Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Bachelorarbeit

Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung

Autor:

André Weigelt

12. August, 2013

Betreuer:

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

Institut für Technische und Betriebliche Informationssysteme

Dipl.-Inform. Thomas Thüm

Weigelt, André:
Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2013.

Inhaltsangabe

Der Bedarf nach komplexen Software-Systemen ist seit Ende der 1960er Jahre kontinuierlich gestiegen. Feature-orientierte Programmierung ist ein Programmierparadigma, welches die Wiederverwendung von Software-Modulen unterstützt und zudem Variabilität in Software ermöglicht. Dazu werden Features getrennt voneinander entwickelt und bei der Produkterstellung zusammen komponiert. Mittels Feature-orientierter Programmierung kann die Entwicklung komplexer Systeme beherrschbar gemacht werden. Komplexe Software-Systeme sind meist fehleranfällig, sofern sie nicht genau spezifiziert werden. Design By Contract unterstützt bei der Spezifikation von Methoden, indem Vorbedingungen, Nachbedingungen und konsistente Einschränkungen an Methoden annotiert werden. In der Literatur stehen sechs Techniken zur Verfügung, wie Kontrakte in Feature-orientierter Programmierung angewendet und komponiert werden können. Bislang besteht allerdings nur die Möglichkeit genau eine Technik für ein gesamtes Projekt zu bestimmen.

In dieser Arbeit wird die Notwendigkeit gezeigt, die sechs Kompositionstechniken flexibel innerhalb eines Projekts zu verwenden. Dadurch können die Vorteile, wie bessere Erweiterbarkeit oder größere Garantien, der einzelnen Techniken für unterschiedliche Methodenverfeinerungen ausgenutzt werden. Es werden fünf Schlüsselwörter zur Angabe einer Kompositionstechnik definiert und Konzepte zu dessen Verwendung entwickelt. Für eine benutzerfreundliche und fehlerfreie Verwendung der Schlüsselwörter wird eine Werkzeugunterstützung bereitgestellt. Mit der Methoden-basierten Komposition von Kontrakten wird dem Nutzer die Möglichkeit bereitgestellt, die Kompositionstechnik für einzelne Methoden je nach Bedarf für Erweiterbarkeit oder Garantien auszuwählen. Zuvor konnte ausschließlich für ein gesamtes Projekt eine Kompositionstechnik festgelegt werden.

Danksagung

Hiermit möchte ich bei meinen Betreuern Thomas Thüm und Prof. Gunter Saake bedanken, dass sie mir die Möglichkeit gegeben haben diese Bachelorarbeit zu schreiben. Ohne die konstruktive Kritik und hilfreichen Diskussionen von bzw. mit Thomas Thüm wäre die Arbeit nicht in diesem Umfang möglich gewesen. Vielen Dank für den Aufwand und Mühe zur Betreuung meiner Bachelorarbeit.

Besonderer Dank geht an Fabian Benduhn und Jens Meinicke für die Unterstützung bei der Erweiterung von FeatureHouse und FeatureIDE. Ohne die vorherige Arbeit von Fabian Benduhn wäre die Implementierung der Methoden-basiert nicht in dieser Art und Weise möglich gewesen.

Ebenfalls bedanke ich mich bei Sebastian Breß, Wolfram Fenske, Reimar Schröter, Norbert Siegmund und Janet Siegmund für das kritische Feedback zu meiner Arbeit. Dadurch konnten neue Ergebnisse erzielt werden, die ohne diese Hinweise nicht entstanden wären.

Außerdem möchte ich Britta Weigelt, Katrin Pascual Rio und Nico Schreiber für die kritischen Anmerkungen insbesondere zum sprachlichen Stil meiner Bachelorarbeit danken.

Nicht vergessen möchte ich meine Familie und Freunde, die mich während der gesamten Studienzeit immer moralisch unterstützt und mir Rückhalt gegeben haben.

Inhaltsverzeichnis

Abbildungsverzeichnis **xiv**

Tabellenverzeichnis **xv**

1 Einführung **1**

2 Grundlagen **7**
2.1 Modularisierung von Software 7
2.2 Spezifikation von Software 11
2.3 Software-Produktlinien 16

3 Kontrakte in Feature-orientierter Programmierung **19**
3.1 Techniken zur Kontrakt-Komposition 19
3.1.1 Sprachunabhängige Techniken 20
3.1.2 Sprachspezifische Techniken 26
3.1.3 Vergleich der Techniken 31
3.2 Schlüsselwörter für Methoden-basierte Kontraktverfeinerung 34
3.2.1 Schlüsselwörter für sprachunabhängige Techniken 34
3.2.2 Schlüsselwörter für sprachspezifische Techniken 39
3.2.3 Zusammenfassung 40
3.3 Anwendung der Schlüsselwörter 41
3.3.1 Überschreiben von Schlüsselwörtern 41
3.3.2 Rangfolge der Schlüsselwörter 42
3.3.3 Weitere Aspekte 44
3.4 Zusammenfassung 46

4 Implementierung **49**
4.1 Erweiterung von FeatureHouse 50
4.1.1 Erweiterung der JML-Grammatik 51
4.1.2 Erweiterung der Kompositionsregeln 52
4.1.3 Zusammenfassung 60
4.2 Erweiterung von FeatureIDE 60
4.3 Zusammenfassung 62

5 Evaluierung **65**
5.1 Validierung der Implementierung 65
5.2 Evaluierung der Methoden-basierten Komposition 72
5.3 Diskussion 77

5.4 Zusammenfassung 80

6 Verwandte Arbeiten **83**

7 Zusammenfassung **87**

8 Zukünftige Arbeiten **91**

Literaturverzeichnis **93**

Abbildungsverzeichnis

2.1 Ausschnitt des Kollaborationsdiagramms der Graph-Produktlinie . . 10

2.2 Verwendung des Schlüsselworts `original` für die Verfeinerung von Methoden. 11

2.3 Auszug des Feature Structure Trees der Klasse `Edge`. 11

2.4 Auszug der Komposition der Feature Structure Trees der Features *Base* und *Weigthed*. 12

2.5 Auszug der Komposition der Methode `equals` der Klasse `Edge` der Features *Base* und *Weighted*. 13

2.6 Die Methode `equals` mit Kontrollbedingungen zur Laufzeit bzw. mit Kontrakten. Das Schlüsselwort `requires` gibt die Vorbedingung an und `ensures` die Nachbedingung. 15

2.7 Feature-Diagramm der Graph-Produktlinie 17

2.8 Zwei Phasen der Software-Produktlinien Entwicklung 18

3.1 Die Kompositionstechnik Plain Contracting am Beispiel der Features *Connection* und *OptimalConnection*, wobei der angegebene Kontrakt an der Methodeneinführung von `getConnection` für alle Methodenverfeinerungen gilt. 21

3.2 Mit Hilfe des Schlüsselworts `original` kann beim Explicit Contract Refinement auf die vorherige Vor- bzw. Nachbedingung verwiesen werden. Somit kann der Kontrakt verfeinert werden, was hier am Beispiel der Methode `addEdge` der Features *Base* und *MaxEdges* gezeigt wird. 23

3.3 Bei der Kompositionstechnik Contract Overriding ist das Schlüsselwort `original` nicht erlaubt, wodurch leicht Quelltextreplikationen entstehen können. Dies wird beim erneuten Verwenden der Methode `addEdge` der Features *Base* und *MaxEdges* deutlich. 24

3.4 Schematisches Beispiel einer Kontraktkomposition mittels Cumulative Contract Refinement. Vorbedingungen werden dabei mit einer oder-Verknüpfung und Nachbedingungen mit einer und-Verknüpfung zusammengefügt. 25

3.5 Die Methode `equals` der Klasse `Edge` soll mittels Conjunctive Contract Refinement komponiert werden. Die drei Features *Base*, *Weighted* und *Directed* implementieren bzw. verfeinern diese Methode und können zusammen zu einer Variante komponiert werden. 27

3.6 Die Features *Base* und *Directed* werden einmal ohne und einmal mit dem optionalen Feature *Weighted* zu zwei unterschiedlichen Varianten mittels Conjunctive Contract Refinement komponiert. Dabei werden die Vorbedingungen und Nachbedingungen der Features miteinander verknüpft. 28

3.7 Semantisch verschiedene Kontrakte können mit Hilfe von Consecutive Contract Refinement verknüpft werden, indem sie mit dem `also` Operator unabhängig voneinander betrachtet werden, was bei einer Sortierung von Listen bzw. Mengen der Fall sein kann. 29

3.8 Pure Method Refinement Beispiel. 30

3.9 Multiple Preconditions and Postconditions können leicht in einzelne Vorbedingungen bzw. Nachbedingungen konvertiert werden, indem die Vor- und Nachbedingungen mit einer &&-Verknüpfung zusammengeführt werden. Eine Konvertierung in die andere Richtung ist ebenfalls möglich. 31

3.10 Die Methode `equals` der Klasse `Edge` wird, anstatt mit Conjunctive Contract Refinement, nun mit Explicit Contract Refinement verfeinert, indem das Schlüsselwort `original` mit einer &&-Verknüpfung in der Kontraktverfeinerung in der Vor- und Nachbedingung verwendet wird. 33

3.11 Auszug aus dem Kollaborationsdiagramm der Graph-Produktlinie. Für die einzelnen Methoden werden die verwendeten Kompositionstechniken mit unterschiedlichen Farben markiert. Methoden, die nicht markiert sind, besitzen entweder keine Kontrakte oder werden nicht verfeinert. 35

3.12 Selbst wenn das Schlüsselwort `original` bei Explicit Contract Refinement genutzt werden muss, können Methodenkontrakte überschrieben werden. Der Ausdruck `original && !original` ist immer falsch, sodass nur der Teil hinter der oder-Verknüpfung für die Erfüllung der Vor- bzw. Nachbedingung verantwortlich ist. 37

3.13 Beispielhafte Überschreibung der Kompositionstechnik von Kontrakten. Explicit Contract Refinement wird ab Kontrakt c2 mit Consecutive Contract Refinement überschrieben. 42

3.14 Rangfolge der Kompositionstechniken. Explicit Contract Refinement kann von allen Kompositionstechniken überschrieben werden. `final_method` kann nicht überschrieben werden, aber alle anderen überschreiben. 44

4.1 Architektur von FeatureHouse [Ben12]. 50

4.2 Die hier eingeführten Schlüsselwörter werden der Grammatik für die Java Modeling Language hinzugefügt. 51

4.3 FeatureBNF für einen Kontrakt in der Java Modeling Language ohne eigene Schlüsselwörter. 52

4.4 Erweiterung der Grammatik um Schlüsselwörter zur Angabe der Kontraktkomposition. 53

4.5 Ausschnitt aus der Implementierung des Enumerators `CompositionKeyword`, welcher den Rang eines Schlüsselworts enthält. . . . 54

4.6 Ausschnitt aus der Implementierung der Kompositionsregel für die Schlüsselwörter, um diese weiterzureichen und die Überschreibung der Schlüsselwörter auf Basis der Rangfolge zu überprüfen. 55

4.7 Die Klasse für die Kompositionsregel `ContractComposition` wird um die Implementierung der Kompositionstechnik Cumulative Contract Refinement erweitert. Conjunctive Contract Refinement funktioniert analog. 56

4.8 Erweiterung der Klasse für die Kompositionsregel `ContractComposition`, um die Verwendung der Kompositionstechnik auf Basis des angegebenen Schlüsselworts zu entscheiden. 57

4.9 Erweiterung der Klasse `ContractComposition` um das Feld `contractCompKey` zur Speicherung des Schlüsselworts für die Kontraktkompositionstechnik. 58

4.10 Auswahl der Projekt-basierten bzw. Methoden-basierten Komposition von Kontrakten über Konfigurationsparameter. 59

4.11 Auswahl der Kontraktkompositionstechnik in FeatureIDE. Für ein Projekt kann nun die Methoden-basierte Komposition von Kontrakten gewählt werden. 62

4.12 Hinweis bei fehlerhafter Verwendung eines Schlüsselworts zur Angabe der Kompositionstechnik mit Hilfe eines Fehlermarkers in FeatureIDE. 63

4.13 Hinweis bei unzulässiger Überschreibung eines Schlüsselworts zur Angabe der Kompositionstechnik mit Hilfe einer Warnung in FeatureIDE. 63

5.1 Exemplarische Darstellung möglicher Formatierungen für den selben Kontrakt. Der erste Kontrakt enthält mehr Leerzeichen, Tabs und Zeilenumbrüche als der zweite Kontrakt. Beide Kontrakte sind syntaktisch äquivalent. 70

5.2 Vergleich der erzeugten Java-Klasse `integerSet` der Variante *hash* der Fallstudie `IntegerSetSPL`. Die beiden Dateien sind inhaltlich gleich. 70

5.3 Feature-Diagramm der Paycard-Fallustudie. 71

5.4 Ausschnitt aus den Feature-Diagrammen der Union und Graph_old Fallstudie. Die Fallstudien besitzen Alternative Features, auf die für die Evaluierung konkret eingegangen werden musste. 73

5.5 Auszug aus dem Kollaborationsdiagramm der Graph-Produktlinie. Die Kompositionstechniken wurden so ausgewählt, dass größtmögliche Garantien erreicht werden. Für die einzelnen Methoden werden die verwendeten Kompositionstechniken mit unterschiedlichen Farben markiert. Methoden, die nicht markiert sind, besitzen keine Kontrakte. 74

5.6 Darstellung der Häufigkeit der Kompositionstechniken aus allen Fallstudien ohne `final_method`. 78

Tabellenverzeichnis

3.1 Beschreibende Schlüsselwörter für Methoden-basierte Kontraktverfeinerung der Techniken Cumulative, Conjunctive und Consecutive Contract Refinement. Es wird je ein Schlüsselwort für die Vorbedingung und für die Nachbedingung benötigt. 38

3.2 Sprechende Schlüsselwörter für Methoden-basierte Kontraktverfeinerung der Techniken Cumulative, Conjunctive und Consecutive Contract Refinement. 39

3.3 Schlüsselwörter für Methoden-basierte Kontraktverfeinerung. 41

5.1 Übersicht der untersuchten Fallstudien. Die ersten vier wurden von Beginn an als Software-Produktlinie mit Kontrakten entwickelt. Die nächsten sechs wurden aus bestehenden Java Programmen (mit Kontrakten) in Feature-Module dekomponiert. Für die letzten drei Fallstudien wurden Kontrakte zu deren bestehenden Feature-Modulen hinzugefügt [TBAS13]. 66

5.2 Übersicht der untersuchten Fallstudien mit deren bisher verwendeter Kompositionstechnik (ECR = Explicit Contract Refinement, CO = Contract Overriding, Cons. CR = Consecutive Contract Refinement, PC = Plain Contracting), Anzahl der Features, Anzahl der gültigen Varianten und Anzahl Varianten mit paarweiser Featureinteraktion. . 67

5.3 Übersicht der untersuchten Fallstudien mit der Häufigkeit der verwendeten Kompositionstechniken (CR steht für Contract Refinement). Zusätzlich werden die Anzahl aller Methoden, Kontrakte und Überschreibungen einer Kompositionstechnik aufgeführt. *Graph** ist die in dieser Arbeit eingeführte Graph-Produktlinie. 76

1. Einführung

Als Ende der 1960er Jahre der Bedarf nach größeren und komplexeren Software-Systemen auf Grund der sinkenden Hardwarepreise aufkam, geriet man mit bisherigen Programmiertechniken an seine Grenzen [PP04]. Bauer berichtet [Bau06], dass spätestens 1967 von einer *Software-Krise* die Rede war, als das US-Verteidigungsministerium während des kalten Krieges durch fehlerhafte Software bzw. deren verspätete Auslieferung eine Gefährdung der Sicherheit der Vereinigten Staaten sah. In diesem Zusammenhang führte Bauer 1968 erstmals den Begriff der *Softwaretechnik* (engl. software engineering) ein [Bau06, PP04], mit dem die Bedeutsamkeit von Methoden und Werkzeugen zur wirtschaftlichen und industriellen Herstellung von Software-Produkten bewusst gemacht wurde. Neben der ursprünglichen Informatiksicht wird allgemeines Ingenieurwissen, Projektmanagementkompetenz und domänenspezifisches Wissen in die Entwicklung von Software-Systemen einbezogen. Dadurch können Herausforderungen, wie die Sicherstellung der Korrektheit und Zuverlässigkeit, die Spezifikation von Anforderungen und die Komplexität, bewältigt werden [PP04].

Die Komplexität von Software-Systemen kann verringert werden, indem man die Belange des Systems voneinander trennt (engl. *separation of concerns*) [HGD03, TOHS99]. Dabei wird das Software-System in eigenständige *Module* zerlegt (engl. decomposition) [Par72, TOHS99]. Ein Modul repräsentiert eine konkrete Aufgabe der Software [Par72] und beinhaltet die dazugehörigen Prozeduren/Funktionen und Daten [PP04]. Die Implementierung wird hinter einer Schnittstelle versteckt, die ausschließlich das Verhalten des Moduls beschreibt [PP04, HGD03, Par72]. Parnas [Par72] spricht von drei Vorteilen, die durch Modularität erzielt werden können. Zum einen kann die Entwicklungszeit von Software-Systemen verkürzt werden, indem unterschiedliche Entwicklergruppen an verschiedenen Modulen arbeiten, ohne großes Wissen über andere Module zu benötigen. Ebenfalls können Änderungen oder Erweiterungen am System leicht durch eine Änderung bzw. den Austausch einzelner Module erreicht werden, ohne das gesamte System zu beeinflussen. Als dritten Punkt erwartet Parnas ein besseres Systemdesign auf Grund besserer Verständlichkeit. Mit der Aufteilung in Module ist es möglich, verschiedene Belange getrennt zu entwickeln, wiederzuverwenden und nachzuverfolgen [TOHS99].

Als Ausweg aus der oben genannten *Software-Krise* schlägt McIlroy [McI69] die Massenproduktion von Softwarekomponenten vor, welche in Standardsoftware verwendet werden [PBL05]. Genauer legt er seinen Fokus auf Techniken zur Massenproduktion von Softwarekomponenten, wie z.B. Modularität. Im Gegensatz zum Entwickeln einzelner maßgeschneiderter Systeme kann durch das Wiederverwenden bestehender Software(-Module) die Produktivität der Softwareentwicklung erhöht und damit direkt die Entwicklungsdauer und -kosten gesenkt werden [Kar95]. Zudem kann gleichzeitig die Qualität, Zuverlässigkeit, Verständlichkeit, Teamarbeit und Dokumentation verbessert werden [Kar95, PP04].

Die Idee der Wiederverwendbarkeit findet sich im objektorientierten Programmierparadigma wieder. Lewis et al. [LHKS91] zeigen, dass die Produktivität der Softwareentwicklung durch objektorientierte Programmierung mit Wiederverwendung gesteigert wird. Poetzsch-Heffter [PH00] unterscheidet zwischen zwei Varianten der Wiederverwendung. Zum einen im Voraus geplante Software-Module, die in anderen Software-Systemen in der Regel wiederverwendet werden. Zum anderen das Wiederverwenden von Programm-Modulen, die nicht dafür geplant sind und daher Anpassungsaufwand an das Zielsystem erfordern. Entwurfsmuster (engl. *Design Patterns*) bieten Lösungen für bekannte, häufig auftretende Probleme der Wiederverwendung und eignen sich, um Variabilität in Software umzusetzen. Entwurfsmuster sind weit verbreitet, sodass sich die Einarbeitung und das Verständnis des Systems leicht umsetzen lassen [GHJV93]. Das Wiederverwenden von Software-Modulen bringt auf der einen Seite positiven Nutzen mit sich, wenn diese leicht in ein System eingefügt werden können und dadurch Funktionalität hinzufügen. Auf der anderen Seite kann der Aufwand zur Integration den Nutzen übertreffen und hat somit einen negativen Effekt [Kar95].

Ein Ausweg zwischen individueller Software mit hoher Flexibilität, aber zusätzlichem Anpassungsaufwand in der Entwicklung oder Standardsoftware, welche durch Wiederverwendbarkeit kostengünstiger ist, aber dadurch nicht ohne Weiteres an die Bedürfnisse von einzelnen Kunden angepasst werden kann, können Software-Produktlinien sein [PBL05]. Software-Produktlinien sind Software-Systeme, die eine gemeinsame Menge an Features teilen (die *Basis*) und die Anforderungen eines bestimmten Anwendungsgebiets, der sogenannten *Domäne*, erfüllen. Ein Feature ist eine Funktionalität des Systems, die für den Nutzer direkt sichtbar ist [ABKS13]. Die Basis kann mit zusätzlichen Features angepasst werden, um die Bedürfnisse (z.B. erweitertes Verhalten) verschiedener Kunden der Domäne umzusetzen. Heutzutage haben sich Software-Produktlinien teilweise in der Industrie etabliert. Unternehmen, die Software-Produktlinien verwenden, konnten dadurch ihre Produktivität steigern [CGK+, ABKS13]. Klassische Entwicklungstechniken wie z.B. die Entwicklung von Komponenten, welche Features als modulare, abgeschlossene Einheiten implementieren, erfordern einen hohen Aufwand zum Einbinden der Komponenten (bspw. zusätzlicher Quelltext (engl. *Glue Code*)). Außerdem können querschneidende Belange auftreten, also Anforderungen, die sich auf mehrere Features beziehen [ABKS13].

Das Feature-orientierte Programmierparadigma bietet eine Möglichkeit die Grenzen der klassischen Implementierungstechniken aufzulösen. Dabei werden Features in Module aufgeteilt. Diese Module enthalten alle am Feature beteiligten Software-

Artefakte, wie z.B. Klassen, Bilder oder Dokumentation. Mittels Klassenverfeinerungen können Klassen aus unterschiedlichen Modulen angepasst werden. Diese werden bei der Produkterstellung mit bestimmten Werkzeugen automatisch zusammengesetzt [ABKS13].

Die Entwicklung komplexer Software-Systeme erfordert nicht nur entsprechende Programmiermethoden, sondern macht es umso mehr notwendig, die Anforderungen und Bedingungen der Software bzw. Software-Module zu verstehen, zu dokumentieren und im Entwicklerteam zu kommunizieren [AP98, Mey92a]. Eine gründliche Spezifikation kann unerwartetes Verhalten und Fehler vermeiden. Am Beispiel der *Ariane 5* wird deutlich, dass fehlerhaftes Verhalten des Systems enorme Auswirkungen haben kann. Eine (wieder-)verwendende Komponente der Ariane 4 sorgte nach dem Start für eine falsche Berechnung in der Ariane 5, woraufhin die Ariane 5 explodierte. Ein Schaden von ca. 500 Millionen Dollar und eine Verzögerung um mehrere Jahre waren die Folge [JM97]. Das Verhalten der verwendeten Software-Komponente war im Vorfeld bekannt, wurde jedoch nicht bei der Entwicklung berücksichtigt.

Durch *Design by Contract* ist es möglich Anforderungen, Bedingungen, Einschränkungen und erwartetes Verhalten von Software-Modulen formal als Kontrakte anzugeben. Dadurch kann die Zuverlässigkeit, Verständlichkeit und Modularität [LC06] der Software schon bei der Entwicklung sichergestellt werden, indem Schnittstellen spezifiziert, dokumentiert und gegebenenfalls verifiziert werden. Nach Meyer ist Zuverlässigkeit, d.h. das korrekte Verhalten und Robustheit gegen Fehler, insbesondere bei sicherheitskritischen objektorientierten Programmen, äußerst wichtig [Mey92a]. Kontrakte bestehen aus spezifizierten Vorbedingungen (engl. *precondition*), die vor dem Methodenaufruf durch den Aufrufer erfüllt werden müssen. Im Gegenzug kann sich der Nutzer der Methode darauf verlassen, dass ein erwartetes Verhalten der Methode eingehalten wird, d.h. spezifizierte Nachbedingungen (engl. *postcondition*) erfüllt werden. Weiterhin müssen konsistente Einschränkungen (engl. *invariants*) gewährleistet sein [JM97, LC06]. Vorbedingungen, Nachbedingungen und Einschränkungen werden zusammen als *Assertions* (dt. Behauptung) bezeichnet [LC06]. Die Programmiersprache Eiffel [Mey92b] oder Spec# [BFL+11] bieten native Unterstützung, um Assertions im Quelltext zu schreiben. Die Java Modeling Language (JML) erweitert die Programmiersprache Java um Sprachkonstrukte, welche die Verwendung von Design by Contract ermöglichen [LC06].

Wie oben beschrieben können Software-Produktlinien den Implementierungsaufwand verringern und kann Design by Contract die Qualität von Software verbessern. Für eine Verifikation von Software-Produktlinien muss das Verhalten der Software-Produkte spezifiziert sein. Dazu ist es notwendig bisherige Spezifikationstechniken auf Software-Produktlinien anzupassen. Thüm et al. [TAK+12] stellen fünf Techniken zur Spezifikation von Software-Produktlinien vor. Beim *Produkt-basierten* Spezifizieren werden alle Produkte der Software-Produktlinie einzeln spezifiziert. Dieses Verfahren skaliert nur für wenige Produkte, weshalb als Verbesserung vorgeschlagen wird, dass nur Produkte spezifiziert werden, die tatsächlich genutzt werden [TAK+12]. Als zweiter Ansatz wird die *Feature-basierte* Spezifikation vorgestellt, bei der das Verhalten jedes Features in Isolation und ohne Bezug auf andere Feature angegeben wird. Die *Familien-basierte* Spezifikation fordert Kenntnis über alle Features im Voraus. Dabei wird die gesamte Software-Produktlinie

spezifiziert, wobei mittels aussagenlogischer Ausdrücke auf einzelne Features bzw. Feature-Kombinationen eingegangen werden kann. Als vierte Spezifikationstechnik zeigen die Autoren [TAK+12] die *globale Spezifikation*. Eine globale Spezifikation muss von allen Produkten der Software-Produktlinie erfüllt werden. Die bisherigen Spezifikationstechniken beziehen sich auf Domänen-spezifische Softwareprodukte. In einigen Fällen kann es notwendig sein, Domänen-übergreifende Spezifikationen anzugeben. Dies kann mit der *Domänen-unabhängigen* Spezifikation umgesetzt werden, welche sich auf eine Programmiersprache anstatt auf eine Software-Produktlinie bezieht [TAK+12].

Die traditionellen Konzepte für Design by Contract richten sich speziell an objektorientierte Programmiertechniken. Diese Konzepte auf Feature-orientierte Programmierung anzuwenden, ist direkt nicht möglich, weil dabei ebenfalls auch Kontrakte verfeinert werden können [TSK+12]. Thüm et al. [TSK+12] stellen fünf Möglichkeiten vor, wie man Kontrakte in Feature-orientierter Programmierung verfeinern kann. Benduhn [Ben12] hat diese fünf um zwei weitere Möglichkeiten erweitert und auf Software-Produktlinien angewendet [Thü]. Die sieben Ansätze können nur auf Projektebene angegeben werden und nicht auf Methodenebene. Dadurch entstehen Herausforderungen bei der Projekterstellung, weil entschieden werden muss, welche Art der Kontraktverfeinerung angewendet werden soll. So wird beim *Plain Contracting* eine Spezifikation angegeben, die für alle Methodenverfeinerungen gelten muss. Im Gegensatz dazu kann für jede verfeinerte Methode eine Vor- und Nachbedingung spezifiziert werden, die andere Spezifikationen überschreibt [Ben12].

Zielstellung der Arbeit

In dieser Arbeit soll untersucht werden, ob es notwendig ist eine Kompositionstechnik für Kontrakte nicht nur auf Projektebene anzugeben, sondern für jede Methode eine unterschiedliche Kompositionstechnik zu wählen. Für die Angabe der Kompositionstechnik auf Methoden-Ebene werden spezielle Schlüsselwörter benötigt. Diese Schlüsselwörter sollen im Folgenden konzeptuell entwickelt werden. Weiterhin soll die Komposition der Kontrakte anhand der Schlüsselwörter automatisiert werden. Dazu soll das Kompositionswerkzeug *FeatureHouse* [AKL13] und die Entwicklungsumgebung *FeatureIDE* [TKB+13] um die Verwendung der Methoden-basierten Komposition von Kontrakten erweitert werden. Die Werkzeugunterstützung soll anhand von bestehenden Software-Produktlinien, für die Kontrakte mit jeweils genau einer Kompositionstechnik verwendet wurde [Thü], getestet werden. Anschließend soll herausgestellt werden, wie häufig welche Schlüsselwörter verwendet wurden und ob tatsächlich alle Kompositionstechniken notwendig sind. Dafür soll zusätzlich eine für die Methoden-basierte Komposition von Kontrakten entwickelte Produktlinie betrachtet werden.

Gliederung der Arbeit

Die erforderlichen Grundlagen zum Verständnis dieser Arbeit werden in Kapitel 2 gegeben. In Kapitel 3 wird die Notwendigkeit der Methoden-basierten Kontraktkomposition anhand von Beispielen vorgestellt. Die Schlüsselwörter und das Vorgehen zur Anwendung in Feature-orientierter Programmierung wird konzeptionell diskutiert. Anschließend wird in Kapitel 4 die Werkzeugunterstützung implementiert. Die

Evaluierung der Ergebnisse wird in Kapitel 5 anhand verschiedener Software-Produktlinien besprochen. Zusätzlich werden verwandte Arbeiten in Kapitel 6 vorgestellt. Schlussendlich fasst das Kapitel 7 diese Arbeit zusammen und in Kapitel 8 wird auf zukünftige Arbeiten hingewiesen.

2. Grundlagen

In diesem Kapitel werden die nötigen Grundlagen für das Verständnis dieser Bachelorarbeit vorgestellt. Der Abschnitt 2.1 motiviert die modulare Entwicklung von Software und geht auf Techniken zur Implementierung ein, insbesondere auf Feature-orientierte Programmierung. Spezifikationstechniken für Software werden in Abschnitt 2.2 auf Seite 11 erläutert. Dabei wird genauer auf die sichere Programmierung mit Kontrakten eingegangen. Der letzte Abschnitt 2.3 auf Seite 16 erklärt was Software-Produktlinien sind und wie diese implementiert und spezifiziert werden können.

