt haben.

Inhaltsangabe

Bei der Entwicklung von Software wird heutzutage großer Wert auf die Wiederverwendung von Funktionalität gelegt. Ohne ein systematisches Konzept für Wiederverwendung steigt der Entwicklungs- und Wartungsaufwand mit steigender Anzahl von Produktvarianten erheblich. Um einen hohen Grad an Wiederverwendung bei einer Vielzahl von Produktvarianten zu erreichen, werden neue Konzepte wie Software-Produktlinien benötigt. In vielen Fällen wurden die Produktvarianten ohne systematische Wiederverwendung unabhängig voneinander entwickelt. Vorhandene Produktvarianten können mithilfe einer Migration in eine Software-Produktlinie überführt werden. Ein Problem, das sich dabei ergibt, ist, dass der Prozess der Migration oftmals komplex und aufwendig ist, da beliebige Unterschiede zwischen den Produkten angeglichen werden müssen. Eine automatische Überführung der Produktvarianten in eine Software-Produktlinie ist dadurch nicht möglich. Um den Migrationsaufwand zu verringern, sollten identische Funktionalitäten vereinheitlicht und angeglichen werden, mit dem Ziel diese Funktionalitäten in den verschiedenen Produktvarianten wiederzuverwenden. In dieser Arbeit stelle ich ein Werkzeug zur Unterstützung der Migration vor, um aufbauend auf den Ergebnissen vorhandener Codeklonanalysen und -erkennungen mithilfe von Refactorings eine Angleichung von ähnlicher Funktionalität zu ermöglichen. Anhand einer Migration auf einer Menge von Produkten zeige ich den Nutzen dieser Refactorings.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1. Einführung

In der heutigen Softwareentwicklung wird vielfach aus vorhandenem Quellcode eines Softwareprodukts ein ähnliches Produkt entwickelt. Diese Vorgehensweise wird als *Clone-and-Own* bezeichnet [Clements und Northrop, 2001]. Mithilfe dieses Ansatzes können Softwareprodukte auf unterschiedliche Anforderungen einzelner Kunden oder Kundengruppen zugeschnitten werden. Neue Softwareprodukte werden durch Kopieren des gesamten Quellcodes erzeugt und anschließend so verändert, dass diese neuen Softwareprodukte den gewünschten Anforderungen entsprechen. Die initialen Entwicklungskosten sind im Gegensatz zur kompletten, eigenständigen Neuentwicklung eines Softwareprodukts niedriger, da auf einer vorhandenen Codebasis aufgebaut wird [Krueger, 1992]. Anpassungen und Erweiterungen sind nur an wenigen Codestellen notwendig. Durch diese geringen Entwicklungskosten ist es möglich, schnell auf geänderte Anforderungen zu reagieren, um so neue Softwareprodukte schneller am Markt bereitzustellen. Ein weiterer Vorteil dieses Ansatzes ist, dass dieses neu erzeugte Softwareprodukt komplett unabhängig von anderen Produkten weiterentwickelt werden kann [Dalgarno und Beuche, 2007]. Mit jedem neuen Produkt erhöht sich jedoch die Anzahl von ähnlichen Produktvarianten, die sich nur geringfügig voneinander unterscheiden. Dadurch steigt der Weiterentwicklungs- und Wartungsaufwand erheblich. Wenn zum Beispiel eine gemeinsame Funktionalität angepasst oder erweitert werden muss, weil ein Fehler zu beheben ist, muss diese Änderung in allen Produktvarianten mit gleicher Funktionalität vorgenommen werden [Geiger et al., 2006]. Somit sind die mittel- und langfristigen Kosten hoch, wofür es nur wenig und nur einfachen Tool-Support gibt. Dadurch besteht die Gefahr, dass die Fehler nicht in allen Produktvarianten behoben werden und somit diese Fehler in einzelnen Produktvarianten weiterhin existieren. Um den Entwicklungs- und Wartungsaufwand zu minimieren, werden neue Konzepte wie Software-Produktlinien (SPLs) [Czarnecki und Eisenecker, 2000; Clements und Northrop, 2001; Pohl et al., 2005; Apel et al., 2013a] benötigt.

Eine SPL ist eine Menge von Softwareprodukten, die sich eine gemeinsame Codebasis teilen [Clements und Northrop, 2001; Pohl et al., 2005; Apel et al., 2013a]. Statt Produktvarianten einzeln, getrennt voneinander zu verwalten, wird eine Plattform entwickelt, die alle wiederverwendbaren Artefakte (z.B. Quellcode) enthält.

Alle Produktvarianten der SPL werden auf Basis dieser wiederverwendbaren Artefakte entwickelt. Somit teilen sich Produktvarianten gemeinsame Eigenschaften. Dadurch müssen Änderungen oder Erweiterungen an mehrfach verwendeter Funktionalität nur an einer zentralen Stelle im Quellcode vorgenommen werden, anstatt in mehreren Produktvarianten separat.

Im Rahmen der Entwicklung von SPLs wird eine Eigenschaft oder ein Verhalten als Feature beschrieben [Pohl et al., 2005; Apel et al., 2013a]. Bei der Erzeugung einer Produktvariante können die gewünschten Features je nach Anforderung ausgewählt werden. Dadurch können generierte Produktvarianten genau auf Bedürfnisse der Kunden angepasst werden. Anhand unterschiedlicher Kombinationen von Features kann eine Vielzahl von Produktvarianten erstellt werden. Somit vereint eine SPL neben der systematischen Wiederverwendung auch eine große Variabilität innerhalb der Domäne.

Eine Möglichkeit, eine SPL umzusetzen, ist Feature-orientierte Programmierung (FOP) [Prehofer, 1997; Batory et al., 2003]. Der Grundgedanke von FOP ist Modularisierung des Quellcodes anhand von Features. Dabei werden Features in Feature-Module aufgeteilt. Diese Feature-Module enthalten alle dazugehörigen Software-Artefakte wie z.B. Klassen, Text- oder Bilderdateien. In den Feature-Modulen können verschiedene Codeelemente wie beispielsweise Klassen, Methoden oder Felder verfeinert werden. In einem Feature-Modell werden benötigte Features einer Produktvariante ausgewählt. Aufgrund der Modularisierung von Features eignet sich der FOP-Ansatz für eine Migration von Produktvarianten.

In meiner Arbeit will ich Clone-and-Own erzeugte Produktvarianten mithilfe einer Migration in eine SPL überführen. Die in der Literatur vorhandenen Ansätze zur Migration von Produktvarianten lassen sich in zwei Kategorien einteilen. Die erste Kategorie von Ansätzen beschreibt die Migration eines einzelnen Produkts in eine SPL. Vertreter für diese Kategorie sind [Liu et al., 2006; Trujillo et al., 2006; Olszak und Jørgensen, 2009; Lopez-Herrejon et al., 2011]. Für meine Arbeit sind diese Migrationsansätze nicht ausreichend, da sie nur die Migration einer Produktvariante in eine SPL unterstützen. Diesen Migrationsansätzen fehlt somit ein Konzept um ähnliche oder identische Funktionalitäten verschiedener Produktvarianten zu erkennen, herauszulösen und in gemeinsam genutzte Funktionalitäten zusammenzuführen. Diese migrierte SPL bildet dann das Grundgerüst für weitere, neu zu entwickelnde Produktvarianten. Somit gibt es in diesem Fall schon eine SPL und damit ein Domänenmodell, welches für die Migration/Integration genutzt werden kann. Mit den Ansätzen der zweiten Kategorie können eine Vielzahl von Produktvarianten migriert werden, z.B. [Alves et al., 2005, 2006; Xue, 2011]. Diese Ansätze sind für meine Arbeit ebenfalls nicht geeignet, weil sie zum einen nicht für FOP entwickelt sind und somit nur unter großem Aufwand auf FOP übertragen werden können. Zum anderen erlauben diese Verfahren keine schrittweise Migration, d.h. die Wartung und Weiterentwicklung der überführten Produktvarianten müssen für die Dauer der Migration gestoppt werden. Da die Migration langwierig sein kann und das Migrationsvorhaben auch scheitern kann, stellen diese Verfahren für ein Software-Unternehmen ein erhebliches Risiko dar. Die Ansätze beider Kategorien eignen sich somit für mein Vorhaben nicht.

Aufgrund der Entstehung der Produktvarianten mithilfe des *Clone-and-Own*-Ansatzes ist anzunehmen, dass ein Großteil der Funktionalität in mehreren Produktvarianten gleich ist. Die Produktvarianten weisen somit eine hohe Anzahl von ähnlichen oder identischen Codestellen auf. Diese Codestellen sind als Codeklone (CC) [Roy et al., 2009] erkennbar. Bei der Migration wird gleiche Funktionalität aus den verschiedenen Produktvarianten in gemeinsam genutzte Artefakte einer SPL extrahiert. Dazu ist es notwendig, identische Funktionalitäten zwischen den Produktvarianten zu vereinheitlichen. Mit dieser Angleichung der Unterschiede der Produktvarianten wird die Grundlage für eine Wiederverwendung auf einer gemeinsamen Codebasis geschaffen, um somit den Grad an Wiederverwendung zu erhöhen. Aus diesem Grund werden vorbereitende Aktionen auf der zu migrierenden SPL durchgeführt. In einem ersten Schritt sollen gleiche Funktionalitäten mithilfe einer Codeklonerkennung erkannt und diese Funktionalitäten mittels Refactoring in gemeinsam genutzte Features extrahiert werden. Refactorings zum Bereinigen von CCs sind etabliert und weit verbreitet [Fowler, 1999; Dudziak und Wloka, 2002; Van Emden und Moonen, 2002; Tourwé und Mens, 2003]. Das Refactoring zeichnet sich dadurch aus, dass das Verhalten (Funktionalität) erhalten bleibt, während die interne Programmstruktur überarbeitet wird. Opdyke [1992] und Fowler [1999] haben Regeln für die Durchführung von Refactorings definiert. Die Extraktion über Refactoring ist nur möglich, wenn identische Funktionalität auch identisch implementiert ist. Dies ist aber nicht immer gegeben, weil es beliebige Unterschiede zwischen den Produktvarianten gibt. Diese Unterschiede ergeben sich zum einen durch den Clone-And-Own-Ansatz selbst und zum anderen durch die separate Weiterentwicklung verschiedener Produktvarianten. Daher müssen vor der Extraktion die variantenspezifischen Unterschiede angeglichen werden.

Das Zusammenspiel von FOP und Refactorings ist bislang wenig erforscht [Schulze et al., 2012, 2013]. Da objektorientierte Refactorings die Variabilität einer SPL nicht berücksichtigen und FOP-spezifische Sprachkonstrukte (insbesondere Refinement) nicht unterstützen, muss im Rahmen von FOP-Refactorings die Variabilität einer SPL berücksichtigt werden. Des Weiteren existiert keine Tool-Unterstützung für Refactorings in FOP.

Um die Attraktivität von SPLs für ein breiteres Spektrum sowohl für Unternehmen, Anwendungen als auch für Entwickler zu erhöhen und damit den Einsatz voranzutreiben, ist es erforderlich, dass Tools zur Migration von vorhandenen Produkten in eine SPL zur Verfügung stehen. Dabei spielt ein zuverlässiges Refactoring eine wichtige Rolle.

Zielstellung der Arbeit

Mit der vorliegenden Arbeit wird das Ziel verfolgt, die Migration von vorhandenen Produktvarianten in eine SPL zu erleichtern. Dafür ist es notwendig, identische Funktionalität zwischen den verschiedenen Produktvarianten aneinander anzugleichen. Da aktuell die Angleichung manuell an den unterschiedlichen Codestellen der Produktvarianten durchgeführt werden muss, stellt diese Arbeit einen teilautomatischen Ansatz bereit, um die benötigte Voraussetzung für eine Migration zu schaffen. Aufbauend auf FeatureIDE [Thüm et al., 2014], einer integrierten Entwicklungsumgebung zur Entwicklung von SPLs, wird ein prototypischer und werkzeuggestützter Ansatz für Refactorings entwickelt. Die Umsetzung dieses Ansatzes

unterstützt dabei FeatureHouse [Apel et al., 2013b], einen sprachunabhängigen Ansatz für Feature-orientierte Programmierung. Das Refactoring berücksichtigt FOP-spezifische Sprachkonstrukte und damit die Variabilität innerhalb einer SPL. Anhand vorhandener Codeklonanalysen [Tonscheidt, 2015] werden ausgewählte Refactorings auf den zu migrierenden Produktvarianten angewandt. Im ersten Schritt wird die Angleichung des Quellcodes der verschiedenen Produktvarianten durchgeführt, um in dem darauffolgenden zweiten Schritt die Codeklone zu beseitigen. Mittels einer Fallstudie wird der Prozess der Migration anhand von ApoGames, einer Menge von *Clone-and-Own* erstellten Produktvarianten, gezeigt.

Gliederung der Arbeit

Im Kapitel 2 werden die Grundlagen über SPLs mit dem Fokus auf die Feature-orientierten Programmierung beschrieben. Aufbauend auf Eigenschaften von variantenerhaltenden Refactoring und der Migration wird in Kapitel 3 ein Ansatz zur Migration von Produktvarianten unter Verwendung des Feature-orientierten Refactorings vorgestellt. Die beiden Schritte meines Migrationsansatzes, Angleichung und Extraktion, werden im Kapitel 4 ausführlich erläutert. Kapitel 5 veranschaulicht die Implementierungsdetails meines Migrationsansatzes. Die Evaluierung des vorgestellten Konzepts der Migration von Produktvarianten erfolgt in Kapitel 6. Anschließend gibt das Kapitel 7 einen Überblick zu verwandten Arbeiten aus den Themengebieten Migration von Produktvarianten und Refactorings mit Bezug zu SPLs. Zum Abschluss fasst Kapitel 8 die gewonnenen Ergebnisse zusammen und stellt Ausgangspunkte für zukünftige Arbeiten vor.

2. Grundlagen

In diesem Kapitel erläutere ich die Grundlagen über SPLs. Dabei ist wichtig zu verstehen, welche Ziele eine Softwareproduktlinie verfolgt, wie sie aufgebaut ist und wie die einzelnen Softwarekomponenten (Features, Feature-Modelle und Feature-orientierte Programmierung) miteinander agieren, damit eine Migration erfolgreich durchgeführt werden kann. Dazu beschreibe ich den Weg von den Grundlagen der Software-Produktlinen über Modellierung von Features hin zu der Generierung eines Produktes mit Unterstützung der Feature-orientierten Programmierung.

2.1 Software-Produktlinien

Die ersten Ideen einer Produktlinie reichen bis in die Anfänge des 18. Jahrhunderts zurück. Damals revolutionierte Eli Whitney die Herstellung von Gewehren, indem er austauschbare Komponenten verwendete. Jahrzehnte später griff Henry Ford die Idee der austauschbaren Komponenten auf und perfektionierte den Ansatz bei der Fließband-Herstellung von Model T-Automobilen. Durch den Einsatz einer Produktlinie konnten die Produkte für Konsumenten auf Basis eines wiederverwendeten gemeinsamen Kerns kostengünstiger und schneller hergestellt werden. Dieser Ansatz wird ebenfalls bei einer Softwareproduktlinie (*SPL*) verfolgt [Pohl et al., 2005].

Nachdem in den letzten zwei Jahrzehnten der Prozess von Standardsoftware zu kundenindividueller Software vollzogen wurde, gewann der Begriff Produktlinien auch für Software an Bedeutung [Pohl et al., 2005]. In der Literatur existieren für den Begriff SPL viele unterschiedliche Definitionen [Kang et al., 1990; Czarnecki und Eisenecker, 2000; Clements und Northrop, 2001; Pohl et al., 2005; Apel et al., 2008]. Clements und Northrop [2001] definieren SPL wie folgt:

> 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.

Wie aus der Definition hervorgeht, handelt es sich bei einer SPL um eine Menge von Software-Produkten mit gemeinsamen Eigenschaften, die auf ein bestimmtes Marktsegment (Domäne) zugeschnitten sind. Alle Software-Produkte werden auf einer gemeinsamen Basis entwickelt und teilen sich somit die Kernkomponenten. Damit vereint eine SPL zum einen die Variabilität innerhalb einer Domäne und zum anderen die Wiederverwendung auf Basis einer gemeinsamen Plattform [Pohl et al., 2005].

Eine SPL besteht aus einer Vielzahl von Features. Unter einem Feature wird eine Eigenschaft oder ein für den Nutzer sichtbares Verhalten eines Softwaresystems verstanden [Kang et al., 1990; Apel et al., 2013a]. Features werden benutzt, um die Variabilität der SPL zu beschreiben und die Kommunikation mit allen Beteiligten (Entwickler, Kunden) zu vereinfachen. Jede Produktvariante einer SPL wird aus Kombinationen einer Teilmenge aller vorhandenen Features erzeugt. Durch die Kombination der Features können Produktvarianten maßgeschneidert auf die Anforderungen der Nutzer erstellt werden.

Zu Beginn der SPL-Entwicklung muss der Aufwand für die Planung der Produktvarianten berücksichtigt werden. Dazu werden für die Features der verschiedenen Produktvarianten, ihre Gemeinsamkeiten und Unterschiede identifiziert [Pohl et al., 2005]. Dieser Mehraufwand erhöht zu Beginn die Entwicklungskosten gegenüber der klassischen Entwicklung von Software-Produkten. Mit zunehmender Anzahl an Produktvarianten verringern sich jedoch die Kosten durch Wiederverwendung von vorhandenen Features. Sobald der Break-Even-Point überschritten ist, sind die Kosten für eine SPL im Gegensatz zur klassischen Entwicklung niedriger. In der Literatur wurde festgestellt, dass der Break-Even-Point zwischen drei und vier Produkten liegt [Pohl et al., 2005]. In Abbildung 2.1 wird ein Vergleich der Projektkosten zwischen einer SPL und Einzelsystemen gezeigt

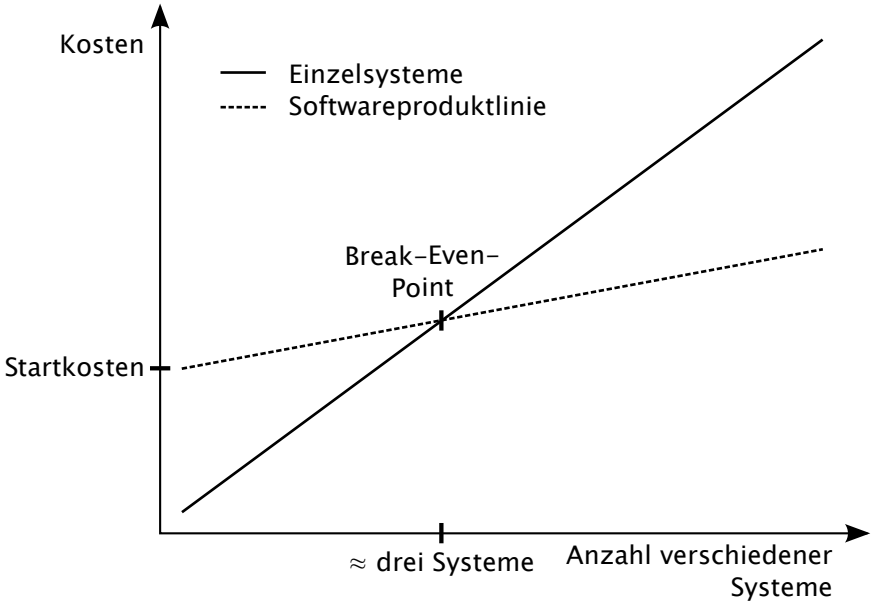


Abbildung 2.1: Vergleich der Kosten zwischen Produktvarianten einer SPL und Einzelsystemen [Pohl et al., 2005])

Durch die Wiederverwendung von Features werden nicht nur die Kosten, sondern auch die Zeit, die zur Markteinführung neuer Produktvarianten benötigen wird, gesenkt. Bei der SPL ist die Markteinführungszeit initial höher, weil die wiederverwendbaren Komponenten geplant und entwickelt werden müssen. Im Laufe der Zeit sinkt der Wartungs- und Weiterentwicklungsaufwand für SPL deutlich unter die Zeit einzelner Systeme. Der Grund dafür ist, dass bei SPLs der Implementierungsaufwand nur einmal anfällt anstatt n-Mal für jede Produktvariante. Somit können Fehler schneller behoben und neue Anforderungen verhältnismäßig zügiger umgesetzt werden.

2.2 Feature-Modellierung

Bei der Entwicklung von SPLs spielt die Modellierung der Variabilität eine entscheidende Rolle. Ein gängiger Ansatz, um Variabilität zu beschreiben, ist die Feature-Modellierung. Dabei werden die Features und ihre Beziehungen untereinander mithilfe von Feature-Modellen dargestellt [Czarnecki und Eisenecker, 2000; Apel et al., 2013a]. Der Vorteil von Feature-Modellen besteht darin, dass sie unabhängig von der Implementierung sind, da die Abhängigkeiten der Features abstrakt beschrieben werden [Czarnecki und Eisenecker, 2000].

Zur Erzeugung einer Produktvariante wird eine Konfiguration benötigt. Bei einer Konfiguration im Kontext von SPLs handelt es sich um eine Teilmenge von Features. Diese Teilmenge enthält alle ausgewählten Features, die für die gewünschten Bedürfnisse benötigt werden. Diese Konfigurationen können gültig oder ungültig sein. Gültige Konfigurationen müssen zu einem funktionierenden Produkt kompiliert werden können. Die Gültigkeit der Konfiguration wird durch das Feature-Modell bestimmt. Somit beschreibt das Feature-Modell die gültigen Produktvarianten einer SPL.

Feature-Modell

Für die grafische Repräsentation eines Feature-Modells wird ein Feature-Diagramm genutzt [Kang et al., 1990; Apel et al., 2013a]. Das Feature-Diagramm wird als Baum dargestellt, bei dem die Kanten die Abhängigkeiten zwischen den Features repräsentieren und die Knoten die Features selbst enthalten [Czarnecki und Eisenecker, 2000]. Ausgehend vom Wurzelknoten wird ein Feature, das im Baum einem anderen Feature untergeordnet ist, als Subfeature bezeichnet. Ein Feature kann eine beliebige Anzahl an Subfeatures haben. Ein Subfeature kann nur ausgewählt werden, wenn das Elternfeature ebenfalls ausgewählt ist.

Um die Gemeinsamkeiten und die Variabilität in einem Feature-Diagramm auszudrücken, kann zwischen obligatorischen, alternativen und optionalen Features gewählt werden. Darüber hinaus ist es auch möglich, Features in einer logischen Oder-Beziehung zu verknüpfen [Batory, 2005]. Ein obligatorisches Feature ist immer dann ausgewählt, wenn das übergeordnete Feature ausgewählt ist. Im Gegensatz dazu muss ein optionales Feature bei Auswahl des übergeordneten Features nicht notwendiger Weise ausgewählt sein. Bei einer Alternative muss aus einer Gruppe von Features genau ein Feature ausgewählt werden. Innerhalb einer Oder-Gruppe muss mindestens ein Feature aus der Gruppe ausgewählt werden.

Um beliebige Abhängigkeiten zwischen Features, die sich nicht durch die Baumhierarchie ergeben, abzubilden, wird ein weiterer Abhängigkeitstyp benötigt. Bei diesem Abhängigkeitstyp handelt es sich um einen aussagenlogischen Ausdruck.

Beispiel einer Feature-Modellierung

In folgender Abbildung 2.2 stellt ich ein Feature-Diagramm einer vereinfachten Version der Graphen-Produktlinie (GPL) dar, die FeatureIDE [Thüm et al., 2014] als Beispiel beiliegt. Die GPL ist eine Familie von Graphen-Anwendungen, die durch frühere Arbeiten von Holland [1992]; Van Hilst und Notkin [1996] entwickelt wurde.

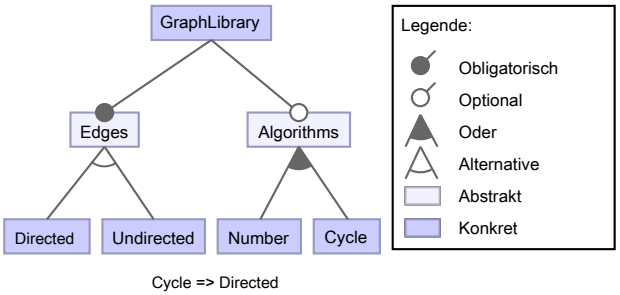


Abbildung 2.2: Feature-Diagramm der Graphen-Produktlinie

Die GPL besteht aus ihrem Basis-Feature *GraphLibrary* und den sechs weiteren Features *Egdes*, *Directed*, *Undirected*, *Algorithms*, *Number* und *Cycle*. Das Basis-Feature ist per Definition immer Bestandteil der Konfiguration. Das Feature *Egdes* ist ein obligatorisches Feature, welches durch einen ausgefüllten Kreis an dem zum Feature führenden Ende der Kante ausgedrückt wird. Da *Egdes* ein Kind vom Basis-Feature *GraphLibrary* ist, muss es in der Konfiguration immer ausgewählt werden und ist deshalb immer Bestandteil jedes Produkts der GPL. Im Gegensatz kann das optionale Feature *Algorithms*, durch einen nicht ausgefüllten Kreis gekennzeichnet, einer Konfiguration hinzugefügt werden, es besteht aber keine Pflicht. Beide Features, *Egdes* und *Algorithms*, sind zusätzlich als abstrakte Features definiert. Dies bedeutet, dass sie keine eigenen Implementierungsartefakte haben, sodass kein Quelltext zum Produkt hinzugefügt oder verändert wird. Sie dienen nur zum Gruppieren einer Menge von Features.

Die Features *Directed* und *Undirected* sind in einer alternativen Gruppe, durch einen nicht ausgefüllten Kreisbogen zwischen den Kanten dargestellt, mit ihrem übergeordneten Feature *Egdes* verknüpft. Da genau ein Feature einer Alternative ausgewählt werden muss, und das Elternfeature *Egdes* mit selektiert ist, kann ein Graph nur gerichtete oder ungerichtete Kanten haben, nicht jedoch beide Kantenarten zur gleichen Zeit.

Bei der Auswahl von Algorithmen, die ein Graph unterstützen soll, eröffnet sich die Wahl, das Feature *Number*, *Cycle* oder beide Features auszuwählen. Um diese Konstellation in der Abbildung oben darzustellen, wurde eine Oder-Gruppe (ausgefüllter Kreisbogen zwischen den Kanten) gewählt. Wenn das Feature *Algorithms* in der Konfiguration selektiert wurde, muss mindestens ein Feature von beiden ebenfalls ausgewählt werden. Da *Algorithms* ein optionales Feature ist, können die Subfeatures *Number* und *Cycle* auch als optionale Features angesehen werden.

2.3 Feature-orientierte Programmierung

Nachdem im vorherigen Kapitel die Grundlagen für Modellierung von Features behandelt wurden, beschriebt der Autor dieser Arbeit in diesem Abschnitt, wie Features im Quellcode implementiert werden. Um die Variabilität in Software-Produktlinien umzusetzen, existieren verschiedene Implementierungsansätze. Diese lassen sich in zwei Kategorien einteilen: annotationsbasiert oder kompositionsbasiert [Kästner et al., 2008]. Im folgenden Absatz erläutere ich Eigenschaften sowie Vor- und Nachteile der beiden Kategorien von Implementierungsansätzen.

Arten von Implementierungsansätzen

Beim annotationsbasierten Ansatz liegen die Features nicht in physisch getrennten Codefragmenten vor, sondern in einer gemeinsamen Quellcodebasis. Dazu werden im Quellcode mithilfe geeinigter Markierungen (Annotationen) die Codefragmente, dem entsprechenden Feature zugeordnet. Ein Problem, das sich daraus ergibt, ist, dass der Quellcode immer alle Feature-Implementierungsartefakte enthält. Dadurch wird der Code deutlich komplexer und schwerer zu verstehen, da Features häufig quer über den Quellcode zerstreut sind [Apel et al., 2013a]. Ein weitverbreiteter Vertreter von annotationsbasierten Ansätzen sind Präprozessoren. Diese werden häufig in C und C++-Projekten verwendet, u. a. Apache, Linux oder PostgreSQL [Liebig et al., 2010].

Im Gegensatz zum Annotationsansatz zeichnet sich der kompositionsbasierte Ansatz dadurch aus, dass Features einer SPL in eigenständige, physisch getrennte Codefragmente aufgeteilt werden. Dies führt zu einer Trennung von Funktionalität und erleichtert damit das Zurückverfolgen, welche Produktvariante welche Eigenschaften implementiert. Zu den Repräsentanten des Kompositionsansatzes zählen aspektorientierte Programmierung (AOP) [Kiczales et al., 1997, 2001; Webber und Gomaa, 2004] und Feature-orientierte Programmierung (*FOP*) [Prehofer, 1997; Batory et al., 2003]. Diese Arbeit fokussiert sich auf den kompositionsbasierten Ansatz FOP.

Bei FOP handelt es sich um einen sprachunabhängigen Ansatz zur Überwindung des Feature-Traceability-Problems. Damit wird die Grundlage geschaffen, um alle Implementierungsartefakte eines Features an einer Stelle im Quellcode zu kapseln. Features werden dabei durch je ein Feature-Modul implementiert, was zu einer Trennung und Modularisierung von Features und damit zu einer guten Feature-Traceability führt [Apel et al., 2013a]. Ursprünglich wurde FOP als Erweiterung für objektorientierte Programmierung (OOP) angedacht [Prehofer, 1997]. Über die Jahre wurden weitere Sprachkonstrukte, wie XML oder funktionale Programmierung und Dokumente, in Form von Text- oder HTML-Dateien, unterstützt. Zur Implementierung von Feature-Modulen werden in FOP verschiedene Werkzeuge unterstützt, darunter sind AHEAD [Batory et al., 2003] und FeatureHouse [Apel et al., 2010, 2013b]. In der vorliegenden Arbeit liegt der Schwerpunkt auf FeatureHouse.

FeatureHouse

Im Rahmen dieser Arbeit spielt FeatureHouse eine bedeutende Rolle, da es die benötigten Grundlagen für die Angleichungen der beliebigen Unterschiede zur Verfügung

stellt. Erst mit diesen bereitgestellten Informationen und Methoden sind Refactorings im Zuge der Migrationsvorbereitung durchführbar. Zu diesen Informationen zählen u. a. implementierte Klassen und Verfeinerungen eines Features. Mit den vorhandenen Methoden ist es über die Grenzen von verschiedenen Features hinweg möglich, alle Vorkommen inklusive Verfeinerungen einer anzupassenden Codestelle zu ermitteln und im darauffolgenden Schritt in einer gemeinsamen Aktion anzugleichen.

Bei FeatureHouse handelt es sich um ein sprachunabhängiges Framework mit einer Sammlung von Programmen zur Komposition von Feature-Modulen [Apel et al., 2010]. Die Repräsentation des Feature-Moduls erfolgt dabei anhand eines *Feature Structure Tree* (*FST*). Mit diesem FST kann jede Art von Software-Artefakten mithilfe einer hierarchischen Struktur dargestellt werden [Apel et al., 2013b]. Bei einem FST handelt es sich um einen vereinfachten Abstract Syntax Tree (AST). Im Gegensatz zum AST enthält ein FST nur Informationen, die für die Spezifikation der modularen Struktur eines Software-Artefakts notwendig sind sowie Details zur Implementierung [Apel et al., 2013b]. Zum Beispiel werden bei einem Java-Artefakt Packages, Klassen, Methoden etc. als Knoten des FST repräsentiert. Jeder Knoten eines FST hat zwei Eigenschaften: einen Namen und ein Typ, nicht jedoch einzelne Statements oder Expressions innerhalb einer Methode. Der Typ entspricht dabei dem strukturellen Element des Software-Artefakts. Der Wurzelknoten eines solchen Baumes repräsentiert jeweils ein Feature. Die inneren Knoten eines FST, als Kindknoten bezeichnet, charakterisieren ein Strukturelement (beispielsweise Klasse oder Methode). Jeder Kindknoten kann wiederum eine beliebige Anzahl an Kindknoten enthalten. Somit ist jeder Kindknoten ein Teilbaum, der eingeführte und erweiterte Klassen und Methoden sowie eingeführte Felder als Kindknoten besitzen kann. Die Blätter der Kindknoten enthalten den Inhalt des Strukturelements (z. B. Methodenrumpf oder Feldinitialisierung). Die Komposition von Software-Artefakten erfolgt durch die Superimposition der entsprechenden FSTs.

Superimposition

Der Prozess der Generierung eines Produkts (Feature-Komposition) bei FeatureHouse wird als *Superimposition* bezeichnet. In der Literatur wird die Superimposition als Prozess der Zusammensetzung von Software-Artefakten durch Verschmelzen ihrer dazugehörigen Strukturen beschrieben [Apel et al., 2013b]. Zu den Strukturen in objektorientierten Sprachen zählen Klassen, Interfaces, Methoden und Felder. Die Vereinigung der Strukturen erfolgt dabei anhand der gleichen Signatur (Name und Typ) [Apel et al., 2010].

Mathematisch wird die Superimposition mithilfe des Operator $\bullet$ über einer Menge von Features F beschrieben [Apel et al., 2010].

$$\bullet : F \times F \rightarrow F$$

Die Komposition von FSTs erfolgt dabei anhand von definierten Regeln. Ausgehend vom Wurzelknoten werden die Knoten vereinigt, die die gleiche Signatur aufweisen. Dieser Vorgang wird rekursiv für alle Knoten der FSTs durchgeführt. Das Ergebnis einer Komposition zweier FSTs ist ein neuer FST mit den Eigenschaften aller Knoten

beider FSTs. Die Komposition erzeugt in Bezug auf Java ein syntaktisch korrektes Java-Programm [Apel et al., 2013b].

Der Prozess der Komposition wird auch *Verfeinerung* genannt, da bei einer Vereinigung von zwei Features das zweite Feature die Klassen, Methoden oder Feldern des ersten Features verfeinert. Verfeinerungen erweitern die Funktionalität von vorher eingeführten Klassen, Feldern und Methoden. Für das Strukturelement Klasse besteht die Möglichkeit, neue Methoden, Felder und innere Klassen hinzuzufügen. Bei der Feldverfeinerung wird der Initialwert verändert. Bei der Verfeinerung von Methoden kann die originale Methode vollständig überschrieben und damit neu eingeführt oder die Funktionalität mit dem Schlüsselwort `original` erweitert werden.

Eine Programmvariante ist damit eine Aneinanderreihung von Kompositionen auf einer Menge von FSTs.

$$v = F_1 \bullet F_2 \bullet ... \bullet F_n$$

Bei der Superimposition werden die Feature-Module aller in der Konfiguration gewählten Features in einer bestimmten Reihenfolge zusammengesetzt. Die Reihenfolge, wie Feature-Module der gewählten Features komponiert werden sollen, kann durch den Nutzer selbst beeinflusst werden. Durch eine fehlerhafte Reihenfolge bei der Feature-Komposition kann es passieren, dass eine Methodenverfeinerung fehlschlägt, weil die verfeinernde Methode erst mit einem späteren Feature eingeführt wird.

Beispiel für eine Superimposition

In der Abbildung 2.5 zeige ich die Funktionsweise von Superimposition anhand eines Beispiels aus der GPL. Dafür werden auf der linken Seite die Ausschnitte des Quellcodes und auf der rechten Seite der passende FST zu den Feature-Modulen *GraphLibrary* und *Number* dargestellt. Die beiden Feature-Module enthalten die gleiche Klasse `Graph`. Das Feature *GraphLibrary* enthält zwei Felder,`nodes` und `edges`), und eine Methode `run`. Feature *Number* enthält ebenfalls eine Methode `run` und zusätzlich eine Methode `numberVertex`.

Bei einer Komposition beider FSTs werden beide `Graph`-Teilbaum zu einem gemeinsamen `Graph`-Teilbaum zusammengesetzt. Dabei werden gleiche Methoden verfeinert. Um von der verfeinernden Methode die originale Methode aufzurufen, existiert das Schlüsselwort `original`, ähnlich dem aus der OOP bekannten Schlüsselwort `super`. Die Methode `run` des Features *GraphLibrary* wird von dem Feature *Number* verfeinert. Während der Komposition wird das Schlüsselwort `original` durch den Aufruf der originalen Methode ersetzt. Dazu wird die originale Methode `run` in `run__GraphLibrary` umbenannt und der Aufruf von `original` durch die umbenannte Methode ersetzt. Der Quellcode, der durch den Komposer erzeugt wurde, kann mit einem Standard-Java-Compiler interpretiert werden.

Eine Besonderheit bei der Komposition in FeatureHouse nehmen Konstruktoren ein. Konstruktoren können ebenfalls verfeinert werden, aber im Gegensatz zu Methoden wird der Code des verfeinernden Konstruktors dem originalen Konstruktor hinzugefügt. Aus diesem Grund wird `original` nicht benötigt. Im unteren Drittel von Abbildung 2.5 wird der komponierte Konstruktor aus den beiden Features *GraphLibrary* und *Number* dargestellt. Dabei lässt sich erkennen, dass der entstandene

Konstruktor zuerst den Code des originalen Konstruktors und danach den Code der verfeinernden Konstruktoren in Reihenfolge der Komposition des Features enthält.

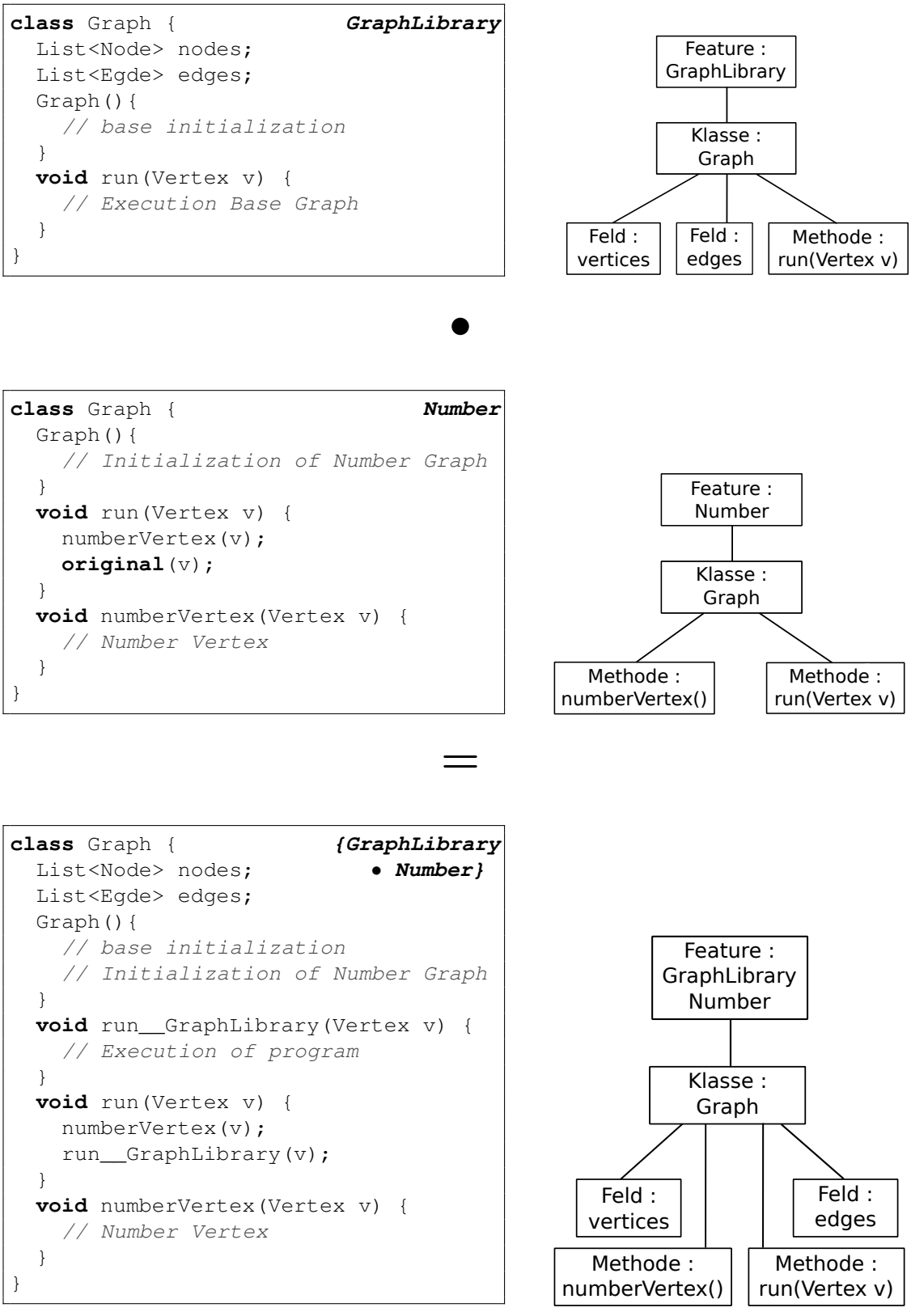


Abbildung 2.5: Komposition der Features GraphLibrary und Number

2.4 Zusammenfassung

In diesem Kapitel wurden Grundlagen von Software-Produktlinien charakterisiert. Dazu wurden in Abschnitt 2.1 allgemeinen Informationen inklusive Definition vorgestellt. Anschließend wurde das Konzept der Feature-Modellierung eingeführt und an einem Beispiel veranschaulicht. Im Abschnitt 2.3 wurden verschiedene Implementierungsansätze mit Fokus auf Feature-orientierte Programmierung erläutert. Zum Abschluss wurde die Generierung eines Produktes mithilfe von FOP skizziert.

3. Feature-orientiertes Refactoring zur Migration von Produktvarianten

In diesem Kapitel beschreibe ich meinen Ansatz zur Migration von Produktvarianten unter Verwendung des Feature-orientierten Refactorings. Dazu gehe ich zuerst auf Codeklone und die Möglichkeit der Erzeugung von Produktvarianten mithilfe des Clone-and-Own-Ansatzes ein. Im anschließenden Abschnitt befasse ich mich mit Refactoring im Zusammenhang mit SPLs. Dazu wird gezeigt, warum objektorientiertes Refactoring für SPLs nicht ausreichend ist. Im letzten Abschnitt stelle ich ausgehend von den Problemen bei der Migration von Produktvarianten und meinen Migrationsprozess vor.

3.1 Codeklone

Dieser Abschnitt stellt Informationen zu Codeklonen dar. Dabei erfolgt eine Einteilung dieser Klone in verschiedene Typen und Möglichkeiten der Erkennung solcher Klone. Zum Abschluss erläutere ich die Erzeugung von Produktvarianten mithilfe des Clone-and-Own-Ansatzes. Auf Produktvarianten, die mittels Clone-and-Own erzeugt wurden, führe ich die Migration in eine SPL durch.

Für meine Arbeit spielen Codeklone eine bedeutende Rolle, da sie auf ähnliche oder identische Funktionalität zwischen verschiedenen Produktvarianten hinweisen. Sie dienen somit der Erkennung dieser anzugleichenden und zu extrahierenden Funktionalitäten im Rahmen der Migration.

Ein Codeklon beschreibt ein Codefragment, das wiederholt in ähnlicher oder gleicher Form innerhalb des Quellcodes auftritt [Roy und Cordy, 2007]. Diese semantisch und syntaktisch gleichen Codefragmente entstehen in den meisten Fällen durch Kopieren. Das Vorhandensein solcher Codeklone ist in der Regel ein Indikator für mangelnde Softwarequalität [Kim et al., 2005]. Zahlreiche Studien haben gezeigt, dass die Codeklonrate in Softwaresystemen zwischen 7 und 23 Prozent liegt [Baker, 1995;

Baxter et al., 1998; Li et al., 2004]. Die Motive dieser Codeklone sind vielfältig. In manchen Fällen sind Codeklone gewollt [Kim et al., 2005] und eine wohlüberlegte Methode, um bestimmte Probleme (z.B. technische Einschränkungen, Verständlichkeit des Codes oder schnelle Markteinführung) zu lösen [Kapser und Godfrey, 2008]. Meistens haben sie aber einen negativen Effekt auf Wartbarkeit und Erweiterbarkeit von Software [Baxter et al., 1998; Juergens et al., 2009]. Beispielsweise müssen bei einer Fehlerbehebung alle Vorkommen des Codeklons, zusätzlich zum zu ändernden Code, ebenfalls geändert werden. Dies ist zeitaufwendig und fehleranfällig, da die Erkennung von Codeklonen im Allgemeinen nicht trivial ist und somit nicht immer alle Vorkommen gefunden werden. Dabei ist zu beachten, dass Codeklone Fehler nicht verursachen, aber bei inkonsequenten Anpassungen zu unerwartetem Programmverhalten führen können [Juergens et al., 2009]. Diese Probleme machen das Auffinden von Codeklonen wünschenswert. Zum Erkennen von Codeklonen wird dabei der Copy/Paste Detector (CPD) aus den Werkzeugen der statischen Codeanalyse von Quelltexten (PMD[1]) verwendet. Diese Codeklonerkennung wurde von Tonscheidt [2015] im Rahmen seiner Arbeit in FeatureIDE integriert, sodass ich in meinem Konzept auf diese Erkennung zurückgreifen kann. Innerhalb dieses Kapitels wird noch mal detaillierter auf die Codeklonerkennung eingegangen.

3.1.1 Klassifizierung von Codeklonen

Nach Roy et al. [2009] lassen sich Codeklone in vier Typen einteilen. Diese Typen unterschieden sich in der Art ihrer Ähnlichkeit. Dabei weisen Codeklone textuelle (Typ-I bis III) und semantische (Typ-IV) Ähnlichkeiten auf.

Typ-I

Bei Typ-I-Klonen handelt es sich um identische Codefragmente, die sich nur durch Leerzeichen, Tabulatoren und Zeilenumbrüchen voneinander unterscheiden. Außerdem werden bei diesen Codefragmenten Kommentare nicht berücksichtigt. Diese Klone werden auch als *exakte Kopien* (*Exact Clones*) bezeichnet.

Typ-II

Typ-II-Klone sind Klone mit syntaktisch identischen Codefragmenten, aber Unterschiede in den Bezeichnern. Sie beinhalten zudem alle Typ-I-Klone. Bei diesen Beinahe-Klonen (Near-Miss Clones), wie Typ-II-Klone bezeichnet werden, sind zusätzlich Veränderungen an den Bezeichnern von Feldern, Methoden, Klassen usw. vorhanden.

Typ-III

Der Typ-III schließt alle Typ-II-Klone mit ein und erlaubt darüber hinaus strukturelle Änderungen. Dazu zählt das Verändern, Hinzufügen oder Löschen von Anweisungen.

[1] http://pmd.sourceforge.net/pmd-4.3.0/cpd.html

Typ-IV

Bei dem letzten Typ, Typ-IV, handelt es sich ausschließlich um semantische Ähnlichkeiten von Codefragmenten. Dabei führen mehrere Codefragmente eine identische Operation aus, die aber auf verschiedene Arten implementiert ist.

Im nachfolgenden Listing 3.1.1 werden die verschiedenen Klontypen in einem originalen Fragment und einem geklonten Fragment veranschaulicht. Dabei werden die Codefragmente von Typ-I-Klons rot, von Typ-II-Klons gelb und von Typ-III-Klons grün umrandet dargestellt. Das Listing 3.1.2 stellt einen Typ-IV-Klon dar. Als Beispiel dient die Berechnung der Fibonacci-Zahlen. Dabei präsentiert das linke Codefragment die iterative und das rechte Codefragment die rekursive Implementierung dieser Berechnung (nach [Tonscheidt, 2015]).

```
void calc(int a)                Typ II
{
  if (a >= n) {                 Typ I
    c = a + 1; } //Comment1
  else
    c = a - 1;//Comment2
}
```

(a) originales Codefragment für Methode `calc`

```
void recalc(int a)              Typ II
{
  if(a>=n) {                    Typ I
   c=a+1;
    d=a+1;}                     Typ III
  else
    c=a-1;//Comment2
}
```

(b) geklontes Codefragment für Methode `recalc`

Listing 3.1.1: Codebeispiel für ein Codefragment vom Typ-I bis Typ-III

```
long fibonacci(int n) {
  if (n <= 2)
    return (n > 0) ? 1 : 0;

  long fib1 = 0, fib2 = 1;
  for (int i = 1; i < n; i++) {
    long newFib = fib1 + fib2;
    fib1 = fib2;
    fib2 = newFib;
  }
  return fib2;
}
```

(a) Typ IV-Klon für Methode `fibonacci` (iterativ)

```
long fibonacci(int n) {
  if (n <= 2)
    return (n > 0) ? 1 : 0;

  return fibonacci(n - 1) +
         fibonacci(n - 2);
}
```

(b) Typ IV-Klon für Methode `fibonacci` (rekursiv)

Listing 3.1.2: Codebeispiel für ein Codefragment vom Typ-IV (nach [Tonscheidt, 2015])

3.1.2 Clone-And-Own

Im Abschnitt 2.1 wird das SPL-Engineering als Möglichkeit zur Entwicklung von Varianten vorgestellt. Bei diesem Ansatz steht die systematische Wiederverwendung von Codefragmenten auf Basis gemeinsamer Software-Artefakte im Vordergrund.

In der industriellen Praxis werden neue Produktvarianten jedoch häufig über einen Clone-and-Own-Ansatz erzeugt [Rubin et al., 2012; Rubin und Chechik, 2013; Rubin et al., 2013; Antkiewicz et al., 2014]. Im Gegensatz zum SPL-Engineering kennzeichnet sich Clone-and-Own dadurch aus, dass neue Produktvarianten durch Klonen vorhandener Produktvarianten erzeugt und diese erzeugten Varianten anschließend an ihre Bedürfnisse (z.B. Verwendungszweck, Anforderungen oder Kundenwünsche) angepasst werden. Dieser Ansatz führt zu einer unsystematischen Wiederverwendung, da Anpassungen in einer geklonten Produktvariante keinen Einfluss auf die anderen geklonten Produktvarianten haben. In der Praxis wird dieser Ansatz auch als *Forking* bezeichnet [Kapser und Godfrey, 2008; Rubin et al., 2012]. Dabei bauen die Entwickler auf funktionierenden und getesteten Code auf und haben zudem die Freiheit, notwendige Anpassungen unabhängig von anderen Produktvarianten vorzunehmen. Die Vorteile sind niedrige initiale Entwicklungskosten und schnellere Markteinführung neuer Produktvarianten. Wenn ein Kunde eine neue Funktionalität wünscht, welche schon in einer anderen Produktvariante existiert, kann diese Funktionalität in die gewünschte Produktvariante kopiert werden. Da initiale Entwicklungskosten um ein Vielfaches im Gegensatz zur kompletten Neuentwicklung gesenkt werden können, sind Unternehmen in der Lage auf Veränderungen und Anforderungen ihrer Kunden schnell zu reagieren. Des Weiteren reduziert sich der Testaufwand von geklonten Funktionalitäten, da sie in anderen Produktvarianten diverse Funktions- und Systemtests durchlaufen und bestanden haben [Kapser und Godfrey, 2008]. Clone-and-Own stellt somit einen einfachen und schnellen Ansatz dar, um Implementierungsartefakte wiederzuverwenden.

Mittel- und langfristig ist Clone-and-Own jedoch mit Nachteilen verbunden, da sich mit jeder neuen Produktvariante die Wartungskosten erhöhen [Kapser und Godfrey, 2008; Rubin et al., 2012]. Wenn eine Funktionalität, die in vielen Produktvarianten vorliegt, aufgrund einer Fehlerbeseitigung oder Weiterentwicklung angepasst werden muss, müssen an allen duplizierten Codefragmenten Änderungen vorgenommen werden. Diese Anpassungen sind sehr zeitintensiv und fehleranfällig. Sollten Änderungen nicht an allen Codestellen durchgeführt werden, erhöht sich der Wartungsaufwand [Dalgarno und Beuche, 2007] und es kann zu Fehlern in den einzelnen Varianten kommen [Roy und Cordy, 2007].

3.2 Refactoring

In diesem Abschnitt wird der Begriff Refactoring im Zusammenhang mit SPLs eingeführt. Das umfasst Grundlagen des objektorientierten Refactorings, Grenzen dieses Refactorings sowie Erweiterungen, welche im Kontext der Feature-orientierten Programmierung notwendig sind.

Im Laufe der Zeit unterliegen Software-Produkte ständigen Änderungen. Aufgrund neuer oder geänderter Bedürfnisse von Kunden müssen Software-Produkte kontinuierlich verbessert, verändert und an neue Anforderungen angepasst werden [Mens und Tourwé, 2004]. Dieser Prozess der Änderungen nach Auslieferung eines Software-Produktes wird als Software-Evolution bezeichnet. Oft werden diese Änderungen ohne vollständiges Verständnis der zugrunde liegenden Struktur und des Designs der Software durchgeführt. Dadurch wird der Code komplexer und weicht von seinem ursprünglichen Design ab, wodurch die Qualität sinkt [Parnas, 1994; Fowler,

1999]. Dadurch können weitere Anpassungen nur unter großem Aufwand realisiert werden [Opdyke, 1992]. Laut Studien nehmen die Kosten für Wartung den größten Teil der Entwicklungskosten über die gesamte Lebensdauer eines Software-Produkts ein [Coleman et al., 1994; Zhou und Leung, 2007]. Um Software-Qualität schrittweise zu verbessern, ist eine Umstrukturierung (*Restructuring* [Arnold, 1986; Griswold und Notkin, 1993]) der betroffenen Codestellen notwendig. In regelmäßigen Abständen sollte eine Umstrukturierung stattfinden, um die gewünschte Software-Qualität zu erhalten.