2.1 Modularisierung von Software

Parnas prägt 1972 erstmals den Begriff Modularität [Par72] mit dem Ziel die Komplexität von Software zu verringern. Unter Modularität werden zwei grundlegende Prinzipien für den Entwurf und die Entwicklung von Software zusammengefasst: *Kapselung* und *Dekomposition* [Par72, SDW+10, SGCH01]. Unter Kapselung versteht man das Verstecken von Implementierungsdetails (engl. *information hiding*) hinter einer *Schnittstelle* (engl. *Interface*), welche zur Kommunikation im Gesamtsystem dient [Par72, SDW+10] und die Funktionalität beschreibt. Dekomposition beschreibt das Zerlegen eines Software-Systems in kleine, überschaubare Teile (*Module*) [Par72, SDW+10, TOHS99]. Dadurch können Belange voneinander getrennt werden (engl. *separation of concerns*) [TOHS99], was zur Verringerung der Komplexität führt [Par72, SDW+10, TOHS99]. Nach Parnas [Par72] ergeben sich drei wesentliche Vorteile durch die Modularisierung von Software-Systemen:

- Parallele Entwicklung - Die Entwicklung von Software kann effizienter durchgeführt werden, weil verschiedene Entwicklergruppen separat Programmteile entwickeln und testen können, ohne Kenntnis über andere Module zu benötigen.
- Flexibilität - Änderungen einzelner Software-Module können variabel vorgenommen werden, indem Module verändert oder durch andere Module ausgetauscht werden, sofern sie dieselbe Schnittstelle implementieren. Dabei muss

ausschließlich an Modulen, die konkret an der Änderung beteiligt sind, gearbeitet werden. Andere Programmteile des Software-Systems müssen nicht angepasst werden.

- Verständlichkeit - Die kleinen Module des gesamten Software-Systems sind leichter zu lernen und zu verstehen, da sie unabhängig voneinander und abgeschlossen sind. Aus dem verbesserten Verständnis wird ein besseres Design für das Software-System erwartet.

Durch die von Parnas genannten Vorteile verbessert sich zusätzlich die Wartbarkeit und Evolution vorhandener Software [TOHS99]. Einzelne Module können in anderen Programmen wiederverwendet werden [Kar95, PP04], wodurch die Qualität von Software gesteigert wird und die Kosten für die Entwicklung gesenkt werden.

Die Prinzipien der Modularität werden im objektorientierten Programmierparadigma umgesetzt. Module werden in Form von *Klassen* repräsentiert, welche ein Grundgerüst für *Objekte* bereitstellen. Objekte bilden Daten und Methoden (Funktionen) ab und können einen internen Status besitzen [SB86]. Methoden einzelner Objekte können von anderen Objekten aufgerufen werden, indem eine Aufforderung in Form einer Nachricht an das jeweilige Objekt gesendet wird [SB86]. Durch die Verwendung von Schnittstellen können Details der Implementierung versteckt werden, sodass Objekte miteinander kommunizieren können, ohne bestimmte Annahmen machen zu müssen [SDW+10, SB86]. Eine weitere Eigenschaft der objektorientierten Programmierung ist die Möglichkeit Objekte zu spezialisieren. Objekte, die Ähnlichkeiten zu anderen Objekten haben, sich aber um bestimmtes Verhalten unterscheiden, können durch Vererbung ihre eigene Funktionalität umsetzen [SB86], ohne redundanten Quelltext schreiben zu müssen. Solche Subklassen können Verhalten bestehender Klassen implementieren, um vorhandene Funktionen wiederzuverwenden. Das Konzept der *Polymorphie* (Vielseitigkeit) ermöglicht es, eine Variable mit unterschiedlichen Typen von (Sub-)Klassen zu initialisieren, wodurch verschiedene Implementierungen ausgenutzt werden können.

Die Herausforderung, Belange im objektorientierten Paradigma zu trennen, liegt darin, dass Belange nur entlang einer Dimension modularisiert werden können (*Tyrannei der dominanten Dekomposition*) [TOHS99, Ari07]. Daraus folgt, dass andere Belange querschneidend über mehrere Module verteilt sind. Der Quelltext eines *querschneidenden Belangs* (engl. *crosscutting concern*) ist im Quelltext anderer Belange enthalten (engl. *code scattering*). Verteilter Quelltext führt dazu, dass mehrere Belange innerhalb eines Moduls implementiert sind (engl. *code tangling*), was dem Gedanken von Parnas zur Verbesserung der Verständlichkeit und Evolution von Software entgegenwirkt [TOHS99, Ari07]. Änderungen eines querschneidenden Belangs können weitreichende Folgen am gesamten Software-System haben, was sich zusätzlich auf die Wiederverwendung auswirkt. Querschneidende Belange finden sich meist in größeren Software-Komponenten mit speziellem Funktionsumfang. Die Wiederverwendung solcher Komponenten wird gehemmt, weil höherer Anpassungsaufwand an die eigene Anwendung erforderlich ist [SB02]. Aus diesem Grund werden neue Techniken für modulare Implementierungen mit hohem Grad an Wiederverwendung benötigt.

Feature-orientierte Programmierung ist eine Erweiterung des objektorientierten Paradigmas [Pre97], mit der querschneidende Belange modular implementiert werden können. Ein Feature ist dabei ein Funktionsmerkmal, was für den Nutzer der Software sichtbar ist [ALB+07]. Ein Modul beinhaltet idealerweise ausschließlich Software-Artefakte eines Features, sodass eine direkte Zuordnung von Features auf Module stattfindet. Diese Feature-Module ermöglichen eine bessere Nachvollziehbarkeit und Verständlichkeit der Funktionsweise eines Features, wodurch die Fehlerbehebung verbessert wird [ABKS13]. In jedem Feature-Modul sind alle Klassen, welche an der Implementierung eines Feature beteiligt sind, enthalten. Diese Menge an Klassen wird als *Kollaboration* bezeichnet. Häufig werden mehrere Features von einer Klasse implementiert, d.h. in verschiedenen Kollaborationen können die gleichen Klassen beteiligt sein, wie im Kollaborationsdiagramm der Graph-Produktlinie in Abbildung 2.1 auf der nächsten Seite zu sehen ist. Dort wird bspw. die Klasse `Edge` von den Features *Base*, *Weighted* und *Directed* implementiert. Der Quelltext für ein bestimmtes Feature in einer Klasse wird als *Rolle* bezeichnet [ABKS13, SB02]. Mittels Rollen ist es möglich, bestehende Klassen zu verfeinern, d.h. neue Klassenvariablen und Methoden hinzuzufügen oder bestehende Methoden zu überschreiben oder erweitern. Das Feature *Weighted* hat eine Rolle in der Klasse `Edge`. Es verfeinert auf der einen Seite die Methode `equals` und fügt auf der anderen Seite die Klassenvariable `weight` hinzu. Wie in der objektorientierten Programmierung können also bestehende Methoden erweitert oder überschrieben. Mit dem Schlüsselwort `Super` [Bat06] bzw. `original` [AKL13] ist es möglich bestehende Methoden in Verfeinerungen aufzurufen, wie in der Methodenverfeinerung der Methode `equals` der Klasse `Edge` im Feature *Weighted* in Abbildung 2.2 auf Seite 11 zu sehen ist. Ob das Schlüsselwort `Super` oder `original` genutzt wird, hängt von dem verwendeten Kompositionswerkzeug ab. In dieser Arbeit wird für die Feature-orientierte Programmierung *FeatureHouse* [AKL13] genutzt, bei dem `original` als Schlüsselwort für das Verfeinern von Methoden verwendet wird.

Wie zuvor erwähnt handelt es sich bei der Feature-orientierten Programmierung um einen Kompositions-basierten Ansatz. Komposition bezeichnet „den Bausteinartigen Zusammenbau von vorgefertigten Softwareeinheiten“ [Gri01], d.h. die bei der Dekomposition entstandenen Feature-Module werden wieder zu einer Softwareeinheit *komponiert*. Die Komposition in FeatureHouse findet mittels „*Superimposition*“ [ALB+07] von *Feature Structure Trees* statt. Feature Structure Trees sind hierarchische Strukturen zur Repräsentation der grundlegenden modularen Struktur von Software-Artefakten [AKL13]. Die Strukturen solcher Software-Artefakte sind dabei von der verwendeten Sprache bzw. der Art des Artefakts abhängig. In Java kann die Hierarchie beginnend bei Paketen (engl. *package*) über Klassen bis hin zu Methoden oder Feldern abgebildet werden. In Abbildung 2.3 auf Seite 11 ist ein Auszug des Feature Structure Trees der Klasse `Edge` abgebildet.

Bei der Komposition mittels Superimposition werden die Software-Artefakte der Kollaborationen eines Features als Feature Structure Tree dargestellt und zusammengefügt, was in Abbildung 2.4 auf Seite 12 zu sehen ist. Die Features werden nach einer globalen Reihenfolge, die im Vorfeld angegeben wird, komponiert. Dabei werden die Knoten rekursiv zusammengefasst, wenn der Typ und Name gleich sind. Unterschieden wird zwischen Terminalknoten und Nicht-Terminalknoten. Nicht-Ter-

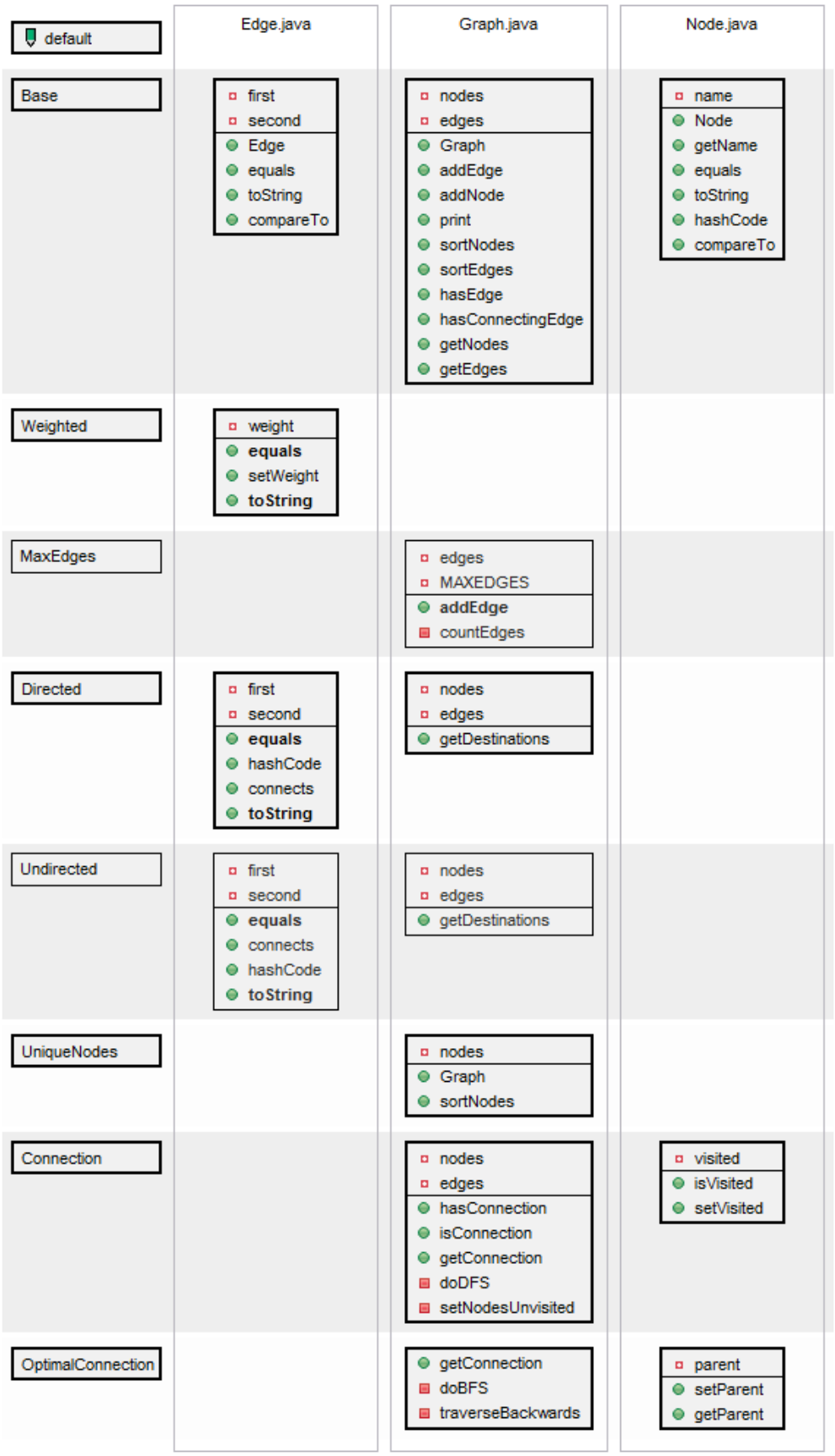

Abbildung 2.1: Ausschnitt des Kollaborationsdiagramms der Graph-Produktlinie

```
class Edge {                                                    Weighted
    private Integer weight = 0;

    public boolean equals(Object ob) {
        if (original(ob) && weight == ((Edge)ob).weight) {
            return true;
        }
        return false;
    }
}
```

Abbildung 2.2: Verwendung des Schlüsselworts `original` für die Verfeinerung von Methoden.

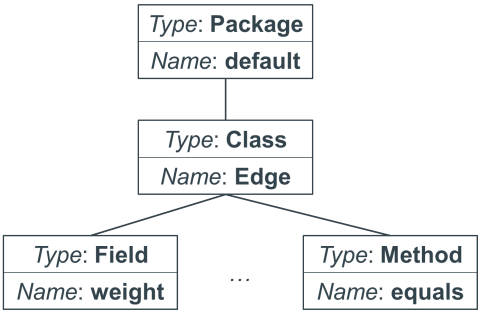

Abbildung 2.3: Auszug des Feature Structure Trees der Klasse `Edge`.

minalknoten sind Knoten, die weitere Kinder haben (im Allgemeinen Paket oder Klasse), wobei Terminalknoten keine weiteren Kinder besitzen (z.B. Felder oder Methoden). Nicht-Terminalknoten werden zusammengefügt, indem ihre Kindknoten überlagert werden. Knoten, die keine Partner in anderen Feature Structure Trees haben, werden einfach an dieselbe Stelle kopiert [Ben12]. Formal wird die Komposition mit dem $\bullet$ Operator angegeben [BSR04, ALB+07]:

$$\bullet : F \times F \rightarrow F \tag{2.1}$$

Daraus folgt, dass eine Komposition von Features selbst wieder ein Feature ist. D.h. ein aus aus n Features erzeugtes Produkt wird ebenfalls als Feature betrachtet [ALB+07]:

$$p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1 \tag{2.2}$$

Weiterhin ist zu beachten, dass die Komposition assoziativ, aber nicht kommutativ ist [ALB+07]. D.h. es gibt eine Reihenfolge, in der die Features miteinander komponiert werden. Diese Reihenfolge ist ausschlaggebend für das entstehende Produkt. Dabei beginnt die Komposition bei dem ersten Feature f_1 und endet beim letzten Feature f_n. Eine Komposition des Quelltext der Methode `equals` aus der Klasse `Edge` der Features *Base* und *Weighted* ist in Abbildung 2.5 auf Seite 13 zu sehen.

2.2 Spezifikation von Software

Wie in Abschnitt 2.1 beschrieben, hilft die Dekomposition von Software in Module die Komplexität zu senken, die Verständlichkeit der Software zu verbessern und

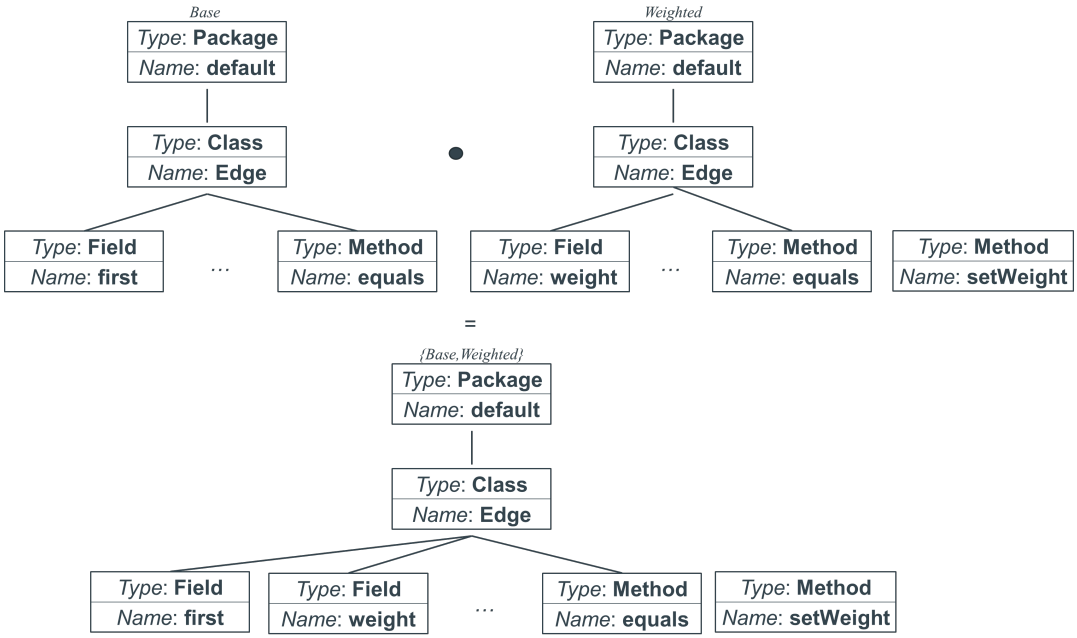

Abbildung 2.4: Auszug der Komposition der Feature Structure Trees der Features *Base* und *Weigthed*.

die Produktivität der Softwareentwicklung zu steigern [AP98]. Allerdings genügt das nicht, um fehlerfreie Software zu gewährleisten, was das Beispiel der Ariane 5 verdeutlicht [JM97]. Eine Herausforderung ist es zu verstehen, wie Module zu verwenden sind und welches Verhalten bzw. welche Funktion geliefert wird, ohne Verständnis über die Implementierung zu besitzen.

Softwarespezifikation bezeichnet nach Alagar et al. [AP98] „die genaue Beschreibung der Objekte im System, die Menge der Methoden mit denen Objekte manipuliert werden und die Angabe über ihr gemeinsames Verhalten für die Dauer ihrer Existenz im System“. Unterschieden wird zwischen *informaler* und *formaler* Spezifikation. Informale Spezifikation bezieht sich dabei auf eine verbale, natürlichsprachliche Beschreibung des Software-Systems, ohne jegliche Vorgaben hinsichtlich Syntax oder Sprache. Im Gegensatz dazu ist formale Spezifikation „eine mathematisch-basierte Sprache, mit wohldefinierter Syntax und Semantik“ [AP98]. Meyer [Mey85] bewertet die Vorteile formaler Spezifikation stärker als die der informalen Spezifikationen, weil die formale Spezifikation Grammatiken besitzt, die eindeutig sind und auf der ganzen Welt verstanden werden. Ferner können Spezifikationen auch als Test-Orakel für das automatische Erstellen von Testfällen genutzt werden. Zusätzlich können Verifikationstechniken wie *Model-Checking* und *Theorem Proving* [TAK+12] auf formale Spezifikationen angewendet werden. Dies hat den Vorteil, dass die Korrektheit der Software schon während der Entwicklung automatisch überprüft werden kann und die aufwändige Erstellung von Testfällen während der Testphase nicht manuell getätigt werden muss.

Design by Contract ist eine Technik um objektorientierte Software zu entwickeln [Mey92a, LC06] und sowohl formale als auch informale Spezifikationen anzugeben. Die Idee bei Design by Contract ist, dass eine Methode und dessen Nutzer einen

```
public class Edge{                                                    Base
    private Node first, second;

    public boolean equals(Object ob) {
        return (ob instanceof Edge) ? true : false;
    }
}
```

```
public class Edge {                                                Weighted
    private Integer weight = 0;

    public boolean equals(Object ob) {
        if (original(ob) && weight == ((Edge)ob).weight) {
            return true;
        }
        return false;
    }
}
```

```
public class Edge {                                       Weighted • Base
    private Node first, second;
    private Integer weight = 0;

    public boolean equals(Object ob) {
        if ((ob instanceof Edge) && weight == ((Edge)ob).weight) {
            return true;
        }
        return false;
    }
}
```

Abbildung 2.5: Auszug der Komposition der Methode `equals` der Klasse `Edge` der Features *Base* und *Weighted*.

Kontrakt (engl. *contract*) abschließen, bei dem der Nutzer der Methode vor Methodenausführung spezifizierte Vorbedingungen (engl. *precondition*) erfüllen muss, damit die Methode ausgeführt werden kann. Gleichzeitig kann der Nutzer davon ausgehen, dass die Methode ein erwartetes Verhalten aufweist. Die Methode selbst ist verpflichtet spezifizierte Nachbedingungen (engl. *postcondition*) zu erfüllen, was bedeutet, dass ein bestimmtes Verhalten oder Zustände nach der Methodenausführung gelten [Mey92a, LC06]. Zusätzlich können konsistente Einschränkungen, sogenannte *Klasseninvarianten*, angegeben werden, die bei jedem Zustand eines Objekts und während jeder Methodenausführung erhalten bleiben müssen [Mey92a]. Zusammengefasst werden Vorbedingungen, Nachbedingungen und Klasseninvarianten als *Assertions* bezeichnet. Programmiersprachen wie Eiffel [Mey92b] oder Spec# [BFL+11] bieten eine native Unterstützung für Design by Contract. Für Java gibt es die Spracherweiterung *Java Modeling Language* (JML) [LC06], welche in dieser Arbeit als Werkzeug für Spezifikationen in Feature-orientierter Programmierung dient. Angelehnt an Leavens et al. [LC06] bringt die Verwendung von Design by Contract folgende Vorteile mit sich:

- Dokumentation - Design by Contract bietet eine bessere Dokumentation für Software-Module als reine Quelltextkommentare oder Javadoc [Ora], weil für jede Methode bzw. Schnittstelle eine abstrakte und formale Spezifikation der Bedingungen und des erwarteten Verhaltens angegeben wird.

- Verifikation - Formale Spezifikationen können automatisch kontrolliert werden und helfen bei der Fehlerbehebung. Aus Spezifikationen können Testfälle generiert werden, anhand derer die Korrektheit von Software überprüft werden kann. Ebenfalls können Verifikationsverfahren wie Theorem Proving oder statische Verifikation auf Spezifikationen angewendet werden.

- Schuldzuweisung - Bei fehlerhaften Programmen kann leicht nachvollzogen werden, an welcher Stelle der Fehler auftaucht. So kann entweder der Entwickler oder der Aufrufer der Methode zur Nachbesserung herangezogen werden, wenn er den Kontrakt nicht erfüllt.

- Effizienz - Kontrollbedingungen während der Ausführungszeit können Programmabläufe stark verlangsamen. Durch Design by Contract können Bedingungen an die Methode ebenfalls im Quelltext deklariert werden. Im Gegensatz zu Kontrollbedingungen muss dabei jedoch nicht die Implementierung verändert werden. Kontrakte lassen sich einfacher aus Produktivsystemen entfernen, sodass die Performanz der Programme gesteigert werden kann. Außerdem können Bedingungen, die nicht zur Laufzeit geprüft werden müssen, im Vorfeld durch statische Verifikation überprüft werden. Ein Beispiel, wie Kontrollbedingungen durch Kontrakte ersetzt werden können, ist in Abbildung 2.6 auf der nächsten Seite zu sehen.

- modulares Verständnis - Die Spezifikation und somit die Beschreibung des Software-Systems wird auf Methodenebene festgelegt, da kleine Spezifikationen einfacher zu verstehen sind. Somit kann das System Schrittweise verstanden werden, indem erst die einzelnen Software-Module verstanden werden. An-

schließend kann das Verständnis für die Interaktion von Modulen gebildet werden, woraus sich iterativ die Funktionsweise des kompletten Software-Systems erschließt.

```
public class Edge{                                  mit Kontrollbedingungen
    public boolean equals(Object ob) {
        if(ob != null && (ob instanceof Edge)) {
            return true;
        return false
    }
}
```

```
public class Edge{                                          mit Kontrakt
    /*@ requires ob != null;
      @ ensures \result ==> ob instanceof Edge;
      @*/
    public boolean equals(Object ob) {
        return (ob instanceof Edge) ? true : false;
    }
}
```

Abbildung 2.6: Die Methode `equals` mit Kontrollbedingungen zur Laufzeit bzw. mit Kontrakten. Das Schlüsselwort `requires` gibt die Vorbedingung an und `ensures` die Nachbedingung.

Zur formalen Beschreibung von Kontrakten wird in dieser Arbeit das *Hoare-Kalkül* verwendet [Hoa69]. Die Formel $\{\phi\}\, m\, \{\psi\}$ aus dem Hoare-Kalkül sagt aus, dass vor der Ausführung der Methode m die Vorbedingung ϕ und nach der Ausführung der Methode die Nachbedingung ψ erfüllt sein muss. Ein Kontrakt c wird also als $c = \{\phi\}\, m\, \{\psi\}$ angegeben.

Hatcliff et al. [HLL+12] stellen die Schwierigkeiten für Spezifikationen in objektorientierten Programmen heraus. Insbesondere Vererbung bzw. Polymorphie, welche in der Objektorientierung einen großen Nutzen haben, bringen einige Herausforderungen bei der Entwicklung mit Design by Contract mit sich. Die Möglichkeit, dass Variablen mit verschiedenen Typen, also allen möglichen (Sub-)Klassen, belegt werden können, benötigt Konzepte, wie in diesen Fällen mit den Kontrakten der Superklassen verfahren wird. *Behavorial subtyping* beschreibt den Ansatz, dass Bedingungen des Verhaltens von Superklassen der Subklasse auferlegt werden, wodurch Probleme der statischen Verifikation gelöst werden können [HLL+12]. D.h. die Spezifikation einer Klasse wird auch auf deren Subklassen übertragen, damit die möglichen Belegungen einer Variable bzw. dessen Verhalten nicht je Subklasse explizit definiert werden müssen und somit keine Typ-Fehler entstehen. Formal bedeutet dies für ein Supertyp A und den Subtyp B mit der Methode m:

$$c_A = \{\phi_A\} m \{\psi_A\} \tag{2.3}$$

$$c_B = \{\phi_A \rightarrow \phi_B\} m \{\psi_B \rightarrow \psi_A\} \tag{2.4}$$

Um die Anwendung von Behavorial Suptyping zu erleichtern, nutzen viele Spezifikationssprachen, wie auch die Java Modeling Language, *Specification Inheritance* [HLL+12]. Dabei werden die Spezifikationen der Superklassen an die Subklassen vererbt. Die Subklassen haben zusätzlich die Möglichkeit weitere Spezifikationen hinzuzufügen. Um die Ausführung der Methode für den Aufrufer zu erleichtern, werden die Vorbedingungen der Super- und Subklassen mit einer Disjunktion verknüpft, sodass es genügt, dass eine der Vorbedingungen erfüllt wird. Die Nachbedingungen der Super- und Subklassen werden zusammen konjugiert, wobei die Nachbedingung der Subklasse von der Erfüllung dessen Vorbedingung abhängt. Dazu muss ein Operator eingefügt werden, der den Zustand der Vorbedingung festhält, damit diese in der Nachbedingung verwendet werden kann. Dies geschieht mit Hilfe des `old` Schlüsselworts:

$$c = \{\phi_A \vee \phi_B\} m \{(\backslash old(\phi_A) \rightarrow \psi_A) \wedge (\backslash old(\phi_B) \rightarrow \psi_B)\} \tag{2.5}$$

Die Umsetzung in der Java Modeling Language erfolgt mit dem Schlüsselwort `also`, welches mit dieser Formel (2.5) definiert ist [RL01].

2.3 Software-Produktlinien

Das Ziel der Wiederverwendung von Software-Artefakten und die Notwendigkeit der *individualisierten Massenproduktion (engl. mass customization)* bzw. Variabilität von Software-Systemen wird in Software-Produktlinien umgesetzt [ABKS13]. Software-Produktlinien erhöhen die Produktivität und die Qualität von Software-Systemen [SPK06, SM12]. Eine Software-Produktlinie ist nach Apel et al. [ABKS13] eine Menge an Software-intensiven Systemen, die sich eine gemeinsam verwaltete Menge an Features teilen. Diese Features erfüllen die speziellen Bedürfnisse eines bestimmten Marktsegments (*Domäne*). Ein Feature ist als charakteristische Funktionalität des Software-Systems definiert, mit der Produkte bzw. Varianten der Software-Produktlinie unterschieden werden. Mit Hilfe von Features wird Variabilität und Wiederverwendung ermöglicht.

Zur Beschreibung der Variabilität der Software-Produktlinie werden *Feature-Modelle* verwendet [ABKS13]. Ein Feature-Modell beinhaltet alle Features und deren Beziehungen. Für die Kommunikation mit den Kunden der Domäne genügt es die Features zu abstrahieren, d.h. ausschließlich den Namen der Features im Modell anzugeben. Oftmals können nicht alle Features frei miteinander kombiniert werden, weil sie ggf. gegenseitig in der Funktion ausschließen oder ein Feature ein anderes benötigt. Diese Einschränkungen werden ebenfalls im Feature-Modell dargestellt. Eine Variante ist eine Kombination von Features, die im Feature-Modell ausgewählt wird, um ein Produkt zu erstellen. Dabei ist darauf zu achten, dass es sich um eine gültige Variante handelt, d.h. die im Feature-Modell angegeben Einschränkungen müssen eingehalten werden. Eine mögliche Form des Feature-Modells sind *Feature-Diagramme* [ABKS13]. Feature-Diagramme stellen Features in einer hierarchischen Form dar, in der die Namen und deren Abhängigkeiten bzw. Bedingungen in Form von logischen Ausdrücken an andere Features in Verbindung gebracht werden. Abbildung 2.7 auf der nächsten Seite zeigt ein Feature-Diagramm der Graph-Produktlinie.

Dabei sind vier verschiedene Operatoren zu sehen. *Mandatory* bedeutet, dass immer, wenn der Elternknoten gewählt wird, zwangsweise auch der Kindknoten ausgewählt werden muss. *Optional* ermöglicht den Kindknoten auszuwählen, was aber nicht zwingend ist. Der *Oder*-Operator ermöglicht es mindestens ein Feature auszuwählen. Es können auch mehrere sein. Eine *Alternative* hat die Bedeutung eines exklusiven Oder, bei dem genau ein Kindknoten gewählt werden muss. Als letztes ist neben den hierarchischen Beziehungen eine Bedingung in Form eines logischen Ausdrucks zu sehen, was als *Constraint* bezeichnet wird [ABKS13].

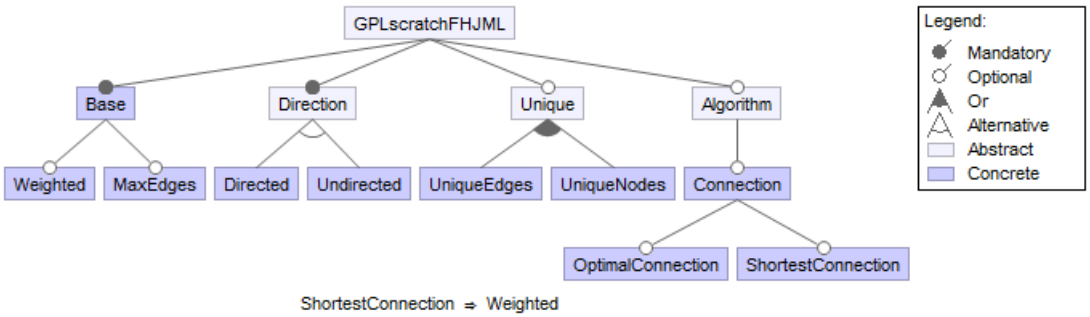

Abbildung 2.7: Feature-Diagramm der Graph-Produktlinie

Die Entwicklung von Software-Produktlinien unterscheidet sich von der Entwicklung einzelner Software-Produkte unter anderem am Entwicklungsmodell. Die Software-Produktlinienentwicklung besteht aus zwei Entwicklungsphasen, wie in Abbildung 2.8 auf der nächsten Seite zu sehen ist. Beim *Domain Engineering* [Kna01], der Entwicklung für die Wiederverwendung, werden die Bedürfnisse der Domäne analysiert, modelliert und implementiert. D.h. es werden Software-Artefakte entwickelt, die später in konkreten Anwendungen (wieder-)verwendet werden. Das *Application Engineering* [Kna01], das Entwickeln mit Wiederverwendung, nimmt die Anforderungen konkreter Kunden auf, modelliert diese Anforderungen, fügt gegebenenfalls nicht implementierte Funktionalität hinzu und generiert schlussendlich aus den Features der Domäne ein maßgeschneidertes Software-System für den Kunden [ABKS13]. Feature-orientierte Programmierung ist eine Möglichkeit, um Software-Produktlinien zu entwickeln. Dabei werden die Features in Feature-Modulen modularisiert. Diese Feature-Module werden im Feature-Modell repräsentiert und bei Auswahl miteinander komponiert.

Wie in der objektorientierten Programmierung ist auch bei der Entwicklung von Software-Produktlinien die Korrektheit der Software eine wichtige Aufgabe. Allerdings ist die Spezifikation von Software-Produktlinien herausfordernd, weil traditionelle Techniken nicht direkt angewendet werden können [TAK+12]. Software-Produktlinien beinhalten ggf. viele Produktvarianten, die exponentiell mit der Anzahl an Features steigen kann. Dennoch soll auch bei der Spezifikation der Vorteil der Wiederverwendbarkeit und Variabilität von Software-Produktlinien ausgenutzt werden. Thüm et al. [TAK+12] stellen fünf Strategien zur Spezifikation von Software-Produktlinien vor:

- Produkt-basierte Spezifikation - Jedes Software-Produkt wird eigenständig spezifiziert. Für eine große Anzahl an Varianten ist der Aufwand dabei sehr

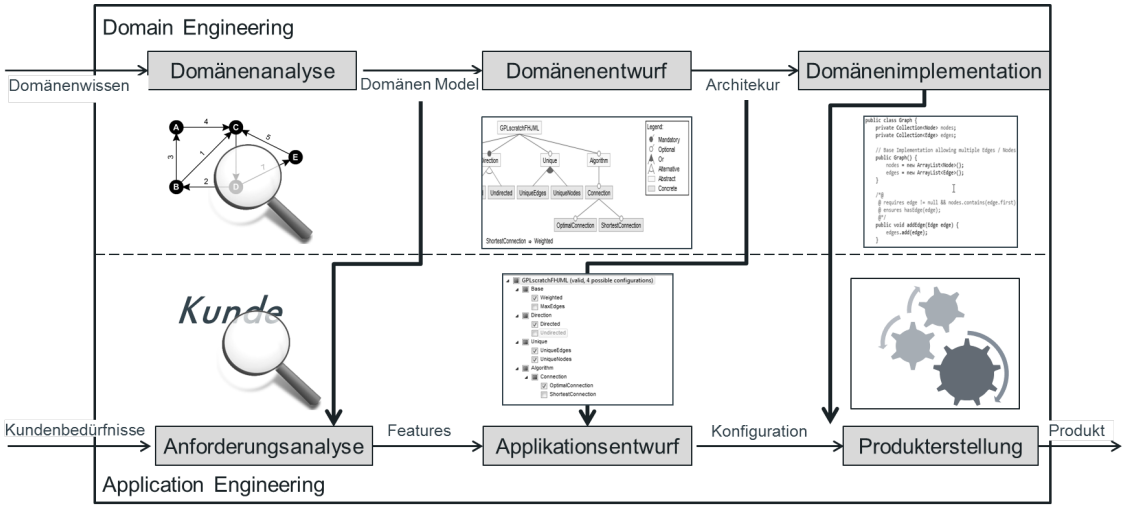

Abbildung 2.8: Zwei Phasen der Software-Produktlinien Entwicklung

hoch, weshalb als Verbesserung des Ansatzes vorgeschlagen wird, nur Produktvarianten zu spezifizieren, die tatsächlich für einen Kunden aus der Software-Produktlinie erzeugt werden. Die Produkt-basierte Spezifikation eignet sich speziell für Software-Produktlinien, bei denen die einzelnen Produkte viele Unterschiede aufweisen und daher die Wiederverwendung von Spezifikationen nicht möglich ist.

- Feature-basierte Spezifikation - Bei der Feature-basierten Spezifikationen werden alle Features ohne Bezug zu anderen Features in Isolation spezifiziert, sodass das erwartete Verhalten eines einzelnen Feature genau angeben werden kann. Bei der Komposition von Features zu einer Variante werden Spezifikationen der Features zu einer Spezifikation komponiert, die genau diese Variante beschreibt. Die Feature-basierte Spezifikation eignet sich, um Feature-Interaktionen [ASLK10] aufzudecken.

- Familien-basierte Spezifikation - Der Familien-basierte Ansatz setzt voraus, dass alle Features im Vorfeld bekannt sind. Es wird eine Spezifikation für die gesamte Software-Produktlinie angegeben, die variable Teile mit Bezug zu Features bzw. Feature-Kombinationen enthält. Somit muss nur eine Spezifikation für die gesamte Software-Produktlinie angegeben werden.

- Globale Spezifikation - Verhalten, welches in allen Varianten der Software-Produktlinie gilt, kann mittels globaler Spezifikation angegeben werden. Zu beachten ist dabei, dass sobald eine Variante dieses Verhalten nicht aufweist, die globale Spezifikation nicht mehr angewendet werden kann.

- Domänen-unabhängige Spezifikation - Bisherige Ansätze beziehen sich auf Eigenschaften einer konkreten Software-Produktlinie. Domänen-unabhängige Spezifikation spezifiziert typische Eigenschaften von Programmiersprachen, die in allen Domänen bzw. Software-Produktlinien auftauchen.

3. Kontrakte in Feature-orientierter Programmierung

In diesem Kapitel werden Techniken zur Spezifikation mit Kontrakten in Feature-orientierter Programmierung vorgestellt. Weiterhin wird aufgezeigt, dass man mit bisherigen Verfahren [TSK+12, TBAS13] zur Umsetzung von Kontrakten in Feature-orientierter Programmierung eingeschränkte Möglichkeiten hat. Deshalb werden neue Schlüsselwörter vorgeschlagen, die eine flexible Spezifikation auf Methodenebene ermöglichen sollen. Ferner werden Konzepte diskutiert, wie die Schlüsselwörter angewendet werden können.

3.1 Techniken zur Kontrakt-Komposition

Bei der Entwicklung von Feature-orientierten Programmen, werden Methoden inkrementell in Feature-Modulen verfeinert. Das gilt auch für die Spezifikation dieser Methoden. Dafür wird ein Kompositionsmechanismus benötigt, der die schrittweise verfeinerten Spezifikationen bei der Produkterstellung zusammenführt. Zur Komposition von Feature-Spezifikationen haben Thüm et al. [TBAS13] den Kompositionsoperator $\bullet$, ähnlich wie bei der Komposition von Methoden, wie folgt definiert: $\bullet : C \times C \rightarrow C$. C ist dabei eine Menge von möglichen Kontrakten.

Die Komposition eines Kontrakts $c = \{\phi\}m\{\psi\}$ mit einem verfeinerten Kontrakt $c' = \{\phi'\}m'\{\psi'\}$ wird als komponierter Kontrakt $c'' = c' \bullet c = \{\phi''\}m' \bullet m\{\psi''\}$ bezeichnet [TBAS13]. Es gibt verschiedene Techniken, wie der Kontrakt letztendlich komponiert wird. Thüm et al. [TSK+12] schlagen fünf Techniken zur Komposition von Kontrakten in Feature-orientierter Programmierung vor, welche Benduhn [Ben12] um zwei weitere Techniken erweitert. Insgesamt werden in dieser Arbeit sechs sprachunabhängige Techniken zur Komposition von Kontrakten in Feature-orientierter Programmierung betrachtet: *Plain Contracting, Explicit Contract Refinement, Contract Overriding, Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement*, sowie die sprachspezifische

Technik *Pure Method Refinement*. Zusätzlich wird auf zwei weitere Spezifika der Java Modeling Langauge eingegangen: *Multiple Specification Cases und Multiple Preconditions and Postconditions*.

3.1.1 Sprachunabhängige Techniken

In diesem Abschnitt werden Techniken erläutert, die generell für jede Programmiersprache, welche Design by Contract ermöglicht, angewendet werden können. Im Fokus steht dabei die Komposition von Vor- und Nachbedingungen. Es werden sechs Kompositionstechniken von Kontrakten in Feature-orientierter Programmierung diskutiert.

Plain Contracting

Der einfachste Fall der Kontrakt-Komposition wird als *Plain Contracting* bezeichnet [TSK+12, TBAS13], bei der ein Kontrakt nur einmal, bei der Methodeneinführung, angegeben wird. Verfeinerte Methoden müssen das Verhalten der Basis-Methode besitzen, d.h. die Implementierung der Methode kann getauscht werden, jedoch müssen die selben Vor- und Nachbedingungen erfüllt werden: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi\}m' \bullet m\{\psi\}$ [TBAS13]. In der Abbildung 3.1 auf der nächsten Seite soll die Methode `getConnection` immer eine Verbindung von Knoten `source` nach Knoten `target` zurückliefern, sofern eine Verbindung vorhanden ist. Das Feature *Connection* nutzt zur Rückgabe dabei eine Tiefensuche, welche bspw. bei Systemen mit wenig Speicher bevorzugt werden könnte, da der Speicherverbrauch bei der Tiefensuche geringer sein kann als bei der Breitensuche [Kor85]. Eine andere Variante könnte eine Breitensuche im Feature *OptimalConnection* sein, welche im Gegensatz zur Tiefensuche immer eine optimale Lösung zurückliefert, wenn es eine gibt [Kor85]. Trotz verschiedener Implementierungen soll der Kontrakt für alle Methodenverfeinerungen gelten, weshalb sich hier Plain Contracting anbietet. Der Vorteil liegt darin, dass Entwickler nur einmal einen Kontrakt für eine Methode angeben müssen. Außerdem werden die durch den Kontrakt gewährleisteten Garantien in allen Verfeinerungen erhalten. Auf der anderen Seite ist dieser Ansatz sehr einschränkend für Erweiterungen. Wenn später ein neues Feature mit neuer Implementierung der Methode hinzugefügt wird und eine andere Spezifikation benötigt, ist dieser Ansatz untauglich. Plain Contracting ist assoziativ und idempotent, aber nicht kommutativ [TBAS13].

Explicit Contract Refinement

Ähnlich wie bei der Methodenverfeinerung von Feature-orientierten Programmen können Kontrakte mit *Explicit Contract Refinement* verfeinert werden. Mit dem Schlüsselwort `original` kann sich der Programmierer auf den originalen Kontrakt, d.h. den Kontrakt der zu verfeinernden Methode, beziehen, um diese Methode mit weiteren Spezifikationen zu verfeinern. Die Komposition zweier Kontrakte wird folgendermaßen beschrieben: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \bullet \phi\}m' \bullet m\{\psi' \bullet \psi\}$. Die Komposition von $\phi' \bullet \phi$ bzw. $\psi' \bullet \psi$ bedeutet, dass alle Vorkommnisse von `original` in ϕ' bzw. ψ' durch ϕ bzw. ψ ersetzt werden [TBAS13]. Explicit Contract Refinement

```
public class Graph {                                                Connection
    /*@ requires source != null && target != null;
      @ ensures hasConnection(source, target)
      @   ==> isConnection(\result, source, target);
      @*/
    public ArrayList<Node> getConnection(Node source, Node target) {
        setNodesUnvisited();
        return doDFS(new ArrayList<Node>(), source, target);
    }
}
```

•

```
public class Graph {                                          OptimalConnection
    public ArrayList<Node> getConnection(Node source, Node target) {
        setNodesUnvisited();
        return doBFS(new ArrayList<Node>(), source, target);
    }
}
```

=

```
public class Graph {                          OptimalConnection • Connection
    /*@ requires source != null && target != null;
      @ ensures hasConnection(source, target)
      @   ==> isConnection(\result, source, target);
      @*/
    public ArrayList<Node> getConnection(Node source, Node target) {
        setNodesUnvisited();
        return doBFS(new ArrayList<Node>(), source, target);
    }
}
```

Abbildung 3.1: Die Kompositionstechnik Plain Contracting am Beispiel der Features *Connection* und *OptimalConnection*, wobei der angegebene Kontrakt an der Methodeneinführung von `getConnection` für alle Methodenverfeinerungen gilt.

setzt die Idee der Komposition von Methoden in Feature-orientierter Programmierung für die Komposition von Kontrakten um, wodurch der Nutzer kein neues Verständnis zur Komposition entwickeln muss. Außerdem hat er die Möglichkeit flexible Spezifikationen, abhängig von der Feature-Auswahl, zu schreiben. Diese Flexibilität kann jedoch auch nachteilig sein, weil Spezifikationen schwerer verständlich sein können, wenn sie von den beteiligten Features abhängig sind [TSK+12, TBAS13]. Zusätzlich können Garantien, die durch vorherige Kontrakte festgelegt wurden, aufgelöst werden. D.h. im Gegensatz zu Plain Contracting ist Explicit Contract Refinement weniger restriktiv im Bezug auf Garantien, die von Methoden angenommen werden können. Dieser Ansatz wird in Abbildung 3.2 auf der nächsten Seite deutlich. Das Feature *Base* bietet die Methode `addEdge`, welche eine neue Kante zu dem Graphen hinzufügt, wenn er die Knoten der Kante besitzt und die Kante initialisiert ist. Dem Aufrufer der Methode wird versichert, dass der Graph anschließend diese Kante besitzt, wenn die Vorbedingungen erfüllt sind. Mit der Auswahl des Features *MaxEdges* kann eine maximale Anzahl an Kanten im Graph festgelegt werden. Diese obere Grenze wird in der Variablen `MAXEDGES` gespeichert und muss für die Ausführung der Methode ebenfalls initialisiert sein, damit eine Überprüfung der Grenze vorgenommen werden kann. Die Nachbedingung von *Base* gilt nur, wenn die Anzahl an Kanten noch nicht größer als die obere Grenze ist. Die Methode `countEdges` liefert dabei die Anzahl vorhandener Kanten zurück. Explicit Contract Refinement ist assoziativ, aber weder kommutativ noch idempotent [TBAS13].

Contract Overriding

Ein Spezialfall des Explicit Contract Refinement ist das *Contract Overriding*. Hierbei wird das Schlüsselwort `original` nicht verwendet, sodass der Originalkontrakt nicht in der Verfeinerung referenziert werden kann und somit überschrieben wird [TSK+12, TBAS13]. Im Gegensatz zu Plain Contracting wird nicht ein Methodenkontrakt angegeben, der in allen Features gültig ist, sondern besteht die Möglichkeit für jede Methodenverfeinerung einen neuen Kontrakt anzugeben, wodurch der Originalkontrakt überschrieben wird. Falls keine Vor- oder Nachbedingung angegeben wird, gilt der Originalkontrakt. Der Kompositionsmechanismus für Contract Overriding ersetzt alle Vor- bzw. Nachbedingungen der ursprünglichen Methode, durch die Vor- bzw. Nachbedingungen der verfeinerten Methode, sofern eine (Vor-/Nach-)Bedingung angegeben ist: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi'\}m' \bullet m\{\psi'\}$ [TBAS13]. Mit Contract Overriding ist man flexibel bei der Kontrakterstellung. Allerdings kann es schnell zu Quelltextreplikationen kommen, indem bestehende Kontrakte kopiert und auf verfeinerte Methoden angewandt werden, weil ein Bezug auf diese Spezifikationen nicht möglich ist [TSK+12, TBAS13]. Veranschaulicht wird die Replikation in Abbildung 3.3 auf Seite 24, bei dem das Beispiel aus Abbildung 3.2 übernommen wurde. Allerdings wurden die Vorkommen des Schlüsselworts `original` durch die konkreten Spezifikationen ersetzt. Deutlich wird dabei, dass Contract Overriding leicht durch Explicit Contract Refinement ausgetauscht werden kann, indem das Schlüsselwort `original` nicht erlaubt bzw. benutzt wird. Contract Overriding ist sowohl assoziativ als auch idempotent, jedoch nicht kommutativ [TBAS13].

```
public class Graph  {                                                    Base
    private Collection<Edge> edges;
    private Collection<Node> nodes;

    /*@ requires edge != null && nodes.contains(edge.first)
      @   && nodes.contains(edge.second);
      @ ensures hasEdge(edge);
      @*/
    public void addEdge(Edge edge) {
        edges.add(edge);
    }

    public /*@pure@*/ boolean hasEdge(Edge edge){...}
}
```

•

```
public class Graph {                                                 MaxEdges
    private static Integer MAXEDGES = new Integer(10);

    /*@ requires \original && MAXEDGES != null;
      @ ensures (\old(countEdges()) < MAXEDGES) ==> \original;
      @*/
    public void addEdge(Edge edge) {
        if(countEdges() < MAXEDGES)
            original(edge);
    }

    private /*@pure@*/ int countEdges(){
        return edges.size();
    }
}
```

=

```
public class Graph {                                        MaxEdges • Base
    private Collection<Edge> edges;
    private Collection<Node> nodes;
    private static Integer MAXEDGES = new Integer(10);

    /*@ requires edge != null && nodes.contains(edge.first)
      @   && nodes.contains(edge.second) && MAXEDGES != null;
      @ ensures (\old(countEdges()) < MAXEDGES) ==> hasEdge(edge);
      @*/
    public void addEdge(Edge edge) {
        if(countEdges() < MAXEDGES)
            edges.add(edge);
    }
}
```

Abbildung 3.2: Mit Hilfe des Schlüsselworts `original` kann beim Explicit Contract Refinement auf die vorherige Vor- bzw. Nachbedingung verwiesen werden. Somit kann der Kontrakt verfeinert werden, was hier am Beispiel der Methode `addEdge` der Features *Base* und *MaxEdges* gezeigt wird.

```
public class Graph  {                                                    Base
    private Collection<Edge> edges;
    private Collection<Node> nodes;

    /*@ requires edge != null && nodes.contains(edge.first)
      @   && nodes.contains(edge.second);
      @ ensures hasEdge(edge);
      @*/
    public void addEdge(Edge edge) {
        edges.add(edge);
    }
}
```

•

```
public class Graph {                                                 MaxEdges
    private static Integer MAXEDGES = new Integer(10);

    /*@ requires edge != null && nodes.contains(edge.first)
      @   && nodes.contains(edge.second) && MAXEDGES != null;
      @ ensures (\old(countEdges()) < MAXEDGES) ==> hasEdge(edge);
      @*/
    public void addEdge(Edge edge) {
        if(countEdges() < MAXEDGES)
            original(edge);
    }
}
```

=

```
public class Graph {                                        MaxEdges • Base
    private Collection<Edge> edges;
    private Collection<Node> nodes;
    private static Integer MAXEDGES = new Integer(10);

    /*@ requires edge != null && nodes.contains(edge.first)
      @   && nodes.contains(edge.second) && MAXEDGES != null;
      @ ensures (\old(countEdges()) < MAXEDGES) ==> hasEdge(edge);
      @*/
    public void addEdge(Edge edge) {
        if(countEdges() < MAXEDGES)
            edges.add(edge);
    }
}
```

Abbildung 3.3: Bei der Kompositionstechnik Contract Overriding ist das Schlüsselwort `original` nicht erlaubt, wodurch leicht Quelltextreplikationen entstehen können. Dies wird beim erneuten Verwenden der Methode `addEdge` der Features *Base* und *MaxEdges* deutlich.

Cumulative Contract Refinement

Eine ähnlich restriktive Kompositionstechnik im Bezug auf Garantien, auf die sich der Aufrufer einer Methode verlassen kann, ist *Cumulative Contract Refinement*. Im Gegensatz zu Plain Contracting können dabei Methodenkontrakte verfeinert werden. Die Originalvorbedingung und die verfeinerte Vorbedingung werden mit einer oder-Verknüpfung zusammengefügt. Dies gewährleistet dem Nutzer eine einfache Ausführung der Methode, weil es genügt eine Vorbedingung zu erfüllen. Im Gegenzug kann sich der Nutzer auf starke Nachbedingungen verlassen, was im Fall von Cumulative Contract Refinement bedeutet, dass alle angegeben Nachbedingungen von der Methode erfüllt werden müssen. Aus der Sicht des Nutzers erhält man so viele Garantien, die durch leichte Ausführung der Methode gewährleistet werden. Formal wird Cumulative Contract Refinement wie folgt angegeben: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\}$ [TBAS13]. Cumulative Contract Refinement ist sowohl assoziativ, kommutativ als auch idempotent [TBAS13]. Für die Graph-Produktlinie scheint Cumulative Contract Refinement zu restriktiv zu sein. Es konnte kein Anwendungsfall gefunden werden, indem es genügt eine beliebige Vorbedingung zu erfüllen, um alle Nachbedingungen zu fordern. Aus diesem Grund kann kein konkretes Beispiel aus der Graph-Produktlinie angegeben werden. In Abbildung 3.4 wird eine schematische Kontraktkomposition angedeutet.

```
public class Klasse {                                                    FeatureA
    /*@ requires VorbedingungA;
      @ ensures NachbedingungA;
      @*/
    public void methode() { }
}
```

•

```
public class Klasse {                                                    FeatureB
    /*@ requires VorbedingungB;
      @ ensures NachbedingungB;
      @*/
    public void methode() { }
}
```

=

```
public class Klasse {                                   FeatureB • FeatureA
    /*@ requires VorbedingungA || VorbedingungB;
      @ ensures NachbedingungA && NachbedingungB;
      @*/
    public void methode() { }
}
```

Abbildung 3.4: Schematisches Beispiel einer Kontraktkomposition mittels Cumulative Contract Refinement. Vorbedingungen werden dabei mit einer oder-Verknüpfung und Nachbedingungen mit einer und-Verknüpfung zusammengefügt.

Conjunctive Contract Refinement

Etwas weniger restriktiv als Cumulative Contract Refinement ist *Conjunctive Contract Refinement*, bei der ebenso alle Nachbedingungen von der Methode erfüllt

werden müssen. Im Gegensatz zu Cumulative Contract Refinement müssen hierbei aber auch alle Vorbedingungen erfüllt werden. Formalisiert bedeutet dies: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}$ [TBAS13]. Bei der Komposition werden die Vorbedingungen und Nachbedingungen zusammengefasst. In Abbildung 3.5 wird die Methode `equals` der Klasse `Edge` verfeinert. Im Feature *Base* wird gewährleistet, dass Kanten sich gleichen, wenn sie vom selben Typ sind. Das Feature *Directed* konkretisiert dies, indem es die Richtung der Kante betrachtet. Die Vor- und Nachbedingung von *Base* gelten weiterhin und werden mit dem Kontrakt von *Directed* konjugiert, indem die Kontrakte zusammengefügt werden. In der zweiten Variante wird zusätzlich zu der Richtung der Kanten auch dessen Gewicht aus dem Feature *Weighted* verglichen. Auch hier sollen die Kontrakte der Features *Base* und *Directed* weiterhin gelten. Dieser Mechanismus ist zwar nicht ganz so restriktiv wie Cumulative Contract Refinement, ist für den Aufrufer einer Methode allerdings schwerer zu erfüllen, was zu Problemen bei der Verwendung führen kann [Ben12]. Conjunctive Contract Refinement ist sowohl assoziativ, kommutativ als auch idempotent [TBAS13].

Consecutive Contract Refinement

Die Idee bei *Consecutive Contract Refinement* ist, dass der Originalkontrakt zwar verfeinert werden kann, jedoch der Originalkontrakt und die Kontraktverfeinerung sich nicht gegenseitig ausschließen bzw. beeinflussen. Konkret heißt das, dass die Nachbedingung des originalen Kontrakts nur erfüllt sein muss, wenn die Vorbedingung des originalen Kontrakts erfüllt ist. Analog gilt das für die Kontraktverfeinerung, was zu folgendem Kompositionsmechanismus führt: $c'' = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{(old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\}$. Die Funktion $old()$ transformiert dabei den Status der Vorbedingung in die Nachbedingung [TBAS13]. Die beiden Kontrakte werden mit dem `also` Operator komponiert, wie in Abbildung 3.7 auf Seite 29 zu sehen ist. Consecutive Contract Refinement ist für Feature-orientierte Programmierung das, was für objektorientierte Programmierung Specification Inheritance (s. Abschnitt 2.2) ist. Durch Consecutive Contract Refinement werden die Kontrakte weniger restriktiv behandelt als bei den vorherigen Mechanismen, weil Garantien für den Nutzer der Methode verloren gehen können, wenn nicht alle Vorbedingungen erfüllt sind [TSK+12, TBAS13]. Auf der anderen Seite ist es leichter für den Aufrufer der Methode diese auszuführen und ebenfalls kann es für die Methode leichter sein ihre Bedingungen zu erfüllen. Consecutive Contract Refinement ist sowohl assoziativ, kommutativ als auch idempotent [TBAS13].

3.1.2 Sprachspezifische Techniken

In Abschnitt 3.1.1 wurden Techniken zur Komposition von Kontrakten in allen Feature-orientierten Programmiersprachen, welche Design By Contract nutzen, vorgestellt. Im Folgenden wird der Umgang mit drei Spezifika der Java Modeling Language diskutiert, weil diese als Spracherweiterung für Java genutzt wird und in dieser Arbeit verwendet wird. Daher ist es notwendig auf diese Spezialfälle einzugehen.

Pure Method Refinement

In der Java Modeling Language ist es möglich innerhalb des Kontrakts Methoden zu verwenden, die keine Seiteneffekte besitzen. D.h. existierende Objekte werden durch

```
public class Edge {                                                    Base
    /*@ requires ob != null;
      @ ensures \result ==> ob instanceof Edge;
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        return (ob instanceof Edge);
    }
}
```

```
public class Edge {                                                 Weighted
    private Integer weight = new Integer(0);

    /*@ requires weight != null;
      @ ensures \result ==> (weight == ((Edge)ob).weight);
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        return original(ob)
            && this.weight.equals(((Edge)ob).weight);
    }
}
```

```
public class Edge {                                                 Directed
    private Node first, second;

    /*@ requires first != null && second != null;
      @ ensures \result ==> (first.equals(((Edge) ob).first)
      @  && second.equals(((Edge) ob).second));
      @*/
    @Override
    public/*@pure@*/boolean equals(Object ob) {
        if (original(ob)) {
            Edge edge = (Edge) ob;
            return first.equals(edge.first)
                && second.equals(edge.second);
        }
        return false;
    }
}
```

Abbildung 3.5: Die Methode `equals` der Klasse `Edge` soll mittels Conjunctive Contract Refinement komponiert werden. Die drei Features *Base*, *Weighted* und *Directed* implementieren bzw. verfeinern diese Methode und können zusammen zu einer Variante komponiert werden.

```
    public class Edge {                                    Directed • Base
    /*@ requires ob != null && first != null
      @  && second != null;
      @ ensures (\result ==> ob instanceof Edge)
      @  && (\result ==> (first.equals(((Edge) ob).first)
      @     && second.equals(((Edge) ob).second)));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        if (ob instanceof Edge) {
            Edge edge = (Edge) ob;
            return first.equals(edge.first)
                    && second.equals(edge.second);
        }
        return false;
    }
}
```

```
    public class Edge {                         Directed • Weighted • Base
        private Integer weight = 0;

    /*@ requires ob != null && weight != null
      @  && first != null && second != null;
      @ ensures (\result ==> ob instanceof Edge)
      @  && (\result ==> (weight == ((Edge)ob).weight))
      @  && (\result ==> (first.equals(((Edge) ob).first)
      @     && second.equals(((Edge) ob).second)));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        if (ob instanceof Edge) {
            Edge edge = (Edge) ob;
            return weight.equals(((Edge)ob).weight)
                    && first.equals(edge.first)
                    && second.equals(edge.second);
        }
        return false;
    }
}
```

Abbildung 3.6: Die Features *Base* und *Directed* werden einmal ohne und einmal mit dem optionalen Feature *Weighted* zu zwei unterschiedlichen Varianten mittels Conjunctive Contract Refinement komponiert. Dabei werden die Vorbedingungen und Nachbedingungen der Features miteinander verknüpft.

```
public class Graph {                                                    Base
    /*@ requires nodes != null && nodes instanceof List<Node>;
      @ ensures \result ==>
      @   ((\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = -1)
      @ || (\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = 0));
      @*/
    public Collection<Node> sortNodes(Collection<Node> nodes) {
        List<Node> list = new ArrayList<Node>(nodes);
        Collections.sort(list);
        return list;
    }
}
```

•

```
public class Graph {                                            UniqueNodes
    /*@ requires nodes != null && nodes instanceof Set<Node>;
      @ ensures \result ==>
      @   (\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = -1);
      @*/
    public Collection<Node> sortNodes(Collection<Node> nodes) {
        return = new TreeSet<Node>(nodes);
    }
}
```

=

```
public class Graph {                                  UniqueNodes • Base
    public Graph() { nodes = new HashSet<Node>(); }

    /*@ requires nodes != null && nodes instanceof List<Node>;
      @ ensures \result ==>
      @   ((\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = -1)
      @ || (\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = 0));
      @also
      @ requires nodes != null && nodes instanceof Set<Node>;
      @ ensures \result ==>
      @   (\forall int i; 0 <= i && i < nodes.size() -1;
      @     nodes.toArray()[i].compareTo(nodes.toArray()[i+1]) = -1);
      @*/
    public Collection<Node> sortNodes(Collection<Node> nodes) {
        return = new TreeSet<Node>(nodes);
    }
}
```

Abbildung 3.7: Semantisch verschiedene Kontrakte können mit Hilfe von Consecutive Contract Refinement verknüpft werden, indem sie mit dem `also` Operator unabhängig voneinander betrachtet werden, was bei einer Sortierung von Listen bzw. Mengen der Fall sein kann.

die Methode nicht verändert. Diese Methoden werden als *pure* bezeichnet. Mit der Verfeinerung von pure Methoden werden unter Umständen auch die Kontrakte anderer Methoden verfeinert. Mittels *Pure Method Refinement* ist eine feingranulare Verfeinerung von Methodenkontrakten möglich, indem als pure deklarierte Methoden in Kontrakten verwendet und verfeinert werden. Thüm et al. [TBAS13] formalisieren keinen Mechanismus zur Komposition solcher Kontrakte, da es mit den bisher vorgestellten Techniken abbildbar ist. In der Abbildung 3.8 verwendet die Methode `hasEdge` die in Abbildung 3.5 auf Seite 27 eingeführte Methode `equals` in der Nachbedingung des Kontrakts. Da die `equals` Methode in verschiedenen Features verfeinert wird, wird auch der Kontrakt der Methode `hasEdge` verfeinert, sodass die Semantik abhängig von der Features-Auswahl ist.

```
public class Graph {                                                    Base
    private List<Edge> edges;

    /*@ requires edge != null;
      @ ensures \result ==> (\exists int i; 0 <= i
      @   && i < edges.size(); edges.get(i).equals(edge));
      @*/
    public /*@pure@*/ boolean hasEdge(Edge edge) {
        for(Edge e : edges)
            if(edge.equals(e))
                return true;
        return false;
    }
}
```

Abbildung 3.8: Pure Method Refinement Beispiel.