3.2.1 Objektorientiertes Refactoring

Im Rahmen objektorientierter Softwareentwicklung wird diese Umstrukturierung als *Refactoring* bezeichnet. Erstmals hat sich Opdyke [1992] mit dem Thema Refactoring auseinandergesetzt. Aufbauend auf Opdykes Arbeiten definiert Fowler [1999] Refactoring als einen Prozess der Veränderung eines Softwaresystems, bei dem das Verhalten aus Sicht des Nutzers erhalten bleibt, während die interne Struktur jedoch verbessert wird. Refactoring verfolgt das Ziel, die Wartung zu erleichtern, um somit Verständlichkeit, Erweiterbarkeit und Anpassbarkeit des Quellcodes zu erhöhen. Somit wird durch die Bereinigung des Codes die Wahrscheinlichkeit minimiert, im späteren Verlauf Fehler einzuführen [Fowler, 1999]. Sollten Änderungen am Code nicht korrekt durchgeführt werden, kann dies jedoch zu Fehlern führen. Refactoring ist also riskant und sollte daher immer systematisch geplant und methodisch durchgeführt werden. Um den Entwickler in diesem Prozess zu unterstützen, haben Opdyke [1992] und Fowler [1999] Regeln für die Durchführung von Refactorings definiert. Damit soll sichergestellt werden, dass das Programm nach dem Refactoring bei gleichen Eingaben dieselben Ergebnisse berechnet wie davor.

Zum Bewahren des Verhaltens eines Programms kann beim Refactoring auf Vor- und Nachbedingungen zurückgegriffen werden [Opdyke, 1992; Fowler, 1999; Roberts, 1999]. Die Vorbedingungen werden vor dem Refactoring geprüft. Und nur, wenn alle definierten Vorbedingungen erfüllt wurden, werden entsprechende Aktionen auf dem Quellcode ausgeführt. Leider hat sich gezeigt, dass Vorbedingungen nicht immer eingesetzt werden können, um das Verhalten zu garantieren. Ein Problem ist, dass die statische Überprüfung der Vorbedingungen sehr zeitintensiv oder in einigen Fällen auch unmöglich ist [Mens und Tourwé, 2004]. Aus diesem Grund definiert Roberts [1999] in seiner Dissertation zusätzlich Nachbedingungen, die nach dem Refactoring überprüft werden. Diese Nachbedingungen können verwendet werden, um den Analyseaufwand gegenüber der ausschließlichen Verwendung von Vorbedingungen zu reduzieren. Des Weiteren können Nachbedingungen benutzt werden, um Abhängigkeiten zwischen Refactorings zu ermitteln und um Vorbedingungen von verketteten Refactorings zu berechnen Roberts [1999]. Im Gegensatz zu Vorbedingungen müssen alle durchgeführten Änderungen wieder rückgängig gemacht werden, wenn die Prüfung der Nachbedingungen nicht bestanden wird.

Grundlagen des Refactorings

Die Dissertation von Opdyke [1992] präsentiert eine Menge von Refactorings für objektorientierte Software-Produkte. Der Fokus liegt dabei auf automatischen Refactorings, um durch definierte Vorbedingungen das Verhalten der Software zu bewahren. Dafür teilte Opdyke Refactorings in zwei Gruppen ein, die Low-Level- und

High-Level-Refactorings. High-Level-Refactorings beschäftigen sich mit der Generalisierung und Spezialisierung von Vererbungshierarchien und Benutzung von Aggregation zum Beschreiben von Beziehungen zwischen den Klassen. Sie setzen sich aus einer Menge Low-Level-Refactorings zusammen. Bei Low-Level-Refactorings handelt es sich um Änderungen, wie zum Beispiel das Umbenennen eines Felders oder einer Methode. Dazu wird das entsprechende Refactoring `Rename Field` oder `Rename Method` vorgeschlagen. Zu jedem Low-Level-Refactoring stellt Opdyke die benötigten Argumente und die Vorbedingungen detailliert dar. Die Vorbedingungen werden dabei in Form von logischen Ausdrücken repräsentiert und stellen sicher, dass sich das beobachtbare Verhalten auch tatsächlich nicht verändert. Für die Verhaltenserhaltung sind semantische und syntaktische Eigenschaften eines Software-Produkts relevant. Dabei gilt es besonders, die Vererbung, den Gültigkeitsbereich, die Typkompatibilität und die semantische Gleichheit zu erhalten [Opdyke, 1992].

Auf Grundlage von Opdykes Arbeit erweitert Fowler den Katalog von Refactorings um zusätzliche Informationen. Dabei präsentiert er für 72 Refactorings jeweils die Motivation sowie detaillierte Einzelschritte, die aber manuell durchgeführt werden müssen. Fowler empfiehlt Änderungen am Code immer nur in kleinen Schritten zu realisieren, um so das Risiko eines Fehlers zu reduzieren.

Aus dem Katalog von Refactorings werden im Rahmen dieser Arbeit die beiden Refactorings `Rename` und `Pull Up` umgesetzt. Bei `Rename` wird eine Klasse, eine Methode oder ein Feld umbenannt. Mit diesen Umbenennungen verfolge ich in meiner Arbeit das Ziel, Codeelemente zwischen unterschiedlichen Produktvarianten anzugleichen. Zur Extraktion von Codeklonen wird das zweite Refactoring, `Pull-Up-to-Parent-Feature`, verwendet. Dabei handelt es sich um eine Sonderform des `Move`-Refactorings, bei der Klassen, Methoden oder Felder von einer Unterklasse in eine Superklasse verschoben werden. Bei Umsetzung der Refactorings ist zu beachten, dass Änderungen nicht auf dem durch Feature-Komposition generierten Code, sondern auf dem FOP-Code ausgeführt werden müssen. Um einen Eindruck von der Vorgehensweise eines Refactorings inklusive Vorbedingungen zu erhalten, wird im Folgenden ein Beispiel eines `Rename`-Refactorings aufbauend auf den vorgestellten Arbeiten von Opdyke [1992] und Fowler [1999] präsentiert.

Beispiel - Umbenennen einer Methode

In diesem Beispiel soll eine Methode umbenannt werden. Diese Namensänderung hat Auswirkungen auf alle Referenzen dieser Methode über den gesamten Quellcode. Dazu zählen beispielsweise Methodendeklarationen oder Aufrufe auf diese Methode in anderen Methoden.

Argumente:

Methode `f`, neuer Name `n`

Vorbedigungen:

1. Eine Methode mit dem Namen `n` und derselben Signatur wie `f` darf in der Klasse von `f` nicht existieren.

2. Ist `f` eine vererbte oder globale (public, protected oder default-Sichtbarkeit) Methode, darf f mit n in der Vererbungshierarchie nicht vorhanden sein.

Die Vorbedingungen stellen sicher, dass der neue Methodenname in keinem Konflikt zu einer bereits existierenden Methode steht. Die Änderung des Methodennamens hat keinen Einfluss auf das Verhalten des Programms. Weitere Programmeigenschaften bleiben ebenso erhalten. Wenn die Vorbedingungen erfüllt wurden, können folgende Schritte durchgeführt werden, um das Refactorings erfolgreich auszuführen.

Vorgehen:

1. Suchen aller Referenzen die Methode `f`
2. Erstellen einer Methode mit neuem Namen
3. Kopieren des gesamten Inhalts der Methode in die neu erzeugte Methode
4. Aktualisieren der Referenzen
5. Entfernen der alten Methode

Im Listing 3.2.1 wird das Refactoring `Rename Method` auf die Methode `getinvcdltmt` der Klasse `Customer` angewendet. Dabei werden diese Methode und alle Referenzen auf den neuen Namen `getInvoiceableCreditLimit` geändert. Dieses Beispiel wurde aus dem Buch von Fowler [1999] entnommen.

```
class Customer
{
  void getinvcdltmt
  {
    ...
  }
}
```

(a) Klasse `Customer` vor dem Refactoring

```
class Customer
{
  void getInvoiceableCreditLimit
  {
    ...
  }
}
```

(b) Klasse `Customer` nach dem Refactoring

Listing 3.2.1: Codebeispiel für die Umbenennung einer Methode in Klasse `Customer`

Im ersten Schritt werden Codestellen identifiziert, die refaktorisiert werden sollen. Dazu bietet sich eine automatische oder manuelle Programmanalyse an. Nach Ermittlung der Codestellen kann aus den Katalogen (u.a. von Fowler [1999] und Kerievsky [2004]) das geeignete Refactoring bestimmt werden. Anschließend werden Vorbedingungen überprüft, um das Verhalten des Programms nach dem Refactoring weiterhin zu garantieren. Nachdem die Vorbedingungen erfüllt wurden, kann das Refactoring ausgeführt werden.

3.2.2 Refactoring von Software-Produktlinien

In diesem Abschnitt beschreibe ich Konzepte für die Umsetzung des Feature-orientierten Refactorings in einer SPL zur Migration von Produktvarianten. Objektorientiertes Refactoring reicht für FOP nicht aus. Die Gründe erläutere ich im Folgenden und stelle Anpassungen an den Definitionen von Refactorings als auch an den Konzepten (Vor-, Nachbedingungen und Vorgehensweise) vor.

Grenzen objektorientierten Refactorings

Objektorientierte Refactorings, wie im vorherigen Abschnitt beschrieben, sind nur für einzelne Produktvarianten korrekt. Wenn das Refactoring auf mehrere Produktvarianten gleichzeitig ausgeführt wird, ist eine Sicherstellung des Verhaltens aller betroffenen Produktvarianten nicht mehr gewährleistet, wie ich anhand des Beispiels in Listing 3.2.2 zeige. In diesem Beispiel stelle ich die Auswirkung eines OO- und eines FOP-Refactorings auf FOP-Code dar. Konkret wird eine Umbenennung der Klasse `Graph` in `SimpleGraph` des Features *GraphLibrary* ausgeführt. Da OO-Refactorings die Variabilität innerhalb der SPL nicht berücksichtigen, werden die Anpassungen immer nur innerhalb des Features *GraphLibrary* durchgeführt. Das Listing 3.2.2b zeigt, dass das Refinement der `Graph`-Klasse im Feature *Number* durch das OO-Refactoring nicht berücksichtigt und damit nicht umbenannt wird. Dadurch ergeben sich zwei Compilerfehler im Feature *Number*. Der erste Fehler bezieht sich auf die Methode `run`. Diese Methode stellt ein Refinement dar, was durch das Schlüsselwort `original` verdeutlicht wird. Aufgrund der Umbenennung existiert für die Klasse `Graph` keine einführende Methode mehr, wodurch ein Compilerfehler erzeugt wird. Der zweite Fehler zeigt sich beim Hinzufügen einer Kante (`addEgde`) in Methode `main`. Da die einführende Klasse umbenannt wurde und die Methode `addEgde` in der Klasse `Graph` nicht vorhanden ist, verursacht der Aufruf von `addEgde` einen weiteren Compilerfehler. Diese Fehler werden durch einen gelben Hintergrund in der Abbildung kenntlich gemacht. Daraus resultierend können die Produktvarianten nicht mehr fehlerfrei erzeugt werden.

Da eine SPL mehrere Produktvarianten enthält, müssen bei einem Refactoring alle Produktvarianten berücksichtigt werden. Somit betreffen die Änderungen des Refactorings eine Vielzahl von Klassen und deren Referenzen. Um das beobachtete Verhalten einer SPL zu bewahren und damit die Variabilität zu garantieren, bedarf es deshalb bestimmter Erweiterungen. Daher besteht beim *variantenerhaltendem Refactoring* das Ziel, sowohl die Variabilität als auch das unveränderte Verhalten jeder Produktvariante zu erhalten [Schulze et al., 2012]. Um dies zu gewährleisten müssen weitere Bedingungen definiert werden. Dazu ist es unumgänglich, die verhaltenserhaltende Definition zu erweitern sowie Vor- und Nachbedingungen und Vorgehensweise bekannter OO-Refactorings anzupassen, sodass die Refactorings im FOP-Kontext genutzt werden können.

Ein Feature besteht aus einer Menge von Klassen. Weiterhin kann aber auch eine Klasse zu verschiedenen Features gehören. Soll ein Refactoring an einem Feature durchgeführt werden, müssen sämtliche Einführung und Verfeinerungen der refaktorisierten Klassen betrachtet werden. Wenn eine Klasse angepasst wird, sollten alle Features berücksichtigt werden, die diese Klasse beinhalten. Zusätzlich müssen alle Referenzen auf die Klasse in das Refactoring mit einbezogen werden. Im Listing

3.2.2d wird das Ergebnis eines korrekten variantenerhaltenden Refactorings für die `Graph`-Klasse im Feature *Number* dargestellt. Wie in dem Listing zu erkennen ist, wurde die verfeinerte Klasse `Graph` in `SimpleGraph` umbenannt. Zudem wurde auch die Referenz auf diese Klasse in Methode `main` aktualisiert. Dadurch kann im Gegensatz zu dem Listing 3.2.2b eine Implementierung eines Graphes wieder ohne Compilerfehler erzeugt werden und die Produktvarianten korrekt gebaut werden.

```
class Graph{          GraphLibrary
  void run(Vertex v){}
  void addEgde(Egde egde){}
}
```

$\Longrightarrow$

```
class SimpleGraph{    GraphLibrary
  void run(Vertex v){}
  void addEgde(Egde egde){}
}
```

(a) Klasse `Graph` vor OO-Refactoring

(b) Klasse `Graph` nach OO-Refactoring

```
class Graph{                  Number
  void run(Vertex v){
    original(v);
  }
  static void main(String[] a){
    Graph g = new Graph();
    g.run(getVertex(a));
    g.addEgde(new Egde());
  }
}
```

$\Longrightarrow$

```
class Graph{                  Number
  void run(Vertex v){
    original(v);
  }
  static void main(String[] a){
    Graph g = new Graph();
    g.run(getVertex(a));
    g.addEgde(new Egde());
  }
}
```

(c) Klasse `Graph` vor FOP-Refactoring

(d) Klasse `Graph` nach FOP-Refactoring

Listing 3.2.2: Codebeispiel für das fehlerhafte Umbenennen einer Klasse

Definition von variantenerhaltenden Refactoring

Variantenerhaltendes (Variant-preserving) Refactoring wurde erstmals in den Arbeiten von Schulze et al. [2012, 2013] vorgestellt. Apel et al. [2013a] beschreiben verfeinernde Definitionen für Refactorings in SPLs. Die von Apel et al. getroffenen Differenzierungen sind im Kontext dieser Arbeit jedoch von nachrangiger Bedeutung. Daher verwende ich in meiner Arbeit die einfachere Definition von Schulze et al.. Um von einem variantenerhaltenden Refactoring bei einer Änderung des Feature-Modells und/oder der Implementierung eines Features zu sprechen, müssen folgende zwei Bedingungen erfüllt sein:

1. Jede gültige Feature-Kombination muss nach dem Refactoring gültig sein. Dabei wird Gültigkeit durch das Feature-Modell bestimmt.

2. Jede gültige Feature-Kombination, die vorher kompilierbar war, muss auch nach dem Refactoring kompilierbar sein und dasselbe beobachtete Verhalten wie vor dem Refactoring aufweisen.

Die erste Bedingung bezieht sich auf Refactorings von Feature-Modellen. Bei einer Anpassung des Feature-Modells müssen dementsprechend die vor dem Refactoring gültigen Kombinationen von Features auch nach dem Refactoring gültig seien. Es dürfen somit keine Produktvarianten aus der Menge aller gültigen Produktvarianten entfernt werden. Die zweite Bedingung beschreibt Refactorings am Feature-Code. Dabei muss bei einer Änderung am Feature-Code das Verhalten aller gültigen Produktvarianten bewahrt werden. Im Vordergrund meiner Arbeit werde ich mich auf die zweite Bedingung fokussieren. Auf die erste Bedingung kann trotzdem nicht verzichtet werden, weil das umzusetzende `Pull-Up-to-Parent-Feature`-Refactoring neue Features und Constraints dem Feature-Modell hinzufügen können.

Anpassungen für Feature-orientiertes Refactoring

Beim variantenerhaltenden Refactoring müssen nicht nur die Klassen betrachten werden, sondern auch Features und Beziehungen zwischen Features und Klassen. Schulze et al. [2012, 2013] unterscheiden dabei zwischen zwei Arten von Refactorings: feature-übergreifendes (inter-feature refactoring) und feature-internes Refactoring (intra-feature refactoring). Feature-internes Refactoring zeichnet sich dadurch aus, dass sich die Änderungen immer nur auf Klassen desselben Features beziehen. Ein Beispiel für diese Art des Refactorings ist `Extract Method`. Müssen Änderungen des Refactorings hingegen an einer oder mehreren Klassen verschiedener Features durchgeführt werden, wird von einem feature-übergreifenden Refactoring gesprochen. `Pull Up` ist ein Beispiel für feature-übergreifendes Refactoring. Die Möglichkeiten der Auswirkung feature-interner und feature-übergreifender Refactorings auf Klassen und Features lassen sich mithilfe von Dimensionen veranschaulichen [Schulze et al., 2012, 2013]. Dazu kann jedes Refactoring einer der vier Dimensionen, D1 bis D4, zugeordnet werden. Feature-internes Refactoring beinhaltet die Dimensionen D1 und D2, während feature-übergreifendem Refactoring D3 und D4 zugeordnet werden. Für jedes Refactoring existieren bestimmte Risiken und Herausforderungen, die gemeistert werden müssen, damit die Änderung der oben genannten Definition genügt. Diese werden im Folgenden beschrieben.

Mit der Abbildung 3.1 stellt ich die vier verschiedenen Dimensionen dar. Dabei sind die Dimensionen des feature-übergreifenden Refactorings (rot) und die des feature-internen Refactorings (blau) gekennzeichnet. Zur Darstellung verwende ich die Kollaborationsansicht. Kollaborationen bestehen aus einer Menge von Klassen, die miteinander interagieren, um die Funktionalität eines Features zu implementieren [Apel et al., 2013a]. Innerhalb einer Kollaboration spielen verschiedene Klassen verschiedene Rollen. Eine Rolle kapselt die Funktionalität einer Klasse, welche für eine Kollaboration relevant ist [Apel et al., 2013a]. Jede Rolle wird getrennt von anderen Features und Klassen in einer eigenen Datei gespeichert. Eine Kollaboration besteht somit aus allen Rollen eines Features. In einer Rolle kann es Einführungen (Introductions) und Verfeinerungen (Refinements) von Klassen, Feldern und Methoden geben.

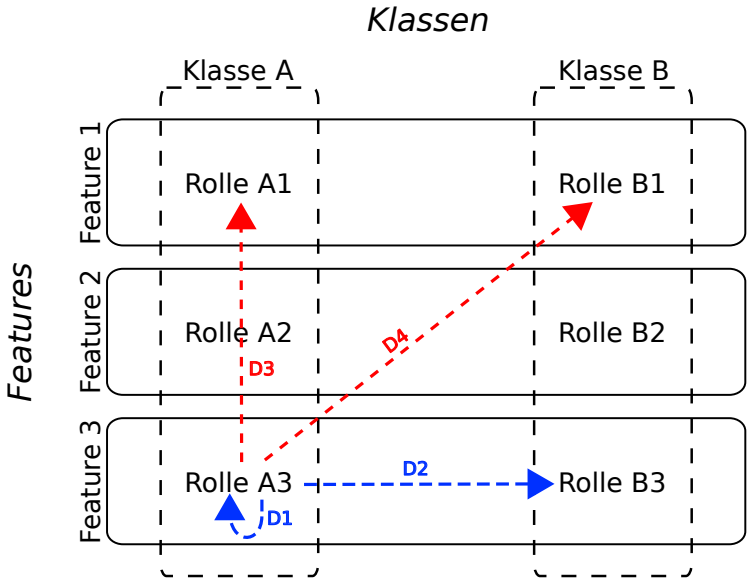


Abbildung 3.1: Dimensionen des feature-orientierten Refactoring (basierend auf [Schulze et al., 2013])

Bei der Dimension D1 handelt es sich um die einfachste Form des Refactorings. Der Grund dafür ist, dass Auswirkungen des Refactorings nur die eigene Klasse eines einzigen Features betreffen. Die Änderungen haben somit keinen Einfluss auf andere Klassen oder Features. Die Dimension D2 beschränkt sich nur auf ein Feature, aber über Klassengrenzen hinweg. Da beide Dimensionen zu feature-internen Refactoring gehören und keine Feature-Grenzen überschreiten, argumentieren Schulze et al. [2012, 2013], dass feature-interne Refactorings mithilfe von objektorientierten Refactorings (OOR) durchgeführt werden können. Auch wenn in diesem Fall OOR angewendet werden kann, müssen bei Prüfung der Vorbedingungen trotzdem alle Features betrachtet werden, um sicherzustellen, dass keine unerwünschten Nebeneffekte entstehen [Schulze et al., 2012, 2013]. Diese Prüfung der Vorbedingungen kann jedoch nicht von OORs geleistet werden. Die Dimension D3 des feature-übergreifenden Refactorings erfordert Änderungen an mehreren Features einer Klasse. Die letzte und komplexeste Dimension D4 des feature-übergreifenden Refactorings ist eine Kombination der Dimension D2 und D3, da Änderungen neben verschiedenen Klassen auch mehrere Features betreffen.

Zusammenfassend lässt sich festhalten, dass das Verhalten aller Produktvarianten vor und nach Refactoring sichergestellt werden muss. Mit objektorientiertem Refactoring kann dies aber nicht in allen Fällen garantiert werden. Somit müssen die Definitionen des Refactoring erweitert werden, um als variantenerhaltend zu gelten. Dafür werden für das variantenerhaltende Refactoring weitere Dimensionen berücksichtigt.

3.3 Migration

Dieser Abschnitt erläutert mein Konzept der Migration von Clone-And-Own-Varianten in eine Software-Produktlinie. Dazu wird auf die Probleme bei der Migra-

tion von Clone-And-Own-Varianten eingegangen und der Migrationsprozess vorgestellt.

3.3.1 Migration von Produktvarianten

Da Clone-And-Own-Varianten mittel- und langfristig Probleme verursachen (z.B. hohe Wartungskosten siehe Abbildung 2.1), besteht bei Unternehmen der Wunsch nach einem alternativen Konzept. Dabei hat sich erwiesen, dass langfristig diese Probleme durch Softwareproduktlinien besser gelöst werden können. Da eine Neuimplementierung des gesamten Codes in einer SPL sehr kostspielig ist, müssen andere Wege gefunden werden, um mit möglichst wenig Aufwand den vorhandenen Code der Clone-And-Own-Varianten in eine SPL zu überführen. Im Zusammenhang mit der Migration von Produktvarianten wird von einer variantenerhaltenden Migration gesprochen. Eine variantenerhaltende Migration ist ein Prozess, der eine oder mehrere Produktvarianten in eine SPL überführt. Dabei muss jede Produktvariante vor und nach der Migration dasselbe Verhalten aufweisen [Fenske et al., 2014]. Dazu gehört, dass aus einer migrierten SPL durch Feature-Komposition dieselben Produkte mit den gleichen Eigenschaften generiert werden können wie vor der Migration. Im nächsten Abschnitt beschreibe ich Probleme, welche bei einer Migration gelöst werden müssen.

Probleme bei der Migration

Bei einer Migration werden gleiche Funktionalitäten verschiedener Produktvarianten in gemeinsam genutzte Software-Artefakte einer SPL extrahiert. Aufgrund der Implementierung der Produktvarianten mithilfe des *Clone-and-Own*-Ansatzes ist anzunehmen, dass ein Großteil der Funktionalität in mehreren Produktvarianten gleich ist [Tonscheidt, 2015]. Somit können Produktvarianten eine erhöhte Anzahl von ähnlichen oder identischen Codestellen aufweisen. Das Problem ähnlicher Funktionalitäten zwischen den Produktvarianten entsteht dadurch, dass jede Produktvariante nach dem Klonen isoliert von anderen Produktvarianten weiterentwickelt wurde. Auf diese Weise kann es vorkommen, dass unterschiedliche Implementierungen von gleichen oder annähernd gleichen Funktionalitäten in verschiedenen Produktvarianten vorliegen [Yoshimura et al., 2006].

Ich vermute, dass sich ähnliche Funktionalitäten unter anderem in verschiedenen Namensausprägungen unterscheiden. Die Vermutung liegt beispielsweise darin begründet, dass unterschiedliche Entwickler unterschiedliche Namenskonventionen bevorzugen. In Listing 3.3.1 werden zwei Produktvarianten (3.3.1a und 3.3.1b) dargestellt, die jeweils eine Berechnung der Quadratsumme enthalten. Beide Implementierungen sind bis auf den Methodennamen (`squaresSum` vs. `sumOfSquares`) identisch. Durch diese unterschiedlichen Namen wird die Extraktion in ein gemeinsames Software-Artefakt erschwert. Dies hängt unter anderem auch damit zusammen, dass die Produktvarianten zusätzlich zu den unterschiedlichen Namen auch eine Vielzahl an unterschiedlich benannten Referenzen enthalten. Diese müssen in jeder Variante konsistent angeglichen werden. Sollte die Angleichung in einer oder mehreren Varianten aus verschiedenen Gründen (beispielsweise Unachtsamkeit) nicht durchgeführt werden und kann die Extraktion in dem Fall nicht (vollständig) durchgeführt werden. Damit hätte sich im Gegensatz zum Clone-and-Own-Ansatz kaum etwas geändert,

da die Wartung weiterhin an verschiedenen Codestellen erfolgen muss und somit wird das Ziel der Migration gemindert. Mit der Migration wird aber das Ziel verfolgt, gleiche Funktionalität nur in einem geteilten Software-Artefakt zu implementieren, um die Wartung und Wiederverwendung zu erhöhen.

```
int squaresSum(int n) {
  int sum = 0;

  for (int i = 0; i <= n; i++)
    sum += i * i;

  return sum;
}
```

(a) Produktvariante a

```
int sumOfSquares(int n) {
  int sum = 0;

  for (int i = 0; i <= n; i++)
    sum += i * i;

  return sum;
}
```

(b) Produktvariante b

Listing 3.3.1: Berechnung der Quadratsumme in zwei Produktvarianten

Schon dieses kleine Beispiel zeigt, dass der Migrationsprozess bei einer Vielzahl von Produktvarianten komplex und aufwendig ist. Der Grund dafür ist, dass beliebige Unterschiede zwischen den Produktvarianten erkannt und im Prozess der Migration angeglichen und konsolidiert werden müssen. Mit jeder zu migrierenden Produktvariante erhöht sich der Aufwand für die Erkennung und Angleichung.

Sollte diese Angleichung von Funktionalitäten aus verschiedenen Produktvarianten nicht korrekt durchgeführt werden, kann dies dazu führen, dass einzelne Produktvarianten nicht mehr generiert werden können. Im schlimmsten Fall können Varianten zwar noch kompiliert werden, aber diese das gewünschte Verhalten nicht mehr widerspiegelt. Diese Fehler sind dann schwerer zu finden als Fehler bei der Kompilierung oder beim Start des Programms. Damit ist die zweite Bedingung der Eigenschaft einer verhaltensbewahrenden Migration nicht mehr gegeben.

Aufgrund der Unterschiede zwischen den Produktvarianten, die manuelle Eingriffe durch die Entwickler erfordern, ist eine vollautomatische Migration nicht möglich. Vielmehr benötigen wir vorbereitende Refactorings um die Unterschiede zwischen den Produktvarianten anzugleichen, damit im nachfolgenden Schritt eine Vielzahl von identischen Funktionalitäten in gemeinsame Software-Artefakte auslagern kann.