Multiple Specification Cases

Mit dem Schlüsselwort `also` bietet die Java Modeling Language dem Nutzer die Option nicht nur einen Methodenkontrakt anzugeben, sondern mehrere Kontrakte für eine Methode. Dabei stellt sich die Frage, wie die Kompositionstechniken bei mehreren Methodenkontrakten funktionieren. Benduhn [Ben12] hat diese Frage beantwortet. Bei Plain Contracting bringt dies keine Probleme mit sich, weil der Kontrakt nicht verfeinert werden darf. Wird der Kontrakt mittels Contract Overriding überschrieben, so werden alle Kontrakte des originalen Kontrakts entfernt und nur der verfeinerte Kontrakt wird in der Komposition verwendet. Um den auf die Kontrakte der originalen Methode in der Verfeinerung mittels Explicit Contract Refinement zu verweisen, hat Benduhn [Ben12] zwei zusätzliche Schlüsselwörter neben `original` eingeführt. Mit dem Schlüsselwort `originalSpec` wird sich auf die gesamte Spezifikation der originalen Methode bezogen. Eine Verfeinerung kann vorgenommen werden, indem mit `also` ein neuer Kontrakt hinzugefügt wird. Soll ein spezieller Kontrakt der originalen Spezifikation verfeinert werden, so wird mittels `originalCase` der Kontrakt an der jeweiligen Position des aktuellen Kontrakts referenziert. D.h. es muss bei der Verfeinerung auf die Reihenfolge der einzelnen Kontrakte geachtet werden. Bei Consecutive Contract Refinement ist nach Definition nur erlaubt einen zusätzlichen Kontrakt zur Methodenspezifikation hinzuzufügen, was mit dem Schlüsselwort `also` umgesetzt wird. Somit entstehen hierbei keine

Probleme. Bei Cumulative Contract Refinement und Conjunctive Contract Refinement muss die Methodenspezifikation vor der Komposition so umgeformt werden, dass das Schlüsselwort `also` nicht mehr vorkommt und nur noch eine Vorbedingung und Nachbedingung in der Methodenspezifikation vorliegt. Diese Umformung kann mit dem aussagenlogischen Ausdruck aus Abschnitt 2.2 auf Seite 16 bzw. Abschnitt 3.1.1 auf Seite 26 vorgenommen werden. Anschließend können die Nachbedingungen mit einer und-Verknüpfung und die Vorbedingungen entweder mit einer und-Verknüpfung oder einer oder-Verknüpfung komponiert werden.

Multiple Preconditions and Postconditions

Zusätzlich ist es in der Java Modeling Language erlaubt, mehrere Vorbedingungen bzw. Nachbedingungen für einen Kontrakt anzugeben, indem die Verwendung der Schlüsselwörter `requires` bzw. `ensures` mehrfach verwendet werden. Semantisch bedeutet dies eine Konjunktion der Vorbedingungen bzw. Nachbedingungen [TBAS13]. So kann der Kontrakt der Methode `equals` im Feature *Directed* in zwei verschiedenen Darstellungsformen geschrieben werden, wie in Abbildung 3.9 zu sehen ist. In beiden Fällen beinhaltet der Kontrakt die gleichen Bedingungen. Aus Gründen der Lesbarkeit kann sich zum Beispiel für die Schreibweise mit Multiple Preconditions und Postconditions entschieden werden, weil während der Komposition eine Transformation in eine Konjunktion stattfinden kann [TBAS13].

```
    public class Edge {
    /*@ requires first != null && second != null;
      @ ensures \result ==> (first.equals(((Edge) ob).first)
      @   && second.equals(((Edge) ob).second));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {...}
}
```

```
    public class Edge {
    /*@ requires first != null
      @ requires second != null;
      @ ensures \result ==> first.equals(((Edge) ob).first)
      @ ensures \result ==> second.equals(((Edge) ob).second));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {...}
}
```

Abbildung 3.9: Multiple Preconditions and Postconditions können leicht in einzelne Vorbedingungen bzw. Nachbedingungen konvertiert werden, indem die Vor- und Nachbedingungen mit einer &&-Verknüpfung zusammengeführt werden. Eine Konvertierung in die andere Richtung ist ebenfalls möglich.

3.1.3 Vergleich der Techniken

Um Kontrakte in Feature-orientierter Programmierung zu verfeinern, gibt es unterschiedliche Ansätze. Die vorgestellten Techniken unterscheiden sich anhand der Komposition von Kontrakten. Dadurch entstehen Garantien an den Methoden.

In der Literatur [TSK+12] wird zwischen *verstärkenden (engl. strengths)* und *abschwächenden (engl. weaknesses)* Kontraktverfeinerungen unterschieden. Dabei gibt es zwei verschiedene Sichtweisen - die Sicht des Methodenaufrufers und die Sicht der Methode. Verstärken bzw. Abschwächen meint dabei die Schwierigkeit ein Kontrakt zu erfüllen. D.h. die Verstärkung eines Kontrakts bedeutet für den Aufrufer der Methode, dass es für ihn einfacher wird den Kontrakt zu erfüllen. Dies wird umgesetzt, indem die Vorbedingung schwächer bzw. die Nachbedingung stärker gemacht wird, d.h. weitere Bedingungen hinzugefügt werden. Die Betrachtung aus Sicht des Aufrufers soll in dieser Arbeit im Mittelpunkt stehen. Erleichtert man im Gegensatz die Erfüllung des Kontrakts für die Methode und erschwert es somit dem Aufrufer, können Methoden weniger genutzt und wiederverwendet werden. Cumulative, Conjunctive und Consecutive Contract Refinement besitzen die Eigenschaft der Kontraktverstärkung aber nicht der Abschwächung von Kontrakten. Plain Contracting hat weder verstärkende noch abschwächendes Verhalten, wohingegen Contract Overriding beide Eigenschaften besitzt [Ben12].

Es hat sich gezeigt, dass restriktive Kompositionstechniken zu Problemen führen können. So ist es beispielsweise bei Plain Contracting nicht möglich unvorhergesehene Methodenimplementierungen mit unterschiedlichen Kontrakten einzuführen. Bei Cumulative Contract Refinement ist es schwer einen Anwendungsfall zu finden, auf den diese Technik angewendet werden kann. Auch Conjunctive Contract Refinement kann in der Anwendung problematisch sein, wenn Kontrakte durch die verknüpften Vorbedingungen gar nicht mehr erfüllt werden können und dadurch die Methode nicht ausgeführt wird.

Aus diesem Grund stellt sich die Frage, warum solch restriktive Kompositionstechniken nicht durch Explicit Contract Refinement ausdrückt werden. Das Beispiel in Abbildung 3.10 auf der nächsten Seite zeigt, wie die Kontrakte, die mittels Conjunctive Contract Refinement komponiert werden (s. Abbildung 3.5 auf Seite 27 und Abbildung 3.6 auf Seite 28) durch Explicit Contract Refinement ausgedrückt werden können. Das kann analog auch für Plain Contracting durchgeführt werden, indem nur die Methodeneinführung einen Kontrakt bekommt. Für Cumulative Contract Refinement muss bei der Kontraktverfeinerung der Vorbedingung das Schlüsselwort `original` mit einer oder-Verknüpfung und in der Nachbedingung das Schlüsselwort `original` mit einer und-Verknüpfung eingeführt werden. Consecutive Contract Refinement kann mit Hilfe der für Multiple Specification Cases eingeführten Schlüsselwörter auch mit Explicit Contract Refinement umgesetzt werden.

Man sieht, dass alle Kompositionstechniken durch Explicit Contract Refinement ausgedrückt werden können. Schreibt man nun bestehende Kontrakte anderer Kompositionstechniken mit Explicit Contract Refinement um, so entsteht ein großer Nachteil. Explicit Contract Refinement bietet keine Garantien, was von Kontrakten der originalen Methode erwartet werden kann. Solche Garantien können wichtig für die Sicherheit von Programmen sein, verbessern die Verständlichkeit der Kontrakte und können insbesondere bei der Entwicklung in Gruppen Fehler vermeiden. Deswegen ist es notwendig alle vorgestellten Techniken nutzen zu können.

Anhand der Beispiele der Graph-Produktlinie wird genau das gezeigt. Es ist wichtig unterschiedliche Kompositionstechniken für Kontrakte innerhalb einer Software-Produktlinie zu verwenden. Einen Überblick der verwendeten Techniken zeigt das

```
public class Edge {                                                    Base
    /*@ requires ob != null;
      @ ensures \result ==> ob instanceof Edge;
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        return (ob instanceof Edge);
    }
}
```

•

```
public class Edge {                                                 Directed
    private Node first, second;

    /*@ requires \original && first != null && second != null;
      @ ensures \original
      @ && (\result ==> (first.equals(((Edge) ob).first)
      @   && second.equals(((Edge) ob).second)));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        if (original(ob)) {
            Edge edge = (Edge) ob;
            return first.equals(edge.first)
                && second.equals(edge.second);
        }
        return false;
    }
}
```

=

```
public class Edge {                                          Directed • Base
    private Node first, second;

    /*@ requires ob != null && first != null && second != null;
      @ ensures (\result ==> ob instanceof Edge)
      @ && (\result ==> (first.equals(((Edge) ob).first)
      @   && second.equals(((Edge) ob).second)));
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
        if (ob instanceof Edge) {
            Edge edge = (Edge) ob;
            return first.equals(edge.first)
                && second.equals(edge.second);
        }
        return false;
    }
}
```

Abbildung 3.10: Die Methode `equals` der Klasse `Edge` wird, anstatt mit Conjunctive Contract Refinement, nun mit Explicit Contract Refinement verfeinert, indem das Schlüsselwort `original` mit einer &&-Verknüpfung in der Kontraktverfeinerung in der Vor- und Nachbedingung verwendet wird.

Kollaborationsdiagramm der Graph-Produktlinie in Abbildung 3.11 auf der nächsten Seite. Die verwendeten Kompositionstechniken werden farblich an den Methoden gekennzeichnet. Es wird deutlich, dass alle Kompositionstechniken, bis auf Cumulative Contract Refinement, in der Graph-Produktlinie verwendet werden.

Bisherige Ansätze [Ben12] ermöglichen dem Nutzer die Festlegung der Kompositionstechnik auf Projektebene. D.h. es wird eine Kompositionstechnik angegeben, die für die gesamte Software-Produktlinie gilt. Dieser Mechanismus ist nicht ausreichend, um unterschiedliche Garantien über eine Software-Produktlinie zu gewährleisten. In Abschnitt 3.2 wird ein Konzept vorgestellt, wie man Methoden-basiert die zu verwendende Kompositionstechnik angeben kann. D.h. es muss die Möglichkeit geben, explizit für jede Methode eine konkrete Kompositionstechnik anzugeben.

3.2 Schlüsselwörter für Methoden-basierte Kontraktverfeinerung

In Abschnitt 3.1 wurden Techniken vorgestellt, wie man Kontrakte in Feature-orientierter Programmierung verfeinern kann. Wie dort zu sehen ist, stammen die verwendeten Beispiele zur Veranschaulichung der Techniken aus einer Software-Produktlinie, der Graph-Produktlinie. Damit bei der Komposition der Methoden - genauer der Methodenkontrakte - entschieden werden kann, welche dieser Techniken angewendet werden soll, ist es notwendig für jede Methode eine Anweisung anzufügen, die die zu verwendende Technik beschreibt. Im Folgendem sollen Schlüsselwörter definiert werden, mit denen die Kompositionstechnik Methoden-basiert angegeben werden kann. Methoden-basiert bedeutet, dass für jede einzelne Methode eine unterschiedliche Kompositionstechnik angewendet werden kann.

3.2.1 Schlüsselwörter für sprachunabhängige Techniken

In diesem Abschnitt sollen für die sprachunabhängigen Techniken Plain Contracting, Explicit Contract Refinement, Contract Overriding, Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement entschieden werden, ob ein Schlüsselwort notwendig ist und wenn ja, wie dieses heißen soll.

Standardkompositionstechnik

Die grundlegende Idee ist, den Aufwand beim Schreiben und Verfeinern von Kontrakten für Entwickler so gering wie möglich zu halten. Aus diesem Grund soll eine der oben genannten Techniken als Standardkompositionstechnik für die Verfeinerung gelten, damit nicht an jede Methode ein Schlüsselwort geschrieben werden muss. Sollte eine andere Technik für eine Methode benötigt werden, so kann dies durch Verwendung eines Schlüsselworts gekennzeichnet werden.

Da Feature-orientierte Programmierung als Erweiterung der objektorientierten Programmierung gesehen wird, sollen auch die Prinzipien der Objektorientierung verwendet werden. Das Open-Closed Prinzip wird von Meyer [Mey88] für jede modulare Dekompositionstechnik vorausgesetzt und beschreibt, dass Module offen für Erweiterungen sind und geschlossen für Veränderungen, damit es beim Wiederverwenden

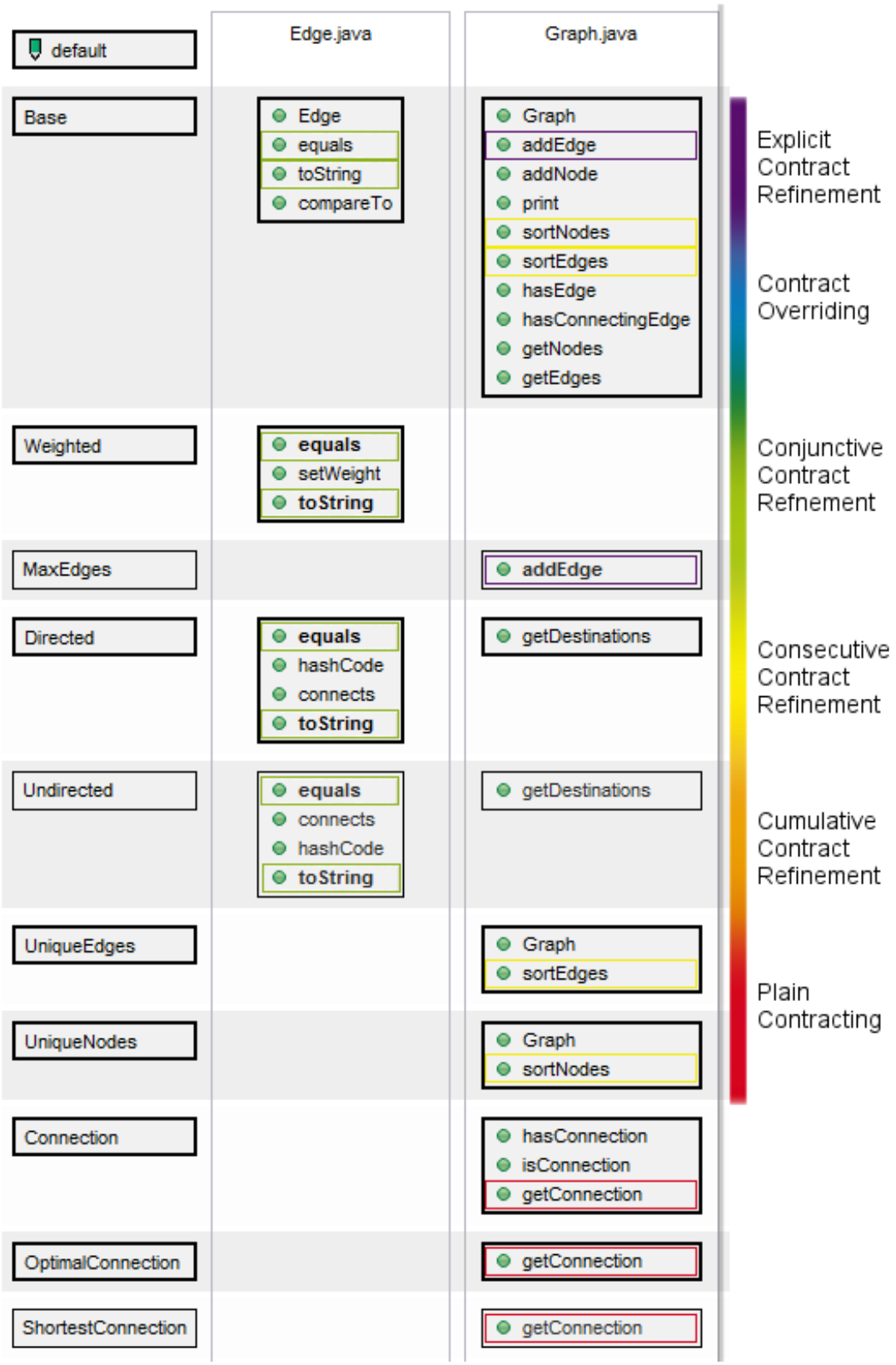

Abbildung 3.11: Auszug aus dem Kollaborationsdiagramm der Graph-Produktlinie. Für die einzelnen Methoden werden die verwendeten Kompositionstechniken mit unterschiedlichen Farben markiert. Methoden, die nicht markiert sind, besitzen entweder keine Kontrakte oder werden nicht verfeinert.

immer das gleiche Verhalten aufweist. Dieses Prinzip kann mit Explicit Contract Refinement umgesetzt werden, da Kontrakte offen für Erweiterungen sind. Im Gegensatz dazu kann man bspw. mit Plain Contracting Erweiterungen verbieten bzw. mit den anderen Techniken die Erweiterbarkeit einschränken.

Zusätzlich können mit Explicit Contract Refinement die Prinzipien Feature-orientierter Programmierung umgesetzt werden, indem Kontrakte mit dem Schlüsselwort `original` verfeinert werden. Diese Art der Verfeinerung ist für Entwickler Feature-orientier Software bekannt, sodass die Verfeinerung von Kontrakten intuitiv anwendbar ist.

Im Gegensatz dazu könnten Plain Contracting, Cumulative, Conjunctive und Consecutive Contract Refinement zu restriktiv für die breite Anwendung sein. Aus diesem Grund soll als Standardkompositionstechnik Explicit Contract Refinement verwendet werden. Jede Methode, welche nicht mit einem Schlüsselwort gekennzeichnet ist, wird mittels Explicit Contract Refinement komponiert.

Contract Overriding

Wie in Abschnitt 3.1.1 auf Seite 22 gezeigt, ist Contract Overriding ein Spezialfall von Explicit Contract Refinement. Dies gilt genau dann, wenn Explicit Contract Refinement das Schlüsselwort `original` nicht verwendet. Die Frage ist, ob es sinnvoll ist, die Verwendung von `original` in Explicit Contract Refinement zu erzwingen und somit zwischen beiden Techniken zu unterscheiden oder ob Contract Overriding durch Explicit Contract Refinement abgebildet werden soll, indem das Schlüsselwort `original` nicht verwendet wird.

Auf der einen Seite könnte der Nutzer mit Hilfe von Entwicklungswerkzeugen bei der Gestaltung von Kontrakten mittels Explicit Contract Refinement unterstützt werden, indem bei Nichtverwendung des Schlüsselworts eine Warnung eingeblendet wird. Außerdem könnte leicht nachvollzogen und unterschieden werden, um welche Kompositionstechnik es sich bei der aktuellen Methode handelt. Wenn das Schlüsselwort `original` im Kontrakt enthalten ist, so weiß der Nutzer, dass es sich um Explicit Contract Refinement handelt und ggf. Kontrakte von Features, die verfeinert werden, mit beachtet werden müssen.

Auf der anderen Seite wird der Nutzer dadurch ggf. weiter eingeschränkt als notwendig. Es muss entschieden werden, ob Kontrakte entweder verfeinert oder überschrieben werden sollen. Dadurch kann flexibel entschieden werden, wie der Kontrakt verfeinert werden soll bzw. ob der Originalkontrakt überschrieben wird. Diese Flexibilität kann besonders in Abhängigkeit der Feature-Auswahl genutzt werden, sodass bspw. optionale Features Kontrakte mit unterschiedlicher Bedeutung umsetzen können. Diese Flexibilität ist auch Prinzip der Feature-orientierten Programmierung. Dort kann der Entwickler ebenfalls selbst frei entscheiden, wann er Methoden verfeinern oder überschreiben will. Dieses Konzept soll auch bei der Kontraktverfeinerung gelten.

Weiterhin kann, selbst wenn eine Benutzung von `original` bei Explicit Contract Refinement gefordert ist, die Einschränkung umgangen werden, um somit die Semantik des originalen Kontrakts zu überschreiben. Dies ergibt sich aus dem Komplementärgesetz der boolschen Algebra: $a \wedge \neg a = 0$ [Das05], wodurch die Entscheidung, ob

```
public class Graph {                                        Verfeinerung
    /*@ requires \original && !\original
      @ || MAXEDGES != null;
      @ ensures \original && !\original
      @ || (\old(countEdges()) < MAXEDGES);
      @*/
    public void addEdge(Edge edge) {}
}
```

Abbildung 3.12: Selbst wenn das Schlüsselwort `original` bei Explicit Contract Refinement genutzt werden muss, können Methodenkontrakte überschrieben werden. Der Ausdruck `original && !original` ist immer falsch, sodass nur der Teil hinter der oder-Verknüpfung für die Erfüllung der Vor- bzw. Nachbedingung verantwortlich ist.

ein Kontrakt erfüllt wird, der Verfeinerung überlassen wird, wie in Abbildung 3.12 gezeigt wird. Der Ausdruck `original && !original` ist immer falsch, sodass nur der Teil hinter der oder-Verknüpfung für die Erfüllung der Vor- bzw. Nachbedingung verantwortlich ist. Das zeigt, dass die zwangsweise Verwendung von `original` leicht umgangen werden kann, um dennoch Kontrakte zu überschreiben.

Aus diesen Gründen ist es nicht notwendig für Contract Overriding ein zusätzliches Schlüsselwort zu definieren. Wird eine Kontraktüberschreibung benötigt kann dies durch die Nichtverwendung des Schlüsselworts `original` bei Explicit Contract Refinement umgesetzt werden.

Für die Techniken Plain Contracting, Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement müssen konkrete Schlüsselwörter definiert werden, weil eine Umsetzung mit Explicit Contract Refinement, sodass die verwendeten Garantien erhalten bleiben, nicht möglich ist.

Cumulative, Conjunctive und Consecutive Contract Refinement

Ein mögliches Konzept für die Schlüsselwörter für Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement könnte die Wiederverwendung der Schlüsselwörter sein. Diese drei Kompositionstechniken besitzen maximal paarweise Gemeinsamkeiten. Cumulative und Consecutive Contract Refinement nutzen zur Komposition der Vorbedingung die oder-Verknüpfung. Die Komposition der Nachbedingung wird bei Cumulative bzw. Conjunctive Contract Refinement mit einer Konjunktion durchgeführt. So könnte man Schlüsselwörter definieren, die in unterschiedlichen Kompositionstechniken umgesetzt werden können und zudem die Funktionsweise der Technik beschreiben.

Für die drei Kompositionstechniken würden vier Schlüsselwörter notwendig sein. Zusätzlich müsste eine Kompositionstechnik mit zwei Schlüsselwörtern beschrieben werden, um sowohl die Vorbedingung als auch die Nachbedingung anzugeben, wie in Tabelle 3.1 auf der nächsten Seite veranschaulicht ist.

Offensichtlich werden bei der Wiederverwendung beschreibender Schlüsselwörter insgesamt mehr Schlüsselwörter benötigt, als vorhandene Techniken. Außerdem wird eine Technik mit zwei Schlüsselwörtern definiert, was aufwändiger ist.

Technik	Schlüsselwort	Bedeutung
Consecutive C. R.	`pre_or`	$\phi' \vee \phi$
	`post_implies`	$(old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)$
Conjunctive C. R.	`pre_and`	$\phi' \wedge \phi$
	`post_and`	$\psi' \wedge \psi$
Cumulative C. R.	`pre_or`	$\phi' \vee \phi$
	`post_and`	$\psi' \wedge \psi$

Tabelle 3.1: Beschreibende Schlüsselwörter für Methoden-basierte Kontraktverfeinerung der Techniken Cumulative, Conjunctive und Consecutive Contract Refinement. Es wird je ein Schlüsselwort für die Vorbedingung und für die Nachbedingung benötigt.

Zusätzlich kann dieses Konzept Missverständnisse hervorrufen bzw. sogar Fehler verursachen. Wird nur ein Schlüsselwort benutzt, weil die Verwendung bspw. noch nicht bekannt ist, würde dies zu Fehlern führen, da es keine Kompositionstechnik gibt, die nur mit einem dieser Schlüsselwörter vollständig beschrieben werden kann. Des Weiteren sind insgesamt vier mögliche Kombinationen der beschreibenden Schlüsselwörter möglich - es gibt aber nur drei Techniken. Die Verwendung der Schlüsselwortkombination `pre_and` und `post_implies` ist überflüssig und kann sich daher verwirrend auswirken, obwohl die Bedeutung Conjunctive Contract Refinement entspricht. Zur Erfüllung eines Kontrakts muss sowohl die Vorbedingung als auch die Nachbedingung erfüllt werden. Die Schlüsselwortkombination `pre_and` und `post_implies` sieht formal folgendermaßen aus:

$$\begin{aligned} & old(\phi') \wedge old(\phi) \wedge ((old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)) \\ \Rightarrow \quad & old(\phi') \wedge old(\phi) \wedge \psi' \wedge \psi \end{aligned}$$

Die Umformung geschieht mit Hilfe des Modus Ponens [Das05], wobei $old(\phi')$ dem ϕ' und $old(\phi)$ dem ϕ aus der Vorbedingung von Conjunctive Contract Refinement entspricht. Die Schlüsselwortkombination `pre_and` und `post_implies` drückt folglich ebenfalls Conjunctive Contract Refinement aus:

$$\begin{aligned} & \phi' \wedge \phi \wedge \psi' \wedge \psi \\ \equiv \quad & old(\phi') \wedge old(\phi) \wedge \psi' \wedge \psi \end{aligned}$$

Eine Alternative zu den beschriebenen Schlüsselwörtern ist die Definition von Schlüsselwörtern, die einen sprechenden Namen im Bezug auf die zu verwendende Technik besitzen. D.h. der Name des Schlüsselworts soll den Namen der Technik ähneln. Insbesondere für Anwender, denen die Kompositionstechniken bekannt sind, wird die Benutzung der Schlüsselwörter vereinfacht. Außerdem sind anstatt vier Schlüsselwörter nur drei Schlüsselwörter notwendig und zur Angabe einer Technik auch nur ein Schlüsselwort (s. Tabelle 3.2 auf der nächsten Seite).

Technik	Schlüsselwort
Consecutive Contract Refinement	`consecutive_contract`
Conjunctive Contract Refinement	`conjunctive_contract`
Cumulative Contract Refinement	`cumulative_contract`

Tabelle 3.2: Sprechende Schlüsselwörter für Methoden-basierte Kontraktverfeinerung der Techniken Cumulative, Conjunctive und Consecutive Contract Refinement.

In beiden vorgeschlagenen Konzepten für die Schlüsselwörter benötigt der Anwender das Verständnis über die Kompositionstechnik. Andernfalls ist die Verwendung der Kompositionstechniken nicht sinnvoll. In dieser Arbeit wird die Annahme getroffen, dass das Verständnis nicht aus der Beschreibung des Schlüsselworts hergeleitet wird, sondern über vorherige Kenntnis der Kompositionstechnik. Aus diesem Grund und der einfacheren Anwendung soll der zweite Vorschlag der sprechenden Schlüsselwörter umgesetzt werden.

Plain Contracting

Mit Plain Contracting wird das Verfeinern bzw. Überschreiben von Kontrakten verhindert. Dies kann mit den vorherigen Techniken bzw. eingeführten Schlüsselwörtern noch nicht erreicht werden, weil diese konkret für das Verfeinern von Kontrakten genutzt werden. Daher benötigt Plain Contracting ein Schlüsselwort, mit dem deutlich wird, dass der Kontrakt nicht verfeinert werden darf.

Analog zu dem Schlüsselwort `final`, welches in Java unter anderem dazu genutzt werden kann, dass Variablen nur einmal initialisiert werden dürfen, soll das Schlüsselwort für Plain Contracting eingeführt werden. Daher soll das Schlüsselwort für Plain Contracting ausdrücken, dass der Kontrakt in einer finalen Version ist und nicht verfeinert werden kann. Das Schlüsselwort `final_contract` bezieht sich dabei sowohl auf den Kontrakt und beschreibt zusätzlich durch das Wort *final*, dass dieser Kontrakt nicht verfeinert werden kann. Daher wird `final_contract` als Schlüsselwort für Plain Contracting gewählt.

3.2.2 Schlüsselwörter für sprachspezifische Techniken

Wie in Abschnitt 3.1.2 erwähnt, kann Pure Method Refinement mit den anderen Kompositionstechniken abgebildet werden, wodurch dafür kein konkretes Schlüsselwort notwendig ist. Allerdings ergibt aus sich der Verwendung von `pure` gekennzeichneten Methoden in anderen Kontrakten ein anderes Problem. `pure` gekennzeichnete Methoden können theoretisch für die Spezifikation von allen anderen Methoden der Software-Produktlinie verwendet werden. Sollte diese Methode unerwünscht verfeinert werden, so könnte sie die Spezifikation der gesamten Software-Produktlinie bzw. aller Varianten, die diese Verfeinerung enthalten, verändern oder sogar fehlerhaft machen.

In Feature-orientierter Programmierung ist es nicht vorgesehen Methodenverfeinerungen zu verbieten. Aus dem genannten Problem wird es allerdings notwendig

insbesondere Methodenverfeinerungen von `pure` gekennzeichneten Methoden verbieten zu können, damit die Korrektheit von anderen Kontrakten gewährleistet ist. Allerdings ist es wichtig, dass nicht generell bei allen `pure` gekennzeichneten Methoden eine Methodenverfeinerung verboten wird, weil auch Verfeinerungen solcher Methoden wünschenswert sein können, um sowohl Methodenverhalten zu ändern und damit auch die Bedeutung von Kontrakten.

Daher soll nur in konkreten Fällen die Möglichkeit bestehen eine Methodenverfeinerung zu verbieten. Ein solches Verbot von Methodenverfeinerungen soll mit Hilfe eines Schlüsselworts gekennzeichnet werden können. Ähnlich wie bei Plain Contracting soll sich das Schlüsselwort an bestehenden Konzepten aus Java orientieren. Dort können auch Methoden als `final` deklariert werden, wodurch eine Methodenüberschreibung durch Subklassen verhindert wird. Das Schlüsselwort `final_method` beinhaltet zum einen den Ausdruck `final` und zum anderen `method`, wodurch deutlich gemacht werden soll, dass diese Methode nicht verfeinert werden darf.

Zusätzlich soll das Schlüsselwort `final_method` auch `final_contract` implizieren. Die Verfeinerung von Kontrakten baut auf dem Konzept der Methodenverfeinerung auf. Wenn eine Methodenverfeinerung nicht mehr möglich ist, so kann auch die Verfeinerung des Kontrakts der Methode nicht mehr erfolgen.

In Abschnitt 3.1.2 ist beschrieben, dass Multiple Specification Cases mit der Verwendung von Explicit Contract Refinement weitere Schlüsselwörter benötigt. Benduhn [Ben12] hat zwei zusätzliche Schlüsselwörter definiert, mit dessen Hilfe ganze Spezifikationen bzw. Kontrakte dieser Spezifikationen referenziert werden können. Für alle anderen Techniken stellen Multiple Specification Cases keine Probleme dar, sodass kein zusätzliches Schlüsselwort für die Methoden-basierte Komposition von Kontrakten benötigt wird.

Die Verwendung von mehreren Vorbedingungen bzw. Nachbedingungen in einem Kontrakt stellt durch die Möglichkeit der Umwandlung in einen Kontrakt mit einer Vor- und Nachbedingung ebenfalls kein Problem dar (s. Abschnitt 3.1.2). Somit wird kein zusätzliches Schlüsselwort für Multiple Preconditions and Postconditions benötigt.

3.2.3 Zusammenfassung

Für die Methoden-basierte Kontraktverfeinerung in Feature-orientierter Programmierung ist es notwendig verschiedene Schlüsselwörter zur Angabe einer Kompositionstechnik zu definieren. Um die Verfeinerung von Kontrakten nah an der Methodenverfeinerung zu halten, wird Explicit Contract Refinement als Standardkompositionsverfahren definiert. Alle Methoden werden, ohne Angabe eines konkreten Schlüsselworts, mit dieser Technik komponiert. Ein Schlüsselwort ist nicht notwendig. Außerdem soll Contract Overriding mit Explicit Contract Refinement abgebildet werden, indem das Schlüsselwort `original` nicht angegeben werden muss.

Für die weiteren Kompositionstechniken wird je ein Schlüsselwort definiert (s. Tabelle 3.3 auf der nächsten Seite). Zusätzlich wird ein Schlüsselwort benötigt, um Methodenverfeinerungen, insbesondere für `pure` gekennzeichnete Methoden, zu verbieten: `final_method`. Das Schlüsselwort `final_method` impliziert außerdem,

dass der Kontrakt der Methode nicht verfeinert werden darf, was auch direkt durch das Schlüsselwort `final_contract` angegeben werden kann.

Technik	Schlüsselwort
Plain Contracting	`final_contract`
Explicit Contract Refinement	-
Contract Overriding	-
Consecutive Contract Refinement	`consecutive_contract`
Conjunctive Contract Refinement	`conjunctive_contract`
Cumulative Contract Refinement	`cumulative_contract`
Verbot von Methodenverfeinerung	`final_method`

Tabelle 3.3: Schlüsselwörter für Methoden-basierte Kontraktverfeinerung.

Mit den definierten Schlüsselwörtern ist es nun möglich Methoden-basiert Kontrakte anzugeben. Dadurch können Garantien bezüglich Methoden über gesamte Software-Produktlinien gewährleistet werden und die Verständlichkeit verbessert werden.

3.3 Anwendung der Schlüsselwörter

In Abschnitt 3.2 wurde die Definition der Schlüsselwörter für die Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung diskutiert. Allerdings genügt es nicht diese Schlüsselwörter zu definieren. Zusätzlich ist es notwendig Restriktionen bei der Anwendung dieser Schlüsselwörter festzulegen. Unter anderem sind noch Fragen offen wie: „Ist das Überschreiben von Schlüsselwörtern erlaubt?“ oder „Dürfen alle Schlüsselwörter von anderen überschrieben werden?“. Diese Fragen zur Anwendung der vorgeschlagenen Schlüsselwörter werden in diesem Abschnitt diskutiert.

3.3.1 Überschreiben von Schlüsselwörtern

Eine der essentiellen Fragen zur Anwendung der Schlüsselwörter ist, ob das Überschreiben von Schlüsselwörtern erlaubt werden soll, um die Kompositionstechnik zu verändern. Ähnlich wie beim Verfeinern bzw. Überschreiben von Methoden oder Kontrakten wäre es denkbar auch dessen Kompositionstechnik zu verfeinern oder überschreiben. Genauer bedeutet dies, dass an jedem Kontrakt einer Methodenverfeinerung ein Schlüsselwort für eine Kompositionstechnik angegeben werden kann.

Die Kompositionstechnik würde sich immer dann ändern, wenn ein neues Schlüsselwort angeben wird, wie in Abbildung 3.13 auf der nächsten Seite beispielhaft angedeutet ist. Dabei würden die ersten beiden Kontrakte c1 und c2 mittels Explicit Contract Refinement komponiert werden. Die weitere Komposition würde mit Consecutive Contract Refinement erfolgen, weil an dem Kontrakt c2 das Schlüsselwort `consecutive_contract` angeben ist.

Die Vorteile beim Überschreiben von Schlüsselwörtern liegen darin, dass Entwickler die Möglichkeit haben auf nicht vorhergesehene Features flexibel zu reagieren,

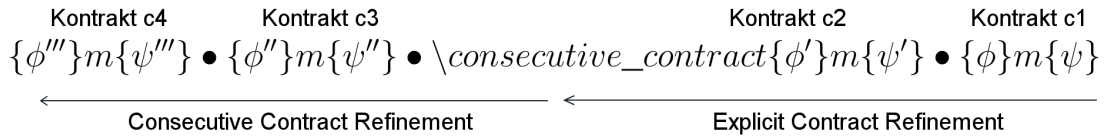

Abbildung 3.13: Beispielhafte Überschreibung der Kompositionstechnik von Kontrakten. Explicit Contract Refinement wird ab Kontrakt c2 mit Consecutive Contract Refinement überschrieben.

wenn diese neu hinzugefügt werden und dessen Kontrakte mit bisheriger Kompositionstechnik nicht ausgedrückt werden können. Zusätzlich bleibt der Gedanke des Verfeinerns von Methoden und Kontrakten auch bei dessen Kompositionstechnik erhalten. D.h. das Verfeinern der Kompositionstechnik ist ebenso erlaubt, wie das Verfeinern von Methoden und Kontrakten.

Auf der anderen Seite ergeben sich durch das Überschreiben von Schlüsselwörtern auch Nachteile und Herausforderungen. Die Garantien, auf die sich Entwickler verlassen können, wie z.B. ein Kontrakt muss immer gelten (wie bei Conjunctive Contract Refinement) oder dass nur genau ein angegebener Kontrakt gilt und dieser nicht verändern werden darf (wie beim Plain Contracting), können durch das Überschreiben verletzt werden. Wird bspw. Plain Contracting durch Explicit Contract Refinement überschrieben, sind diese Garantien nicht mehr gültig. Um diese Garantien auch beim Überschreiben von Schlüsselwörtern zu erhalten, ist es notwendig eine Rangfolge der Schlüsselwörter zu definieren.

3.3.2 Rangfolge der Schlüsselwörter

Die Rangfolge, welche Schlüsselwörter von anderen überschrieben werden dürfen, wird gefordert, damit Garantien eingehalten werden. Konkret bedeutet dies, dass sich Aufrufer von Methoden auf Nachbedingungen verlassen können, sofern bestimmte Vorbedingungen erfüllt sind. Des Weiteren sichert ein Kontrakt mehr Garantien, je restriktiver er ist.

Plain Contracting ist dabei die restriktivste sprachunabhängige Kompositionstechnik, welche die meisten Garantien bei der Methodenausführung gewährleistet, da Kontrakte nicht verfeinert werden dürfen. Somit kann sich der Aufrufer immer auf die gleichen Nachbedingungen verlassen, wenn er die Vorbedingungen erfüllt. Damit diese Eigenschaft erhalten bleibt, soll festgelegt werden, dass Plain Contracting von keiner sprachunabhängigen Kompositionstechnik überschrieben werden darf.

Im Gegensatz dazu steht Explicit Contract Refinement. Mit dieser Kompositionstechnik ist es möglich alle anderen Kompositionstechniken zu simulieren, um sowohl Kontrakte zu verfeinern, als auch zu überschreiben. Allerdings können dabei keine Garantien gewährleistet werden. Jede andere Kompositionstechnik ist restriktiver als Explicit Contract Refinement. Daher soll es möglich sein Explicit Contract Refinement mit jeder anderen Kompositionstechnik zu überschreiben.

Conjunctive Contract Refinement verspricht dem Aufrufer einer Methode, dass alle Nachbedingungen erfüllt werden, sofern alle Vorbedingungen erfüllt sind. Auf der einen Seite entstehen dadurch starke Garantien. Auf der anderen Seite könnte es für

den Aufrufer der Methode schwer sein, alle Vorbedingungen zu erfüllen, wodurch die Wiederverwendung der Methode erschwert wird. Conjunctive Contract Refinement soll ausschließlich Explicit Contract Refinement überschreiben können.

Einen ähnlichen Effekt erreicht man mit Consecutive Contract Refinement. Wenn alle Vorbedingungen erfüllt sind, kann man sich auch auf alle Nachbedingungen verlassen. Allerdings ist es einfacher eine Methodenausführung zu erreichen, weil es genügt mindestens eine Vorbedingung und die dazugehörige Nachbedingung zu erfüllen. Somit können die selben Garantien wie bei Conjunctive Contract Refinement gewährleistet werden und dennoch kann eine Methodenausführung leichter erreicht werden. Aus diesem Grund können sowohl Conjunctive Contract Refinement als auch Explicit Contract Refinement durch Consecutive Contract Refinement überschrieben werden.

Weiterhin können die Garantien von Conjunctive Contract Refinement und Consecutive Contract Refinement mit Cumulative Contract Refinement erreicht werden. Im Gegensatz zu den beiden vorherigen Techniken genügt es bei Cumulative Contract Refinement allerdings mindestens eine Vorbedingung zu erfüllen, um alle Nachbedingungen zu erhalten. Für den Aufrufer einer Methode ist dessen Ausführung dadurch deutlich leichter zu erreichen. Dennoch kann er von den selben Garantien, wie bei Conjunctive Contract Refinement bzw. Consecutive Contract Refinement ausgehen. Cumulative Contract Refinement soll daher die Kompositionstechniken Consecutive Contract Refinement, Conjunctive Contract Refinement und Explicit Contract Refinement überschreiben können.

Das sprachspezifische Schlüsselwort `final_method` bietet die gleichen Garantien wie `final_contract` und hat bezüglich der Methodenkontrakte die gleiche Semantik. Außerdem verbietet `final_method` die Verfeinerung von Methoden, was eine zusätzliche Restriktion mit sich bringt. Aus diesem Grund soll das sprachspezifische Schlüsselwort `final_method` von keinem sprachunabhängigen Schlüsselwort überschrieben werden dürfen, aber all diese Schlüsselwörter überschreiben können.

Zusammengefasst zeigt die Abbildung 3.14 auf der nächsten Seite die Rangfolge der Schlüsselwörter. Die gepunkteten Pfeile drücken transitive Abhängigkeiten aus. `final_method` kann alle Kompositionstechniken überschreiben, aber von keiner anderen überschrieben werden. Das Gegenteil gilt für Explicit Contract Refinement. Plain Contracting kann ausschließlich von `final_method` überschrieben werden, allerdings alle anderen sprachunabhängigen Kompositionstechniken überschreiben. Conjunctive Contract Refinement kann von Consecutive Contract Refinement überschrieben werden, welche wiederum von Cumulative Contract Refinement überschrieben werden kann.

Aus der Rangfolge der Schlüsselwörter folgt ein allgemeines Problem. Generell kann nicht sichergestellt werden, dass Entwickler von Kontrakten die Rangfolge einhalten. Bspw. kann im Basis-Feature das Schlüsselwort `final_contract` angegeben werden und trotzdem in einer folgenden Verfeinerung ein anderes Schlüsselwort wie z.B. `consecutive_contract` verwendet werden. Dies würde zu Fehlern während der Komposition führen, da diese Reihenfolge der Schlüsselwörter nicht erlaubt ist. Daher ist es notwendig den Entwickler bestmöglich bei der Verwendung zu unterstützen, was mit Hinweisen in der Entwicklungsumgebung erreicht werden kann.

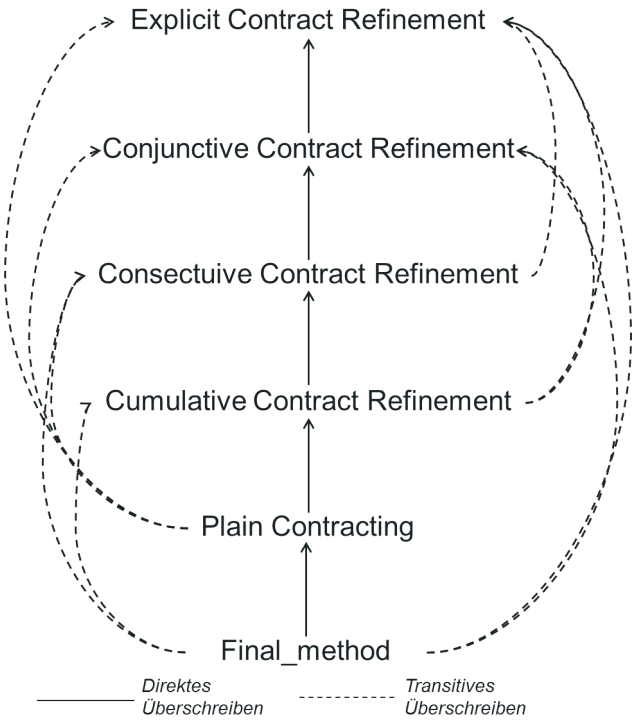

Abbildung 3.14: Rangfolge der Kompositionstechniken. Explicit Contract Refinement kann von allen Kompositionstechniken überschrieben werden. `final_method` kann nicht überschrieben werden, aber alle anderen überschreiben.

3.3.3 Weitere Aspekte

Dieser Abschnitt diskutiert weitere Aspekte zur Anwendung der Schlüsselwörter. Unter anderem werden allgemeine Probleme bei der Verwendung von eigenen Schlüsselwörtern in der Java Modeling Language dargelegt. Außerdem wird darauf eingegangen, welche Vor- und Nachteile eine Entscheidung gegen das Überschreiben der Schlüsselwörter hätte.

Fehlerhafte Benutzung der Schlüsselwörter

Wie im vorherigen Abschnitt erwähnt, liegt die Verantwortung für die korrekte Anwendung der Schlüsselwörter letztendlich beim Anwender. Wird eine falsche Reihenfolge beim Überschreiben der Schlüsselwörter gewählt, so entstehen Fehler bei der Kontraktkomposition.

Durch optionale bzw. alternative Features kann dieses Problem verstärkt werden. Führt ein optionales Feature ein Schlüsselwort ein, besteht die Möglichkeit, dass eine zuvor gültige Variante (ohne das optionale Feature) aufgrund fehlerhafter Kontraktkomposition ungültig wird (mit dem optionalen Feature). Wenn z.B. das optionale Feature das Schlüsselwort `final_method` verwendet, so entstehen Fehler, sobald es eine Verfeinerung des optionalen Features gibt, da weder dessen Kontrakt noch dessen Methode verfeinert werden darf. Der gleiche Effekt kann mit alternativen Features erzeugt werden, bei dem eine Alternative zu einer gültigen Variante führt, während eine andere Alternative durch die fehlerhafte Überschreibung eines Schlüsselworts zu einer ungültigen Variante führt.

Zusätzlich können fehlerhafte Kontrakte durch eine falsche Verwendung des Schlüsselworts `original` führen. Wird dieses Schlüsselwort in einer Vor- bzw. Nachbedingung einer Verfeinerung genutzt, muss das vorherige Feature ebenfalls eine Vor- bzw. Nachbedingung enthalten, damit das Vorkommen von `original` ersetzt werden kann. Ist das nicht der Fall, so bleibt in der Komposition das Schlüsselwort erhalten, was zu einem ungültigen Kontrakt führt.

Das ungültige Vorkommen des Schlüsselworts `original` in einer Kontraktkomposition kann durch das Überschreiben von Schlüsselwörtern für die Kontraktkomposition in optionalen Features verstärkt werden. Wird in einer Variante Explicit Contract Refinement verwendet, welches die einzige Kompositionstechnik ist, bei der `original` verwendet werden darf, und in einem optionalen Feature ein anderes Schlüsselwort eingeführt, so werden in allen nachfolgenden Features die Vorkommen des Schlüsselworts `original` nicht mehr ersetzt. Dadurch entstehen fehlerhafte Kontrakte. Deshalb ist es zwingend notwendig den Nutzer darauf aufmerksam zu machen, dass die Verwendung der eingeführten und bestehenden Schlüsselwörter fehlerhaft ist, damit eine Korrektur erfolgen kann.

Ein weiterer Aspekt bei der Erweiterung der Java Modeling Language um eigene Schlüsselwörter ist, dass diese Schlüsselwörter von Werkzeugen für die Analyse und Ausführung von Kontrakten nicht unterstützt werden. Daher ist es erforderlich nach der Komposition der Kontrakte die eigenen Schlüsselwörter aus den erzeugten Varianten zu entfernen.

Anwendung der Schlüsselwörter ohne Überschreiben

In Abschnitt 3.3.1 wurde die Entscheidung getroffen, dass das Überschreiben von Schlüsselwörtern erlaubt ist, um eine flexible Komposition von Kontrakten zu gewährleisten. Das Verbieten der Überschreibung der Kontraktkomposition würde weitere Probleme mit sich bringen.

So müsste die Angabe der Komposition mit einem Schlüsselwort immer bei Methodeneinführung geschehen, d.h. in dem Feature, welches in der Kompositionsreihenfolge als erstes steht. Ansonsten würde man Explicit Contract Refinement immer dann überschreiben, wenn ein Schlüsselwort später eingeführt wird, was nicht gewollt ist. Jedoch kann in der Feature-orientierten Programmierung im Allgemeinen nicht entschieden werden, welches Feature das Methoden-einführende Feature ist, weil die Möglichkeit besteht, dass das erste Feature in der Kompositionsreihenfolge optional ist. Somit würde dieses optionale Feature die Methode nur dann einführen, wenn es auch ausgewählt ist. Daher ist eine Angabe der Kompositionstechnik im Methoden-einführendem Feature nicht möglich.

Eine Lösung dafür könnte die Verwendung einer zentralen Einheit, z.B. dem Basis-Feature, sein, in der alle Methodeneinführungen (ggf. mit leerem Methodenkörper) stattfinden. Dies würde zusätzlich die Verständlichkeit der Methodenkontrakte verbessern. Allerdings ist der Nachteil dabei, dass das Ziel von der Trennung von Belangen nicht mehr umgesetzt wird. Verschiedene Belange finden sich in einem zentralen Feature wieder. Außerdem wird die Größe des Quelltexts und der Programme unnötig durch die Verwendung von leeren Methoden gesteigert.

Daher wird deutlich, dass sich die Angabe von Schlüsselwörtern an einer zentralen Stelle nicht für die Feature-orientierte Programmierung eignet. Folglich soll die Angabe von Schlüsselwörtern in allen Features möglich sein. Die Verständlichkeit von Kontrakten kann durch Werkzeugunterstützung verbessert werden, indem im Kollaborationsdiagramm bspw. die Kompositionstechnik an die entsprechenden Methoden geschrieben bzw. farblich markiert werden, wie in Abbildung 3.11 auf Seite 35 angedeutet ist.

3.4 Zusammenfassung

In diesem Kapitel wurde das Konzept für die Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung vorgestellt. Dazu wurden die vorhandenen sprachunabhängigen Kompositionstechniken Plain Contracting, Explicit Contract Refinement, Contract Overriding, Cumulative Contract Refinement, Conjunctive Contract Refinement und Consecutive Contract Refinement anhand von Beispielen in Abschnitt 3.1 erläutert. Zusätzlich wurde gezeigt, dass diese Kompositionstechniken auch innerhalb einer Software-Produktlinie verwendet werden sollten. Ebenfalls wurde auf sprachspezifische Techniken in der Java Modeling Language eingegangen. Pure Method Refinement ist eine Möglichkeit zur Verfeinerung von Methodenkontrakten durch die Verfeinerung von `pure` gekennzeichneten Methoden. Deshalb ist diese Technik ein wichtiger Bestandteil für das Konzept der Methoden-basierten Kontraktkomposition.

Durch die Notwendigkeit der Verwendung von unterschiedlichen Kontraktkompositionstechniken innerhalb einer Software-Produktlinie, musste ein Mechanismus gefunden werden, wie die Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung erfolgen kann. In Abschnitt 3.2 wurden dazu Schlüsselwörter (s. Tabelle 3.3 auf Seite 41) definiert, mit denen für jede Methode eine unterschiedliche Kompositionstechnik gewählt werden kann. Dabei wurde festgelegt, dass Explicit Contract Refinement als Standkompositionstechnik verwendet wird. D.h. wird kein Schlüsselwort angegeben gilt diese Kompositionstechnik. Für Contract Overriding wurde kein zusätzliches Schlüsselwort benötigt, weil diese mit Explicit Contract Refinement ausgedrückt werden kann.

Auch für die sprachspezifische Technik Pure Method Refinement musste ein Schlüsselwort definiert werden, mit dem sowohl der Verfeinerung des Kontrakts als auch der Methode selbst verboten wird. Für die restlichen sprachspezifischen Techniken Multiple Preconditions und Postconditions sowie Multiple Specification Cases wurden keine Schlüsselwörter benötigt, weil diese durch Kontraktumformungen eliminiert werden können.

In Abschnitt 3.3 wurde sich dafür entschieden, dass eine Überschreibung der Schlüsselwörter bei dessen Anwendung erlaubt ist. Dies gewährleistet eine flexible Anwendung. Trotzdem ist es notwendig eine Rangfolge der Schlüsselwörter festzulegen, damit erwartete Garantien nicht verloren gehen. Diese Rangfolge ist der Abbildung 3.14 auf Seite 44 zu entnehmen.

Bisher kann die Methoden-basierte Komposition von Kontrakten mit den vorgestellten Techniken nur manuell erfolgen, wodurch die Anwendung aufwändig ist. Damit

die Komposition automatisch auf Basis der Schlüsselwörter erfolgen kann, muss eine Werkzeugunterstützung implementiert werden, was in Kapitel 4 auf Seite 49 umgesetzt wird. Ebenfalls wird damit die korrekte Anwendung der Schlüsselwörter bzw. Rangfolge der Schlüsselwörter überprüft.

4. Implementierung

In Kapitel 3 wurden Schlüsselwörter für die Methoden-basierte Angabe der Kompositionstechniken von Kontrakten in Feature-orientierter Programmierung definiert und Konzepte für deren Anwendung diskutiert. Damit die Komposition der Kontrakte nicht manuell durchgeführt werden muss, sondern automatisiert mit Hilfe von Werkzeugen, sollen die Schlüsselwörter in eine Entwicklungsumgebung eingebunden und die Komposition automatisch durchgeführt werden.

Wie zuvor beschrieben, soll für Design by Contract die Java Modeling Language genutzt werden. Es muss also sichergestellt werden, dass das zu verwendende Werkzeug für Feature-orientierte Programmierung in Java ausgelegt ist und die Java Modeling Language unterstützt. Die zurzeit einzige plattformunabhängige und graphische Entwicklungsumgebung, welche diese Anforderungen erfüllt, ist *FeatureIDE* [TKB+13]. FeatureIDE ist eine Eclipse-basierte Entwicklungsumgebung, die mit verschiedenen Werkzeugen die Entwicklung von Feature-orientierten Programmen in allen Phasen unterstützt und ebenfalls die Komposition von Kontrakten ermöglicht. Weiterhin ist FeatureIDE ein Open Source Produkt, wodurch die Erweiterung durch den zugänglichen Quelltext erleichtert wird.

Für die Komposition von Feature-Modulen stehen zwei Werkzeuge in FeatureIDE zur Verfügung. Die *AHEAD Tool Suite* [Bat06] ist ein Kommandozeilenwerkzeug zur Komposition für die Programmiersprachen Java 1.4, XML und JavaCC. Allerdings ist eine Komposition von Kontrakten nicht möglich. Im Gegensatz dazu ist in dem sprachunabhängigen Kompositionsframework *FeatureHouse* [AKL09] die Komposition von Kontrakten in Java 1.5 bereits vorgesehen. Aus diesem Grund soll in dieser Arbeit FeatureHouse als Kompositionswerkzeug verwendet werden.

In diesem Kapitel soll die Werkzeugunterstützung für die Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung ermöglicht werden. Dazu wird in Abschnitt 4.1 auf der nächsten Seite FeatureHouse um die eingeführten Schlüsselwörter und die Konzepte für eine Methoden-basierte Komposition erweitert. In Abschnitt 4.2 auf Seite 60 wird die Auswahl für die Methoden-basierte Komposition ermöglicht und die Erweiterung von FeatureHouse in FeatureIDE integriert. Sowohl die Erweiterung von FeatureHouse als auch FeatureIDE soll auf

Basis der aktuellen, produktiven Entwicklungsstände geschehen und anschließend als verwendbares Produkt zur Verfügung stehen.

4.1 Erweiterung von FeatureHouse

Für die Komposition von Feature-Modulen soll in dieser Arbeit FeatureHouse erweitert werden. FeatureHouse ermöglicht die Komposition von Software-Artefakten mittels Superimposition. Zwei grundlegende Komponenten werden bei der Komposition verwendet. Mit dem *FSTGenerator* können mit Hilfe einer Grammatik neue Programmiersprachen zu FeatureHouse hinzugefügt werden, welche als Feature Structure Trees dargestellt werden. Der *FSTComposer* ist für die Komposition der Feature Structure Trees verantwortlich.

Die Abbildung 4.1 verdeutlicht die Funktionsweise von FeatureHouse und die Rollen der beiden Komponenten. Um ohne großen manuellen Aufwand neue Sprachen zu FeatureHouse hinzuzufügen, generiert der FSTGenerator die notwendigen Klassen zur Verarbeitung von Quelldateien der jeweiligen Programmiersprache. Der *Lexical Analyzer* bereitet den Inhalt der Quelldateien auf und übergibt diesen als Stream aus Tokens an den *Parser*. Der Parser erstellt daraus einen Feature Structure Tree für die Komposition mittels Superimposition in der zweiten Komponente - dem FSTComposer. Es besteht die Möglichkeit dem FSTComposer eigene Kompositionsregeln zu übergeben. Der komponierte Feature Structure Tree wird schlussendlich durch den *Pretty Printer* als Datei ausgegeben.

Für die Generierung des Lexical Analyzers, des Parsers und des Pretty Printers benötigt der FSTGenerator eine Grammatik der jeweiligen Programmiersprache in einem speziellen Format: die *FeatureBNF*. Das Format der FeatureBNF ähnelt der Backus-Naur-Form.

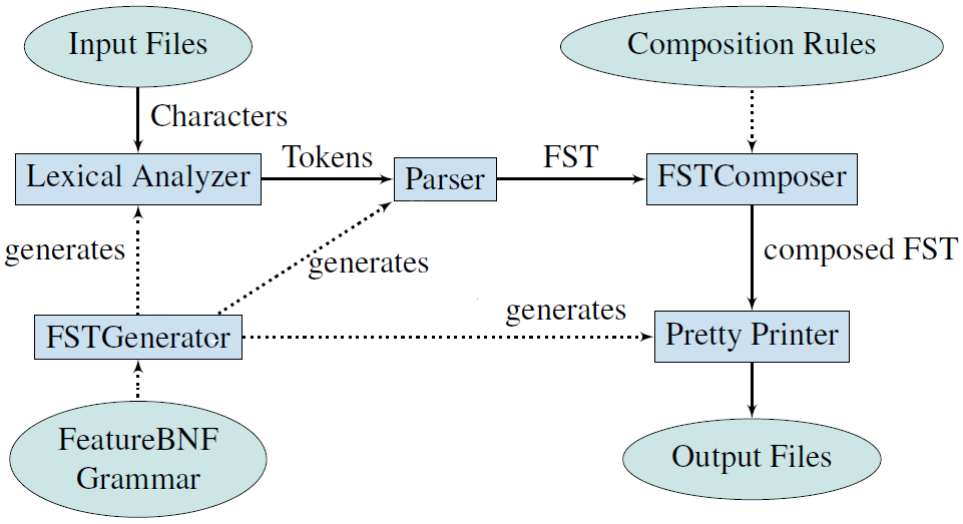

Abbildung 4.1: Architektur von FeatureHouse [Ben12].

Zur Zeit ist in FeatureHouse die Komposition von Kontrakten in der Java Modeling Language mit Plain Contracting, Explicit Contract Refinement, Contract Overriding und Consecutive Contract Refinement möglich. Für diese Kompositionstechniken sind bereits Kompositionsregeln implementiert, die vom FSTComposer

genutzt werden. Ebenfalls sind die Schlüsselwörter `original`, `original_spec` und `original_case` in der Grammatik enthalten und können für Explicit Contract Refinement verwendet werden.

Um die in Abschnitt 3.2 eingeführten Schlüsselwörter in FeatureHouse verfügbar zu machen, muss die FeatureBNF Grammatik für die Java Modeling Language in FeatureHouse erweitert werden, damit die Feature Structure Trees korrekt erzeugt werden. Zusätzlich müssen Kompositionsregeln sowohl für die Kompositionstechniken Cumulative Contract Refinement und Conjunctive Contract Refinement, als auch für die Methoden-basierte Komposition von Kontrakten implementiert werden.