3.3.2 Migrationsprozess

Der Migrationsprozess wird in Abbildung 3.2 dargestellt. Anhand der Abbildung ist zu sehen, dass der Migrationsprozess aus drei Teilschritten besteht. Im ersten Schritt erfolgt eine Überführung der Produktvarianten in eine initiale Software-Produktlinie. Da diese Überführung nicht Teil meines Konzeptes ist, verwende ich dafür den Ansatz von Tonscheidt [2015]. Dabei werden die Produktvarianten in einer Alternative angeordnet. Ausgehend von der initialen SPL kann direkt die Extraktion von Codeklonen erfolgen oder vorbereitende Schritte vor der Extraktion von Codeklonen durchgeführt werden. Das Angleichen in diesem Zusammenhang bezeichnet das Vereinheitlichen von Namensunterschieden zwischen den Codeklonen. Ein Beispiel für das Angleichen ist, alle Methoden, die in verschiedenen Produktvarianten die gleiche Funktionalität aufweisen, auf einen gemeinsamen aussagekräftigen Namen zu

ändern. Nachdem identische Funktionalitäten eines Klontyps II vereinheitlicht wurden, kann die Extraktion in gemeinsam genutzte Software-Artefakte durchgeführt werden. Dieser Zyklus der Angleichung und der Extraktion wird schrittweise für alle gefundenen Codeklone ausgeführt. Können keine Codeklone mehr vereinheitlicht oder extrahiert werden, ist die Migration abgeschlossen. Die Ermittlung der anzugleichenden oder extrahierenden Codeklone wird mittels einer Codeklonerkennung/-analyse durchgeführt, welche Bestandteil einer jeden Angleichung oder Extraktion ist.

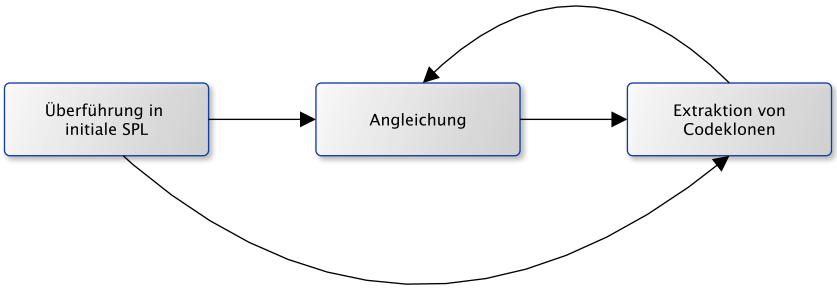


Abbildung 3.2: Ablauf des Migrationsprozesses

In den beiden anschließenden Abschnitten werden benötigte Bestandteile der Migration betrachtet, auf denen meines Konzeptes aufbaut. Dies ist zum einen der Prozess der Überführung von Produktvarianten in eine initiale SPL und zum anderen die Codeklonerkennung/-analyse. Kapitel 4 beschreibt detailliert die beiden umsetzenden Schritte Angleichung und Extraktion meines Konzeptes und veranschaulicht diese Teilschritte mithilfe eines Beispiels.

Überführung in eine initiale SPL

Zu Beginn der Migration werden die Produktvarianten in eine initiale SPL überführt. Eine initiale SPL für eine Menge n von Produktvarianten ist in Abbildung 3.3 dargestellt. Bei der Überführung wird im Feature-Modell zuerst ein abstraktes Basis-Feature erzeugt. Anschließend wird jede Produktvariante nacheinander in ein Feature transformiert. Dabei werden alle Implementierungsartefakte jeder Produktvariante ins dazugehörige Feature kopiert [Tonscheidt, 2015]. Diese Features werden als Alternativen des Basis-Features dargestellt. Die initiale SPL enthält für jede Produktvariante eine eigene Konfiguration. Jede dieser Konfigurationen beinhaltet neben dem Basis-Feature das Feature der jeweiligen Variante.

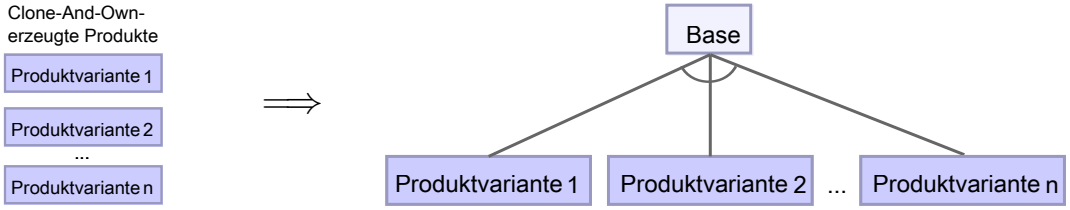


Abbildung 3.3: Überführung von Produktvarianten in initiale SPL (basierend auf Tonscheidt [2015])

Codeklonerkennung/-analyse

Um ähnliche Funktionalitäten zu erkennen, wird ein Klondetektor benötigt. Dafür verwende ich die Codeklonanalyse aus der Arbeit von Tonscheidt [2015], welche einen entsprechenden Detektor enthält. Der Grund für die Verwendung dieser Analyse ist die Berücksichtigung der Feature-Beziehungen und die Darstellung der Analyseergebnisse in einer individuellen Ansicht. Mit Berücksichtigung der Beziehungen zwischen den Features wurde die Möglichkeit geschaffen, Codeklone in die zwei Gruppen, feature-interne und feature-übergreifende Codeklone, einzuteilen. Auf Grundlage dieser Einteilung sowie der individuellen Ansicht kann ein Entwickler, auf einen Blick Codeklone erkennen, die relevant für einen Migrationsprozess sind [Tonscheidt, 2015]. Wie in Abschnitt 3.1 erwähnt, wird CPD zur Erkennung von Codeklonen verwendet. Beim CPD handelt es sich um einen Klondetektor, der auf einem textbasierten Klonerkennungalgorithmus aufbaut [Karp und Rabin, 1987]. Dabei sucht der Algorithmus mit Hilfe von Hash-Werten in einer Zeichenkette nach Mustern einer anderen Zeichenkette. Der Klondetektor wird auf einer Menge von Java-Dateien und/oder Features ausgeführt. Zudem können bestimmte Elemente ausgeschlossen werden oder die Erkennung auf eine bestimmte Klasse oder ein bestimmtes Feature beschränkt werden. Da für jeden Extraktionsschritt nur Codeklone einer Klasse in den unterschiedlichen Features betrachtet werden müssen, kann in diesem Fall eine Beschränkung auf einen bestimmten Klassennamen erfolgen. Der verwendete Klondetektor bietet keine Unterscheidung von Klontypen (siehe Abschnitt 3.1.1) an. Für beide Migrationsschritte wird jedoch die Einteilung benötigt. Dabei arbeitet der Angleichungsschritt auf Typ-II Klonen, während Codeklone vom Typ-I für den Extraktionsschritt verwendet werden. Die Codeklone vom Typ-III und Typ-IV spielen in meinem Konzept der Migration keine Rolle und können dementsprechend vernachlässigt werden. Aus diesem Grund habe ich die Ergebnisse von CPD um die notwendige Unterscheidung der Klontypen erweitert. Mit der Einteilung in Klontypen und in feature-interne und feature-übergreifende Klone ist die Codeklonanalyse abgeschlossen. An dieser Stelle setzt meine Arbeit an, die die Migration von Produktvarianten in eine SPL mit Unterstützung von Feature-orientiertem Refactoring behandelt.

3.4 Zusammenfassung

In diesem Kapitel wurden ausgehend von den Grundlagen des objektorientierten Refactorings die Grenzen für die Anwendung in SPLs erläutert. Anschließend wurde das variantenerhaltende Refactoring eingeführt, welches die notwendigen Erweiterungen beinhaltet, damit das Verhalten aller Varianten gewahrt bleibt. In Abschnitt 3.3 wurden die Probleme bei der Migration von Produktvarianten präsentiert und daraus folgend mein Migrationsansatz vorgestellt.

4. Variantenerhaltendes Refactoring

In diesem Kapitel beschreibe ich die umzusetzenden variantenerhaltenden Refactorings meines Konzeptes der Migration von Produktvarianten. Vorneweg kläre ich die Voraussetzungen, die vorliegen müssen, damit die beiden variantenerhaltenden Refactorings zur Angleichung und Extraktion von Codeklonen durchgeführt werden können. Anschließend werden die Refactorings erläutert. Hierzu beschreibe ich die Vorgehensweise dieser Refactorings und lege dar, welche Ergänzungen in den Vor- und Nachbedingungen zum Bewahren des Verhaltens der zu migrierenden Produktvarianten notwendig sind.

4.1 Voraussetzungen

Vor der Angleichung und Extraktion von Codeklonen müssen bestimmte Voraussetzungen erfüllt sein, damit die Konsolidierung überhaupt systematisch durchgeführt werden kann. Diese vorliegenden Voraussetzungen werde ich anhand der Eigenschaften einer Methode erläutern.

Die Extraktion einer Methode kann nur durchgeführt werden, wenn die Signaturen dieser Methode in verschiedenen Produktvarianten komplett identisch ist. Dies bedeutet, dass sowohl gleiche Methodennamen, gleiche Typen und Namen der Parameter, gleiche Anweisungen und eben auch gleich benannte Klassen vorliegen müssen. Ist auch nur ein Kriterium nicht erfüllt, kann die Extraktion für eine gegebene Methode nicht durchgeführt werden und die Codeklone damit nicht konsolidiert werden. Codeklone werden mithilfe der Codeklonerkennung ermittelt und durch Analyse dieser Ergebnisse in die verschiedenen Typen (siehe Abschnitt 3.1.1) eingeteilt werden. Sollte eine Methode in verschiedenen Produktvarianten als Typ-I-Klon erkannt werden, kann es immer noch das Problem geben, dass die Klassennamen, in denen die geklonte Methode definiert ist, unterschiedlich sind. Da unterschiedliche Klassennamen vorliegen, kann das `Pull-Up-to-Parent-Feature`-Refactoring diese

Methode nicht in ein übergeordnetes Feature verschieben. Dennoch lässt sich festhalten, dass die Extraktion nur auf Typ-I-Klonen durchgeführt werden kann, mit der Einschränkung, dass die Klassennamen der betroffenen Klone identisch sein müssen.

In meinem Konzept zielt die Angleichung auf Vereinheitlichen von Namensunterschieden ab und kann somit nur auf Codeklone vom Typ-II erfolgen. Bei den Typ-II-Klonen existieren unterschiedliche Namen der Methode oder ihrer Elemente (z.B. Parameter, lokale Variablen). Diese Namensunterschiede können durch das `Rename`-Refactoring angeglichen werden.

Bei einer Methode vom Typ-III können neben Umbenennungen auch Statements, Ausdrücke, Parameter etc. hinzukommen oder fehlen. Typ-IV-Klone entziehen sich weitgehend einer systematischen Konsolidierung mithilfe von Refactoring, da es sich um unabhängige Implementierungen derselben Funktionalität handelt. Für Typ-III- und Typ-IV-Klone ist unser Ansatz somit nicht geeignet. Zur Angleichung von Typ-III-Klonen sind weitere variantenerhaltende Refactorings wie beispielsweise `Extract Method` notwendig. Bei dieser Angleichung werden einzelne Anweisungen des Methodenblocks in eigene Methoden extrahiert. Nachdem dieser Extraktionsschritt durchgeführt wurde, kann anschließend die Vereinheitlichung von Namensunterschieden ausgeführt werden. Um den Aufwand der vorliegenden Arbeit in einem machbaren Rahmen zu halten, wird die Behandlung von Typ-III und IV-Klonen zukünftigen Arbeiten überlassen.

4.2 Angleichung von Codeklonen

In diesem Abschnitt wird erklärt, was ich für mein Konzept unter einer Angleichung verstehe und welche Schritte notwendig sind, damit eine Angleichung fehlerfrei durchgeführt werden kann. Dabei werde ich immer wieder den Bezug zur Migration herstellen.

Unter einer Angleichung von Codeklonen verstehe ich das Beseitigen von Unterschieden zwischen ähnlichen Codefragmenten. Da die Angleichung nur auf Codeklone vom Typ-II angewendet wird, zielt sie auf unterschiedliche Namen ab, welche die Konsolidierung von Typ-I und -II-Klonen verhindert. Dazu müssen alle Codeklone eines bestimmten Codefragments so angepasst werden, dass vorhandene Unterschiede entfernt werden, d.h. Codeklone des Typs-II in werden Typ-I-Klone transformiert. Bei der Angleichung von Codeklonen wird für jeden anzugleichenden Unterschied ein Codeklonfragment ausgewählt, welches als Modell für alle anderen Codeklonfragmente dient. Nachdem ein Codeklonfragment als Modell ausgewählt ist, erfolgen Schritt für Schritt die Anpassungen aller Klone an das Modell. Die Codeklonerkennung wird somit beim Angleichen zur Ermittlung von Codeklonen, die umbenannt werden sollen, verwendet.

Mein Ansatz unterstützt die beschriebene Vorgehensweise des schrittweisen Angleichens von Codeklonen. Diese schrittweise Angleichung benötigt weiterhin Unterstützung von Entwicklern, kann aber mithilfe von Codeklonerkennung und Refactorings semi-automatisch durchgeführt werden. Die Aufgabe des Entwicklers besteht in der Auswahl des als Modell genutzten Codefragments. Dadurch bleibt in meinem Ansatz der Einfluss des Entwicklers auf die Angleichung erhalten, sodass er entscheidet, welche Angleichung wie durchgeführt wird.

Für die Angleichung von Codeklonen habe ich mich entschieden, das Refactoring `Rename` zu verwenden. Eine Studie zur Ermittlung der Häufigkeit der Verwendung von Refactorings zeigt, dass `Rename` und `PullUp` Refactorings zu den am häufigsten eingesetzten Refactorings zählen [Murphy-Hill et al., 2009]. Dabei sticht besonders das Refactoring `Rename` hervor, welches rund 75 Prozent aller Aufrufe auf sich vereinen kann. Damit leistet meine Arbeit auch einen Beitrag für Feature-orientierte Programmierung im Allgemeinen über den hier fokussierten Anwendungsfall der Migration hinaus.

Rename-Refactoring

Das Refactoring `Rename` bietet für verschiedene Codeelemente (wie z.B. Klassen, Methoden oder Felder) Möglichkeiten der Umbenennung an. In meinem Ansatz setze ich es dafür ein, unterschiedliche Namen in Typ-II-Klonen anzugleichen.

Wie in meinen Einführungen zu variantenerhaltendem Refactoring erklärt, müssen für `Rename` vier verschiedene Dimensionen (siehe Abbildung 3.1) betrachtet werden. Anhand der Tabelle 4.1 wird ersichtlich, bei welcher Sichtbarkeitsausprägung der Codeelemente welche Dimensionen anzupassen sind. Allgemein lässt sich festhalten, dass sich bei drei von vier Ausprägungen Umbenennungen auf mehr als eine Dimension erstrecken. Nur im Fall der lokalen Sichtbarkeit (z.B. lokale Variablen) bezieht sich `Rename` nur auf eine Dimension (D1). Auch wenn Refactorings für private Codeelemente nur in zwei Dimensionen durchgeführt werden, müssen bei Prüfungen der Vor- und Nachbedingungen alle Dimensionen auf Namenskollisionen überprüft werden. Dies bedeutet, es müssen sämtliche Features der betroffenen Klassen sowie deren Super- und Subklassen betrachtet werden.

Dimension / Sichtbarkeit	D1	D2	D3	D4
lokal	x			
private	x		x	
protected	x	x	x	x
public	x	x	x	x

Tabelle 4.1: Sichtbarkeitsausprägung der Codeelemente für die verschiedenen Dimensionen

Das Refactoring `Rename` wird von uns für folgende Codeelemente unterstützt:

- Klassen und Interfaces
- Methoden
- Felder
- Lokale Variablen
- Konstruktoren und Parameter

Nachfolgend stelle ich einzeln die genannten `Rename`-Refactorings vor. Dazu wird erläutert, welche Argumente für Vor- und Nachbedingungen notwendig sind. Im Anschluss daran werden Vor- und Nachbedingungen verdeutlicht. Dafür erweitere ich Vor- und Nachbedingungen um zusätzliche Bedingungen, um das Verhalten aller Varianten zu bewahren. Zum Abschluss beschreibe ich die Vorgehensweise jedes `Rename`-Refactorings und veranschauliche diese anhand eines Codebeispiels.

4.2.1 Klassen und Interfaces

Beim `Rename`-Refactoring von Klassen und Interfaces müssen alle Introductions und Refinements der umzubenennenden Klassen und Interfaces einbezogen werden.

Argumente:

Klasse `C`, neuer Name `n`

Vorbedingungen:

1. Die Klasse `C` darf keine Klassen mit neuem Namen `n` beinhalten.
2. In einem Feature darf keine Klasse mit neuem Namen `n` existieren.
3. Die beiden Vorbedingungen müssen für alle Features gelten, die die Klasse `C` implementieren.

Die Vorbedingungen stellen sicher, dass die umzubenennende Klasse in keinem Namenskonflikt zu einer bereits existierenden Klasse innerhalb eines Features steht. Dazu betrachte ich nicht nur das Feature der umzubenennenden Klasse, sondern neben den Introductions auch alle Refinements der Klasse in allen Features. Sobald die Prüfung der Vorbedingungen in einem Feature fehlschlägt, kann das Refactoring nicht durchgeführt werden.

Nachbedingungen:

1. Kein Feature darf zwei Klassen mit neuem Namen `n` enthalten.
2. Kein Feature enthält eine Klasse `C` mit dem alten Namen.
3. Alle Referenzen müssen auf die umbenannten Klassen zeigen.
4. Es dürfen keine Referenzen auf die Klassen mit dem alten Namen existieren.

Die Bewahrung des Verhaltens aller Varianten nach dem Refactoring wird durch die Nachbedingungen gewährleistet. Da es neben Introductions auch Refinements für die umbenannte Klasse geben kann, müssen unter Umständen mehrere Klassen umbenannt werden. Dazu dürfen in einem Feature keine zwei Klassen mit dem gleichen Namen vorhanden sein. Auch darf keine Klasse mit dem alten Namen in den Features existieren. Des Weiteren müssen alle Referenzen der umbenannten Klasse in allen betroffenen Features aktualisiert sein.

Vorgehensweise:

1. Ermitteln aller einführenden und verfeinernden Klassen der Klasse `C`
2. Suchen aller Referenzen (u.a. Importe, Instanziierungen) auf allen gefundenen Klassen
3. Erstellen aller Klassen mit neuen Namen `n` in den betroffenen Features
4. Kopieren des gesamten Inhalts der Klassen `C` in die neu erzeugten Klassen
5. Aktualisieren aller Referenzen
6. Entfernen aller alten Klassen `C`

Codebeispiel:

Das Codebeispiel in Listing 4.2.1 zeigt die Umbenennung der Klasse `Graph` in `SimpleGraph`. In dem Codebeispiel sind zwei Features *GraphLibrary* und *Number* zu erkennen. Beide Features implementieren die umzubenennende Klasse `Graph`. Auf der linken Seite wird die Klasse vor und auf der rechten Seite nach dem Refactoring darstellt. Die entsprechenden Codestellen sind zudem unterschiedlich farblich hervorgehoben. Wie dargestellt, müssen in der Klasse `Graph` des Features *GraphLibrary* zwei Codestellen umbenannt werden. Dabei handelt es sich zum einen um den Klassennamen und zum anderen um die Instanziierung der Klasse. In der verfeinernden Klasse `Graph` des Features *Number* müssen die Änderungen nur am Klassennamen vorgenommen werden.

```
class Graph{                    GraphLibrary
  void run(Vertex v){
    // Execution Base Graph
  }
  static void main(String[] a){
    Graph g = new Graph();
    g.run(getVertex(a));
  }
}
```

⟹

```
class SimpleGraph{              GraphLibrary
  void run(Vertex v){
    // Execution Base Graph
  }
  static void main(String[] a){
    SimpleGraph g =
        new SimpleGraph();
    g.run(getVertex(a));
  }
}
```

```
class Graph{                    Number
  void run(Vertex v){
    numberVertex(v);
    original(v);
  }
  void numberVertex(Vertex v){
    // Number Vertex
  }
}
```

⟹

```
class SimpleGraph{              Number
  void run(Vertex v){
    numberVertex(v);
    original(v);
  }
  void numberVertex(Vertex v){
    // Number Vertex
  }
}
```

Listing 4.2.1: Codebeispiel für das variantenerhaltende Refactoring der Umbenennung einer Klasse

4.2.2 Methoden

Dieser Abschnitt beschreibt die Umbenennung des Methodennamens, aller dazugehörigen Methoden in Unter- und Superklassen sowie deren Refinements. Einen Sonderfall sowohl in OOP und FOP stellen statische Methoden dar, welche im Gegensatz zu Instanzmethoden in Unterklassen nicht überschrieben werden können. Dies bedeutet, dass bei einer Umbenennung nur die Klasse, in der die Methode definiert ist, und alle Refinements dieser Methode berücksichtigt werden. Eine Namensänderung gleicher Methoden in Unter- und Superklasse wird nicht durchgeführt.

Argumente:

Klasse `C`, Methode `M`, neuer Name `n`

Vorbedingungen:

1. Eine Methode `M` mit dem neuen Namen `n` und gleicher Signatur darf in Klasse `C` nicht vorhanden sein.
2. Wenn eine Methode mit dem neuen Namen `n` in Unterklassen existiert und es sich bei der umbenennenden Methode `M` um keine private Methode handelt, darf die Sichtbarkeit in den Unterklassen nicht eingeschränkt werden.
3. Wenn eine nicht-lokale Methode mit dem neuen Namen `n` in einer Superklasse existiert, darf die Sichtbarkeit in Klasse `C` nicht eingeschränkt werden.
4. Die Bedingungen eins und zwei gelten für alle Features, die die Methode `m` verfeinern.

Mit der ersten Vorbedingung soll sichergestellt werden, dass eine Methode mit dem neuen Namen in der eigenen Klasse noch nicht existiert, um die Eindeutigkeit der Namen von Codeelementen zu gewährleisten. Es darf immer nur eine Methode mit demselben Namen und derselben Signatur pro Klasse inklusive aller vererbten Methoden vorhanden sein. Sollte eine Methode mit demselben neuen Namen in Unterklassen schon vorhanden und die umzubenennenden Methode nicht nur lokal sichtbar sein, dann muss die Methode in der Unterklasse die gleiche oder eine größere Sichtbarkeit haben. Dies bedeutet, wenn die umzubenennende Methode `protected` ist, dann darf die Methode in Unterklassen nur `protected` oder `public` sein. Da in FOP Refinements von Methoden möglich sind, müssen diese Verfeinerungen zusätzlich betrachtet werden. Daraus folgt, dass die Prüfung der Vorbedingungen nicht nur in einem Feature, sondern in allen Features, die umzubenennende Methode verfeinert. Somit müssen die Vorbedingungen eins bis drei zusätzlich zu der originalen Methode auch für alle verfeinernden Methoden geprüft werden.

Nachbedingungen:

1. In keinem Feature, das die Klasse `C` einführt und verfeinert, dürfen zwei Methoden mit neuem Namen `n` und gleicher Signatur enthalten sein.
2. In keinem Feature, das die Klasse `C` einführt und verfeinert, darf die Methode `M` mit dem alten Namen vorhanden sein.

3. Alle Referenzen müssen auf die umbenannten Methoden zeigen.
4. Keine Referenzen dürfen auf Methoden mit dem alten Namen existieren.

Vorgehensweise:

1. Suchen aller Referenzen auf Methode `M` und die verfeinernden Methoden
2. Erstellen neuer Methoden (originale und verfeinerte) mit neuem Namen `n`
3. Kopieren des gesamten Inhalts der entsprechenden Methoden in die neu erzeugten Methoden
4. Aktualisieren der Referenzen aller geänderten Methoden
5. Entfernen der alten originalen und verfeinernden Methoden

Codebeispiel:

Das Listing 4.2.2 zeigt die Umbenennung der Methode `run` aus Klasse `Graph` in die Methode `execute`. Dazu werden zuerst alle Vorkommen der Methode `run` gesucht. Wie dem Code zu entnehmen, existiert neben der umzubenennenden Methode auch ein Aufruf diese Methode in der `main`-Methode. Des Weiteren ist eine Verfeinerung in dem Feature *Number* vorhanden. Im gezeigten Codebeispiel müssen somit alle drei gefundenen Codestellen umbenannt werden. Angenommen, ich würde die Methode `run` in die bereits vorhandene Methode `numberVertex` aus dem Feature *Number* umbenennen wollen, dann wären die Vorbedingungen eins und vier verletzt. In diesem Fall dürfte das Refactoring nicht durchgeführt werden.

```
class Graph{                    GraphLibrary
  void run(Vertex v){
    // Execution Base Graph
  }
  static void main(String[] a){
    Graph g = new Graph();
    g.run(getVertex(a));
  }
}
```

⟹

```
class Graph{                    GraphLibrary
  void execute(Vertex v){
    // Execution Base Graph
  }
  static void main(String[] a){
    Graph g = new Graph();
    g.execute(getVertex(a));
  }
}
```

```
class Graph{                              Number
  void run(Vertex v){
    numberVertex(v);
    original(v);
  }
  void numberVertex(Vertex v){
    // Number Vertex
  }
}
```

⟹

```
class Graph{                              Number
  void execute(Vertex v){
    numberVertex(v);
    original(v);
  }
  void numberVertex(Vertex v){
    // Number Vertex
  }
}
```

Listing 4.2.2: Codebeispiel für das variantenerhaltende Refactoring der Umbenennung einer Methode

4.2.3 Felder

Dieser Abschnitt beschreibt die Umbenennung eines Feldes. Dabei ist die Einschränkung zu beachten, dass Felder in Unterklassen nicht überschrieben werden. Somit müssen bei einer Umbenennung eines Feldes die Unter- und Superklassen nicht in das Refactoring einbezogen werden.

Argumente:

Klasse `C`, Feld `F`, neuer Name `n`

Vorbedingungen:

1. Ein Feld `F` mit dem neuen Namen `n` darf in Klasse `C` nicht vorhanden sein.
2. Die Bedingung eins gilt für alle Features, die die Klasse `C` implementieren.

Wenn das Umbenennen eines Feldes durchgeführt werden soll, darf es in der Klasse, welches das umzubenennende Feld beinhaltet, kein Feld mit dem neuen Namen geben. Diese Einschränkung reicht für FOP nicht aus, da damit nur eine Variante abgedeckt ist. Um alle Varianten zu berücksichtigten, müssen diese Einschränkungen auf alle zu dem Feld zugehörigen, verfeinernden Klassen anwendet werden.