4.1.1 Erweiterung der JML-Grammatik

Die Erweiterung der bestehenden Grammatik für die Java Modeling Language in FeatureHouse um die Verwendung der in Abschnitt 3.2 eingeführten Schlüsselwörter gliedert sich in zwei Teile. Zum einen müssen die definierten Schlüsselwörter der Grammatik bekanntgemacht werden. Die Terminale bzw. Token in der FeatureBNF werden in der Form `<TERMINAL: „Schlüsselwort">` angegeben. Alle Schlüsselwörter aus der Java Modeling Language (s. JML Reference Manual [LPC+11]), sowie die Schlüsselwörter `original`, `original_spec` und `original_case` wurden bereits in dieser Form implementiert [Ben12]. Daher müssen noch die Schlüsselwörter `final_method`, `final_contract`, `conjunctive_contract`, `consecutive_contract` und `cumulative_contract` als Token hinzugefügt werden, was in Abbildung 4.2 zu sehen ist.

```
<IN_JML_SINGLE_LINE, IN_JML_MULTI_LINE>
TOKEN :
{
<ORIGINAL_SPEC: "\\original_spec">
|<ORIGINAL_CASE: "\\original_case">
|<ORIGINAL_CLAUSE: "\\original_clause">
|<ORIGINAL: "\\original">
|<FINAL_CONTRACT: "\\final_contract">
|<FINAL_METHOD: "\\final_method">
|<CONSECUTIVE_CONTRACT: "\\consecutive_contract">
|<CONJUNCTIVE_CONTRACT: "\\conjunctive_contract">
|<CUMULATIVE_CONTRACT: "\\cumulative_contract">
|<HENCE_BY: "hence_by">
...
| < MODEL_PROGRAM: "model_program" >
}
```

Abbildung 4.2: Die hier eingeführten Schlüsselwörter werden der Grammatik für die Java Modeling Language hinzugefügt.

Um die Schlüsselwörter zur Methoden-basierten Komposition von Kontrakten verwenden zu können, muss die Syntax der Methodenspezifikationen in der Grammatik angepasst werden. Bislang besteht eine Spezifikation aus einer `SpecCaseSeq`, also mehreren Specification Cases, und optional aus einem `RedundantSpec`, wie in Abbildung 4.3 auf der nächsten Seite zu sehen ist. Dabei wird angegeben, dass

eine SpecCaseSeq als Terminalknoten im Feature Structure Tree betrachtet werden soll. Für die Komposition zweier SpecCaseSeq wird die Kompositionsregel ContractComposition verwendet.

```
...
@FSTNonTerminal()
Specification:
@% SpecCaseSeq [ RedundantSpec ] @&
;
@FSTTerminal(compose="ContractComposition")
SpecCaseSeq:
SpecCase (LOOK_AHEAD(3) AlsoSpecCase )*
;
AlsoSpecCase:
<ALSO> SpecCase
;
...
RedundantSpec:
Implications
;
Implications:
<IMPLIES_THAT> SpecCaseSeq
;
...
```

Abbildung 4.3: FeatureBNF für einen Kontrakt in der Java Modeling Language ohne eigene Schlüsselwörter.

In der bestehenden Grammatik können die definierten Schlüsselwörter noch nicht verwendet werden. Aus diesem Grund müssen weitere Regeln zur Grammatik hinzugefügt werden, welche die Syntax einer Spezifikation mit Schlüsselwörtern beschreibt. In Abbildung 4.4 auf der nächsten Seite wird diese Syntax angegeben. Diese drückt aus, dass man vor der SpecCaseSeq optional eines der neu definierten Schlüsselwörter angeben kann. Auch dafür wird eine neue Kompositionsregel ContractKeywordComposition definiert.

Mit der erweiterten Grammatik der Java Modeling Language ist es nun möglich Kontrakte mit den neu definierten Schlüsselwörtern zu beschriften. Dabei ist die Verwendung der Schlüsselwörter final_method, final_contract, conjunctive_contract, consecutive_contract und cumulative_contract optional, aber es kann auch nur genau ein Schlüsselwort am Anfang des Kontrakts verwendet werden. Wird ein Schlüsselwort falsch benutzt (z.B. Angabe am Ende des Kontrakts oder fehlerhafte Schreibweise), wird automatisch von FeatureHouse eine ParseException geworfen. Die dazugehörigen Feature Structure Trees werden auf Basis der Quelltextdateien der Feature-Module generiert. Allerdings ist es notwendig die Auswahl der richtigen Kompositionstechnik für die Feature Structure Trees in Abhängigkeit des verwendeten Schlüsselworts zu implementieren, was im nächsten Abschnitt erläutert wird.

4.1.2 Erweiterung der Kompositionsregeln

Für die Komposition von Kontrakten stehen wie zuvor beschrieben zwei Kompositionsregeln zur Verfügung. Die Kompositionsregel ContractKeywordCompo-

```
...
@FSTNonTerminal()
Specification:
@% ContractCompKey SpecCaseSeq [ RedundantSpec ] @&
;
@FSTTerminal(compose="ContractKeywordComposition")
ContractCompKey:
  [ContractCompKeyList]
;
ContractCompKeyList:
  <FINAL_CONTRACT>
 |<FINAL_METHOD>
 |<CONSECUTIVE_CONTRACT>
 |<CUMULATIVE_CONTRACT>
 |<CONJUNCTIVE_CONTRACT>
;
...
```

Abbildung 4.4: Erweiterung der Grammatik um Schlüsselwörter zur Angabe der Kontraktkomposition.

`sition` ist notwendig, um zum einen das verwendete Schlüsselwort an die Komposition weiterzureichen und zum anderen die Rangfolge beim Überschreiben der Schlüsselwörter zu überprüfen, was in Abbildung 4.6 auf Seite 55 zu sehen ist. Die Klasse der Kompositionsregel `ContractKeywordComposition` wird in dieser Arbeit von Grund auf implementiert, da sie in der bisherigen Implementierung von FeatureHouse nicht existierte.

Bei der Komposition der Feature Structure Trees $terminalA \bullet terminalB$ ist darauf zu achten, dass dessen Resultat `terminalComp` in jedem Fall ein Schlüsselwort enthält, wenn eines angeben wurde. Dies ist notwendig, weil `terminalComp` selbst Teil einer folgenden Komposition sein kann. Das Resultat muss das Schlüsselwort des originalen Kontrakts aus `terminalB` enthalten, wenn in dem verfeinerten Kontrakt aus `terminalA` kein Schlüsselwort angegeben ist. Dies ist notwendig, damit eine Kompositionstechnik nicht mit Explicit Contract Refinement überschrieben wird, also wenn kein Schlüsselwort verwendet wird. Oder das Resultat muss das Schlüsselwort aus dem verfeinerten Kontrakt aus `terminalA` enthalten, wenn dort ein Schlüsselwort angeben ist und die Überschreibung korrekt ist.

Überprüfung der Rangfolge der Schlüsselwörter

Zur Überprüfung der Rangfolge wird ein Enumerator `CompositionKeyword` erstellt, der eine Gewichtung für die jeweilige Kompositionstechnik besitzt (s. Abbildung 4.5 auf der nächsten Seite). Auf dessen Basis wird mit der Methode `isValidOrder` (s. Abbildung 4.6 auf Seite 55) überprüft, ob die Überschreibung des Schlüsselworts zulässig ist. Falls die Überschreibung nicht zulässig ist, wird dies durch das Werfen einer `CompositionException` mitgeteilt.

Zusätzlich wird überprüft, ob generell eine Kontraktkomposition stattfindet, wenn der Originalkontrakt das Schlüsselwort `final_contract` und der verfeinerte Kontrakt nicht das Schlüsselwort `final_method` enthält bzw. im Originalkontrakt das

Schlüsselwort `final_method` angegeben wurde. In beiden Fällen soll der Entwickler mit einer Warnung in Form einer `CompositionException` darauf hingewiesen werden, dass der verfeinerte Kontrakt nicht in der Komposition enthalten ist.

```
public enum CompositionKeyword {
    FINAL_METHOD("\\final_method", 0),
    FINAL_CONTRACT("\\final_contract", 1),
    CUMULATIVE_CONTRACT("\\cumulative_contract", 2),
    CONSECUTIVE_CONTRACT("\\consecutive_contract", 3),
    CONJUNCTIVE_CONTRACT("\\conjunctive_contract", 4),
    EXPLICIT_CONTRACT("", 5);

    private String label;
    private int rank;

    private CompositionKeyword(String label, int rank) {
        this.label = label;
        this.rank = rank;
    }
    ...
    public int getRank() {return rank;}
    ...
}
```

Abbildung 4.5: Ausschnitt aus der Implementierung des Enumerators `CompositionKeyword`, welcher den Rang eines Schlüsselworts enthält.

Mit der jetzigen Implementierung kann die korrekte Reihenfolge beim Überschreiben der Schlüsselwörter überprüft werden. Allerdings ist es erforderlich die Komposition der Kontrakte weiter anzupassen, damit die Kompositionstechnik in Abhängigkeit des angegebenen Schlüsselworts angewendet wird. Außerdem besteht noch die Anforderung, dass die Schlüsselwörter nicht mehr in der Ausgabe enthalten sind.

Hinzufügen von Cumulative Contract Refinement und Conjunctive Contract Refinement

Bisher waren nur die Kompositionstechniken Plain Contracting, Explicit Contract Refinement und Consecutive Contract Refinement implementiert. Daher müssen die Techniken Cumulative Contract Refinement und Conjunctive Contract Refinement zuerst umgesetzt werden (s. Abbildung 4.7 auf Seite 56), bevor sie anhand der Schlüsselwörter verwendet werden können. Dazu wird die existierende Implementierung der Kompositionsregel `ContractComposition` erweitert.

Für die Umsetzung der beiden Kompositionstechniken ist es notwendig, dass sowohl Multiple Specification Cases als auch Multiple Preconditions und Postconditions so transformiert werden, dass ein Methodenkontrakt nur genau eine Vorbedingung und eine Nachbedingung enthält. Dies vereinfacht die Komposition der Kontrakte.

Zur Transformation von Multiple Specification Cases (s. Abschnitt 3.1.2 auf Seite 30) wird die Methode `desugarAlso` (s. Abbildung 4.7 auf Seite 56) realisiert. Dabei werden alle Vorkommen des Schlüsselworts `also` in die äquivalente Schreibweise

```
public class ContractKeywordComposition
    extends AbstractCompositionRule {
    ...
    @Override
    public void compose(FSTTerminal terminalA, FSTTerminal terminalB,
    FSTTerminal terminalComp, FSTNonTerminal nonterminalParent) {
        String keywordA = terminalA.getContractCompKey();
        String keywordB = terminalB.getContractCompKey();

        ...
        if (keywordB.equals("\\final_contract")
                    && !keywordB.equals("\\final_method")) {
            throw new CompositionException(null, terminalA,
                "Contracts␣which␣contain␣the␣keyword␣"
                + "\\final_contract␣can␣only␣be␣refined␣"
                + "with␣the␣keyword␣\\final_method!");
        } else if(keywordB.equals("\\final_method")) {
            throw new CompositionException(null, terminalA,
                "It␣is␣not␣allowed␣to␣refine␣a␣contract"
                + "␣which␣contains␣the␣keyword␣\\final_method!");
        }

        if (keywordA.equals("")) {
            terminalComp.setContractCompKey(keywordB);
        } else if (keywordB.equals("") ||
            isValidOrder(keywordA, keywordB)) {
            terminalComp.setContractCompKey(keywordA);
        } else if (!isValidOrder(keywordA, keywordB)) {
            terminalComp.setContractCompKey(keywordB);
            throw new CompositionException(null, terminalA,
                "Overriding␣Keyword␣" + keywordB + "␣with␣"
                    + keywordA + "␣is␣not␣allowed!");;
        }
        ...
        terminalComp.setBody("\n\t");
    }

    private boolean isValidOrder(String keywordA, String keywordB) {
        CompositionKeyword CompKeyA = CompositionKeyword
                .getCompositionKeyword(keywordA);
        CompositionKeyword CompKeyB = CompositionKeyword
                .getCompositionKeyword(keywordB);

        return CompKeyA.getRank() <= CompKeyB.getRank();
    }
}
```

Abbildung 4.6: Ausschnitt aus der Implementierung der Kompositionsregel für die Schlüsselwörter, um diese weiterzureichen und die Überschreibung der Schlüsselwörter auf Basis der Rangfolge zu überprüfen.

transformiert. So erhält man anstatt mehreren Specification Cases einen Kontrakt mit genau einer Vor- und Nachbedingung.

Um Multiple Preconditions und Postconditions zu eliminieren, werden alle Vorbedingungen bzw. alle Nachbedingungen eines Specification Cases mit dem &&-Operator verknüpft (s. Abschnitt 3.1.2 auf Seite 31).

Liegen die Kontrakte der Feature Structure Trees `terminalB` und `terminalA` mit je maximal einer Vor- und Nachbedingung vor, werden die Vorbedingungen beider Kontrakte mit dem ||-Operator für Cumulative Contract Refinement bzw. mit dem &&-Operator für Conjunctive Contract Refinement verknüpft. Die Nachbedingungen werden für beide Kompositionstechniken mit dem &&-Operator verknüpft. Dafür wird die Methode `joinClauses` genutzt (s. Abbildung 4.7).

```
public class ContractComposition extends AbstractCompositionRule {
    ...
    private void cumulativeContracting(FSTTerminal terminalA,
            FSTTerminal terminalB, FSTTerminal terminalComp) {
        if(terminalB.getBody().contains("also"))
            desugarAlso(terminalB);
        if(terminalA.getBody().contains("also"))
            desugarAlso(terminalA);

        List<FSTTerminal> reqClaB = getRequiresClauses(terminalB);
        List<FSTTerminal> reqClaA = getRequiresClauses(terminalA);
        List<FSTTerminal> ensClaB = getEnsuresClauses(terminalB);
        List<FSTTerminal> ensClaA = getEnsuresClauses(terminalA);

        terminalComp.setBody(
            joinClauses(reqClaB, reqClaA, "requires", "||")
                + "\n\t␣" +
            joinClauses(ensClaB, ensClaA, "ensures", "&&"));
    }
    ...
}
```

Abbildung 4.7: Die Klasse für die Kompositionsregel `ContractComposition` wird um die Implementierung der Kompositionstechnik Cumulative Contract Refinement erweitert. Conjunctive Contract Refinement funktioniert analog.

Mit den Implementierungen von Cumulative Contract Refinement und Conjunctive Contract Refinement stehen nun alle Kompositionstechniken für die Methodenbasierte Komposition von Kontrakten zur Verfügung. So kann die Anwendung der Kompositionstechnik auf Basis der Schlüsselwörter realisiert werden, was in Abbildung 4.8 auf der nächsten Seite) gezeigt ist. Dafür wird das Schlüsselwort aus dem Originalkontrakt `terminalB` extrahiert und daraus entschieden, welche Kompositionstechnik verwendet werden muss.

Entfernung des Schlüsselworts in der Komposition

Das Problem bei der Generierung eines Feature Structure Trees für einen Kontrakt bzw. dessen Schlüsselwort ist, dass das angegebene Schlüsselwort in dem Feld `body`

```
public class ContractComposition extends AbstractCompositionRule {
    ...
    private void compositionByKeywords(FSTTerminal terminalA,
            FSTTerminal terminalB, FSTTerminal terminalComp,
            FSTNonTerminal nonterminalParent) {
        String compositionKey = "";

        for (FSTNode n : ((FSTNonTerminal) terminalB.getParent())
            .getChildren())
            if (n.getType().equals("ContractCompKey"))
                compositionKey = ((FSTTerminal) n)
                    .getContractCompose();

        if (compositionKey.equals(CompositionKeyword
                .FINAL_CONTRACT.getLabel())
                || compositionKey.equals(CompositionKeyword
                .FINAL_METHOD.getLabel())) {
            plainContracting(terminalA, terminalB, terminalComp);
        } else if (compositionKey.equals(CompositionKeyword
                .CONSECUTIVE_CONTRACT.getLabel())) {
         consecutiveContracting(terminalA, terminalB, terminalComp);
        } else if (compositionKey.equals(CompositionKeyword
                .CONJUNCTIVE_CONTRACT.getLabel())) {
         conjunctiveContracting(terminalA, terminalB, terminalComp);
        } else if (compositionKey.equals(CompositionKeyword
                .CUMULATIVE_CONTRACT.getLabel())) {
         cumulativeContracting(terminalA, terminalB, terminalComp);
        } else {
            explicitContracting(terminalA, terminalB, terminalComp);
        }

    }
    ...
}
```

Abbildung 4.8: Erweiterung der Klasse für die Kompositionsregel `ContractComposition`, um die Verwendung der Kompositionstechnik auf Basis des angegebenen Schlüsselworts zu entscheiden.

gespeichert wird. Dieses Feld des Feature Structure Trees wird ebenfalls für die Ausgabe in die Zieldatei genutzt. Damit das Schlüsselwort nicht in der Ausgabe enthalten ist, muss das Schlüsselwort aus dem Feld `body` entfernt werden. Allerdings ist es notwendig, dass das Schlüsselwort anderweitig gespeichert wird. Dazu wird die Klasse `FSTTerminal` um das Feld `contractCompKey` erweitert. In diesem Feld wird das Schlüsselwort gespeichert, was in Abbildung 4.9 zu sehen ist.

```
public class FSTTerminal extends FSTNode {
    ...
    private String body;
    ...
    private String contractCompKey;

    public FSTTerminal(String type, String name,
        String body, String prefix) {
        super(type, name);
        this.body = body;
        this.prefix = prefix;
        if (type.equals("ContractCompKey")) {
            this.contractCompKey = body;
            this.body = "\n\t";
        }
    }
    ...
}
```

Abbildung 4.9: Erweiterung der Klasse `ContractComposition` um das Feld `contractCompKey` zur Speicherung des Schlüsselworts für die Kontraktkompositionstechnik.

Konfiguration für den Kompositionsmechanismus

Da weiterhin die Möglichkeit bestehen soll neben der Methoden-basierten Komposition von Kontrakten auch die bisherige Projekt-basierte Komposition zu verwenden, muss die Implementierung der Methoden-basierten Komposition als Erweiterung realisiert werden. Welche Projekt-basierte Technik verwendet werden soll, wird bisher über Konfigurationsparameter gesteuert. Deshalb wird auch für die Methoden-basierte Komposition ein solcher Parameter eingeführt, wie in Abbildung 4.10 auf der nächsten Seite zu sehen ist. Ebenfalls werden Parameter für Cumulative Contract Refinement und Conjunctive Contract Refinement eingeführt, damit diese auch als Projekte-basierte Kompositionstechniken ausgewählt werden können.

Damit der Entwickler auf die fehlerhafte Verwendung des Schlüsselworts `original` (s. Abschnitt 3.3.3) aufmerksam gemacht werden kann, wird mit Hilfe der Methode `checkContainsOriginal` überprüft, ob alle Vorkommen des Schlüsselworts `original` ordnungsgemäß nach der Komposition zweier Kontrakte ersetzt wurden. Andernfalls wird eine Fehlermeldung in Form einer `CompositionException` ausgegeben.

```
public class ContractComposition extends AbstractCompositionRule {
    private static final String PLAIN_CONTRACTING
        = "plain_contracting";
    private static final String CONSECUTIVE_CONTRACTING
    = "consecutive_contracting";
    private static final String EXPLICIT_CONTRACTING
    = "explicit_contracting";
    private static final String CONTRACT_OVERRIDING
    = "contract_overriding";
    private static final String CUMULATIVE_CONTRACTING
    = "cumulative_contracting";
    private static final String CONJUNCTIVE_CONTRACTING
    = "conjunctive_contracting";
    private static final String METHOD_BASED_COMPOSITION
    = "method_based";
    private String contractStyle = PLAIN_CONTRACTING;

    public ContractComposition(String contract_style) {
        if (contract_style != null)
            contractStyle = contract_style.trim();
    }
    @Override
    public void compose(FSTTerminal terminalA, FSTTerminal terminalB,
        FSTTerminal terminalComp, FSTNonTerminal nonterminalParent) {
        if (contractStyle.equals(PLAIN_CONTRACTING)) {
            plainContracting(terminalA, terminalB, terminalComp);
        } else if (contractStyle.equals(CONTRACT_OVERRIDING)) {
            contractOverriding(terminalA, terminalB, terminalComp);
        } else if (contractStyle.equals(EXPLICIT_CONTRACTING)) {
            explicitContracting(terminalA, terminalB, terminalComp);
        } else if (contractStyle.equals(CONSECUTIVE_CONTRACTING)) {
            consecutiveContracting(terminalA,
                terminalB, terminalComp);
        } else if (contractStyle.equals(CUMULATIVE_CONTRACTING)) {
            cumulativeContracting(terminalA,
                terminalB, terminalComp);
        } else if (contractStyle.equals(CONJUNCTIVE_CONTRACTING)) {
            conjunctiveContracting(terminalA,
                terminalB, terminalComp);
        } else if (contractStyle.equals(METHOD_BASED_COMPOSITION)) {
            compositionByKeywords(terminalA, terminalB, terminalComp,
                    nonterminalParent);
        }

        if (checkContainsOriginal(terminalComp))
            throw new CompositionException(terminalA, terminalB,
                "Contract still contains the keyword"
                + "\original, \original_case or \original_spec"
                + "after composition!");
    }
    ...
}
```

Abbildung 4.10: Auswahl der Projekt-basierten bzw. Methoden-basierten Komposition von Kontrakten über Konfigurationsparameter.

Überprüfung von Methodenverfeinerungen

Für die Verwendung des Schlüsselworts `final_method` muss zusätzlich zur Komposition der Kontrakte auch die Methodenkomposition betrachtet werden. D.h. Methoden dürfen nicht verfeinert werden, wenn deren Kontrakt das Schlüsselwort `final_method` enthält. Dazu wird die Kompositionsregel `JavaMethodOverriding`, welche für die Komposition von Methoden zuständig ist, erweitert, indem vor der Komposition der Methoden überprüft wird, ob der originale Kontrakt dieses Schlüsselwort enthält. Ist dies der Fall, wird eine `CompositionException` als Fehlermeldung ausgegeben.

4.1.3 Zusammenfassung

In diesem Abschnitt wurde die Erweiterung von FeatureHouse um die Verwendung von Cumulative Contract Refinement, Conjunctive Contract Refinement und der Methoden-basierten Komposition von Kontrakten vorgestellt. Es können nun die in Abschnitt 3.2 definierten Schlüsselwörter am Anfang eines Methodenkontrakts verwendet werden. Auf dessen Basis wird die Komposition der Kontrakte und im Fall von `final_method` die Methodenkomposition vorgenommen. Dazu wurden alle Konzepte aus Abschnitt 3.3 umgesetzt. Auf Fehler bei der Verwendung der Schlüsselwörter und der Komposition der Kontrakte wird mit einer Fehlermeldung in Form einer `CompositionException` hingewiesen.

Die Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung ist mit der Verwendung des Kommandozeilenwerkzeugs FeatureHouse möglich. Die Anwendung ist allerdings umständlich, da der Kompiliervorgang manuell auf der Kommandozeile durchgeführt werden muss. Außerdem stellt FeatureHouse keine grafische Unterstützung für die Entwicklung Feature-orientierter Programme bereit. Daher soll in Abschnitt 4.2 die Erweiterung von FeatureHouse in die grafische Entwicklungsumgebung FeatureIDE eingebunden werden.

4.2 Erweiterung von FeatureIDE

Die grafische Entwicklungsumgebung FeatureIDE ist ein Eclipse-basiertes Framework zur Entwicklung von Feature-orientierter Software [TKB+13]. FeatureIDE steht als Open Source Produkt zur Verfügung, welches entweder als Eclipse Plugin über den Eclipse Marketplace[1] oder direkt von der Entwicklerseite[2] bezogen werden kann.

FeatureIDE stellt verschiedene Werkzeuge für alle Phasen der Feature-orientierten Software-Entwicklung bereit. Ein grafischer Editor für die Erstellung und Bearbeitung von Feature-Diagrammen hilft bei der Modellierung der Domänenanforderungen. Es besteht die Möglichkeit Feature-Modelle zu importieren oder unter anderem als PDF-Datei oder Bilddatei zu exportieren.

Ein Konfigurationseditor unterstützt den Entwickler während der Anforderungsanalyse, indem auf Basis des Feature-Modells die Konfigurationsmöglichkeiten dargestellt werden. Dabei können Features aus- bzw. abgewählt werden. Zusätzlich werden ungültige Konfigurationen angezeigt.

[1] http://marketplace.eclipse.org/
[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

Für die Domänenimplementierung stehen mehrere unterstützende Werkzeuge zu Verfügung. Implementierungsartefakte werden in Feature-Modulen direkt auf die Features im Feature-Modell bzw. Konfigurationseditor zugeordnet. Eine automatische Erstellung einer Software-Variante wird bei gültiger Feature-Auswahl ermöglicht. Unter anderem kann die Komposition der Feature-Module mit FeatureHouse umgesetzt werden. Zusätzlich werden dem Entwickler ein Kollaborationsdiagramm zur Übersicht, Syntax Highlighting und Fehlerlokalisierung zur besseren Entwicklung angeboten.

FeatureIDE unterstützt auch die Entwicklung von Kontrakten in der Java Modeling Language, sofern FeatureHouse als Kompositionswerkzeug gewählt ist. Bislang standen Plain Contracting, Explicit Contract Refinement, Contract Overriding und Consecutive Contract Refinement als Kompositionstechnik für Kontakte zur Verfügung. Allerdings galt die gewählte Technik für das gesamte Projekt.

Durch die Erweiterung von FeatureHouse in Abschnitt 4.1 um die Kompositionstechniken Cumulative Contract Refinement und Conjunctive Contract Refinement, sollen diese nun auch in FeatureIDE verfügbar gemacht werden. Außerdem soll die Methoden-basierte Komposition von Kontrakten ermöglicht werden.

Konfiguration für den Kompositionsmechanismus

Die Auswahl des Kompositionsmechanismus erfolgt in FeatureHouse wie erwähnt durch Konfigurationsparameter. FeatureIDE erstellt für jedes Projekt eine Konfigurationsdatei, in der unter anderem auch der Parameter für die Technik der Kontraktkomposition gespeichert wird. Der Entwickler kann entweder diese Datei bearbeiten oder über einen grafischen Dialog den zu verwendenden Mechanismus auswählen.

Zur Erweiterung der drei zusätzlichen Kompositionsmechanismen in FeatureIDE wird als erstes die bisherige Implementierung von FeatureHouse, welche als jar-Archiv in FeatureIDE vorliegt, durch die neue Implementierung ausgetauscht. Anschließend wird der Konfigurationsdialog für ein Projekt in FeatureIDE um die drei Kompositionsmechanismen erweitert, wie die Abbildung 4.11 auf der nächsten Seite zeigt. Diese Konfigurationseinstellung kann beim Erstellen eines neuen Projekts ausgewählt und zu jedem späteren Zeitpunkt verändert werden.

Fehlermeldung

FeatureIDE unterstützt den Entwickler bei der Verwendung der Schlüsselwörter, indem bei fehlerhafter Verwendung Fehlermarker an die entsprechende Stelle gesetzt werden. Wird ein Schlüsselwort falsch angegeben, z.B. durch fehlerhafte Schreibweise, wird eine Fehlermeldung mit den erwarteten Schlüsselwörtern ausgegeben, was in Abbildung 4.12 auf Seite 63 zu sehen ist.

Auf eine fehlerhafte Kontraktkomposition wird in FeatureIDE durch Warnungen aufmerksam gemacht. Solche Marker werden in folgenden fällen erzeugt:

- Bei unzulässiger Überschreibung eines Schlüsselworts zur Angabe der Kompositionstechnik (s. Abbildung 4.13 auf Seite 63).

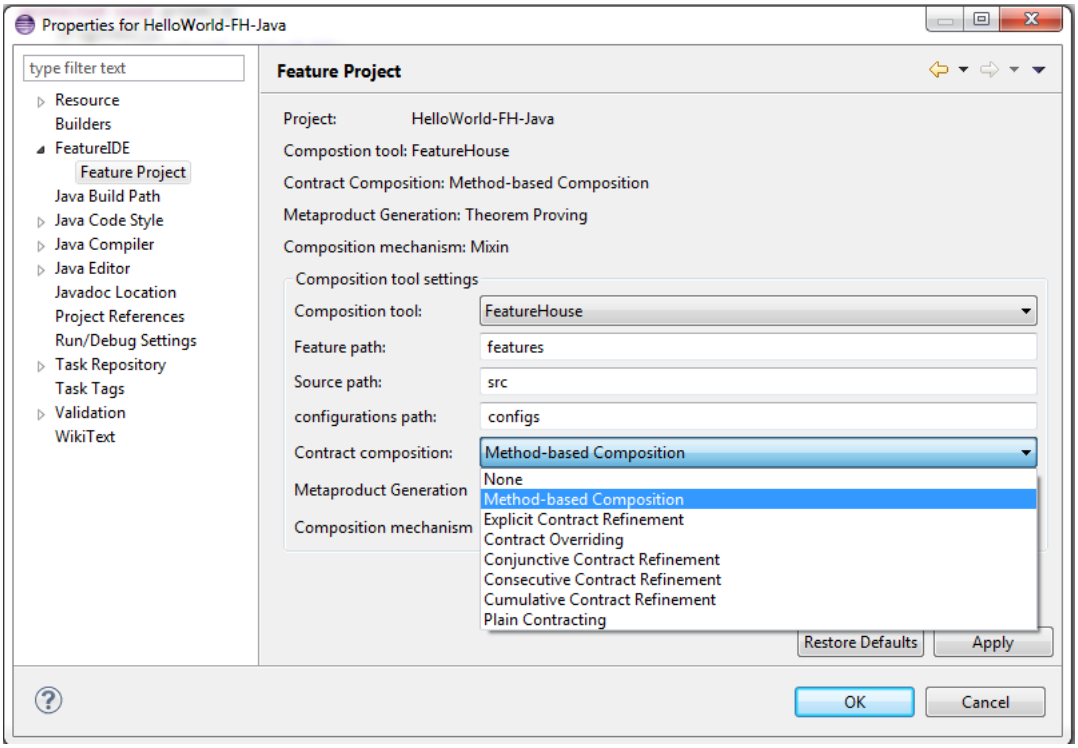

Abbildung 4.11: Auswahl der Kontraktkompositionstechnik in FeatureIDE. Für ein Projekt kann nun die Methoden-basierte Komposition von Kontrakten gewählt werden.

- Bei der Verfeinerung eines mit `final_contract` gekennzeichneten Kontrakts, sofern der verfeinerte Kontrakt nicht das Schlüsselwort `final_method` enthält.
- Bei der Verfeinerung einer mit `final_method` gekennzeichneten Methode bzw. dessen Kontrakt.
- Falls nach der Kontraktkomposition eines der Schlüsselwörter `original`, `original_spec` oder `original_case` enthalten ist.