Nachbedingungen:

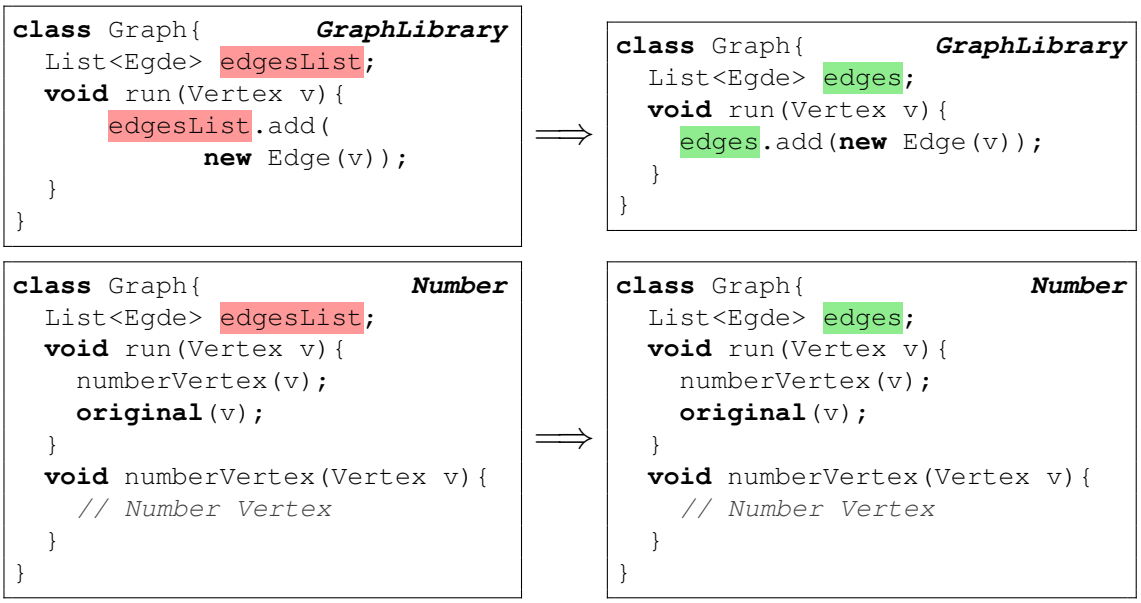
1. Innerhalb der Klasse `C` dürfen keine zwei Felder mit gleichem Namen `n` vorhanden sein.
2. In den verfeinernden Klassen von `C` darf kein Feld `F` mit dem alten Namen existieren.
3. Alle Referenzen müssen auf die neuen Felder aktualisiert sein.
4. Es dürfen keine Referenzen auf Feld `F` mit dem alten Namen existieren.

Vorgehensweise:

1. Ermitteln aller Referenzen auf Feld `F` in allen Features, die die Klasse `C` einschließen
2. Erstellen neuer Felder mit neuem Namen `n`
3. Kopieren der Deklaration des Feldes `F` in neu erzeugte Felder
4. Aktualisieren der Referenzen aller geänderten Felder
5. Entfernen aller alten Felder

Codebeispiel:

Dieses Codebeispiel 4.2.3 zeigt die Umbenennung des Feldes `edges` der Klasse `Graph`.



Listing 4.2.3: Codebeispiel für das variantenerhaltende Refactoring der Umbenennung eines Feldes

4.2.4 Lokale Variablen und Parameter in Methoden

In diesem Abschnitt beschreibe ich die Umbenennung einer lokalen Variable oder eines Parameters innerhalb einer Methode. Aufgrund lokaler Sichtbarkeit werden keine Erweiterungen bezüglich der Feature-Orientierung benötigt. Somit können die Vor- und Nachbedingungen des OOPs verwenden werden. Da keine Unterschiede in der Umbenennung von lokalen Variablen und Parametern bestehen, werden im Folgenden zur Vereinfachung beide Codeelemente als Variable bezeichnet.

Argumente:

Methode `M`, Variable `V`, neuer Name `n`

Vorbedingungen:

1. Eine Variable mit dem neuen Namen `n` darf in Methode `M` nicht vorhanden sein.
2. Eine Variable darf eine Feld-Referenz nicht überschreiben.

Nachbedingungen:

1. In Methode `M` dürfen keine zwei Variablen mit gleichem Namen `n` vorhanden sein.
2. Die umbenannte Variable darf keine Feld-Referenz überdecken.
3. Alle Referenzen sind aktualisiert.
4. Es dürfen keine Referenzen auf Variable `V` mit dem alten Namen existieren.

Vorgehensweise:

1. Ermitteln aller Zugriffe auf Variable `V`
2. Erstellen neuer Variablen mit neuem Namen `n`
3. Kopieren des Variableninhalts `F` in neu erzeugte Variable
4. Aktualisieren der Referenzen der geänderten Variable
5. Entfernen der alten Variable

Codebeispiel:

Der Einfachheit wegen verzichte ich an dieser Stelle auf ein Codebeispiel.

4.2.5 Lokale Variablen und Parameter in Konstruktoren

Als letztes Refactoring aus der Gruppe `Rename` stelle ich den Sonderfall Konstruktoren im Zusammenspiel mit lokalen Variablen und Parametern für FOP vor. Aufgrund der Art und Weise wie FeatureHouse Konstruktoren-Verfeinerungen implementiert werden, müssen Vor- und Nachbedingungen angemessen erweitert werden. In Abbildung 2.5 wird die Komposition zweier Konstruktoren dargestellt. Dabei lässt sich erkennen, dass der entstandene Konstruktor zuerst den Code des originalen Konstruktors und danach den Code der verfeinernden Konstruktoren in Reihenfolge der Komposition des Features enthält. Dies bedeutet, dass bei einer Umbenennung eines Parameters oder einer lokalen Variable in einem Konstruktor alle beteiligten originalen und alle verfeinernden Konstruktoren auf Namenskollisionen überprüft werden müssen. Nur dadurch kann sichergestellt werden, dass alle Varianten nach einer Umbenennung innerhalb eines Konstruktors korrekt funktionieren. Wie in dem vorherigen Abschnitt werden beide Codeelemente als Variablen bezeichnet.

Argumente:

Klasse `C`, Konstruktor `K`, Variable `V`, neuer Name `n`

Vorbedingungen:

1. Eine Variable mit dem neuen Namen `n` darf in Konstruktor `K` der Klasse `C` nicht vorhanden sein.
2. Eine Variable darf eine Feld-Referenz nicht überschreiben.
3. Die Bedingung eins gilt für alle Features, die die Klasse `C` implementieren.

Nachbedingungen:

1. In Konstruktor `K` dürfen keine zwei Variablen mit gleichen Name `n` vorhanden sein.
2. Die umbenannte Variable darf keine Feld-Referenz überdecken.

3. In keinem Feature darf eine Variable `V` innerhalb des Konstruktors `K` mit dem gleichen Namen `n` existieren.

4. In dem originalen und den verfeinernden Konstruktoren von `K` darf keine Variable `V` mehr mit dem alten Namen existieren.

5. Alle Referenzen sind aktualisiert.

Vorgehensweise:

1. Ermitteln aller Zugriffe auf Variable `V`

2. Erstellen neuer Variablen mit neuem Namen `n`

3. Kopieren des Variableninhalts `V` in neu erzeugte Variable

4. Aktualisieren der Referenzen der geänderten Variable

5. Entfernen der alten Variable

4.3 Extraktion von Codeklonen

Nachdem im vorherigen Abschnitt der vorbereitende Migrationsschritt in Form der Angleichung von Codeklonen erläutert wurde, werde ich in diesem Abschnitt auf den Migrationsschritt der Extraktion von Codeklonen eingehen.

Den Voraussetzungen ist zu entnehmen, dass die Extraktion von Codeklonen nur auf Klone vom Typ-I anwendbar ist. Mit der Extraktion von Codeklonen soll sichergestellt werden, dass nach Konsolidierung der Codeklone keine identischen Codefragmente zwischen verschiedenen Produktvarianten mehr vorkommen. Die Extraktion eines Codeklons aus verschiedenen Produktvarianten wird schrittweise durchgeführt. Dabei wird Codefragment eines Codeklons in ein gemeinsames Feature aller von diesem Codeklon betreffenden Produktvarianten verschoben. Die Produktvarianten liegen wie in Abschnitt 3.3.2 beschrieben in einzelnen Features vor. Anhand des Feature-Modells und der betroffenen Features wird das gemeinsame Feature ermittelt. Sollte das gemeinsame Feature nicht existieren, wird es im Feature-Modell neu angelegt. Zusätzlich werden für das gemeinsame Feature und die dazugehörigen Features Constraints erstellt. In Abbildung 4.1 wird das Anpassen des Feature-Modells durch Hinzufügen des Features *Common* bildhaft dargestellt. Die beiden Features *Undirected* und *Number* haben identische Funktionalitäten, die in ein gemeinsames Feature extrahiert werden sollen. Dafür wird im Feature-Modell ein konkretes, optionales Feature *Common* und ein Constraint angelegt. Der Constraint sagt aus, dass bei einer Auswahl des Features *Undirected* oder *Number* ebenfalls Feature *Common* ausgewählt werden muss. Dies gilt auch in die entgegengesetzte Richtung, sodass bei der Auswahl von *Common* eines der beiden Features ausgewählt werden muss. Der Constraint stellt sicher, dass die Produktvarianten ohne Fehler kompiliert werden können. Zusätzlich zu den Änderungen am Feature-Modell müssen nach einer Neuanlage eines Features die Konfigurationen der betroffenen Produktvarianten angepasst werden, um das Verhalten der Varianten weiter zu gewährleisten.

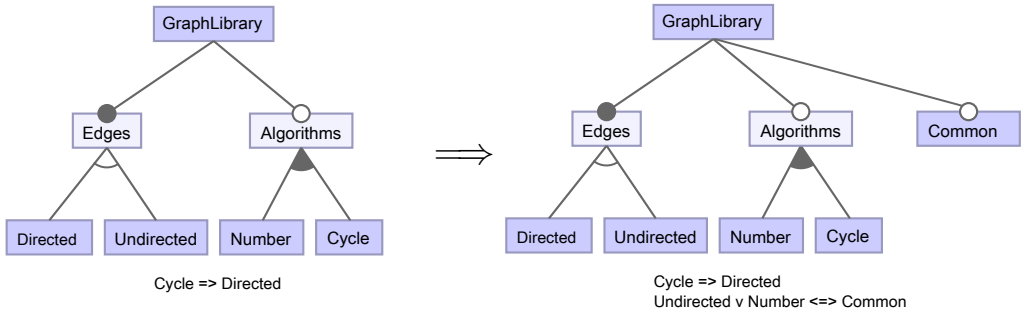


Abbildung 4.1: Extraktion eines gemeinsamen Codes in ein gemeinsames Feature

Im darauffolgenden Schritt werden alle Vorkommen des Codeklons aus allen betreffenden Produktvarianten extrahiert. Diese beiden Schritte können ohne Unterstützung eines Entwicklers mit Hilfe von Refactorings vollautomatisch durchgeführt werden. Zum Extrahieren von Codeklonen bietet mein Konzept das umzusetzende `Pull-Up-to-Parent-Feature`-Refactoring an.

Pull Up to Parent Feature-Refactoring

Das `Pull-Up-to-Parent-Feature`-Refactoring beruht auf dem `Pull Up`-Refactoring aus dem Bereich OOP. Beim `Pull Up`-Refactoring handelt es sich um einen Spezialfall vom `Move`-Refactoring mit der Einschränkung, dass Codeelemente (Methoden und Felder) innerhalb der Vererbungshierarchie nur in die Superklasse verschoben werden können. Mit diesem Refactoring wird das Ziel verfolgt, identische Codeelemente in einer Superklasse zusammenzufassen. Dadurch können Fehleranfälligkeit und Wartungsaufwand gesenkt werden. Im Gegensatz zum OOP-Refactoring wird das FOP-Refactoring nicht auf die Vererbungshierarchie von Klassen sondern auf die Feature-Hierarchie angewendet. Dies bedeutet, dass Codeelemente anstelle der Verschiebung in eine Superklasse in ein übergeordnetes Feature der gleichen Klasse verschoben werden. Da sich die Quell- und Zielklasse in diesem Refactoring nicht ändert und die Verschiebung auf Feature-Ebene stattfindet, können zusätzlich zu den beiden genannten Codeelementen auch ganze Klassen verschoben werden. Wie im originalen `Pull Up`-Refactoring kann die Verschiebung der Codeelemente aber nur in Richtung Generalisierung und nicht in Richtung Spezialisierung durchgeführt werden.

Aktuell existiert für das `Pull Up`-Refactoring von Methoden und Felder eine FOP-Implementierung [Brunswig, 2013; Schulze et al., 2013]. Mit diesem variantenerhaltenden Refactoring können beide Codeelemente in ein übergeordnetes Feature oder in eine generalisierende Klasse verschoben werden. Dieses Refactoring unterstützt keine Codeklonerkennung, die für das Ziel meines umzusetzenden Refactorings, Konsolidierung von Codeklonen, erforderlich ist. Da Anpassungen am vorhandenen `Pull Up`-Refactoring zu aufwendig sind, habe ich mich entschieden, für die Migration ein eigenes `Pull-Up-to-Parent-Feature`-Refactoring zu entwickeln. Mit dem `Pull-Up-to-Parent-Feature`-Refactoring wird die Möglichkeit bereitgestellt, zwei oder mehr identische Codeelemente derselben Klasse aus unterschiedlichen Features in die gleiche Klasse des übergeordneten Features zu verschieben. Dies ist notwendig, da bei der Migration vorhandene Codeklone verschiedener Features der

gleichen Klasse automatisch mit in das übergeordnete Feature verschoben werden können. Dafür habe ich die Codeklonerkennung direkt in das Refactoring integriert. Die Ausführung des Refactorings erfolgt immer auf eine bestimmte Klasse. Dazu werden bei jeder Ausführung des Refactorings die Codeklone innerhalb der Klasse ermittelt und für den Extraktionsschritt aufbereitet. Dadurch muss im Gegensatz zum `Rename`-Refactoring keine separate Erkennung ausgeführt werden. Durch die Kombination des `Pull-Up-to-Parent-Feature`-Refactorings mit der Codeklonerkennung ist das Refactoring in der Praxis deutlich einfacher einsetzbar.

Für das `Pull-Up-to-Parent-Feature`-Refactoring muss nur eine Dimension (siehe Abbildung 3.1) betrachtet werden. Da Codeelemente nur in der Feature-Hierarchie verschoben werden und die Klassenzugehörigkeit nicht geändert wird, hat das Refactorings nur Auswirkungen auf Dimension D3. Des Weiteren besteht keine Notwendigkeiten die Referenzen der zu verschiebenden Codeelemente zu aktualisieren.

Das `Pull-Up-to-Parent-Feature`-Refactoring unterstützt folgende Codeelemente:

- Klassen und Interfaces
- Methoden
- Felder

Im nachfolgenden Abschnitt werden nacheinander die genannten `Pull-Up-to-Parent-Feature`-Refactorings eingeführt. Wie beim `Rename`-Refactoring stelle ich dazu Argumente, Vor- und Nachbedingungen sowie die Vorgehensweise eines jeden `Pull-Up-to-Parent-Feature`-Refactorings vor und veranschauliche diese anhand eines Codebeispiels.

4.3.1 Methoden

Die erste Möglichkeit ist, das `Pull-Up-to-Parent-Feature`-Refactoring auf Methoden anzuwenden. Dabei werden zwei oder mehr identische Methoden derselben Klasse aus unterschiedlichen Features in die gleiche Klasse des übergeordneten Features verschoben. Sollte die Klasse im Zielfeature nicht vorhanden sein, wird sie im Rahmen des Refactorings erzeugt. Dabei spielt es keine Rolle, ob die Methoden in Super- und Subklassen überladen oder überschrieben sind. Sollte es sich um eine verfeinernde Methode handeln, ist Sorgsamkeit geboten, da sich durch Verschiebung der Methode die Feature-Komposition ändern kann. Dies kann zur Folge haben, dass der Feature-Code für bestimmte Konfigurationen nicht kompilierbar ist.

Argumente:

Quellfeature `Q`, Zielfeature `Z`, Klasse `C`, Methode `M`

Vorbedingungen:

1. Das Zielfeature `Z` muss existieren.

2. Das Zielfeature `Z` muss konkret sein. Das heißt, Z darf kein abstraktes Feature sein.
3. Das Zielfeature `Z` muss ein übergeordnetes Feature des Quellfeatures `Q` sein.
4. Wenn die Klasse `C` in Zielfeature `Z` schon existiert, darf in dieser Klasse die Methode `M` nicht enthalten sein.

Mit den Vorbedingungen wird zum einen sichergestellt, dass das Zielfeature ein konkretes existierendes Feature und ein übergeordnetes Feature des Quellfeatures ist. Dies ist notwendig, damit die Methode nur in der Feature-Hierarchie nach oben verschoben werden kann. Zum anderen darf die Methode in der Klasse des Zielfeatures nicht existieren. Sollte die Klasse im Zielfeature nicht existieren, wird die Klasse neu erzeugt.

Nachbedingungen:

1. Das Zielfeature `Z` muss die Klasse `C` enthalten.
2. Das Zielfeature `Z` darf keine zwei Klassen mit dem gleichen Namen der Klasse `C` enthalten.
3. Die Klasse `C` im Zielfeature `Z` muss Methode `M` enthalten.
4. Die Klasse `C` im Zielfeature `Z` darf keine zwei Methoden `M` mit der gleichen Signatur enthalten.
5. In der Klasse `C` des Quellfeatures `Q` darf die verschobene Methode `M` nicht mehr existieren.

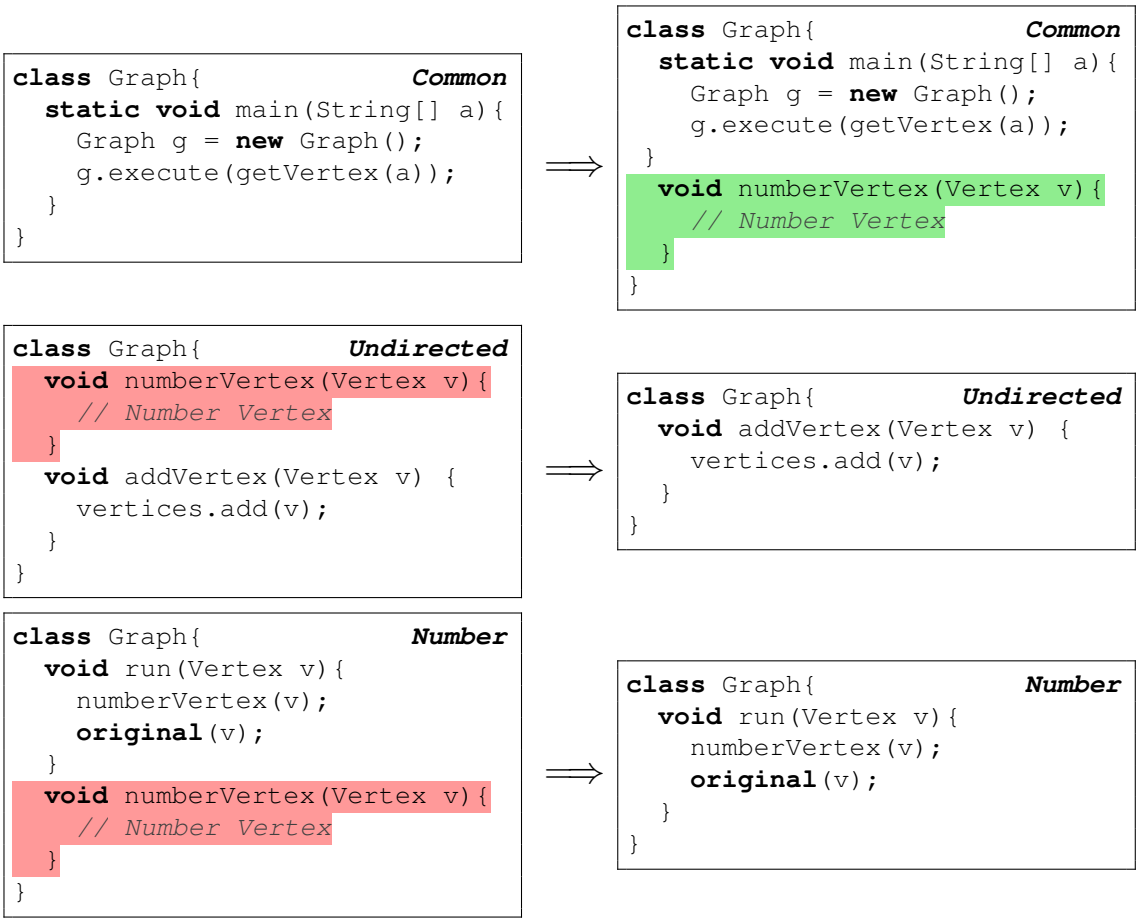
Die Nachbedingungen des variantenerhaltenden Refactorings sollen sicherstellen, dass das Verhalten aller Varianten nach dem Refactoring bewahrt wird. Dazu muss im Zielfeature die verschobene Klasse enthalten sein. Diese Methode darf in dieser Klasse aber nicht doppelt vorhanden sein. Des Weiteren darf die Methode in der Klasse des Quellfeatures nicht mehr existieren.

Vorgehensweise:

1. Ermitteln des gemeinsamen übergeordneten Zielfeatures `Z`
2. Erzeugen eines neuen Zielfeatures `Z` und des Constraints im Feature-Modell, wenn `Z` noch nicht existiert
3. Anpassen der Konfigurationen der beteiligten Features
4. Erzeugen der Klasse `C` in Zielfeature `Z`, wenn sie noch nicht existiert
5. Erstellen einer neuen Methode `M` in Klasse `C` des Zielfeatures `Z`
6. Kopieren des gesamten Inhalts der alten Methode in die neu erzeugte Methode
7. Entfernen der alten Methode `M` aus der Klasse `C` des Quellfeatures `Q`

Codebeispiel:

Das Codebeispiel 4.3.1 zeigt das Verschieben der geklonten Methode `numberVertex` aus der Klasse `Graph` der beiden Features *Undirected* und *Number* in die gleiche Klasse des übergeordneten Features *Common*. Wie zu erkennen ist, existiert die Methode `numberVertex` im Feature *Common* noch nicht und kann aus diesem Grund verschoben werden. Nach Verschiebung der Methode existiert sie nur im Feature *Common*. Aus den Features *Undirected* und *Number* wurde die Methode entfernt.



Listing 4.3.1: Codebeispiel für das variantenerhaltende Refactoring der Verschiebung einer Methode in ein übergeordnetes Feature

4.3.2 Felder

Das `Pull-Up-to-Parent-Feature`-Refactoring von Feldern ist ähnlich dem Refactoring von Methoden. Aus diesem Grund verzichte ich auf die Erklärungen zu den Vor- und Nachbedingungen und auf das Codebeispiel am Ende des Abschnitts.

Argumente:

Quellfeature `Q`, Zielfeature `Z`, Klasse `C`, Felder `F`

Vorbedingungen:

1. Das Zielfeature `Z` muss existieren.
2. Das Zielfeature `Z` muss konkret sein. Das heißt, Z darf kein abstraktes Feature sein.
3. Das Zielfeature `Z` muss ein übergeordnetes Feature des Quellfeatures `Q` sein.
4. Wenn die Klasse `C` in Zielfeature `Z` schon existiert, darf in dieser Klasse das Feld `F` nicht vorhanden sein.

Nachbedingungen:

1. Das Zielfeature darf keine zwei Klassen mit dem gleichen Namen enthalten.
2. Das Zielfeature muss die Klasse `C` enthalten.
3. Die Klasse `C` im Zielfeature `Z` darf keine zwei Felder `F` mit der gleichen Signatur enthalten.
4. Die Klasse `C` im Zielfeature `Z` muss das Feld `F` enthalten.
5. In der Klasse `C` des Quellfeatures `Q` darf die verschobene Methode `F` nicht mehr existieren.

Vorgehensweise:

1. Ermitteln des gemeinsamen übergeordneten Zielfeatures `Z`
2. Erzeugen eines neuen Zielfeatures `Z` und des Constraints im Feature-Modell, wenn `Z` noch nicht existiert
3. Anpassen der Konfigurationen der beteiligten Features
4. Erzeugen der Klasse `C` in Zielfeature `Q`, wenn sie noch nicht existiert
5. Erstellen eines neuen Feldes `F` in Klasse `C` des Zielfeatures `Z`
6. Kopieren der Deklaration des Feldes `F` ins neue Feld
7. Entfernen des alten Feldes `F` aus der Klasse `C` des Quellfeatures `Q`

4.3.3 Klassen und Interfaces

Das `Pull-Up-to-Parent-Feature`-Refactoring verschiebt komplette Klassen und Interfaces aus einem Feature in ein übergeordnetes Feature.

Argumente:

Quellfeature `Q`, Zielfeature `Z`, Klasse `C`

Vorbedingungen:

1. Das Zielfeature `Z` muss existieren.
2. Das Zielfeature `Z` muss konkret sein. Das heißt, Z darf kein abstraktes Feature sein.
3. Das Zielfeature `Z` muss ein übergeordnetes Feature des Quellfeatures `Q` sein.
4. Die Klasse `C` darf in Zielfeature `Z` nicht existieren.

Die ersten drei Vorbedingungen sind analog zu den Vorbedingungen der Methoden oder Felder. Die vierte Vorbedingung sagt aus, dass die Klasse nur dann verschoben werden darf, wenn sie in keinem Konflikt zu einer bereits existierenden Klasse innerhalb eines Zielfeatures steht.

Nachbedingungen:

1. Das Zielfeature `Z` muss die verschobene Klasse `C` enthalten.
2. Das Zielfeature `Z` darf keine zwei Klassen mit dem gleichen Namen der Klasse `C` enthalten.
3. Im Quellfeature `Q` darf die verschobene Klasse `C` nicht mehr existieren.

Nach dem Refactoring muss im Zielfeature die verschobene Klasse enthalten sein. Ebenso darf die Klasse in dem Quellfeature nicht mehr vorhanden sein.

Vorgehensweise:

1. Ermitteln des gemeinsamen übergeordneten Zielfeatures `Z`
2. Erzeugen eines neuen Zielfeatures `Z` und des Constraints im Feature-Modell, wenn `Z` noch nicht existiert
3. Anpassen der Konfigurationen der beteiligten Features
4. Erstellen der Klasse `C` im Zielfeature `F`
5. Kopieren des gesamten Inhalts der Klasse `C` in die neu erzeugte Klasse
6. Entfernen der alten Klasse `C` im Quellfeature `Q`

Codebeispiel:

Ich verzichte an dieser Stelle auf ein Codebeispiel, da dies analog zum ersten Beispiel des `Pull-Up-to-Parent-Feature`-Refactorings ist.

4.4 Zusammenfassung

Dieses Kapitel erläutert ausführlich die beiden Migrationsschritte Angleichung und Extraktion von Codeklonen. Einleitend wurden die Voraussetzungen für die Anwendung dieser beiden Migrationsschritte herausgearbeitet. In den anschließenden Abschnitten wurde für jeden Migrationsschritt das dazugehörige variantenerhaltende Refactoring vorgestellt. Dabei wurde detailliert auf die Vorgehensweise, Vor- und Nachbedingungen eingegangen und diese anhand eines Beispiels gezeigt.

5. Implementierung

In den vorhergehenden Kapiteln wurde ein Konzept zur Migration unter Verwendung variantenerhaltender Refactorings für Feature-orientierte Programmierung vorgestellt. Das Konzept beinhaltet die beiden Migrationsprozesse Angleichung und Beseitigung von Codeklonen. Dieses Kapitel beschreibt die Implementierung anhand eines Refactoring-Prozesses mithilfe des Eclipse Refactoring Frameworks. Dazu werden benötigte Komponenten erklärt, die einen Einfluss auf die Implementierung der Refactorings haben. Mittels eines Beispiels wird die Anwendung der implementierten Refactorings gezeigt.

Sollten meine Refactorings auf einem Element (z.B. Klasse) mit globaler Sichtbarkeit durchgeführt werden, müssen alle Referenzen zu diesem Element ebenfalls geändert werden. Diese Referenzen manuell zu suchen und zu aktualisieren, ist in den meisten Fällen zeitaufwendig und fehleranfällig [Opdyke, 1992]. Fowler [1999] forderte deshalb die Entwicklung von Werkzeugen zur Unterstützung automatisierter Refactoring-Prozesse. Roberts [1999] bekräftigte ebenfalls die Notwendigkeit solcher Tools. In den letzten Jahren wurde die Forderung kontinuierlich in integrierten Entwicklungsumgebungen (IDEs) für verschiedene Sprachen (z.B. C++, C#, Java oder Smalltalk) umgesetzt. Für die Sprachen Java und C++ steht die Eclipse-IDE zur Verfügung, für C# MonoDevelop und für Smalltalk Squeak. Mit diesem IDEs ist es möglich, automatische Refactorings durchzuführen. Leider besteht im Bereich Tool-Unterstützung für das Refactoring von FOP-Code großer Nachholbedarf. Meine Arbeit liefert einen Beitrag zur Automatisierung ausgewählter Refactorings durch Tool-Unterstützung.

5.1 Umsetzung der Refactorings

Die Umsetzung der vorgestellten Refactorings erfolgt mit Unterstützung von Eclipse[1]. Eclipse ist eine integrierte Entwicklungsumgebung, die durch eine Vielzahl von Plugin-Erweiterungen bekannt ist. Zur Umsetzung der Refactorings werden weitere Komponenten benötigt. Diese Komponenten werden nachfolgend kurz vorgestellt.

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

Fuji

Die vorhandenen OO-Refactorings können auf einzelne Produktvarianten durchgeführt werden. Diese Refactorings stellen, aber immer nur das Verhalten einer Produktvariante sicher. Für das Refactoring der gesamten SPL ist dies nicht ausreichend, weil nicht das Verhalten aller Varianten bewahrt werden muss. Dadurch wird ein spezieller Compiler benötigt, der die Variabilität innerhalb der SPL berücksichtigt. Fuji ist ein erweiterter Java Compiler mit Unterstützung für feature-orientierte Programmierung [Kolesnikov, 2011]. Dieser erweiterte Compiler ist kompatibel zu FeatureHouse, in welchem die Refactorings entwickelt wurden. Fuji wird zur Erzeugung eines Abstract Syntax Tree (AST) über die gesamte SPL verwendet.

Ein AST beschreibt den Inhalt des gesamten Programms in Form einer Baumstruktur. Er dient zur Analyse, Modifikation und Übersetzung des Quellcodes. Der erzeugte Fuji-AST beinhaltet somit alle Informationen aller Produktvarianten, die zur Durchführung der Refactorings benötigt werden. Auf Grundlage dieses ASTs können für jedes Refactoring konkrete Vor- und Nachbedingungen geprüft werden. Für die Vor- bzw. Nachbedingungen der Refactorings ist es erforderlich, die Klassen aller konkreten Features zu betrachten, um zu prüfen, ob das Verhalten nach dem Refactoring bewahrt bleibt. Sollten nur Features bestimmter Produktvarianten berücksichtigt werden, können Änderungen durch Refactorings die Funktionsweise der Produktvarianten beeinträchtigen. Für jede Signatur (d.h. Methode, Klasse, Feld oder lokale Variable) werden aus dem Fuji-AST unter anderem der Name, die Länge, die Refinements und alle Referenzen extrahiert. Da alle relevanten Informationen aus dem Fuji-AST identifiziert werden können, besteht keine Notwendigkeit, zusätzliche Informationen aus einem Java-AST herauszufiltern.

Refactoring-Framework

Um den Aufwand der Entwicklung zu reduzieren, wird ein vorhandenes Framework wiederverwendet, das Eclipse Refactoring Framework. Innerhalb dieses Frameworks wird auf die API Language Tool Kit (LTK) zurückgegriffen. Dieses LTK besteht aus zwei Plugins, zum einen aus Basiselementen (*org.eclipse.ltk.core.refactoring*) und zum anderen aus grafischen Elementen (*org.eclipse.ltk.ui.refactoring*) wie Wizard- oder Eingabedialoge. Dieses Framework unterstützt die Entwicklung von eigenen Refactorings durch einen vorhandenen und funktionierenden Refactoring-Mechanismus. Dadurch konzentriert sich meine Implementierung auf wesentliche Punkte (beispielsweise Signaturen, Vorbedingungen oder Änderungen) bei der Entwicklung von Refactorings. Für jedes zu entwickelnde Refactoring muss die Klasse `Refactoring` aus dem Core-Package des LTK erweitert werden. Durch diese Erweiterung kann Einfluss auf den Refactoring-Prozess des LTKs genommen werden, um benötigte Voraussetzungen, beispielsweise Berücksichtigung der Variabilität, festzulegen. Die Steuerung dieses Refactoring-Prozesses läuft weiterhin im LTK-Framework ab. Die einzelnen Schritte des Refactoring-Prozesses werden im Abschnitt 5.2 ausführlich erläutert.

Zudem müssen Dialoge aufbauend auf den Basisdialogen des UI-Packages implementiert werden. Diese Dialoge erlauben die Visualisierung der validierten Vorbedingungen und die Vorschauen auf mögliche Änderungen. Zusätzlich können in diesen

Dialogen aufgetretene Fehler während des Refactorings angezeigt werden. Um das Refactoring auszuführen, wird eine Action verwendet. Diese Action wird bei meiner Implementierung in ein Kontextmenü eingebettet.

FeatureIDE

Die umgesetzten Refactorings wurden in FeatureIDE[2], einem Eclipse-Plugin, integriert. FeatureIDE ist ein Open-Source-Framework zur Entwicklung von SPLs [Thüm et al., 2014]. FeatureIDE unterstützt verschiedene Implementierungstechniken, wie aspekt-orientierte [Kiczales et al., 1997], delta-orientierte [Schaefer et al., 2010] und feature-orientierte Programmierung [Prehofer, 1997] und Präprozessoren [Kästner, 2010]. Für die Analyse, Modellierung und Konfiguration von Feature-Modellen stehen grafische Elemente wie Konfigurations- und Feature-Modell-Editoren zur Verfügung. Weiterhin bietet FeatureIDE die Option, Warnungen und Fehler innerhalb des Quellcodes und des Modells anzuzeigen. Tonscheidt [2015] schaffte die Möglichkeit, ermittelte Codeklone als Warnung innerhalb des Editors der jeweiligen Klasse in Form von Markierungen darzustellen (siehe Abbildung 5.1). Diese Markierungen erlauben, vorhandene Codeklone zu anderen Klassen zu erkennen und diese zeitnah zu beseitigen. In FeatureIDE wird der FeatureHouse-Ansatz (siehe Abschnitt 2.3) unterstützt. Des Weiteren wird die benötigte Fuji-Bibliothek zur Ermittlung der Variabilität im Zusammenhang mit FeatureHouse schon eingesetzt.

Bisher bietet FeatureIDE noch keine Möglichkeit, automatische Refactorings auf den Feature-Code auszuführen. Mit der prototypischen Umsetzung des `Rename`- und `Pull-Up-to-Parent-Feature`-Refactoring ist der Anfang für weitere Refactorings geschaffen. Die Refactorings können somit nicht nur für den Migrationsprozess verwendet werden, sondern auch in der fortlaufenden Weiterentwicklung von SPLs, um automatisch Änderungen am Feature-Code vorzunehmen. Die implementierten Refactorings sind auf FeatureHouse beschränkt. Die beiden Refactorings können über das Kontextmenü *FeatureHouse-Refactoring* ausgewählt werden. Dieses Kontextmenü ist sichtbar, wenn sich der Nutzer im Editor oder in der Feature-Outline eines FeatureHouse-Projektes befindet.

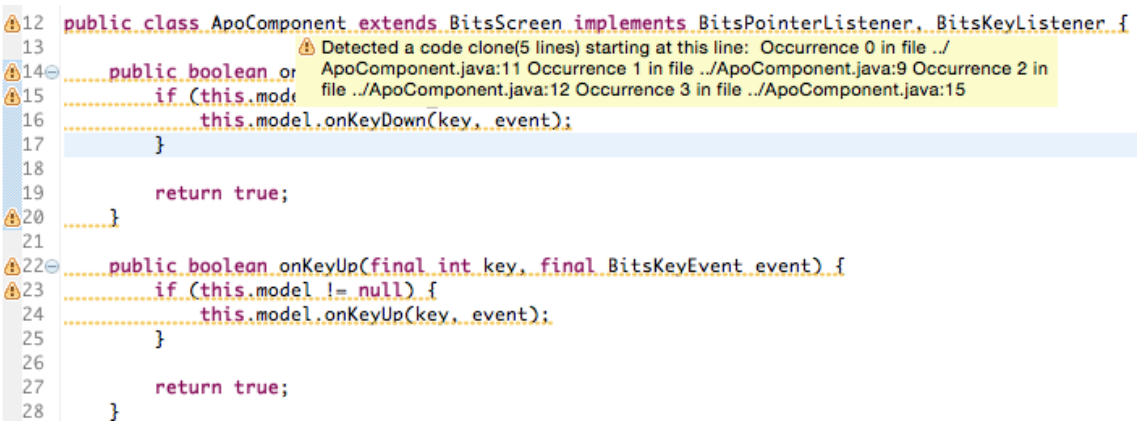


Abbildung 5.1: Codeklone-Warnungen im Quellcode

[2] http://www.fosd.de/fide

5.2 Refactoring-Prozess

Der Refactoring-Prozess wird in Abbildung 5.2 dargestellt. Wie anhand der Abbildung zu erkennen ist, lässt sich der gesamte Prozess in die drei Teilabschnitte Vorbereitungsschritt, Analyseschritt und Änderungsschritt unterteilen. Innerhalb dieser Teilabschnitte werden unter anderem die initialen Bedingungen und die Vorbedingungen überprüft. Leider stellt der Refactoring-Prozess keine Möglichkeit zur Kontrolle der Nachbedingungen zur Verfügung. Aus diesem Grund wird bei Durchführung meiner Refactorings auf Prüfung der Nachbedingungen verzichtet. Im Folgenden werden die einzelnen Teilabschnitte beschrieben.

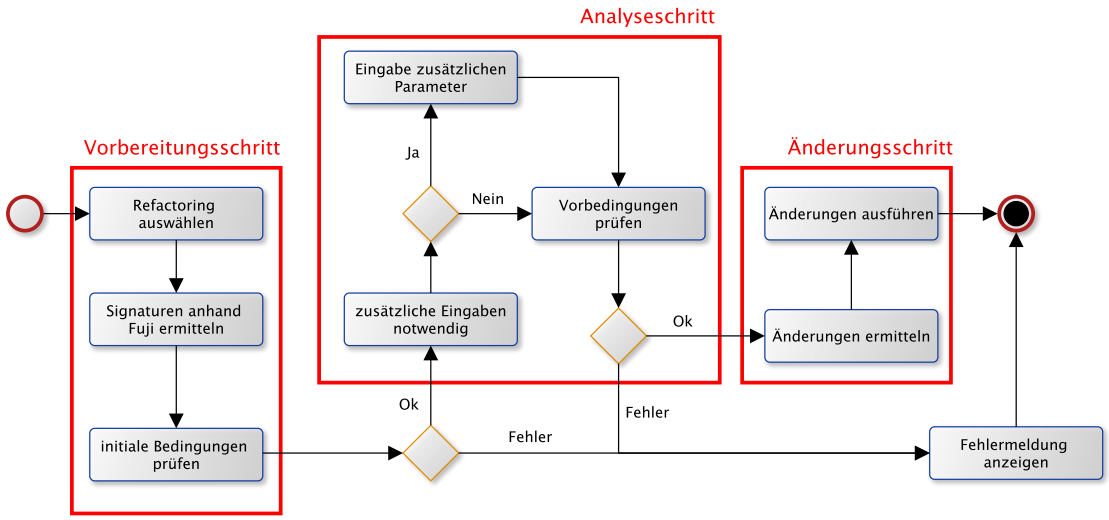


Abbildung 5.2: Refactoring-Prozess

Vorbereitungsschritt

Im ersten Schritt wird für ein Codeelement das gewünschte Refactoring, welches die durchzuführende Aktion bestimmt, ausgewählt. Anschließend wird der Fuji-AST aufgebaut und anhand des ASTs die benötigten Signaturen ermittelt. Auf diesen Signaturen wird während des gesamten Refactoring-Prozesses gearbeitet, sodass der Fuji-AST im weiteren Verlauf des Refactorings nicht mehr benötigt wird. Im letzten Vorbereitungsschritt werden die initialen Bedingungen überprüft. Zu den initialen Bedingungen meiner umgesetzten Refactorings gehören folgende Bedingungen:

- Quelldatei muss beschreibbar sein
- Quelldatei muss Dateiendung .java haben
- keine Compilerfehler dürfen vorhanden sein

Beim Rename-Refactoring wird zusätzlich zu den oben genannten initialen Bedingungen geprüft, ob das ausgewählte Codeelement mit neuem Namen nicht schon in der Klasse vorhanden ist. Sollten eine oder mehrere initiale Bedingungen nicht erfüllt sein, wird das Refactoring mit einer Fehlermeldung abgebrochen.

Analyseschritt

Nachdem die initialen Bedingungen erfüllt wurden, kann mit dem Analyseschritt begonnen werden. Sollten zusätzliche Eingaben notwendig sein, müssen diese vorgenommen werden, damit das Refactoring fortgeführt werden kann. Zu den zusätzlichen Parametern beim Rename-Refactoring zählt der neue Name des zu ändernden Codeelements. An diesen Schritt schließt sich die Überprüfung der Vorbedingungen an. Dabei werden anhand des neuen Namens und der extrahierten Signaturen des Fuji-ASTs die definierten Vorbedingungen überprüft. Wenn diese Vorbedingungen erfüllt sind, ist der Analyseschritt abgeschlossen und es wird der nachfolgende Änderungsschritt ausgeführt.

Änderungsschritt

In diesem Teilabschnitt werden die Änderungen ermittelt, die durch das Refactoring im späteren Schritt ausgeführt werden. Dazu zählen alle Änderungen des Codeelements sowie deren Refinements und alle Referenzen. Für jede Änderung werden die zugrunde liegenden Dateien eingelesen und aus den betroffenen Codestellen ein Änderungsobjekt erstellt. Diese Änderungsobjekte beziehen sich entweder auf das Hinzufügen, Ersetzen, Löschen oder Verschieben von Codeelementen. Nachdem alle Änderungsobjekte erstellt wurden, werden sie vom LTK in die zugrunde liegenden Dateien geschrieben. Die Änderungen an den Codeelementen werden nicht in die extrahierten Signaturen und den Fuji-AST zurückgeschrieben, weil der Aufwand dafür in der aktuellen Implementierung zu groß ist. Zusätzlich zu den Änderungsobjekten müssen für das `Pull-Up-to-Parent-Feature`-Refactoring teilweise weitere Änderungen durchgeführt werden. Dieses Refactoring erfordert, dass eine bestimmte Klasse im Zielfeature vorhanden ist. Sollte diese Klasse in dem Zielfeature nicht existieren, dann muss diese Klasse neu erstellt werden.

Sicherstellen der Korrektheit der Refactorings

Der Refactoring-Prozess ist ein komplexer Prozess, bei dem leicht Änderungen vergessen werden können. Um die Korrektheit des kompletten Refactoring-Prozesses sicherzustellen, habe ich Unit-Tests geschrieben, die alle wesentlichen Refactoring-Testfälle abdecken. Innerhalb dieser Unit-Tests prüfe ich nicht nur die Vor-, sondern auch die Nachbedingungen. Mit diesen Tests soll sichergestellt werden, dass die Refactorings das Verhalten aller Produktvarianten bewahrt.

5.3 Beispiel

Dieser Abschnitt repräsentiert ein Beispiel für die Durchführung eines `Pull-Up-to-Parent-Feature`-Refactorings für die Methode `getVecY`. Zu Starten des Refactorings wählt der Nutzer aus dem Kontektmenü *FeatureHouse-Refactoring* das `Pull-Up-to-Parent-Feature`-Refactorings aus (Abbildung 5.3).

Nach Auswahl des `Pull-Up-to-Parent-Feature`-Refactorings läuft im Hintergrund von dem Nutzer unbemerkt der erste Teilschritt des Refactoring-Prozesses. Nachdem der vorbereitende Schritt abgeschlossen ist und keine Fehler aufgetreten sind, wird der linke Dialog der Abbildung 5.4 zur Auswahl der zu verschiebenden

Abbildung 5.3: Kontextmenü des FeatureHouse-Refactorings

Codeelemente angezeigt. An der Position **1.** kann das Feature ausgewählt werden, in welches die ausgewählten Codeelemente verschoben werden sollen. In der darunterliegenden Tabelle werden alle Codeelemente angezeigt, die in der Klasse vorhanden sind. In dem Beispiel soll die Methode `getVecY` und alle dazugehörigen Codeklone extrahiert werden. Wie zu erkennen ist, sind weitere Einträge selektiert (siehe **2.**). Diese Einträge zeigen an, in welchen Features identische Methoden von `getVecY` enthalten sind. Die zweite Spalte der Tabelle signalisiert zusätzlich, dass es sich bei der Methode um einen Codeklon handelt (siehe **3.**). Dieser Codeklon ist in vier verschiedenen Produktvarianten enthalten.

Nach Auswahl aller gewünschten Codeelemente kann der Nutzer mit den Buttons *Next* oder *Finish* den Dialog schließen und die Fortsetzung der zweiten Phase des Refactoring-Prozesses veranlassen. Sollte die Überprüfung der Vorbedingungen keine Fehler liefern, werden alle Änderungen ermittelt. Diese Änderungen werden im rechten Dialog der Abbildung 5.4 angezeigt und können somit nochmal überprüft werden. Im oberen Bereich werden alle durchzuführenden Änderungen dargestellt (siehe **4.**). Im unteren Bereich wird für jede Änderung auf der linken Seite, der aktuelle und auf der rechten Seite, der geänderte Quellcode in einem Vorschaufenster angezeigt (siehe **5.**). Mit dem Klick auf den Button *Finish* werden die angezeigten Änderungen in die Dateien geschrieben und damit das Refactoring beenden.

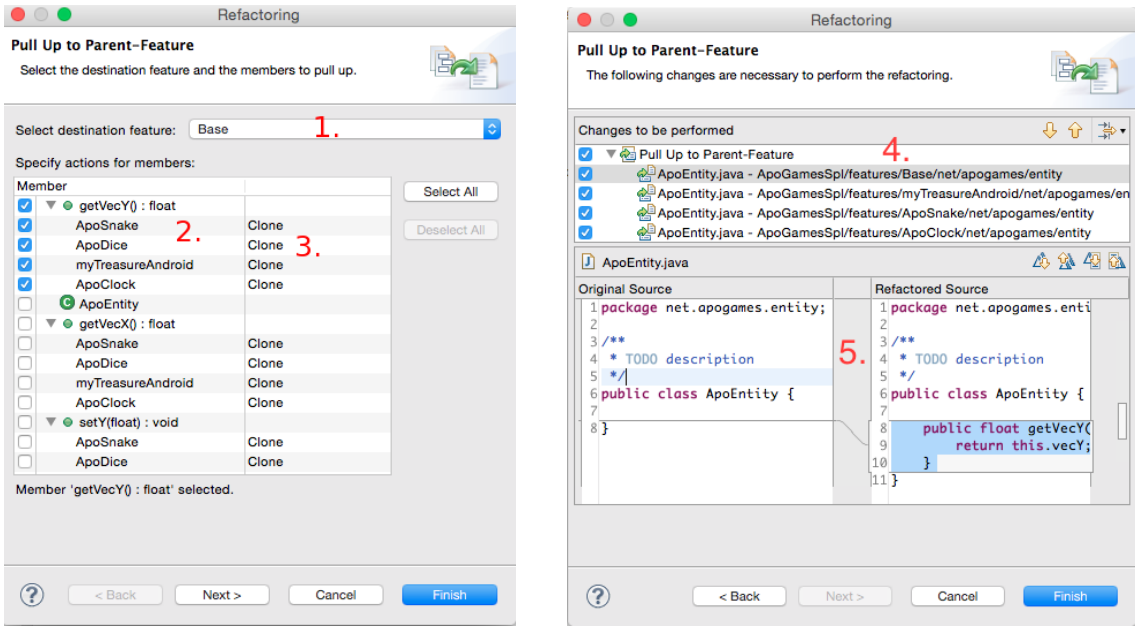


Abbildung 5.4: Dialoge des `Pull-Up-to-Parent-Feature`-Refactoring

5.4 Zusammenfassung

In diesem Kapitel wurde die Umsetzung einer prototypischen Implementierung für die beiden Refactorings (`Rename` und `Pull-Up-to-Parent-Feature`) beschrieben. Dazu wurde im Abschnitt 5.1 ein Einblick in die verwendeten Komponenten gegeben. In den nachfolgenden Abschnitten wurde der Ablauf des Refactoring-Prozesses erläutert und anhand eines Beispiels skizziert.

6. Evaluierung

Dieses Kapitel evaluiert das in Abschnitt 3.3 vorgestellte Konzept der Migration von Produktvarianten in eine SPL. Dafür werden besonders beiden Prozesse Angleichung und Extraktion von Codeklonen betrachtet. Anhand der Fallstudie ApoGames wird die Evaluierung durchgeführt und es wird gezeigt, welche Ergebnisse sich daraus gewinnen lassen.

6.1 Zielstellung

Das Ziel dieser Evaluierung ist den Nutzen meines Konzeptes zur Migration von Produktvarianten in eine SPL zu zeigen. Dazu untersuche ich, ob eine Migration mit meinem Konzept möglich ist und welche Limitierungen dieser Migrationsansatz hat. Im Fokus der Evaluierung stehen die beiden Migrationsschritte Angleichung und Extraktion von Codeklonen.

Die Nutzung meines Konzeptes im Rahmen der Evaluierung wird mittels folgender drei Fragen bewertet.

FF1: *Wie viel geklonter Code kann ohne vorbereitendes Refactoring bereits migriert werden?*

Mithilfe der ersten Forschungsfrage soll beantwortet werden, ob die Extraktion von Codeklonen ohne vorbereitendes Refactoring für die Migration ausreichend ist. Anhand der Anzahl extrahierter und verbliebener Codeklone kann bestimmt werden, ob weitere Schritte in Form eines vorbereitenden Refactorings notwendig sind. Zur Untersuchung wie viel Codeklone zusätzlich migriert werden können sind die Messungen notwendig.

FF2: *Wie viel geklonter Code kann mit vorbereitendem Refactoring migriert werden?*

Die zweite Forschungsfrage beantwortet, ob die Anwendung des vorbereitenden Refactorings die Ergebnisse der Migration verbessern kann bzw. ob das vorbereitende Refactoring einen positiven Einfluss auf die Extraktion von geklontem Code hat. Anhand der Messungen dieser Frage und der vorherigen Frage werden die Auswirkungen meines Konzeptes ermittelt.

FF3: *Wie viel geklonter Code verbleibt in den migrierten Produktvarianten und was sind mögliche Gründe dafür?*

Mit der letzten Forschungsfrage soll beantwortet werden, welche Grenzen mein Konzept hat. Um daran anschließend zu untersuchen, welche Codeklone nicht migriert werden und welche Gründe dafür ausschlaggebend sind. Die Art der verbleibenden Codeklone liefert Limitierungen meines Konzeptes. Anhand derer weitere Maßnahmen wie beispielsweise neue Refactorings vorgeschlagen werden können.

Die Evaluierung soll neben den Forschungsfragen weitere interessante Erkenntnisse liefern. Zum einen, ob bestimmte Eigenschaften vorliegen, die die Ergebnisse der Migration beeinflusst haben. Daraus ergibt sich eine weitere aufschlussreiche Frage, ob die Ergebnisse der Fallstudie auf andere Fallstudien übertragen werden können. Damit kann die Frage beantwortet werden, ob die Ergebnisse dieser Fallstudie auf andere Applikationen (z.B. eingebettete Systeme) übertragen werden können.

6.2 Fallstudie

Als Fallstudie zur Untersuchung meines Kozeptes der Migration von Produktvarianten wird ApoGames[1] verwendet. ApoGames ist eine Sammlung von Spielen, die in Java für die Plattform Android entwickelt wurden. Sie beinhaltet die folgenden fünf Spiele ApoClock, ApoDice, ApoMono, ApoSnake und myTreasure. Diese Spiele eignen sich für eine Migration in eine SPL, weil sie mit dem beschriebenen Clone-and-Own-Ansatz erzeugt wurden. Es ist anzunehmen, dass zwischen den verschiedenen Spielen eine Vielzahl von Codeklonen existiert, die mithilfe einer Migration in ein SPL konsolidiert werden können. Da die Spiele nach und nach entwickelt wurden, beruhen sie auf unterschiedliche Versionen dieser Engine. Dies führte während der Migration zu Problemen, da die Auswahl einer falschen Engine-Version zu Compilerfehlern führte. Die BitsEngineAndroid und weitere spezifische Android-Bibliotheken werden für meine Migration nicht berücksichtigt. In Tabelle 6.1 werden die Produktvarianten kurz beschrieben. Da sich die Spiele vom Spielprinzip her ähnlich sind, eignet sie sich für eine Untersuchung im Rahmen der Migration in eine SPL.

6.3 Durchführung

Die Fallstudie ApoGames wird auf einem FeatureIDE-Projekt für FeatureHouse durchgeführt. Da diese fünf Produktvarianten in separaten Android-Projekten vorliegen, müssen sie zuerst gemeinsam in ein FeatureIDE-Projekt überführt werden. Diese Überführung wird in Abschnitt 3.3.2 beschrieben. Nach Transformation der Produktvarianten in ein initiale SPL und manueller Nachjustierung (u.a. Auswahl

[1] http://www.apo-games.de

Produktvariante	Beschreibung
ApoClock	Puzzlespiel
ApoDice	Würfelpuzzlespiel
ApoMono	Puzzlespiel
ApoSnake	Snake-Spiel
myTreasure	Puzzlespiel

Tabelle 6.1: Übersicht über die Produktvarianten von ApoGames

korrekter Android-Bibliotheken) wird die Codeklonerkennung mit anschließender Analyse auf diese SPL angewendet. Die Codeklonerkennung wurde so konfiguriert, dass alle Zeichenketten der Länge von zehn oder mehr Zeichen (Tokens) als Codeklone erkannt werden. Die Ergebnisse dieser Codeklonerkennung und Metadaten der Features sind in der Tabelle 6.2 darstellt. Diese Tabelle beinhaltet für jedes Feature die Anzahl der Klassen, die Anzahl der Codezeilen, Lines of Code (LOC), die Anzahl der Codeklonzeilen, Lines of Code Clones (LOCC), und die Codeklonrate (CKR).