4.3 Zusammenfassung

Dieses Kapitel stellt die Implementierung der Methoden-basierten Komposition von Kontrakten in Feature-orientierte Programmierung vor. Dazu wurde in Abschnitt 4.1 das Kompositionswerkzeug FeatureHouse um die Verwendung der in Abschnitt 3.2 definierten Schlüsselwörter erweitert, indem diese zur JML-Grammatik hinzugefügt wurden. Zusätzlich wurde die automatische Komposition für die Methoden-basierte Komposition auf Basis des angegebenen Schlüsselworts durchgeführt. Anschließend wurde in Abschnitt 4.1 die Erweiterung von FeatureHouse in FeatureIDE eingebunden, damit die Technik auch dort verfügbar ist. Zur Konfiguration eines Projekts für die Methoden-basierte Komposition wurde der Konfigurationsdialog erweitert. Eine fehlerhafte Verwendung der Schlüsselwörter und eine fehlerhafte Kontraktkomposition wird durch eine Fehlermeldung mitgeteilt.

```
Edge.java[Base]
public class Edge implements Comparable<Edge>{
    /*@ \conjunc_contract
      @ requires ob != null;
      @ ensures \result ==> ob instanceof Edge;
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
```

```
Encountered "\\" (288) at line 2, column 13. Was expecting one of: "}" ... "public"
"synchronized" ... "native" ... "transient" ... "volatile" ... "strictfp" ... "@" ... "spec_
"instance" ... "helper" ... "uninitialized" ... "spec_java_math" ... "spec_safe_math"
"code_bigint_math" ... "non_null" ... "nullable" ... "nullable_by_default" ... "extra
"constraint_redundantly" ... "represents" ... "represents_redundantly" ... "initially
"in_redundantly" ... "maps" ... "maps_redundantly" ... "{" ... "class" ... "interface"
\consecutive_contract" ... "\\cumulative_contract" ... "\\conjunctive_contract" .
"requires" ... "\\pre" ... "requires_redundantly" ... "pre_redundantly" ... "diverges"
"assignable_redundantly" ... "modifiable" ... "modifiable_redundantly" ... "modif
```

```
    /*@ \conjunctive_contract
      @ requires first != null && second != null;
      @*/
    @Override
    public /*@pure@*/ String toString() {
```

Abbildung 4.12: Hinweis bei fehlerhafter Verwendung eines Schlüsselworts zur Angabe der Kompositionstechnik mit Hilfe eines Fehlermarkers in FeatureIDE.

```
Edge.java[Base]
public class Edge implements Comparable<Edge>{
    /*@ \consecutive_contract
      @ requires ob != null;
      @ ensures \result ==> ob instanceof Edge;
      @*/
    @Override
    public /*@pure@*/ boolean equals(Object ob) {
```

```
Edge.java[Weighted]
public class Edge {
    /*@ \conjunctive_contract
     @ requires weight != null;
     @ ensures \result ==> weight == ((Edge)ob).weight;
     @*/
    public /*@pure@*/ boolean equals(Object ob) {
```

```
Overriding keyword \consecutive_contract with \conjunctive_contract is not allowed!
```

Abbildung 4.13: Hinweis bei unzulässiger Überschreibung eines Schlüsselworts zur Angabe der Kompositionstechnik mit Hilfe einer Warnung in FeatureIDE.

Mit der Integration der Methoden-basierten Komposition von Kontrakten in FeatureHouse und FeatureIDE wird eine Werkzeugunterstützung für die Konzepte aus Kapitel 3 bereitgestellt. Es ist nun möglich innerhalb einer Software-Produktlinie unterschiedliche Kompositionstechniken für die Komposition von Kontrakten zu verwenden. Dadurch können unterschiedlich starke Garantien für verschiedene Methoden bzw. Methodenverfeinerungen gewährleistet werden.

Außerdem wird mit der Werkzeugunterstützung eine automatisierte Lösung für die Anwendung bereitgestellt. Eine manuelle Komposition der Kontrakte ist nicht erforderlich, was den Aufwand erleichtert und Fehler durch den Entwickler bei der Komposition vermeidet.

5. Evaluierung

Mit der in Kapitel 4 vorgestellten Erweiterung von FeatureIDE und FeatureHouse besteht nun die Möglichkeit innerhalb einer Software-Produktlinie unterschiedliche Kompositionstechniken auf Methoden bzw. Methodenverfeinerungen anzuwenden. In diesem Kapitel soll zum einen die Implementierung der Methoden-basierten Komposition in Abschnitt 5.1 getestet werden, um zu zeigen, dass die dabei komponierten Kontrakte denen der Projekt-basierten Komposition entsprechen. Zum anderen soll in Abschnitt 5.2 herausgestellt werden, ob bzw. mit welcher Häufigkeit die in Kapitel 3 vorgestellten Kompositionstechniken in Software-Produktlinien angewendet werden können.

Sowohl für die Validierung der Implementierung als auch die Evaluierung der Methoden-basierten Komposition stehen 13 schon mit Kontrakten spezifizierte Fallstudien zur Verfügung. Diese können aus dem Repository für Software-Produktlinien *SPL2go* [Thü] oder als Beispiele in FeatureIDE bezogen werden. Eine kurze Beschreibung der Fallstudien kann der Tabelle 5.1 auf der nächsten Seite entnommen werden. Bei der Evaluierung wird zusätzlich die in dieser Arbeit vorgestellte Graph-Produktlinie betrachtet. Diese Graph-Produktlinie wurde speziell mit dem Fokus auf die Verwendung unterschiedlicher Kontraktkompositionstechniken entwickelt.

5.1 Validierung der Implementierung

In diesem Abschnitt soll die Korrektheit der Implementierung der Methoden-basierten Komposition in FeatureHouse bzw. FeatureIDE überprüft werden. Zur Validierung werden die 13 vorgestellten Fallstudien verwendet, weil diese bereits mit Kontrakten versehen sind und mit vorheriger Implementierung von FeatureHouse und FeatureIDE komponiert werden konnten. Die Kompositionstechnik für Kontrakte der Fallstudien wurde bislang immer projektweit festgelegt. Bisher wurde bei den Fallstudien entweder Explicit Contract Refinement, Contract Overriding, Consecutive Contract Refinement oder Plain Contracting als Kompositionstechnik verwendet (s. Tabelle 5.2 auf Seite 67).

Fallstudie	Beschreibung
BankAccount	Feature-orientiertes Verwaltungssystem für Bankkonten. Umgesetzt werden unter anderem Konzepte wie Konten, Transaktionen, Zinsen, Grenzen für Abbuchungen und Kredite.
IntegerList	Eine Feature-orientierte Implementierung einer Struktur, um Integerwerte in einer Liste (Stack oder sortierte Liste) zu speichern.
UnionFind	Stellt eine Auswahl von Union-Find Implementierungen bereit. Berechnet u.a. die Vereinigung und überprüft, ob zwei Werte in der gleichen Komponente sind.
StringMatcher	Feature-orientierte Implementierung, um zwei Strings auf Gleichheit und Enthalten sein zu testen. Ermöglicht weiterhin den Test auf Präfix bzw. Substring.
DiGraph	Eine Bibliothek, um unterschiedliche Operationen auf gerichteten Graphen durchzuführen, wie z.B. suchen und löschen.
ExamDB	Ein Datenbankmanagementsystem zur Verwaltung von Prüfungen, die von Studenten absolviert werden müssen. Es gibt zusätzliche Features wie z.B. Statistiken und Bonuspunkte.
IntegerSet	Unterschiedliche Darstellung von Mengen durch Bäume oder Hashes. Kann z.B. Werte einfügen oder löschen.
Numbers	Feature-orientierte Spezifikation von Operationen auf realen und komplexen Zahlen, wie Multiplikation, Division oder das Prüfen auf Gleichheit zweier Zahlen.
Paycard	Ein System für die Bezahlung per Smart-Card. Ermöglicht das Loggen von Transaktionen und Anzeigen von Statistiken.
Poker	Feature-orientierte Implementierung eines Poker-Spiels. Durch die Auswahl der Features können die Anzahl der Karten, die ein Spieler maximal haben kann oder die Blinds verändert werden.
Elevator	Ein Fahrstuhl-System, mit dem unter anderem Prioritäten für bestimmte Stockwerke ausgewählt oder das Ignorieren von Knöpfen innerhalb eines leeren Fahrstuhls gesetzt werden können.
Email	Ein Email-System mit Features wie Weiterleitung, Verschlüsselung und Signaturen
GPL_old	Eine Graph-Produktlinie bietet Variabilität von Datenstrukturen und Algorithmen auf Graphen. Wurde geplant als Standardproblem zur Evaluierung von Produktlinientechniken.

Tabelle 5.1: Übersicht der untersuchten Fallstudien. Die ersten vier wurden von Beginn an als Software-Produktlinie mit Kontrakten entwickelt. Die nächsten sechs wurden aus bestehenden Java Programmen (mit Kontrakten) in Feature-Module dekomponiert. Für die letzten drei Fallstudien wurden Kontrakte zu deren bestehenden Feature-Modulen hinzugefügt [TBAS13].

Fallstudie	Kompositionstechnik	Features	Varianten	Paarweise
BankAccount	ECR	8	72	7
IntegerList	ECR	5	16	5
UnionFind	ECR	8	6	6
StringMatcher	ECR	9	28	9
DiGraph	ECR	4	8	5
ExamDB	PC	4	8	4
IntegerSet	Cons. CR	3	2	2
Numbers	ECR	2	2	2
	CO	2	2	2
	Cons. CR	2	2	2
Paycard	Cons. CR	4	6	5
Poker	ECR	10	21	9
Elevator	ECR	6	20	7
Email	ECR	9	40	7
GPL_old	ECR	27	156	19
Gesamt		79	389	91

Tabelle 5.2: Übersicht der untersuchten Fallstudien mit deren bisher verwendeter Kompositionstechnik (ECR = Explicit Contract Refinement, CO = Contract Overriding, Cons. CR = Consecutive Contract Refinement, PC = Plain Contracting), Anzahl der Features, Anzahl der gültigen Varianten und Anzahl Varianten mit paarweiser Featureinteraktion.

Vorgehensweise der Validierung

Zur Validierung der Implementierung der Methoden-basierten Komposition in FeatureHouse bzw. FeatureIDE soll untersucht werden, ob die generierten Kontrakte aus der Methoden-basierten Komposition denen der Projekt-basierten Komposition entsprechen. Dadurch soll sichergestellt werden, dass mit der Methoden-basierten Komposition die Projekt-basierte Komposition abgebildet werden kann. Es wird dabei allerdings nur auf die Korrektheit der Komposition mittels FeatureHouse eingegangen. Eine Verifikation wird nicht durchgeführt. Zu beachten ist, dass schon die Implementierung der Projekt-basierten Komposition fehlerhaft sein kann, wodurch die Komposition nicht korrekt wäre.

Damit die generierten Kontrakte aus der Projekt-basierten Komposition mit denen der Methoden-basierten Komposition verglichen werden können, müssen Anpassungen an den Fallstudien vorgenommen werden. Zum einen muss die Konfiguration jeder Fallstudie verändert werden. Dort muss statt einer konkreten Angabe einer Kompositionstechnik für das gesamte Projekt die Methoden-basierte Komposition ausgewählt werden (s. Abbildung 4.11 auf Seite 62). Anschließend können die eingeführten Schlüsselwörter zur Angabe einer Kompositionstechnik verwendet werden.

Für den Vergleich der generierten Kontrakte ist es notwendig, dass die Komposition bei der Projekt-basierten und Methoden-basierten Komposition mit der selben Kompositionstechnik erfolgt. Deshalb müssen mindestens alle Kontrakte in Methoden-einführenden Features mit dem Schlüsselwort für die entsprechende Kompositionstechnik versehen werden. Es muss kein Schlüsselwort angegeben werden, wenn die Kompositionstechnik Explicit Contract Refinement oder Contract Overriding verwendet wird. Bei der Fallstudie *ExamDB* werden alle Kontrakte in Methoden-einführenden Features mit dem Schlüsselwort `final_contract` versehen, weil die Kompositionstechnik Plain Contracting verwendet wurde. Analog wird das Schlüsselwort `consecutive_contract` bei den Fallstudien *IntegerSet* und *Paycard* zur Angabe der Kompositionstechnik Consecutive Contract Refinement verwendet. Für die Fallstudie *Numbers* müssen die drei Kompositionstechniken Explicit Contract Refinement, Contract Overriding und Consecutive Contract Refinement separat angegeben werden, weil die Fallstudie mehrfach mit diesen Kompositionstechniken vorliegt.

Nachdem die Fallstudien sowohl mit dem Projekt-basierten als auch mit dem Methoden-basierten Kompositionsmechanismus konfiguriert und die entsprechenden Schlüsselwörter angegeben wurden, soll für jede Fallstudie mit beiden Mechanismen gültige Varianten generiert werden. Um den Aufwand für den Vergleich der komponierten Kontrakte gering zu halten und nicht alle Klassen bzw. Interfaces aller Varianten aller Fallstudien vergleichen zu müssen, genügt es Varianten mit paarweiser Featureinteraktion zu betrachten. Varianten mit paarweiser Featureinteraktion verursachen im Allgemeinen die meisten Fehler [JHF+12]. Dazu werden mit dem in FeatureIDE integriertem Werkzeug *SPLCATool*[1] [JHF+12] alle Varianten mit paarweiser Interaktion generiert. Dadurch wird die Anzahl der zu überprüfenden Varianten von anfangs 389 Varianten auf 91 reduziert, wie in Tabelle 5.2 gezeigt ist. Insgesamt werden 539 Klassen und Interfaces für die 91 Varianten generiert.

[1] http://heim.ifi.uio.no/martifag/splcatool/

Die erzeugten Quelltext-Dateien der Projekt-basierten Komposition sollen mit denen der Methoden-basierten Komposition für jede Fallstudie und jede Variante verglichen werden, um zu überprüfen, ob die komponierten Kontrakte gleich sind. Ein manueller Vergleich der Kontrakte aus insgesamt 539 Klassen und Interfaces würde enormen Aufwand erfordern und wäre zudem fehleranfällig. Daher soll der Vergleich automatisiert durch ein Programm zum Vergleichen von Dateien erfolgen.

Vergleichsprogramme mit einer grafischen Oberfläche wie z.B. das *Compare-Plugin*[2] für den Editor *Notepad++*[3] oder das Vergleichswerkzeug in Eclipse erfordern zum einen weiterhin manuellen Aufwand bei der Eingabe der zu vergleichenden Dateien. Außerdem wird das Ergebnis des Vergleichs mit einer grafischen Ausgabe zurückgeliefert, was eine manuelle Dokumentation erfordert.

Eine Alternative sind Kommandozeilenprogramme, bei denen die Dokumente z.B. durch Parameter übergeben werden können und das Ergebnis des Vergleichs durch Rückgabewerte automatisch ausgewertet werden kann, wodurch sich eine Automatisierung mit Hilfe von Skripten anbietet. Das Unix-Werkzeug *diff*[4] ist ein Programm, welches die Kriterien zur Automatisierung des Vergleichs erfüllt.

Die Herausforderung bei dem Vergleich der generierten Kontrakte liegt darin, dass die Formatierung der Kontrakte nach der Komposition sich bei der Projekt-basierten und der Methoden-basierten Komposition unterscheiden kann. Beispielsweise könnten eingefügte Leerzeichen, Tabs oder Zeilenumbrüche zu unterschiedlich formatierten Kontrakten führen, die syntaktisch trotzdem gleich sind. Insbesondere das Problem der Zeilenumbrüche ist mit den vorherigen Programmen nicht zu lösen, weil diese den Vergleich Zeilen-orientiert durchführen. Der Vergleich der Kontrakte in Abbildung 5.1 auf der nächsten Seite würde Ungleichheit als Ergebnis ergeben. Allerdings ist die Bedeutung beider Kontrakte gleich.

Mit dem Kommandozeilenwerkzeug *wdiff*[5] ist es möglich das Problem der unterschiedlichen Formatierung zu lösen. Im Gegensatz zu den vorherig vorgestellten Programmen (z.B. *diff*) vergleicht *wdiff* zwei Dateien Wort für Wort. D.h. *wdiff* erstellt beim Vergleich für jede Datei eine temporäre Datei, in die jedes Wort der Quelldatei in eine neue Zeile geschrieben wird. Anschließend wird ein zeilenweiser Vergleich auf die temporären Dateien mit *diff* ausgeführt und dessen Ausgabe modifiziert ausgegeben. Damit werden Formatierungsunterschiede bei der Komposition nicht betrachtet, sondern ausschließlich die Syntax der Kontrakte betrachtet. Außerdem kann der Vergleich mit *wdiff* automatisiert werden, indem das Kommando über alle Fallstudien und dessen Varianten iteriert. Unterschiede können mit Hilfe des Status Codes von *wdiff* bemerkt werden. Sind die verglichenen Dateien unterschiedlich, so ist der Status Code 1, bei Gleichheit 0 (s. Abbildung 5.2 auf der nächsten Seite).

Ergebnis der Validierung

In diesem Abschnitt werden die Ergebnisse aus der Validierung der Implementierung der Methoden-basierten Kontraktkomposition vorgestellt. Wie zuvor erwähnt wur-

[2] http://sourceforge.net/projects/npp-compare/
[3] http://notepad-plus-plus.org/
[4] http://unixhelp.ed.ac.uk/CGI/man-cgi?diff
[5] http://www.gnu.org/software/wdiff/

```
public class Graph {
    /*@ requires    source != null
&& target != null;
      @ ensures hasConnection( source, target )
      @   ==> isConnection(
\result, source, target );
      @*/
    public ArrayList<Node> getConnection(Node source, Node target) {
        ...
    }
}
```

```
public class Graph {
    /*@ requires source != null && target != null;
      @ ensures hasConnection(source, target)
      @   ==> isConnection(\result, source, target); @*/
    public ArrayList<Node> getConnection(Node source, Node target) {
        ...
    }
}
```

Abbildung 5.1: Exemplarische Darstellung möglicher Formatierungen für den selben Kontrakt. Der erste Kontrakt enthält mehr Leerzeichen, Tabs und Zeilenumbrüche als der zweite Kontrakt. Beide Kontrakte sind syntaktisch äquivalent.

```
IntegerSetSPL/hash$ wdiff integerSet.java_project
    integerSet.java_method > integerSet.java_wdiff
IntegerSetSPL/hash$ echo $?
0
```

Abbildung 5.2: Vergleich der erzeugten Java-Klasse `integerSet` der Variante *hash* der Fallstudie `IntegerSetSPL`. Die beiden Dateien sind inhaltlich gleich.

den 539 Klassen und Interfaces aus 13 Fallstudien sowohl für die Projekt-basierte Komposition, als auch für die Methoden-basierte Komposition erzeugt und anschließend miteinander verglichen.

Eine technische Limitierung der Implementierung bezieht sich auf die Verwendung der Kommentarzeichen in der Java Modeling Language. Es besteht die Möglichkeit zum einen Blockkommentare oder zum anderen einzeilige Kommentare anzugeben. Für die Komposition von Methodenkontrakten muss sichergestellt sein, dass Blockkommentare genutzt werden. Andernfalls können fehlerhafte Kompositionen entstehen.

Der Test der Paycard-Fallstudie (Feature-Diagramm s. Abbildung 5.3) war nur dann korrekt, wenn die Features *Logging* und *MaximumRecord* nicht ausgewählt waren. Andernfalls wurde der Kontrakt der Methode `charge` des Features *Paycard* mit Contract Overriding mit der Verfeinerung aus *LockOut* komponiert, anstatt mit Consecutive Contract Refinement. Der Grund dafür liegt an einem Fehler in der Implementierung der Komposition. Der Fehler konnte durch eine kleine Anpassung korrigiert werden. Das Konzept der Methoden-basierten Komposition wies jedoch keine Fehler auf.

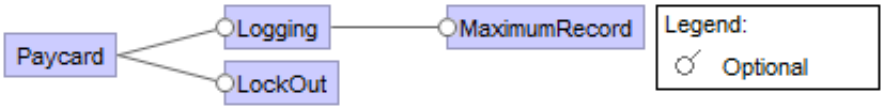

Abbildung 5.3: Feature-Diagramm der Paycard-Fallustudie.

Nachdem alle Methodenkontrakte mit Blockkommentaren angegeben und die korrigierte Version der Methoden-basierten Komposition verwendet wurde, ergab die Validierung der Implementierung keine weiteren Fehler.

Getestet wurden bei der Validierung allerdings nur die Kompositionstechniken Explicit Contract Refinement, Contract Overriding, Consecutive Contract Refinement und Plain Contracting. Für die Kompositionstechniken Conjunctive Contract Refinement und Cumulative Contract Refinement stehen keine Testfälle zur Verfügung, weil diese Kompositionstechniken erst in dieser Arbeit in FeatureHouse und FeatureIDE implementiert wurden. Somit konnten bislang keine Fallstudien mit den Techniken Conjunctive Contract Refinement und Cumulative Contract Refinement in FeatureHouse bzw. FeatureIDE implementiert werden.

In der Fallstudie *Numbers* wird in den Kontrakten das Schlüsselwort `original` sowohl bei Explicit Contract Refinement als auch bei Consecutive Contract Refinement verwendet. Die Verwendung des Schlüsselworts `original` ist allerdings bei Consecutive Contract Refinement nicht erlaubt, weil das Schlüsselwort bei der Komposition nicht ersetzt wird. Sowohl bei der Projekt-basierten als auch bei der Methoden-basierten Komposition ist nach der Komposition das Schlüsselwort `original` im Produkt enthalten. Allerdings wird der Anwender durch die neue Implementierung auf die fehlerhafte Anwendung in Form einer Fehlermeldung aufmerksam gemacht. Fehlerhafte Kontrakte können so besser erkannt und korrigiert werden.

5.2 Evaluierung der Methoden-basierten Komposition

Nachdem die Validierung der Implementierung der Methoden-basierten Komposition erfolgreich anhand von 13 Fallstudien durchgeführt wurde, wird für die Evaluierung die in dieser Arbeit vorgestellte Graph-Produktlinie zusätzlich mit einbezogen. So soll in diesem Abschnitt anhand von 14 Fallstudien herausgestellt werden, welche Kompositionstechnik wie häufig verwendet werden kann.

Vorgehensweise der Evaluierung

Bei der Evaluierung werden, im Gegensatz zur Validierung der Implementierung, 14 Fallstudien betrachtet. Zusätzlich zu den 13 zuvor getesteten Fallstudien wird nun die in dieser Arbeit vorgestellte Graph-Produktlinie für die Evaluierung genutzt. Diese Fallstudie wurde im Zuge dieser Arbeit mit dem Fokus auf die Methoden-basierte Komposition von Kontrakten entwickelt.

Für die Evaluierung der Methoden-basierten Komposition soll untersucht werden, wie häufig die unterschiedlichen Kompositionstechniken in den Fallstudien verwendet werden können. Beim Verwenden der Kompositionstechniken können zwei gegensätzliche Ansätze benutzt werden. Zum einen können Kontrakte mit dem Ziel der Erweiterung geschrieben werden, wodurch möglicherweise schwache Garantien entstehen. Für eine flexible Erweiterbarkeit würde für alle Kontrakte die Kompositionstechnik Explicit Contract Refinement verwendet werden, weshalb sich eine statistische Evaluierung nicht eignen würde. Auf der anderen Seite können größtmögliche Garantien (s. Rangfolge der Schlüsselwörter in Abbildung 3.14) verlangt werden, was die Erweiterbarkeit von Kontrakten bzw. Methoden ggf. einschränkt. Im Allgemeinen wird sich der Entwickler bei der Wahl der Kompositionstechnik zwischen diesen beiden Extremen bewegen. In dieser Arbeit sollen die Kompositionstechniken mit dem Ziel der größtmöglichen Garantien ausgewählt werden, was zu einem deterministischen und vergleichbaren Ergebnis führt.

Zusätzlich wird untersucht, wie oft eine Kompositionstechnik durch eine andere überschrieben wird. Die Herausforderung sind dabei alternative Features. Der erste zu betrachtende Fall ist die Überschreibung von Schlüsselwörtern in einer Alternative, was im Beispiel a) in Abbildung 5.4 auf der nächsten Seite angedeutet ist. Zu entscheiden ist, ob es sich um eine oder zwei Überschreibungen der Kompositionstechnik handelt, wenn in den alternativen Features ein Schlüsselwort eingeführt wird. Da in beiden Fällen das selbe Schlüsselwort (aus *UnionFind*) durch möglicherweise unterschiedliche Schlüsselwörter (in *Quickfind* und *QuickUnion*) überschrieben werden kann, sollen in diesen Fällen zwei Überschreibungen gezählt werden. Dieser Anwendungsfall trat in der UnionFind-Fallstudie und der Graph-Produktlinie auf. Der zweite zu beachtende Fall wird in Beispiel b) in Abbildung 5.4 auf der nächsten Seite veranschaulicht. Es kann sein, dass die Kompositionstechnik von Alternativen Features ebenfalls überschrieben wird. Dabei kann es mehrere Möglichkeiten geben, weil jedes Alternative Feature eine unterschiedliche Kompositionstechnik besitzen kann. Allerdings wird die Kompositionstechnik immer durch dieselbe Kompositionstechnik ersetzt. Daher soll in diesen Fällen eine Überschreibung gezählt werden.

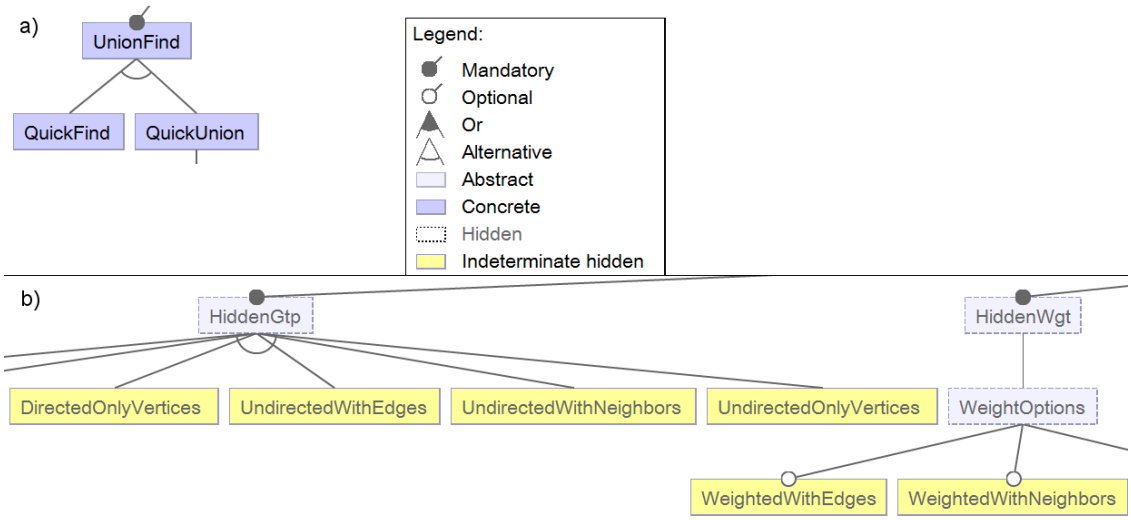

Abbildung 5.4: Ausschnitt aus den Feature-Diagrammen der Union und Graph_old Fallstudie. Die Fallstudien besitzen Alternative Features, auf die für die Evaluierung konkret eingegangen werden musste.

Zum Erreichen der größtmöglichen Garantien werden die Kompositionstechniken der Kontrakte der 14 Fallstudien angepasst. Dabei wird ausschließlich auf die bestehende Implementierung der Fallstudien eingegangen. D.h. es werden keine neuen Kontrakte geschrieben bzw. Kontrakte semantisch verändert. Die bisherigen Kontrakte sollen auch durch die Verwendung der Methoden-basierten Komposition gelten. Methoden mit Kontrakten, die nicht weiter verfeinert werden, sollen mit dem Schlüsselwort `final_method` versehen werden. In den Fällen, in denen derselbe Kontrakt für alle Methodenverfeinerungen gilt, wird das Schlüsselwort `final_contract` verwendet. Andernfalls wird individuell für jede Methode und dessen Verfeinerung überprüft, mit welcher Kompositionstechnik der bisherige Kontrakt mit größten Garantien umgesetzt werden kann. Die Anpassung der Kompositionstechniken für die Graph-Produktlinie kann dem Kollaborationsdiagramm in Abbildung 5.5 auf der nächsten Seite entnommen werden.

Nachdem die Kompositionstechniken in den Fallstudien angepasst wurden, werden für jede Fallstudie folgende Daten erhoben:

- Anzahl aller Methoden: Darunter fallen Methoden, sowohl mit als auch ohne Kontrakte. Methoden werden dabei pro Rolle gezählt, d.h. wenn bspw. eine Methode zwei mal verfeinert wird, werden drei Methoden gezählt. Dies gilt auch für Methoden in Alternativen. Für eine Alternative mit zwei Features und je einer Methode, werden zwei Methoden gezählt.

- Anzahl der Kontrakte: Es werden nur Methoden gezählt, die einen Kontrakt besitzen. Diese Anzahl ist eine Teilmenge aller Methoden.