Features	#Klassen	#LOC	#LOCC	CKR
ApoClock	28	3584	2643	73,7 %
ApoDice	19	2504	2003	80,0 %
ApoMono	25	6483	4382	67,6 %
ApoSnake	19	2946	2350	79,8 %
myTreasure	27	5322	3042	57,2 %
Gesamt	118	20839	14420	69,2 %

Tabelle 6.2: Metadaten der Features inklusive Ergebnisse der Codeklonerkennung

Nach Abschluss der Überführung und Auswertung der Codeklonerkennung kann mit der konkreten Durchführung für diese Fallstudie begonnen werden. Die Migration von ApoGames wird zweimal durchgeführt. Im ersten Durchlauf erfolgt die Extraktion von Codeklonen ohne vorbereitendes Refactoring und liefert somit die Ergebnisse zur Beantwortung der *FF1*. Die Extraktion von Codeklonen wird mithilfe des `Pull-Up-to-Parent-Feature`-Refactoring durchgeführt. Dieser Schritt wird automatisiert ausgeführt, da keine Nutzerinteraktion erforderlich ist. Dafür werden alle Klassen aller Features durchlaufen und auf jede Klasse das `Pull-Up-to-Parent-Feature`-Refactoring angewendet. Für jede Klasse werden geklonte Felder und Methoden ermittelt und diese anschließend in ein gemeinsames Feature verschoben. Sollte die gesamte Klasse ein Codeklon sein, wird sie mit dem gleichen Mechanismus in ein gemeinsames Feature verschoben. Nach Beendigung des ersten Durchlaufs existierten keine Codeklone mehr, die durch das `Pull-Up-to-Parent-Feature`-Refactoring verschiebbar wären.

Beim zweiten Durchlauf wird vor der Extraktion von Codeklonen ein vorbereitendes Refactoring zur Angleichung von Codeklonen angewendet und dadurch die *FF2* abdeckt. Die Auswertungen der Codeklonerkennung lieferte Codefragmente, die durch das `Rename`-Refactoring angeglichen werden müssen. Bei diesem Codefragmenten

handelt es sich um Klassen, Methoden oder Felder, die gleich sind und sich nur durch unterschiedliche Namensbenennungen voneinander unterscheiden. Anschließend werden diese Codefragmente mithilfe des `Rename`-Refactoring umbenannt. Dieser Prozess wird auf alle gefundenen Codefragmente ausgeführt. Nach Beendigung des Angleichungsschrittes muss die Codeklonerkennung nochmal auf den Produktvarianten ausgeführt werden, da durch das Angleichen neue Codeklone entstanden sind. Diese angeglichenen Codeklone werden analog zum ersten Durchlauf extrahiert. Nach Durchführung der beiden Durchläufe können die Ergebnisse ausgewertet werden.

6.4 Ergebnisse

In der Abbildung 6.1 zeige ich den LOC-Vergleich der fünf Features und der gemeinsamen Features (als Feature *Common* dargestellt) nach den beiden Durchläufen des Migrationsprozesses. Für diese Abbildung werden die LOC zu Beginn der Migration, nach dem Durchlauf eins ohne vorbereitendes Refactoring und nach dem zweiten Durchlauf mit vorbereitendem Refactoring ermittelt. Diese Abbildung ist zweigeteilt, auf der linken Seite werden die Zahlen für die einzelnen Features und auf der rechten Seite die Gesamtzahlen dargestellt. Es ist zu erkennen, dass mit vorbereitendem Refactoring mehr Codezeilen eingespart werden konnten. Die Beobachtung ist über alle Features ersichtlich. Im Durchlauf ohne vorbereitendes Refactoring beträgt die Anzahl insgesamt 879 entfernte Codezeilen. Im Gegensatz dazu konnten beim Durchlauf mit vorbereitendem Refactoring 3259 Codeklonzeilen entfernt werden. In den Gesamtsummen sind die ausgelagerten gemeinsamen Codefragmente (Feature *Common*) schon enthalten. Durchschnittlich konnten somit in den verschiedenen Features ca. 3,5 bis viermal mehr Codeklone mit vorbereitendem Refactoring entfernt werden. Aus den fünf Features können insgesamt 479 bzw. 1779 gemeinsame Codezeilen extrahiert werden.

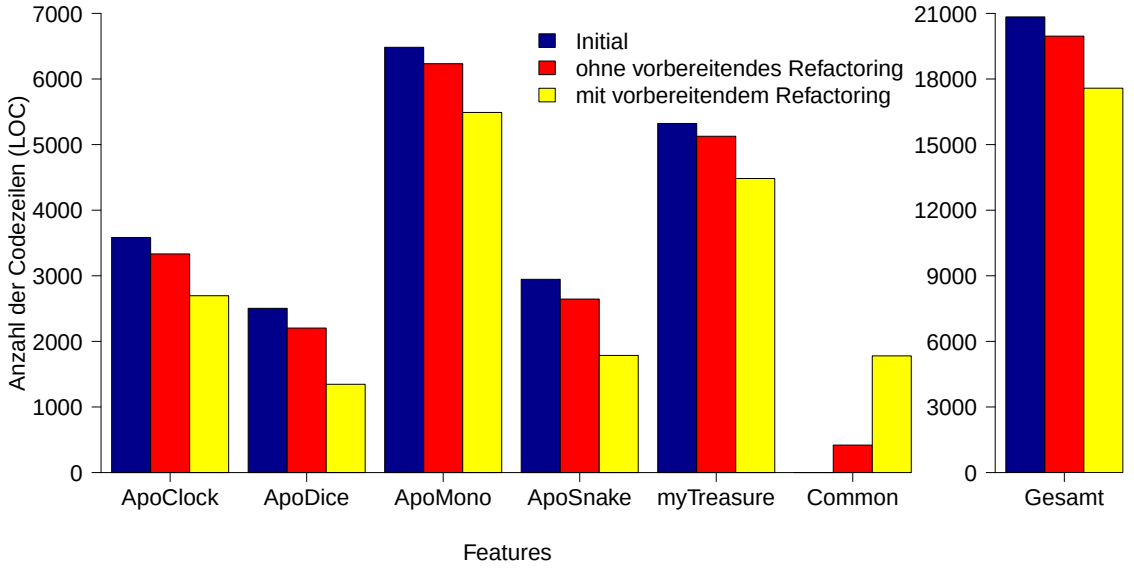


Abbildung 6.1: Vergleich der Anzahl von Codezeilen beider Durchläufe (mit und ohne vorbereitendem Refactoring) des Migrationsprozesses

Dennoch weichen die Codeeinsparungen zwischen den Features teilweise doch stark voneinander ab. Diese Erkenntnis wird in Abbildung 6.2 nochmal bekräftigt. Die Abbildung zeigt die prozentualen Einsparungen von Codezeilen. Ohne vorbereitendes Refactoring können 4,2 Prozent, mit vorbereitendes Refactoring 15,6 Prozent an Codezeilen eingespart werden. Ins Feature *Common* können insgesamt im ersten Durchlauf 3,7 Prozent und im zweiten Durchlauf 10 Prozent extrahiert werden. Das Verhältnisses verschobener Codezeilen zu entfernter Codezeilen liegt in beiden Durchläufen bei 54,5 Prozent. Somit sind 45,5 Prozent aller entfernten Codezeilen Duplikate. Das Einsparpotential beider Features *ApoMono* und *myTreasure* ist in beiden Durchläufen fast identisch, während zwischen den Features *myTreasure* und *ApoDice* große Unterschiede bestehen.

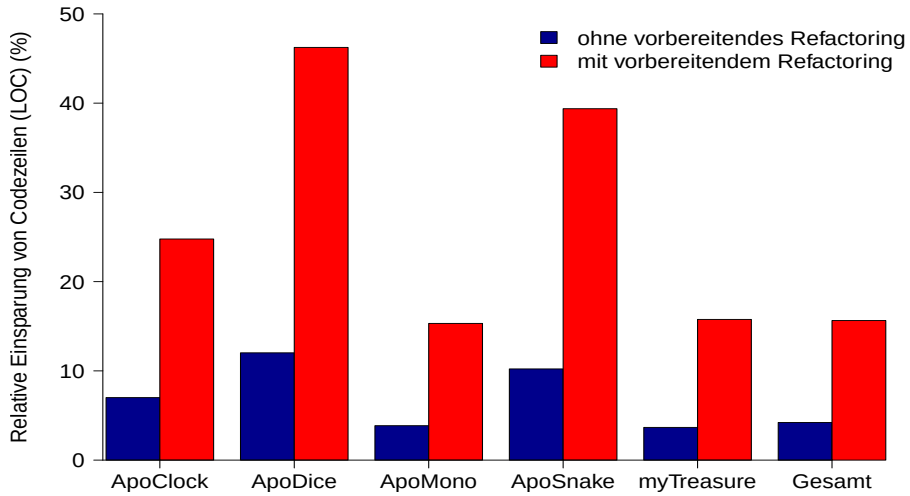


Abbildung 6.2: Prozentualer Vergleich der Anzahl beider Durchläufe des Migrationsprozesses

In den nachfolgenden Abbildungen zeige ich die Anzahl der verschobenen Felder (Abbildung 6.3) und Methoden (Abbildung 6.4). Wie man im Durchlauf ohne vorbereitendes Refactoring erkennt, liegt die Anzahl bei rund 22 verschobenen Feldern in jedem Feature, während im zweiten Durchlauf der Wert zwischen 75 und 102 schwankt. Somit konnten mit vorbereitendem Refactoring ca. vier bis viereinhalb mehr Felder verschoben werden. Diese Beobachtungen spiegeln sich auch in den verschobenen Methoden wider, auch wenn die Zahlen dort mit dem drei- bis dreieinhalbfachen etwas niedriger liegen. Das Maximum liegt bei 63 bzw. 185 (Feature *ApoSnake*) und das Minimum bei 57 bzw. 159 verschobenen Methoden in beiden Durchläufen. Außerdem wird ersichtlich, dass die verschobenen Felder und Methoden für Features `ApoDice` und `ApoSnake` beider Durchläufe nahezu gleich sind. Die Verschiebungen ohne vorbereitendes Refactoring wurden nur aus drei Klassen vorgenommen. Mit vorbereitendem Refactoring wurden Felder und Methoden aus 23 Klassen in gemeinsame Codefragmente ausgelagert.

Die Tabelle 6.3 präsentiert die Anzahl umbenannter Klassen, Methoden und Felder des vorbereitenden Refactorings. Es ist etwas überraschend, dass mehr als 90 Prozent dieser Umbenennung auf Klassen entfallen. Methodenumbenennungen werden nur in vier Features und auch nur in einer sehr geringen Anzahl vorgenommen. Namensan-

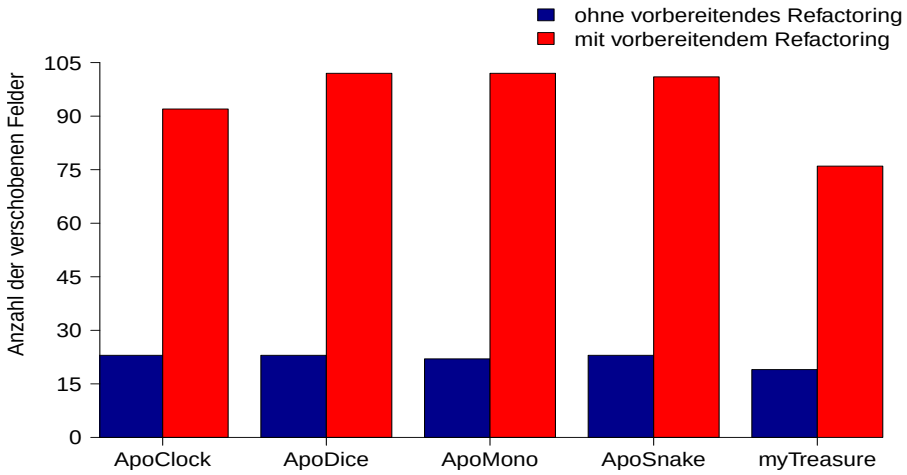


Abbildung 6.3: Vergleich der Anzahl verschobener Felder beider Durchläufe des Migrationsprozesses

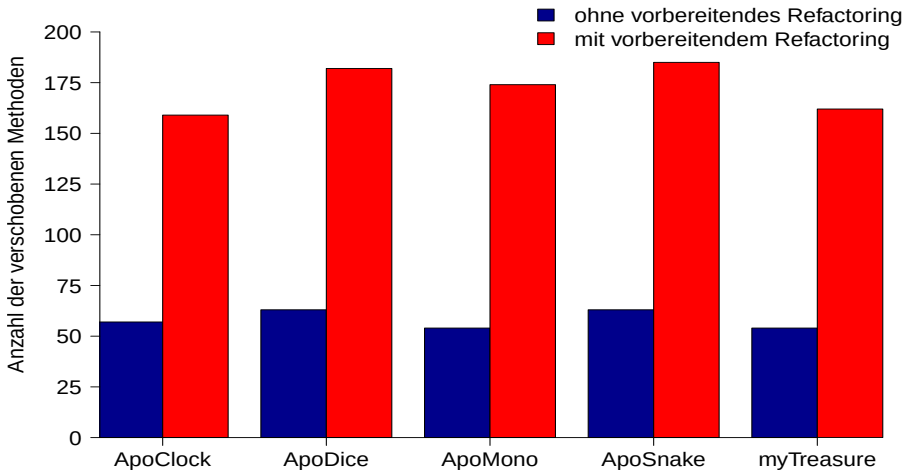


Abbildung 6.4: Vergleich der Anzahl verschobener Methoden beider Durchläufe des Migrationsprozesses

passungen von Feldern konnten nicht durchgeführt, weil aufgrund der definierten Zeichenkettenlänge der Codeklonerkennung in den meisten Fällen keine Codeklone gefunden wurden.

Das Ergebnis der im Code verbleibenden Codeklonen nach Beendigung der Migration zeigt die Tabelle 6.4. Zusätzlich wird die Differenz zur initialen Codebasis aus Tabelle 6.2 ermittelt. Anhand der Tabelle ist zu erkennen, dass die LOCC teils deutlich gesenkt werden konnte. Im Fall des Features *ApoDice* sogar um mehr als 50 Prozent. Da auch die LOC in der Vielzahl herabgesetzt wurde, liegt die Differenz in der Codeklonrate nur zwischen 3 Prozent und 9 Prozent. Insgesamt konnten über alle Features ca. 25 Prozent an Codeklonezeilen entfernt werden. Somit konnte die Codeklonrate um 8 Prozent reduziert werden. Trotzdem zeigt die Tabelle, dass nach Beendigung der Migration weiterhin noch rund 61 Prozent an Codeklonen vorhanden sind. Dies entspricht 10756 geklonten Codezeilenfragmenten.

Features	#Klassen	#Methoden	#Felder
ApoClock	14	0	0
ApoDice	16	2	0
ApoMono	18	2	0
ApoSnake	16	2	0
myTreasure	20	2	0
Gesamt	84	8	0

Tabelle 6.3: Vergleich umbenannter Klassen, Methoden und Felder der fünf Features

Features	#LOC	ΔLOC	#LOCC	ΔLOCC	CKR
ApoClock	2696	-24,8 %	1809	-31,6 %	67,1 (-6,6) %
ApoDice	1346	-46,2 %	957	-52,2 %	71,1 (-8,9) %
ApoMono	5490	-15,3 %	3535	-19,3 %	64,4 (-3,2) %
ApoSnake	1786	-39,4 %	1285	-45,3 %	71,9 (-7,9) %
myTreasure	4483	-15,8 %	2264	-25,6 %	50,5 (-6,7) %
Common	1779	N.A.	906	N.A.	50,9 % (N.A.)
Gesamt	17580	-15,6 %	10756	-25,4 %	61,1 (-8,1) %

Tabelle 6.4: Ergebnisse der Codeklonerkennung nach Beendigung der Migration

6.5 Diskussion

Die fünf migrierten Produktvarianten liefern in beiden Durchläufen der Migration teilweise unterschiedliche Ergebnisse. Die Gründe für diese Ergebnisse und die daraus schließenden Schlussforderungen werden im Nachfolgenden betrachtet.

Zuerst lässt sich festhalten, dass die Migration der fünf Produktvarianten mit und ohne vorbereitende Refactorings erfolgreich durchgeführt werden konnte. In den beiden Durchläufen der Migration konnten eine gewisse Anzahl von Codeklonen in gemeinsame Basiskomponenten ausgelagert werden. Zur Beantwortung der *FF1* betrachte ich die Ergebnisse des ersten Durchlaufs der Migration ohne vorbereitendes Refactoring. Diese Ergebnisse zeigen, dass nur eine geringe Anzahl an Codeklonzeilen (ca. 4 Prozent) entfernt werden konnte. Wie sich aus den Ergebnissen des zweiten Durchlaufs erkennen lässt, wurden in der Migration mit vorbereitendem Refactoring deutlich mehr Codeklone beseitigt. Somit kann als Antwort auf *FF2* festgehalten werden, dass das vorbereitende Refactoring einen positiven Einfluss auf die Migration von Produktvarianten hat. Mithilfe des vorbereitenden Refactorings konnte die Codezeilen um rund 15 Prozent gesenkt werden. Dies verdeutlicht, dass bei einer Migration auf Clone-and-Own-erzeugten Produktvarianten nicht auf vorbereitende Refactorings verzichtet werden kann.

Die Ergebnisse für die Beantwortung der *FF3* liefert die Tabelle 6.4. Anhand der Tabelle erkennt man, dass in den migrierten Produktvarianten weiterhin eine hohe Anzahl von geklontem Code (rund 61 Prozent) vorliegt. Daraus lässt sich ableiten, dass der Migrationsprozess auch mit dem vorbereitenden Refactoring noch nicht als

abgeschlossen gelten kann. Um weitere Codeklone zu extrahieren muss der vorbereitende Schritt ein weiteres Refactoring zur Verfügung stellen. Dafür würde sich das Refactoring `Extract Method/Extract Shared Code` ([Apel et al., 2013a]) oder `Extract Refinement` anbieten. Bei diesen Refactorings werden Codezeilen einer Methode in eine neue Methode extrahiert. Im Fall des Refactorings `Extract Method/Extract Shared Code` werden alle betroffenen Codestellen durch den Aufruf der Methode ersetzt oder beim `Extract Refinement` durch Aufruf des Schlüsselwort `original`.

Zum Abschluss stellt sich die Frage, ob die Ergebnisse der Fallstudie ApoGames zu verallgemeinern sind. Meiner Einschätzung nach würden andere Produktvarianten ebenfalls vorbereitende Schritte benötigen. Darüber hinaus halte ich die Möglichkeit dafür, dass die Migration mit vorbereitendem Refactoring mehr als 15 Prozent Codezeilen einsparen kann, durchaus für sehr realistisch. Dies hat mehrere Gründe. Zum einen sind die Produktvarianten von ApoGames, obwohl mit dem Clone-and-Own-Ansatz erstellt, sehr unterschiedlich. Dadurch existiert in den Varianten wenig identischer Code, der extrahiert werden kann. Somit besteht zwischen den Varianten eine geringe Wiederverwendung von gemeinsamem Code. Demzufolge bieten diese Produktvarianten wenig Potential zur Migration an. In Produktvarianten, die ein realistischeres Abbild des Clone-and-Own-Ansatzes widerspiegeln, sollte dieses Potential viel höher sein. Zum anderen liegen in ApoGames spezifische Namenskonventionen vor, die die Migration von Produktvarianten behindert. Grundsätzlich bedeutet dies, dass mein Ansatz für spezifische Namenskonventionen begrenzt einsetzbar ist.

Diskussion der Validität

Die Ursache für die unterschiedlichen Ergebnisse in beiden Durchläufen liegt darin begründet, dass ApoGames eine spezifische Eigenart in der Benennung von Klassen besitzt. Die Mehrheit von Klassen beinhalten einen Präfix im Namen, der auf die jeweilige Produktvariante verweist. Stellvertretend für ungleiche Namensbenennungen nenne ich die beiden Klassen `ApoClockMenu` und `ApoDiceMenu`. Verdeutlicht wird diese Aussage durch eine Vielzahl von Angleichungen der Klassennamen in Tabelle 6.3. Diese unterschiedlichen Namensbenennungen führen zu dem Problem, dass eigentlich identische Klassen oder Methoden der verschiedenen Produktvarianten im ersten Durchlauf nicht beseitigt werden können. Die Ursache dafür ist, dass die Codeklone bei der Analyse nicht als Typ-I-Klone erkannt werden und somit nicht extrahiert werden können. Erst mit der automatischen Angleichung der Namensbezeichner war es mir möglich, identische Klassen, Methoden und Felder zu extrahieren.

Die Fehlerursachen, warum nach Migration weiterhin eine hohe Anzahl von geklontem Code vorliegt, sind vielfältig. Zum einen liegt es daran, dass die geklonten Produktvarianten vollständig unabhängig voneinander entwickelt wurden. Somit wurden Änderungen in einer Produktvariante nicht auf andere Produktvarianten übertragen, was sich in der Auswahl unterschiedlicher Versionen von Bibliotheken sichtbar macht. Zum anderen unterscheiden sich Methoden mit gleicher Signatur in den verschiedenen Produktvarianten meistens nur in wenigen Codezeilen. Teilweise weichen

diese Methoden sogar nur in einer einzigen Codezeile voneinander ab. Oftmals existieren in den Klassen aber auch sehr lange Methoden, in denen Code mehrfach wiederholt auftritt. An einigen Codestellen wird außerdem anstatt einer Feld-Referenz der konkrete Wert dieser Referenz verwendet. Das vorbereitende Refactoring bietet in diesem Fall keine Maßnahmen zur Angleichung von Typ-III-Codeklonen (siehe Abschnitt 3.1.1).

Eine weitere Fehlerursache, die die Migration erschwert, ist die Verwendung unterschiedlicher Versionen von Bibliotheken in den fünf Produktvarianten der Fallstudie ApoGames. Diese Bibliotheken unterscheiden sich teilweise stark voneinander. In höheren Versionen existieren Methoden nicht mehr oder die Signaturen der Methoden haben sich geändert. Hier wäre es wünschenswert, die Produktvarianten vor der Migration so anzupassen, dass in allen Produktvarianten gleiche Bibliotheken verwendet werden.

Des Weiteren konnten einige Felder nicht verschoben werden, weil sie unter die Grenze von zehn Tokens fallen. Zur Erinnerung, die Codeklonerkennung findet nur Codeklone, die größer gleich diese Schranke sind. Es sollte überlegt werden, auf welchen Wert die Schranke gesetzt wird. Dabei ist aber zu beachten, dass je kleiner die Grenze gesetzt ist, desto mehr Codeklone in der Regel gefunden werden. Dadurch kann in mittleren bis großen Projekten schnell die Übersicht über die zu extrahierenden Codeklone verloren werden. Eine Möglichkeit, die ich für durchführbar halte, ist, die Grenze schrittweise bis zu einer gewissen Schwelle herunterzusetzen. In jedem Schritt werden die gefundenen Codeklone extrahiert. Können keine Codeklone mehr extrahiert werden, wird die Grenze um einen bestimmten Wert gemindert und die Extraktion der Codeklone analog zum vorherigen Schritt durchgeführt werden.

6.6 Zusammenfassung

Die Evaluierung zeigt den Nutzen der beiden Refactorings `Pull-Up-to-Parent-Feature` und `Rename` für die Migration anhand der Fallstudie ApoGames. Mithilfe der beiden Refactorings können die ApoGames in eine SPL migriert werden. Dazu wurde die Migration einmal mit und einmal ohne vorbereitendes Refactoring durchgeführt. Insgesamt zeigen die Ergebnisse der Evaluierung, dass die Migration mit Tool-Unterstützung erfolgreich durchgeführt werden kann. Unter Verwendung des vorbereitenden Refactorings können bei der Migration rund 16 Prozent an Codezeilen entfernt werden. Dennoch sind weitere vorbereitende Schritte notwendig, um den gewünschten Grad an Codeklonen zu erreichen.

7. Verwandte Arbeiten

Mit dieser Arbeit wurde das Ziel verfolgt, Produktvarianten unter Verwendung Feature-orientierter Programmierung in eine SPL zu migrieren. Für den Migrationsprozess wurden dazugehörige Refactorings zur Angleichung und Extraktion von Codeklonen mithilfe von FOP entwickelt. Dieses Kapitel gibt einen Überblick zu verwandten Arbeiten aus den Themengebieten Migration von Produktvarianten und Refactorings mit Bezug zu SPLs.

Migration von Produktvarianten

Wie in Kapitel 1 diskutiert, existieren eine Reihe von Ansätzen zur Migration von Produktvarianten in eine SPL. Von diesen Ansätzen sind insbesondere diejenigen mit meiner Arbeit verwandt, welche die Migration mehrerer Produktvarianten in eine SPL zum Ziel haben. Auf diese gehe ich in den folgenden Abschnitten detailliert ein.

Mit der Migration von Produktvarianten beschäftigten sich bereits Alves et al. [2006]. Dabei zielt dieser Migrationsansatz auf eine SPL ab, die mit AOP implementiert ist. Alves et al. präsentieren ein Menge von Transformationsprozessen für Feature-Modelle, um die Variabilität migrierter Produktvarianten zu bewahren und unterschiedliche Feature-Modelle in ein gemeinsames zu übertragen. Trotzdem bleibt bei dem Ansatz von Alves et al. unklar, wie die Zusammenführung der Produktvarianten auf Code-Ebene erfolgen soll. Darüber hinaus stellt diese Arbeit keine Möglichkeit zum Angleichen und Extrahieren von Codeklonen bereit. Diese beiden Schritte sind für meinen Migrationsansatz von essentieller Bedeutung.

Die Migrationsansätze von Xing und Xue et al. zielen darauf ab, eine Menge von geklonten Produktvarianten in eine annotationsbasierte SPL zu migrieren. Dabei beruht der Ansatz auf zwei Säulen, dem Vergleich der Feature-Modelle aller Produktvarianten und einer Codekloneanalyse. Durch Kombination beider Komponenten können gemeinsame und variable Features ermittelt werden [Xing et al., 2011; Xue et al., 2010; Xue, 2011, 2012]. Bei meinem Ansatz steht die Extraktion von Gemeinsamkeiten auf Code-Ebene im Vordergrund, während die Domänenanalyse zunächst nicht betrachtet wird. Vergleiche auf Feature-Modell-Ebene sind jedoch

Bestandteil zukünftiger Arbeiten. Hier besteht Synergiepotenzial zwischen unserer Arbeit und den Arbeiten von Xue.

Klatt et al. präsentieren einen Ansatz zur Konsolidierung von geklonten Varianten in eine SPL. Dabei werden anhand von Unterschieden im Code zwischen den Varianten Variationspunkte abgeleitet, um das Variabilitätsmodell durch Aggregationen dieser zusammengehörenden Variationspunkte zu erstellen [Klatt und Küster, 2013; Klatt et al., 2013, 2014a,b]. Im Gegensatz zu meinem Ansatz bei dem der Schwerpunkt auf schrittweisem Refactoring von Code liegt, fokussiert der Ansatz von Klatt et al. vorrangig auf die Modell-Ebene welche Teil zukünftiger Arbeiten sind.

In der Arbeit von Rubin et al. wird ein Framework für geklonte Softwaresysteme eingeführt, dass mit Unterstützung von abstrakten Operatoren die Beschreibung und Verwaltung von Codeklonen ermöglicht [Rubin und Chechik, 2013]. Mein Ansatz unterscheidet sich darin, dass konkrete Refactorings und Tool-Unterstützung angeboten werden, welche einen Teil von Operatoren des Frameworks implementieren.

Refactoring von Feature-orientierten Software-Produktlinien

Verschiedene Arbeiten existieren für Refactorings in SPLs. Dieser Abschnitt fokussiert sich auf das Refactoring in Feature-orientierten SPLs.

Basierend auf Analyse der Code-Struktur führen Liu et al. [2006] einen Ansatz für Feature-orientiertes Refactoring ein. Im Vordergrund dieses Ansatzes steht die Abspaltung von Funktionalität eines Programms in einzelne Features. Des Weiteren werden die Probleme der Feature-Interaktion beschrieben. Im Gegensatz zu meiner Arbeit wird dieses Refactoring auf ein einzelnes Produkt angewendet, mit dem Ziel der Überführung in eine SPL. Darüber hinaus beschränkt sich der Ansatz von Liu et al. auf formale Grundlagen des Zerlegungsprozesses, konkrete Refactorings und Tool-Support, wie in meinen Ansatz umgesetzt, werden nicht präsentiert.

Kuhlemann et al. [2009] schlagen vor, Refactorings in Feature-Module einzubinden. Mit diesem Ansatz werden automatisierte objektorientierte Refactorings auf Feature-Modulen während der Generierung von Produktvarianten ausgeführt. Diese Refactorings werden automatisch angewendet, während in meinem Ansatz diese manuell durch den Entwickler angestoßen werden. Zudem wird in meiner Arbeit im Gegensatz zu Kuhlemann et al. die Migration von geklonten Produktvarianten in eine SPL betrachtet.

In einer empirischen Studie wurde das Vorhandensein und die Entfernung von Codeklonen in Feature-orientierten SPLs analysiert [Schulze et al., 2010]. Die Ergebnisse der Studie zeigten eine Vielzahl von Codeklonen. Die Arbeit von Schulze et al. setzt den Fokus auf Code-Qualität in bestehenden SPLs. Meine Arbeit hingegen verwendet die Codeklonerkennung, um Produktvarianten in eine SPL zu migrieren.

Schulze et al. [2012] führen das Konzept variantenerhaltendes Refactoring ein. Dazu definieren sie, was unter variantenerhaltenden Refactoring verstanden wird und stellen Bedingungen auf, die nach einer Änderung am Feature-Modell und/oder Implementierung gelten müssen. Aufbauend auf diesen Ansatz wurde in einer nachfolgenden Arbeit ein konkretes Refactoring (`Pull Up Method`) umgesetzt [Schulze

et al., 2013]. In meinem Ansatz wird das `Pull Up`-Refactoring mit einer Codeklonerkennung kombiniert, wodurch das Refactoring in der Praxis deutlich einfacher einsetzbar ist. Darüber hinaus stelle ich mit einem Ansatz ein `Rename`-Refactoring vor.

Definitionen für das Bewahren des Verhaltens von Varianten und einen Katalog mit sieben Refactoring stellen Apel et al. [2013a] vor. Anhand eines `Extract Feature`-Refactoring erläutern sie, welche Schritte notwendig sind, damit das Refactoring das Verhalten aller Produktvarianten in FOP sicherstellt. Auf die benötigten Vorbedingungen für das Refactoring wird nicht eingegangen und konkrete Implementierungen von Refactorings bleiben offen.

Die Arbeit von Liebig et al. [2015] beschreibt ein Refactoring-Tool zum Bewahren der Variabilität im SPL-Kontext. Dieses Tool bietet das Refactoring von C-Code mit cpp-Direktiven an, während mein Refactoring-Tool auf FOP mit Java abzielt. Darüber hinaus betrachten Liebig et al. keine Migration.

8. Zusammenfassung und Zukünftige Arbeiten

Um Zeit und Kosten bei der Entwicklung neuer Softwareprodukte zu sparen und diese Produkte schnell in den Markt zu bringen, ist der Clone-and-Own-Ansatz weit verbreitet. Dabei wird der gesamte Code eines vorhandenen Softwareprodukts kopiert und anschließend so angepasst, dass er den gewünschten Anforderungen des neuen Softwareprodukts entspricht. Der Nachteil dieses Ansatzes sind die mittel- und langfristig hohen Kosten für die Wartung und Wiederverwendung. Um den Wartungs- und Wiederverwendungsaufwand bei steigender Anzahl von Produkten zu minimieren, werden neue Konzepte wie Software-Produktlinien (SPLs) benötigt. Die Idee von SPLs ist die Entwicklung von Softwareprodukten, die sich eine gemeinsame Codebasis teilen. Dies hat den Vorteil, dass gemeinsame Funktionalität nur einmal entwickelt werden und dann allen Produktvarianten zur Verfügung stehen. Dadurch müssen Änderungen nur am gemeinsamen Software-Artefakt vorgenommen werden, anstatt in mehreren Softwareprodukten. Eine SPL vereint somit neben der systematischen Wiederverwendung auch eine große Variabilität, also die Möglichkeit der Anpassung an unterschiedliche Bedürfnisse. Ein Implementierungsansatz für eine SPL bietet die Feature-orientierte Programmierung (FOP). Dabei wird der Quellcode in einzelne separate Module aufgeteilt. Aus den genannten Gründen wäre es wünschenswert, ab einer bestimmten Anzahl an Varianten, eine Überführung der geklonten Varianten in eine SPL durchzuführen, die aber oftmals komplex und aufwendig. Für diese Überführung bietet meine Arbeit einen Ansatz an.

Aktuell existieren verschiedene Ansätze zur Migration von Produktvarianten, die aber aus den unterschiedlichen Gründen (z.B. Migration nur einer Produktvariante oder keine Unterstützung für FOP) für meinen Ansatz nicht geeignet sind. Mit meinem Ansatz soll die Lücke der Migration auf Basis schrittweisem Refactoring von FOP-Code geschlossen werden. Dazu kombiniere ich das Refactoring mit der Codeklonerkennung, um identische Funktionalitäten zwischen den Produktvarianten zu erkennen und in gemeinsame Artefakte zu extrahieren. Da das Zusammenspiel von FOP und Refactorings bislang wenig erforscht ist, existiert keine Tool-Unterstützung in FeatureIDE, welche mit diesem Ansatz umgesetzt werden soll.

Die vorliegende Arbeit beschäftigt sich mit einer Migration von Clone-and-Own-erzeugten Produktvarianten in eine SPL. Meine Arbeit basiert auf der Annahme, dass ein Großteil der Funktionalitäten in mehreren Produktvarianten gleich ist. Somit können Produktvarianten eine hohe Anzahl von ähnlichen oder identischen Codestellen aufweisen. Das Ziel der Migration ist diese gleichen Funktionalitäten in gemeinsam genutzte Software-Artefakte einer SPL zu extrahieren, um den Grad an Wiederverwendung und Wartung zu erhöhen. Diese Extraktion ist aber nur möglich, wenn die Funktionalitäten komplett identisch implementiert ist. Dies bedeutet, dass es sich um eine exakte Kopie der Funktionalität handelt. Diese Gegebenheit ist im Kontext der späteren separaten Weiterentwicklung als unmittelbare Folge von Clone-and-Own nicht immer gegeben. Daher müssen vorbereitende Maßnahmen ergriffen werden, um die vorhandenen Unterschiede anzugleichen. Anschließend können die angeglichenen Funktionalitäten extrahiert werden.

Zur Migration von Produktvarianten wurde ein Konzept erstellt, dass die schrittweise Angleichung und Extraktion enthält. Die Umsetzung dieser Teilschritte wird mithilfe von Refactorings verwirklicht. Die Erkennung von identischen oder ähnlichen Funktionalitäten zwischen den verschiedenen Produktvarianten erfolgt durch eine Codeklonerkennung [Tonscheidt, 2015]. Anhand dieser Codeklonerkennung erfolgt in meinem Konzept die Analyse der Ergebnisse, mit denen sich bestimmen lässt, welche Funktionalitäten das Potenzial zur Angleichung und/oder Extraktion besitzen. Das objektorientierte Refactoring kann nur auf einzelne Varianten, aber nicht auf die gesamte SPL angewendet werden. Aufgrund dieser Nichtberücksichtigung der Variabilität müssen neue Ansätze für Refactorings in FOP betrachtet werden. Variantenerhaltendes Refactoring stellt so einen Ansatz dar, mit dem Ziel, sowohl die Variabilität als auch das Verhalten jeder Produktvariante zu erhalten [Schulze et al., 2012; Apel et al., 2013a]. Bisher existiert Refactoring von FOP-Code bis auf die Arbeit von Brunswig [2013] kaum Tool-Unterstützung, weshalb die Refactorings derzeit meist manuell durchgeführt werden. Für mein Konzept wurden die beiden variantenerhaltenden Refactorings (`Rename` und `Pull-Up-to-Parent-Feature`) in FeatureIDE umgesetzt, einem Framework zur Entwicklung von SPLs. Die Refactorings, insbesondere das `Rename`-Refactoring können auch unabhängig vom Migrationskontext die Implementierung und Weiterentwicklung von FOP-basierten SPLs vereinfachen. Meine Arbeit liefert somit einen Beitrag zur Automatisierung ausgewählter Refactorings durch Tool-Unterstützung.

Die Evaluierung meines Konzeptes erfolgt mittels der Fallstudie ApoGames, welche fünf Produktvarianten enthält, die mithilfe des Clone-and-Own-Ansatzes erzeugt wurden. Die Beantwortung der ersten Forschungsfrage zeigte, dass bei einer Migration ohne vorbereitendes Refactoring nur eine geringe Anzahl von Codezeilen und damit wenige Funktionalitäten extrahiert werden konnten. Anhand der Ergebnisse der zweiten Forschungsfrage lässt sich feststellen, dass die Migration mit vorbereitendem Refactoring rund 15,6 Prozent an Codezeilen einspart. Dies ist viermal so viel wie bei der Migration ohne vorbereitendes Refactoring. Die Schlussfolgerung daraus ist, dass vorbereitende Maßnahmen notwendig sind, um gemeinsame Funktionalitäten zu extrahieren und damit die Anzahl von Codeklonen zu reduzieren. Dennoch zeigt die dritte Forschungsfrage, dass nach Abschluss der Migration in den migrierten Produktvarianten weiterhin eine hohe Anzahl von Codeklonen existiert.

Insgesamt wurden 3664 Codeklonezeilen (25 Prozent) entfernt. Weiterhin verbleiben aber noch 10756 Zeilen an Codeklonen, was einer Codeklonrate von rund 61 Prozent entspricht. Diese Codeklone können mit dem vorliegenden Konzept nicht entfernt werden. Zum einen liegt es daran, dass die geklonten Produktvarianten vollständig unabhängig voneinander entwickelt wurden. Somit wurden Änderungen in einer Produktvariante nicht auf andere Produktvarianten übertragen, was sich in der Auswahl unterschiedlicher Versionen von Bibliotheken sichtbar macht. Zum anderen unterscheiden sich Methoden mit gleicher Signatur in den verschiedenen Produktvarianten meist nur in wenigen Codezeilen. Somit kann der Migrationsprozess als noch nicht abgeschlossen gelten und benötigt weitere Maßnahmen.

Zusammenfassend lässt sich Folgendes festhalten: Mithilfe des von mir vorgestellten Konzeptes ist es möglich, Produktvarianten durch variantenerhaltendes Refactoring in eine SPL zu migrieren. Der Entwickler wird somit bei der Migration durch einen werkzeuggestützten Ansatz unterstützt. Dieser Ansatz beinhaltet neben der Codeklonerkennung auch zwei Refactorings, zum einem für das Angleichen von ähnlichen Funktionalitäten und zum anderen für die Extraktion von gleichen Funktionalitäten.

Zukünftige Arbeiten

In zukünftigen Arbeiten sollte der Nutzen meines Konzepts der Migration anhand weiterer Fallstudien untersucht werden. Dazu sollte die Frage beantwortet werden, ob andere Fallstudien gleiche oder abweichende Ergebnisse liefern. Aufgrund der Eigenarten der vorliegenden Fallstudie, z.B. Namenskonventionen, ist anzunehmen, dass mein Konzept in anderen Fallstudien bessere Ergebnisse liefert und damit mehr gemeinsame Funktionalitäten extrahiert werden können. Der Grund für diese Vermutung ist, dass die Produktvarianten meiner Fallstudie sehr unterschiedlich sind und lediglich auf gemeinsamer Kernfunktionalität basieren. Bei Produktvarianten, die geringere Unterschiede zueinander aufweisen, erwarte ich, dass die Anzahl der Codeklone größer ist, und damit mein Ansatz auch effektiver ist. Darüber hinaus spielt der Entwicklungsprozess in Clone-and-Own-Varianten vermutlich eine Rolle. Dabei sollte untersucht werden, ob die Art und Weise der Synchronisation von Änderungen (z.B. Fehlerbehebungen) die Migration beeinflusst. In meiner Fallstudie wurden Änderungen lediglich manuell, ohne gesonderten Prozess oder Werkzeugunterstützung übertragen. Zudem könnten die Ergebnisse für Fallstudien aus anderen Anwendungsdomänen (z.B. eingebetteten Systemen) abweichend sein und sollten deshalb in die Betrachtung mit einbezogen werden.

In meiner Fallstudie habe ich festgestellt, dass die Migration von Produktvarianten durchgeführt werden konnte, aber weiterhin eine hohe Anzahl von gemeinsamen Funktionalitäten zwischen den Produktvarianten verbleibt. Aus diesem Grund schlage ich vor, mein Konzept um weitere Refactorings wie das `Extract Method` oder `Extract Refinement` zu erweitern. Anhand dieser neuen Refactorings muss untersucht werden, ob eine Verbesserung erreicht wird. Das heißt, ob mit diesen Refactorings die Anzahl der Codeklone weiter gesenkt werden kann.

Das vorgestellte Konzept der Migration von Produktvarianten ist auf die Programmiersprache Java und den FOP-Ansatz FeatureHouse ausgelegt. Dennoch lässt sich dieses auch Konzept auf andere Ansätze wie beispielsweise FeatureC++, welcher

die Programmiersprache C++ verwendet, anwenden. Dafür müssten die Vor- und Nachbedingungen sowie die Vorgehensweise meiner Refactorings an die Syntax und Semantik der Zielsprache angepasst werden. Zudem sind teilweise große Anstrengungen notwendig, da die Implementierung meiner Refactorings für andere Sprachen von verfügbaren Werkzeugen abhängt.

Im Vergleich zu anderen Migrationsansätzen (wie beispielsweise Xing et al. [2011]) wird die Domänenanalyse in meinem Konzept nicht betrachtet. Diese Analyse sollte jedoch Bestandteil zukünftiger Arbeiten sein. Dazu könnte man beispielsweise die Verbindung von Feature-Location-Techniken mit meinem Ansatz untersuchen.

Das Thema der Skalierbarkeit der Refactorings bleibt in meiner Arbeit offen. Um Skalierbarkeit zu zeigen, sollten die Refactorings auf größeren SPLs, d.h. eine Vielzahl von Features und Produktvarianten, ausgeführt werden. Es ist davon auszugehen, dass eine Erhöhung der Anzahl von Features das Refactoring nicht maßgeblich behindert.

Des Weiteren können eine Reihe von Verbesserungen und Erweiterungen an den implementierten Refactorings vorgenommen werden. Beispielsweise sollten beim `Pull-Up-to-Parent-Feature`-Refactoring die Import-Anweisungen der zu extrahierenden Methoden oder Felder ebenfalls automatisch mit extrahiert werden. Eine sinnvolle Erweiterung wäre, den Analyseaufwand durch den Einsatz von Caching-Mechanismen reduzieren. Dies könnte bei großen Projekten zu einer deutlichen Zeiteinsparung führen. Mit dem aktuellen Stand des Prototypen müssen die Konfigurationen nach einer Änderung am Feature-Modell manuell angepasst werden. Dies könnte ebenfalls automatisiert werden.

Literaturverzeichnis

V. Alves, P. Matos, L. Cole, P. Borba und G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 70–81. Springer, 2005. (zitiert auf Seite 2)

V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba und C. Lucena. Refactoring Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, Seiten 201–210. ACM, 2006. (zitiert auf Seite 2 und 67)

M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski und I. Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 532–535. ACM, 2014. (zitiert auf Seite 18)

S. Apel, C. Lengauer, B. Möller und C. Kästner. An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, Seiten 36–50. Springer, 2008. (zitiert auf Seite 5)

S. Apel, C. Lengauer, B. Möller und C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047, 2010. (zitiert auf Seite 9 und 10)

S. Apel, D. Batory, C. Kästner und G Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, Berlin, 2013a. (zitiert auf Seite 1, 2, 6, 7, 9, 23, 24, 64, 69 und 72)

S. Apel, C. Kästner und C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering (TSE)*, 39(1):63–79, 2013b. (zitiert auf Seite 4, 9, 10 und 11)

R. S. Arnold. An introduction to software restructuring. IEEE Press, 1986. (zitiert auf Seite 19)

B.S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Seiten 86–95. IEEE, 1995. (zitiert auf Seite 15)

D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 7–20. Springer, 2005. (zitiert auf Seite 7)

D. Batory, Jacob Neal Sarvela und Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 187–197. IEEE, 2003. (zitiert auf Seite 2 und 9)

I. D. Baxter, A. Yahin, L. Moura, M. Sant Anna und L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, Seite 368. IEEE, 1998. (zitiert auf Seite 16)

S. Brunswig. Automatisierte Refactorings für Feature-Orientierte Software-Produktlinien. Masterarbeit, TU Braunschweig, 2013. (zitiert auf Seite 42 und 72)

P. C. Clements und L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2001. (zitiert auf Seite 1 und 5)

D. Coleman, D. Ash, B. Lowther und P. Oman. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27(8):44–49, 1994. (zitiert auf Seite 19)

K. Czarnecki und U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM, New York, 2000. (zitiert auf Seite 1, 5 und 7)

M. Dalgarno und D. Beuche. Variant Management. 2007. (zitiert auf Seite 1 und 18)

T. Dudziak und J. Wloka. Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code. Diplomarbeit, TU Berlin, 2002. (zitiert auf Seite 3)

W. Fenske, T. Thüm und G. Saake. A Taxonomy of Software Product Line Reengineering. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2014. (zitiert auf Seite 26)

M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999. (zitiert auf Seite 3, 18, 19, 20, 21 und 49)

R Geiger, B. Fluri, H. C. Gall und M. Pinzger. Relation of Code Clones and Change Couplings. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, Seiten 411–425. Springer, 2006. (zitiert auf Seite 1)

W. G. Griswold und D. Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3): 228–269, 1993. (zitiert auf Seite 19)

I. M. Holland. Specifying Reusable Components Using Contracts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Seiten 287–308. Springer, 1992. (zitiert auf Seite 8)

E. Juergens, F. Deissenboeck, B. Hummel und S. Wagner. Do Code Clones Matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 485–495. IEEE, 2009. (zitiert auf Seite 16)

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon Universität, November 1990. (zitiert auf Seite 5, 6 und 7)

C. J. Kapser und M. W. Godfrey. Cloning Considered HarmfulConsidered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, 13(6):645–692, 2008. (zitiert auf Seite 16 und 18)

R. M. Karp und M. O. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development - Mathematics and computing*, 31(2): 249–260, 1987. (zitiert auf Seite 29)

C. Kästner, S. Apel und M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 311–320. ACM, 2008. (zitiert auf Seite 9)

Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. Doktorarbeit, Otto-von-Guericke-Universität Magdeburg, Magdeburg, 2010. (zitiert auf Seite 51)

J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, Boston, 2004. (zitiert auf Seite 21)

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier und J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Seiten 220–242. Springer, 1997. (zitiert auf Seite 9 und 51)

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm und W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Seiten 327–353. Springer, 2001. (zitiert auf Seite 9)

M. Kim, V. Sazawal, D. Notkin und G. Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the European Software Engineering Conference/-Foundations of Software Engineering (ESECFSE)*, Seiten 187–196. ACM, 2005. (zitiert auf Seite 15 und 16)

B. Klatt und M. Küster. Improving Product Copy Consolidation by Architecture-aware Difference Analysis. In *Proceedings of the International ACM Sigsoft Conference on Quality of Software Architectures (QoSA)*, Seiten 117–122. ACM, 2013. (zitiert auf Seite 68)

B. Klatt, M. Küster und K. Krogmann. A graph-based analysis concept to derive a variation point design from product copies. *International Workshop on Reverse Variability Engineering (REVE)*, Seiten 1–8, 2013. (zitiert auf Seite 68)

B. Klatt, K. Krogmann und C. Seidl. Program Dependency Analysis for Consolidating Customized Product Copies. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, Seiten 496–500. IEEE, 2014a. (zitiert auf Seite 68)

B. Klatt, K. Krogmann und C. Wende. Consolidating Customized Product Copies to Software Product Lines. *Workshop on Software-Reengineering & Evolution (WSRE)*, 2014b. (zitiert auf Seite 68)

S. Kolesnikov. An Extensible Compiler for Feature-Oriented Programming in Java. Diplomarbeit, Passau Universität, 2011. (zitiert auf Seite 50)

C. W. Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992. (zitiert auf Seite 1)

M. Kuhlemann, D. Batory und S. Apel. Refactoring Feature Modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, Seiten 106–115. Springer, 2009. (zitiert auf Seite 68)

Z. Li, S. Lu, S. Myagmar und Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, Seite 20. USENIX Association, 2004. (zitiert auf Seite 16)

J. Liebig, S. Apel, C. Lengauer, C. Kästner und M. Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 105–114. ACM, 2010. (zitiert auf Seite 9)

J. Liebig, A. Janker, F. Garbe, S. Apel und C. Lengauer. Morpheus: Variability-Aware Refactoring in the Wild. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 380–391. IEEE, 2015. (zitiert auf Seite 69)

J. Liu, D. Batory und C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 112–121. ACM, 2006. (zitiert auf Seite 2 und 68)

R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal und A. Egyed. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 181–190. IEEE, 2011. (zitiert auf Seite 2)

T. Mens und T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering (TSE)*, 30(2):126–139, 2004. (zitiert auf Seite 18 und 19)

E. Murphy-Hill, C. Parnin und A. P. Black. How We Refactor, and How We Know It. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 287–297. IEEE, 2009. (zitiert auf Seite 33)

A. Olszak und B. N. Jørgensen. Remodularizing Java programs for comprehension of features. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, Seiten 19–26. ACM, 2009. (zitiert auf Seite 2)

W. F. Opdyke. *Refactoring Object-oriented Frameworks*. Doktorarbeit, Universität von Illinois, Illinois, 1992. (zitiert auf Seite 3, 19, 20 und 49)

D. L. Parnas. Software Aging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 279–287. IEEE, 1994. (zitiert auf Seite 18)

K. Pohl, G. Böckle und F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, New York, 2005. (zitiert auf Seite 1, 2, 5 und 6)

C. Prehofer. Feature-Oriented Programming: A Fresh Look At Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Seiten 419–443. Springer, 1997. (zitiert auf Seite 2, 9 und 51)

D. B. Roberts. *Practical Analysis for Refactoring.* Doktorarbeit, Universität von Illinois, Illinois, 1999. (zitiert auf Seite 19 und 49)

C. K. Roy und J. R. Cordy. A Survey on Software Clone Detection Research. *Technical Report 2007-541, School of Computing, Queen's University*, 2007. (zitiert auf Seite 15 und 18)

C. K. Roy, J. R. Cordy und R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming (SCP)*, 74(7):470–495, 2009. (zitiert auf Seite 3 und 16)

J. Rubin und M. Chechik. A Framework for Managing Cloned Product Variants. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 1233–1236. IEEE, 2013. (zitiert auf Seite 18 und 68)

J. Rubin, A. Kirshin, G. Botterweck und M. Chechik. Managing Forked Product Variants. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 156–160. ACM, 2012. (zitiert auf Seite 18)

J. Rubin, K. Czarnecki und M. Chechik. Managing Cloned Variants: A Framework and Experience. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 101–110. ACM, 2013. (zitiert auf Seite 18)

I. Schaefer, L. Bettini, F. Damiani und N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 77–91. Springer, 2010. (zitiert auf Seite 51)

S. Schulze, S. Apel und C. Kästner. Code Clones in Feature-oriented Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, Seiten 103–112. ACM, 2010. (zitiert auf Seite 68)

S. Schulze, T. Thüm, M. Kuhlemann und G. Saake. Variant-preserving Refactoring in Feature-oriented Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, Seiten 73–81. ACM, 2012. (zitiert auf Seite 3, 22, 23, 24, 25, 68 und 72)

S. Schulze, M. Lochau und S. Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, Seiten 33–40. ACM, 2013. (zitiert auf Seite 3, 23, 24, 25, 42 und 68)

T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake und T. Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Science of Computer Programming (SCP)*, 79:70–85, 2014. (zitiert auf Seite 3, 8 und 51)

K. Tonscheidt. Leveraging Code Clone Detection for the Incremental Migration of Cloned Product Variants to a Software Product Line: An Explorative Study. Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2015. (zitiert auf Seite 4, 16, 17, 26, 27, 28, 29, 51 und 72)

T. Tourwé und T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, Seite 91. IEEE, 2003. (zitiert auf Seite 3)

S. Trujillo, D. Batory und O. Diaz. Feature Refactoring a Multi-representation Program into a Product Line. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, Seiten 191–200. ACM, 2006. (zitiert auf Seite 2)

E. Van Emden und L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Seite 97. IEEE, 2002. (zitiert auf Seite 3)

M. Van Hilst und D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *ISOTAS*, Seiten 22–37. Springer, 1996. (zitiert auf Seite 8)

D. L. Webber und H. Gomaa. Modeling Variability in Software Product Lines with the Variation Point Model. *Science of Computer Programming (SCP)*, 53(3): 305–331, 2004. (zitiert auf Seite 9)

Z. Xing, Y. Xue und S. Jarzabek. CloneDifferentiator: Analyzing Clones by Differentiation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, Seite 576. IEEE, 2011. (zitiert auf Seite 67 und 74)

Y. Xue. Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 1114–1117. ACM, 2011. (zitiert auf Seite 2 und 67)

Y. Xue. *Reengineering Legacy Software Products into Software Product Line*. Doktorarbeit, Nationaluniversität Singapur, 2012. Seite 18 von 20. (zitiert auf Seite 67)

Y. Xue, Z. Xing und S. Jarzabek. Understanding Feature Evolution in a Family of Product Variants. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Seiten 109–118. IEEE, 2010. (zitiert auf Seite 67)

K. Yoshimura, D. Ganesan und D. Muthig. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *Proceedings of the International Workshop on Software Engineering for Automotive Systems (SEAS)*, Seiten 61—-67. ACM, 2006. (zitiert auf Seite 26)

Y. Zhou und H. Leung. Predicting Object-oriented Software Maintainability Using Multivariate Adaptive Regression Splines. *Journal of Systems and Software*, 80 (8):1349–1361, 2007. (zitiert auf Seite 19)

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

Magdeburg, den 08. Dezember 2015