- Häufigkeit der verwendeten Kompositionstechniken bzw. des Schlüsselworts. Ist ein Schlüsselwort an einem Methodenkontrakt angegeben, kann explizit die Häufigkeit der verwendeten Kompositionstechniken ermittelt werden. Ist kein Schlüsselwort angegeben müssen weitere Fälle unterschieden werden:

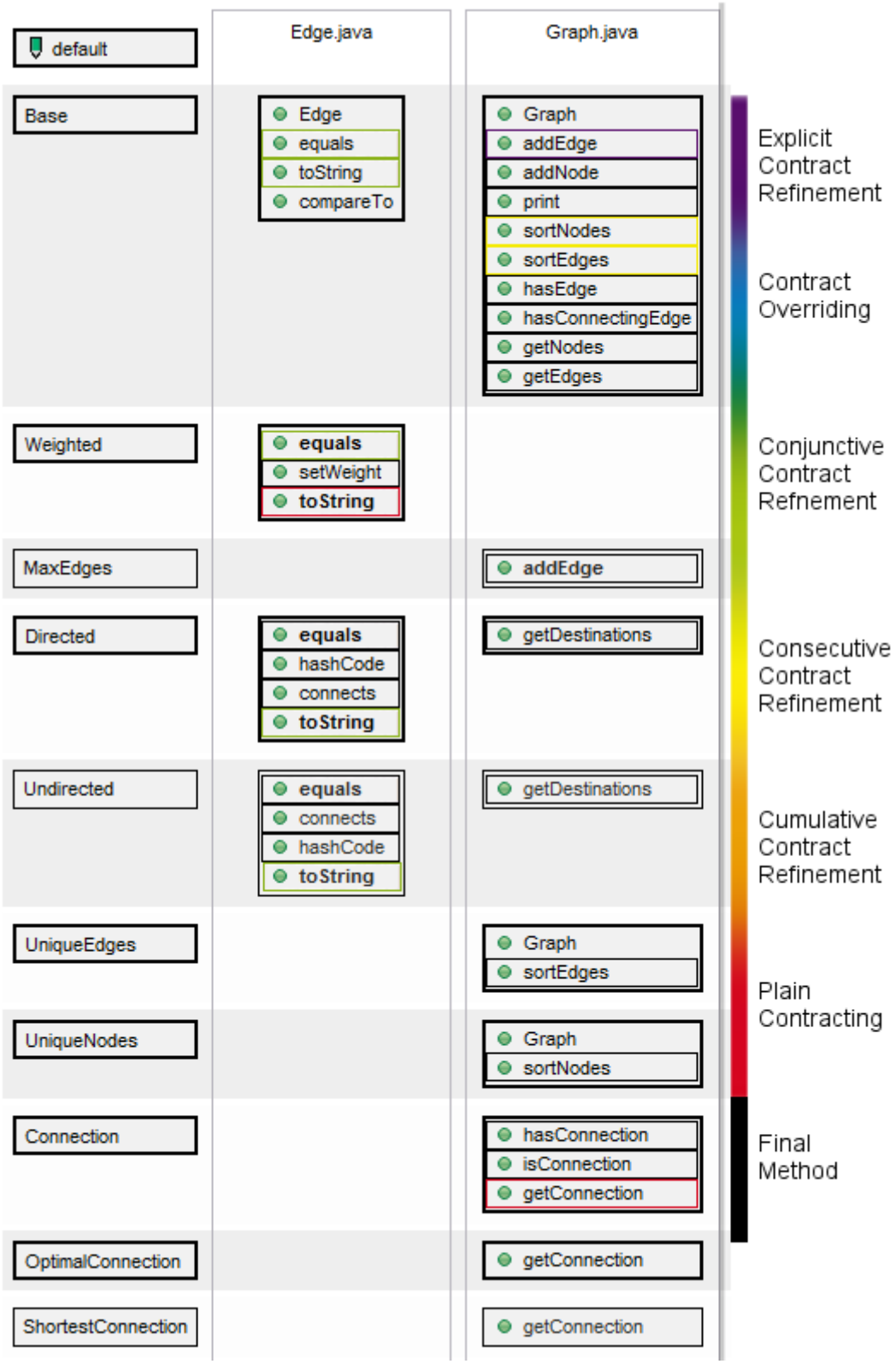

Abbildung 5.5: Auszug aus dem Kollaborationsdiagramm der Graph-Produktlinie. Die Kompositionstechniken wurden so ausgewählt, dass größtmögliche Garantien erreicht werden. Für die einzelnen Methoden werden die verwendeten Kompositionstechniken mit unterschiedlichen Farben markiert. Methoden, die nicht markiert sind, besitzen keine Kontrakte.

- Methode ist nicht im Methoden-einführendem Feature und im vorherigen Kontrakt wurde ein Schlüsselwort verwendet, dann gilt die zuvor verwendete Kompositionstechnik.
- Methode ist nicht im Methoden-einführendem Feature, das Schlüsselwort `original` wurde nicht in der Kontraktverfeinerung benutzt und im vorherigen Kontrakt wurde kein Schlüsselwort zur Angabe einer Kompositionstechnik verwendet und, dann gilt Contract Overriding.
- Ansonsten gilt Explicit Contract Refinement.

Ergebnis der Evaluierung

Die Tabelle 5.3 auf der nächsten Seite zeigt, wie viele Methoden und Methodenkontrakte eine Fallstudie besitzt und wie häufig welche Kompositionstechnik in den Fallstudien verwendet wurde. Deutlich wird, dass weniger als die Hälfte der Methoden mit Kontrakten versehen ist. Dies kann unterschiedliche Gründe haben. Es könnte sein, dass die Methoden auf Grund des Aufwands bei der Spezifizierung ohne Kontrakt entwickelt wurden. Außerdem ist es vorstellbar, dass es mit vorheriger Kompositionstechnik für einige Methoden nicht möglich war Kontrakte entsprechend zu komponieren, weil ausschließlich eine Kompositionstechnik für die gesamte Fallstudie vorgegeben war.

Außerdem ist die häufige Verwendung (83,2% aller Kontrakte) des Schlüsselworts `final_method` auffällig. Das liegt daran, dass viele der Methoden in bisheriger Implementierung nie verfeinert wurden. Dies gilt insbesondere bei den Fallstudien DiGraph und ExamDB bei denen keine Methodenkontrakte mit sprachunabhängigen Techniken verfeinert werden. Allerdings wird Pure Method Refinement für die Kontraktverfeinerung genutzt. Der Grund dafür ist, dass die Variabilität nicht durch die Implementierung, sondern durch dessen Verwendung ermöglicht wird, sodass in der DiGraph keine Methode bzw. in der ExamDB Fallstudie nur eine Methodenverfeinerung ohne Kontrakt vorliegt. Bei der ExamDB-Fallstudie werden die Kontrakte an abstrakte Methoden geschrieben. Diese gelten dann auch für dessen Implementierung.

Zehn Fallstudien verwendeten bei der Projekt-basierten Komposition der Kontrakte Explicit Contract Refinement. Mit der Methoden-basierten Komposition wird diese Kompositionstechnik nur noch in der Elevator-Fallstudie genutzt. Das liegt insbesondere daran, das mit Explicit Contract Refinement die schwächsten Garantien erzeugt werden. Scheinbar können die meisten Kontrakte auch mit anderen Kompositionstechniken umgesetzt werden. Beispielsweise wurde in der StringMatcher Fallstudie für die Kontraktverfeinerungen der Methode `compare` in der Vor- und Nachbedingung ausschließlich das Schlüsselwort `original` verwendet. Dies konnte durch Plain Contracting ebenfalls abgebildet werden.

Cumulative Contract Refinement wird in den Fallstudien ausschließlich dann verwendet, wenn zuvor als Kompositionstechnik Explicit Contract Refinement genutzt wurde. Zusätzlich ist festzustellen, dass fast alle Kontrakte nur aus Nachbedingungen bestanden, die mit einer &&-Verknüpfung komponiert wurden. Im Falle der BankAccount-Fallstudie war in einem Kontrakt eine Vorbedingung angegeben, die

	Methoden	Kontrakte	Explicit CR	Contract Overriding	Conjunctive CR	Consecutive CR	Cumultaive CR	Plain Contracting	Final Method	Überschreibungen
Bank-Account	36	32	0	0	1	0	3	0	28	4
IntegerList	14	7	0	1	0	0	0	1	5	2
UnionFind	17	7	0	0	0	0	1	3	3	2
StringMatcher	13	4	0	0	0	0	0	1	3	0
DiGraph	66	48	0	0	0	0	0	0	48	0
ExamDB	54	25	0	0	0	0	0	0	25	0
IntegerSet	23	15	0	0	0	0	0	3	12	0
Numbers	20	16	0	0	0	6	0	1	9	6
Paycard	28	10	0	0	0	1	0	2	7	1
Poker	63	49	0	0	1	1	0	0	47	2
Elevator	111	13	2	1	0	0	3	1	6	4
Email	71	8	0	4	0	0	0	0	4	3
Graph_old	258	112	0	0	3	6	10	1	92	8
Graph*	60	42	1	0	3	2	0	2	34	6
Gesamt	834	388	3	6	8	16	17	15	323	38
in %	214,9	100,0	0,8	1,5	2,1	4,1	4,4	3,9	83,2	9,8

Tabelle 5.3: Übersicht der untersuchten Fallstudien mit der Häufigkeit der verwendeten Kompositionstechniken (CR steht für Contract Refinement). Zusätzlich werden die Anzahl aller Methoden, Kontrakte und Überschreibungen einer Kompositionstechnik aufgeführt. *Graph** ist die in dieser Arbeit eingeführte Graph-Produktlinie.

allerdings nicht verfeinert wurde. Deshalb konnte auch Cumulative Contract Refinement genutzt werden. Cumulative Contract Refinement konnte jedoch nicht verwendet werden, wenn die Vor- und Nachbedingungen des originalen Kontrakts verfeinert wurden, um größtmögliche Garantien zu erreichen.

Daher stellt sich die Frage, ob Cumulative Contract Refinement wirklich notwendig bzw. zu restriktiv ist. Das Ergebnis der Komposition wäre mit Conjunctive Contract Refinement in allen 17 Fällen bzw. mit Consecutive Contract Refinement in 16 Fällen das Gleiche gewesen. Mit der derzeitigen Implementierung der Fallstudien wären auch die Garantien gleich, was sich allerdings in späteren Versionen der Fallstudien ändern könnte. Cumulative Contract Refinement könnte auch in den Fällen benutzt werden, in denen Plain Contracting oder das Schlüsselwort `final_method` verwendet wurde. Aufgrund schwächerer Garantien (s. Rangfolge der Kompositionstechniken in Abbildung 3.14 auf Seite 44) wurde dies jedoch nicht umgesetzt.

Vier der 14 Fallstudien benötigten keine Überschreibung der Schlüsselwörter. Das liegt daran, dass nur die Kompositionstechniken Plain Contracting bzw. das Schlüsselwort `final_method` verwendet wurden, die nicht überschrieben werden. Im Durchschnitt finden knapp 10% Überschreibungen der Kompositionstechnik statt. Dies verstärkt die Entscheidung zum Überschreiben von Schlüsselwörtern, welche im Abschnitt 3.3 getroffen wurde. Insbesondere wenn größtmögliche Garantien erzeugt werden sollen, ist die Möglichkeit notwendig.

Insgesamt wird aus der Verteilung der Kompositionstechniken in Abbildung 5.6 auf der nächsten Seite deutlich, dass jede der vorgestellten Kompositionstechniken entsprechender Häufigkeit verwendet wurde. Diese Verteilung könnte ein Beispiel für eine einzige Fallstudie sein. Dies kann z.B. erfolgen, wenn alle 14 Fallstudien in einer Fallstudie genutzt werden.

Aus der Art der Entwicklung der Fallstudien mit Kontrakten (von Beginn an als Software-Produktlinie mit Kontrakten, späteres Hinzufügen von Kontrakt zu Feature-Modulen oder Extraktion in Feature-Module) kann keine Tendenz der zu verwendeten Kompositionstechnik gezogen werden, weil die Verteilung der Häufigkeiten in der Tabelle 5.3 keine Rückschlüsse liefert.

5.3 Diskussion

In diesem Abschnitt werden mögliche Kritikpunkte der Validierung der Implementierung und der Evaluierung der Methoden-basierten Komposition vorgestellt.

Validierung der Implementierung

Insgesamt wurden 13 Fallstudien zur Validierung der Implementierung der Methoden-basierten Komposition von Kontrakten in FeatureHouse und FeatureIDE getestet. Der Vergleich der 539 Klassen und Interfaces der 13 getesteten Fallstudien der Projekt-basierten Komposition und Methoden-basierten Komposition zeigt, dass die Komposition in beiden Fällen gleich ist, nachdem ein Fehler in der Implementierung behoben wurde. Allerdings muss beachtet werden, dass dieser Test kein Beweis für die absolute Korrektheit der Implementierung ist. Es könnte möglich sein, dass eine

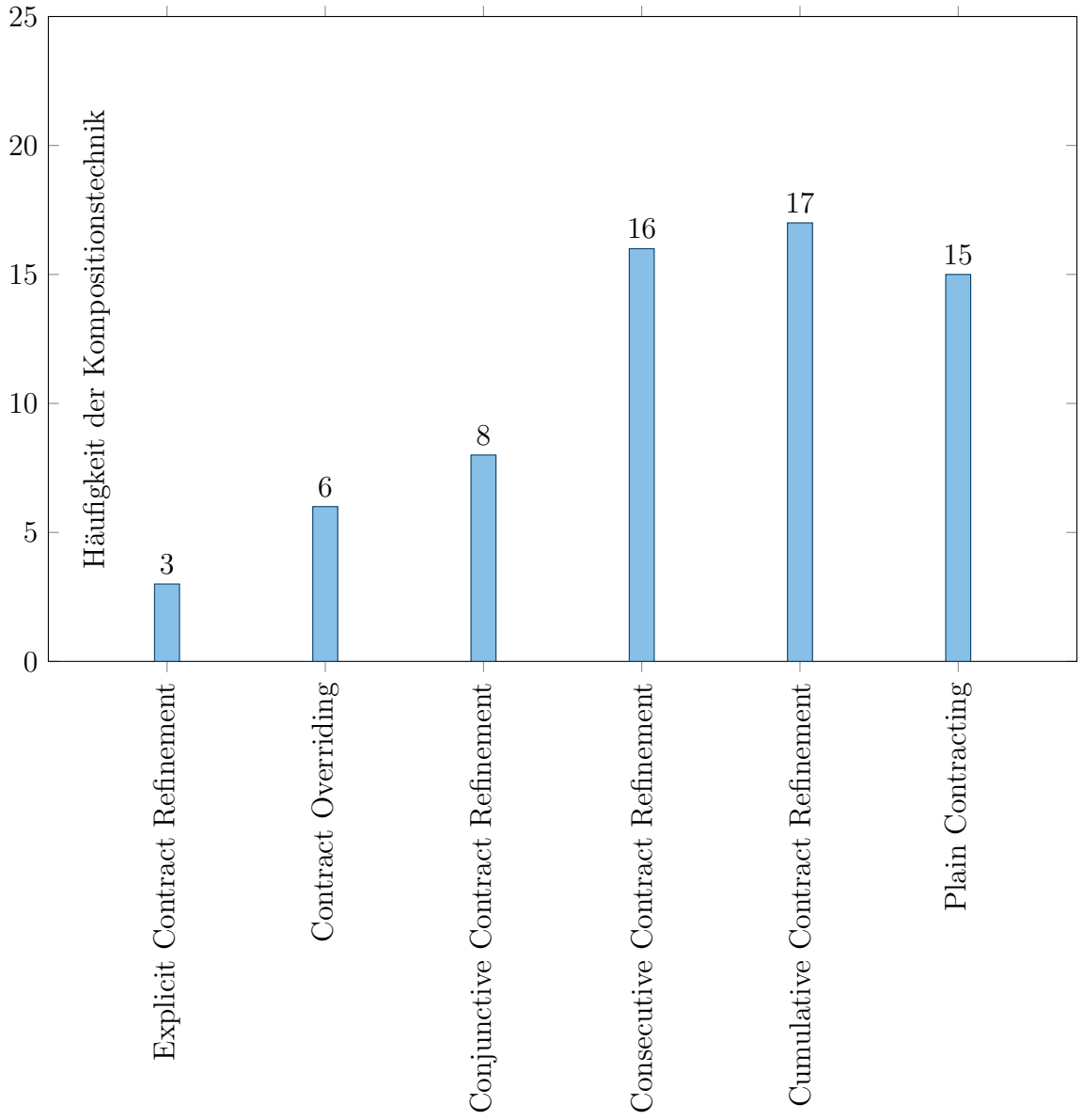

Abbildung 5.6: Darstellung der Häufigkeit der Kompositionstechniken aus allen Fallstudien ohne `final_method`.

neue bzw. andere Fallstudie einen fehlerhaften Vergleich mit sich bringt. Ebenfalls könnte es sein, dass die vorherige Implementierung der Projekt-basierten Komposition fehlerhaft ist.

Um zu gewährleisten, dass die Kontrakte auch semantisch zu den Methoden passen, sollte überprüft werden, ob die Methoden die Kontrakte erfüllen. Mit einer solchen Verifikation kann zusätzlich die Korrektheit der Implementierung unterstützt werden. Da die Verifikation kein direktes Ziel dieser Arbeit ist und den Aufwand übersteigen würde, sollte dies in zukünftigen Arbeiten überprüft werden.

Letztendlich ist mit der Implementierung der Methoden-basierten Komposition noch nicht gewährleistet, dass die Kontrakte korrekt komponiert werden. Allerdings würde eine manuelle Komposition vermutlich häufiger zu fehlerhaften Kontrakten führen, die unregelmäßig bzw. nichtdeterministisch auftreten würden. Mit der automatisierten Lösung würden Fehler schneller auffallen, da diese immer an gleicher Stelle bzw. gleicher Komposition entstehen würden.

Zudem ist die Implementierung eine automatisierte Lösung, um den Aufwand der Komposition zu verringern. Durch den automatischen Build-Prozess in FeatureIDE muss der Entwickler nicht bei jeder Änderung die Komposition manuell ausführen, was weniger zeitintensiv ist.

Evaluierung der Methoden-basierten Komposition

Ein wichtiger Punkt bei der Evaluierung der Methoden-basierten Komposition war, dass größtmögliche Garantien durch die Verwendung der Schlüsselwörter zur Angabe der Kompositionstechniken erreicht werden. Dadurch wird das Ziel der Erweiterbarkeit von Methoden und dessen Kontrakten stark eingeschränkt. Die Schlüsselwörter müssten bei der Erweiterung ggf. umgeschrieben werden. Allerdings wird die Verständlichkeit der Kontrakte durch größtmögliche Garantien gesteigert.

Insbesondere Methoden bzw. Methodenkontrakte, die mit dem Schlüsselwort `final_method` angegeben wurden, müssen bei zukünftigen Erweiterungen angepasst werden. In der derzeitigen Implementierung wurden diese Methoden nicht verfeinert. Eine Verfeinerung dieser Methoden könnte in zukünftigen Erweiterungen allerdings notwendig sein, sodass das Schlüsselwort `final_method` durch eine andere Kompositionstechnik getauscht werden muss.

Zu beachten ist, dass in Menge der Methoden ohne Kontrakte auch die Methoden fallen, bei denen der Kontrakt der Originalmethode mit Plain Contracting geschrieben wurde. Allerdings wurde diese Kompositionstechnik nur für 15 Methodenkontrakte verwendet, sodass trotzdem für ca. die Hälfte der Methoden kein Kontrakt bei der Entwicklung angegeben wurde. Für zukünftige Statistiken sollte dieser Fall separat betrachtet werden, weil es ein Kontrakt für diese Methoden gilt. Insbesondere in der StringMatcher-Fallstudie wird dies deutlich. Laut Tabelle 5.3 haben nur vier der 13 Methoden einen Kontrakt. Allerdings wird die Methode `compare` mit Plain Contracting komponiert, sodass deren sechs Verfeinerungen keine Kontraktverfeinerungen besitzen dürfen.

Zusätzlich muss genauer auf Kontrakte ohne Schlüsselwort zur Angabe der Kompositionstechnik eingegangen werden. Sobald es ein optionales Feature in der Kompositionsreihenfolge gibt, welches ein Schlüsselwort verwendet, und das nachfolgende

Feature kein Schlüsselwort verwendet, ist für das nachfolgende Feature unklar, welche Kompositionstechnik verwendet wird. In den Fallstudien, die in dieser Arbeit betrachtet wurden, gab es diese Konstellation nicht.

Die Verwendung der Methoden-basierten Komposition in den 14 Fallstudien hat gezeigt, dass Cumulative Contract Refinement durch Consecutive bzw. Conjunctive Contract Refinement ausgedrückt werden kann. Dabei werden die Garantien nicht verkleinert. Es gilt zu überprüfen, ob Cumulative Contract Refinement tatsächlich gebraucht wird. Wird diese Kompositionstechnik nicht betrachtet, würde sich die Verteilung in Abbildung 5.6 stärker bei Consecutive oder Conjunctive Contract Refinement anhäufen.

Interessant wäre auch die Evaluierung von realen Software-Produktlinien, die von Anfang an vom Entwickler mit der Methoden-basierten Komposition entwickelt wurden. Durch die fachliche und technische Kenntnis des Entwicklers über die Methoden und Kontrakte würde so nicht nur ein extrem - entweder Erweiterbarkeit oder starke Garantien - fokussiert. Vermutlich würde sich so die Verteilung mehr in Richtung von Kompositionstechniken mit schwächeren Garantien verschieben. Dies gilt es zu überprüfen.

Ein zusätzlicher Punkt bei der Evaluierung ist, dass nicht auf fachliche Punkte der Fallstudien eingegangen wurde. Wahrscheinlich ließen sich weitere Kontrakte in den Fallstudien angeben oder detaillierter spezifizieren. Ebenfalls könnte es durch die Möglichkeit der Verwendung der Methoden-basierten Komposition generell zu anderen Kontrakten kommen. D.h. die Entwickler der Fallstudien hätten schon bei der Entwicklung andere Kontrakte benutzt. Dadurch würde sich eventuell die Häufigkeit der verwendeten Kompositionstechniken anders verteilen.

Außerdem könnte die Größe der evaluierten Fallstudien eine Rollen spielen. Durchschnittlich wurden 28 Kontrakte pro Fallstudie untersucht. Bei deutlich größeren Software-Produktlinien mit mehreren Methoden- und Kontraktverfeinerungen könnte die Verteilung der Kompositionstechniken anders ausfallen.

5.4 Zusammenfassung

In Kapitel 3 wurden die notwendigen Konzepte für die Verwendung der Methoden-basierten Komposition von Kontrakten vorgestellt. Diese wurden in den insgesamt 14 Fallstudien verwendet. Damit die Komposition automatisch ausgeführt werden kann, wurde in Kapitel 4 FeatureHouse und FeatureIDE als Werkzeug für die Komposition erweitert. Diese Implementierung konnte genutzt werden, um die Fallstudien mit Kontrakten bzw. Schlüsselwörtern zu versehen, dessen korrekte Syntax zu überprüfen und die Komposition auszuführen.

In diesem Kapitel wurde zuerst die Implementierung der Methoden-basierten Komposition in FeatureHouse und FeatureIDE getestet. Dazu wurden aus 13 Fallstudien alle Produkte erzeugt, in denen eine paarweise Featureinteraktion stattfindet. Diese wurden zum einem mit der bisherigen Projekt-basierten Komposition erzeugt. Anschließend wurde die Methoden-basierte Komposition verwendet, indem an alle Kontrakte in Methoden-einführenden Features die zuvor verwendete Kompositionstechnik mit den in Abschnitt 3.2 eingeführten Schlüsselwörtern angegeben wurde.

Für die 13 Fallstudien wurden dann mit der Methoden-basierten Komposition die gleichen Produkte wie bei der Projekt-basierten Komposition erzeugt. Die Quelltextdateien der Projekt-basierten Komposition und der Methoden-basierten Komposition wurden dann mit dem Kommandozeilenwerkzeug *wdiff* auf Gleichheit überprüft. Der Vergleich ergab, dass alle 539 generierten Klassen und Interfaces gleich sind.

Im zweiten Schritt wurden die Kontrakte der 13 Fallstudien mit den Schlüsselwörtern der Methoden-basierten Komposition versehen. Dabei stand im Vordergrund größtmögliche Garantien mit der verwendeten Kompositionstechnik zu erreichen. Eine mögliche Verschlechterung der Erweiterbarkeit der Fallstudien musste in Kauf genommen werden. Im Gegenzug wird das modulare Verständnis der Kontrakte verbessert. Anschließend wurde die Häufigkeit der verwendeten Kompositionstechniken in den 13 Fallstudien und der in dieser Arbeit vorgestellten Graph-Produktlinie aufgezeigt. Das Ergebnis zeigt, dass der Großteil der Methoden in den Fallstudien nicht verfeinert wird. Des Weiteren ist zu sehen, dass Cumulative Contract Refinement in den Fallstudien keinen Vorteil mit sich bringt. Für die gleichen Garantien könnte auch Conjunctive oder Consecutive Contract Refinement bzw. für stärkere Garantien Plain Contracting bzw. die das Schlüsselwort `final_method` verwendet werden. Ebenfalls auffällig ist, dass viele Methoden eingangs projektweit Explicit Contract Refinement nutzten, bei der Methoden-basierten Komposition diese Kompositionstechnik durch andere Kompositionstechniken mit stärkeren Garantien verwendeten. Die gute Erweiterbarkeit von Kontrakten durch Explicit Contract Refinement wird in den Fallstudien in vielen Fällen durch Kompositionstechniken mit stärkeren Garantien ersetzt.

Insgesamt wird deutlich, dass bei in den Fallstudien mindestens zwei verschiedene Kompositionstechniken verwendet werden, mit Ausnahme der DiGraph- und ExamDB-Produktlinien, welche keine Methodenverfeinerungen mit Kontrakten haben. Dies zeigt, die Notwendigkeit der Methoden-basierten Komposition von Kontrakten, insbesondere wenn die Garantien maximiert werden sollen, wodurch ein besseres modulares Verständnis erzeugt werden kann.

6. Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten zu den in dieser Arbeit eingeführten Konzepten und der Werkzeugunterstützung für die Methoden-basierten Angabe einer Kompositionstechnik für Kontrakte vorgestellt.

Kontrakte in Feature-orientierter Programmierung

In der Literatur [TSK+12, Ben12, TBAS13] wurde bereits untersucht, wie Design by Contract aus dem objektorientierten Paradigma auf die Feature-orientierte Programmierung angewendet werden kann. Da ein Produkt in Feature-orientierter Programmierung aus mehreren Features besteht, welche bei der Generierung eines Produkt komponiert werden, mussten Konzepte überlegt werden, wie Kontrakte komponiert werden. Dazu wurden insgesamt sieben Techniken vorgeschlagen, wie Kontrakte verfeinert bzw. komponiert werden können [Ben12, TBAS13]. Jede dieser Techniken hat Vor- und Nachteile bezüglich der Erweiterbarkeit von Kontrakten und dem Gewährleisten von Garantien. Es gibt keine Kompositionstechnik, die den anderen überlegen ist. In vergangener Arbeit [Ben12] wurde eine Werkzeugunterstützung zur automatisierten Komposition von Kontrakten umgesetzt. Die zu verwendende Kompositionstechnik wurde dabei für das gesamte Projekt festgelegt. Die Vorteile der einzelnen Kompositionstechniken innerhalb eines Projekts auszunutzen, ist Motivation dieser Arbeit. Mit der Methoden-basierten Komposition von Kontrakten kann nun flexibel für Methoden(-verfeinerungen) Erweiterbarkeit oder starke Garantien erreicht werden. Allerdings ist der Aufwand zur Angabe der Kompositionstechnik höher als bei vorheriger projektweiter Angabe. Deswegen bleibt die Möglichkeit zur Projekt-basierten Angabe einer Kompositionstechnik in FeatureHouse und FeatureIDE weiterhin bestehen.

Kontrakte in weiteren Programmierparadigmen

Design by Contract hat seinen Ursprung in der objektorientierten Programmierung [Mey92a, LC06]. In der objektorientierten Programmierung werden bei der

Vererbung die bisherigen Kontrakte der Superklasse ebenfalls an die Subklasse vererbt. Die Subklasse kann zusätzlich neue Kontrakte in ihrem Geltungsbereich hinzufügen [HLL+12]. Dies entspricht der in dieser Arbeit vorgestellten Kompositionstechnik Consecutive Contract Refinement. Allerdings kann durch das Ergebnis dieser Arbeit auf Methodenebene entschieden werden, wie Kontrakte komponiert werden sollen, sodass die Nachteile von Consecutive Contract Refinement im Bezug auf Erweiterbarkeit und Garantien flexibel behoben werden können.

Neben der Feature-orientierten Programmierung kann das Problem von querschneidenden Belangen mit der Apsect-orientierten Programmierung gelöst werden. Dazu werden Belange in Aspekten modularisiert, welche zusätzliche Funktionalität an bestimmte Stellen im Quelltext einweben [KLM+97]. Aufgrund unterschiedlicher Konzepte in der Aspect-orientierten Programmierung, wie *Join Points* und *Advices* kann formale Spezifikation nicht direkt auf dieses Programmierparadigma angewendet werden. Zhao et al. [ZR03] stellen eine Erweiterung der Java Modeling Language namens *Pipa* vor, mit der Kontrakte für Aspekte verwendet werden können. Aspekte können von anderen abstrakten Aspekten, Klassen und Interfaces erben. Die Kontrakte werden dabei mit vererbt und mit dem Schlüsselwort `also` verknüpft, was Consecutive Contract Refinement entspricht. Die Spezifikationen aus den Aspekten überschreiben beim Einweben in die entsprechenden Stellen im Quelltext deren Spezifikation. Dies entspricht Contract Overriding. Eine solche Überschreibung kann bei querschneidenden Belangen zu Problemen führen, da nicht jede Methode den Kontrakt erfüllen kann. Auf diese Probleme wurde in dieser Arbeit eingegangen. Durch die Verwendung von Schlüsselwörtern zur Angabe der Kompositionstechnik wurde das Problem behoben.

Delta-orientierte Programmierung eignet sich ähnlich wie Feature-orientierte Programmierung für das Entwickeln von Software-Produktlinien. Dabei werden Belange in Delta-Modulen entwickelt. Wie bei der Feature-orientierten Programmierung bestehen Programme aus einer gemeinsamen Menge an Features. Die Delta-Module können Klassen hinzufügen, bearbeiten oder löschen, wodurch Variabilität erzeugt wird [SBB+10]. Hähnle et al. [HS12] haben in vergangener Arbeit Design by Contract für Delta-orientierte Programmierung angewendet. Durch die Möglichkeit der Verfeinerung von Delta-Modulen mussten ebenfalls Konzepte für die Verfeinerung von Kontrakten diskutiert werden. Es ist dabei möglich für Vor- und Nachbedingungen separat zu entscheiden, ob diese hinzugefügt, verändert oder entfernt werden. Dies ist äquivalent zu der in dieser Arbeit vorgeschlagenen Kompositionstechnik Explicit Contract Refinement. Wie in vorherigen Kapiteln diskutiert können mit Explicit Contract Refinement jedoch keine Garantien gewährleistet werden, was einen deutlichen Nachteil haben kann.

Verifikation in Feature-orientierter Programmierung

Durch die Spezifikation von Methodenkontrakten kann, wie in vorherigen Kapiteln erwähnt, die Korrektheit von Feature-Modulen überprüft werden [TAZ+13], indem verifiziert wird, ob die Methoden ihre Kontrakte erfüllen. Eines der wichtigsten Ziele der Verifikation ist die Effizienz zu steigern. D.h. die erforderliche Zeit und Manpower für die Verifikation soll reduziert werden können. In vorheriger Arbeit wurden drei Techniken zur Verifikation von Software-Produktlinien untersucht [TSKA11, TSAH12]. Die Herausforderung bei der Verifikation liegt an der

möglichen Anzahl von Varianten. Beim Produkt-basierten Ansatz werden alle Varianten erzeugt und separat verifiziert. Insbesondere bei steigender Anzahl an Varianten wird der Aufwand zur Verifikation sehr aufwändig und komplex, da viele redundante Prüfungen vorgenommen werden. Für die effizientere Verifikation aller Programmvarianten ohne redundante Überprüfung, kann der *Feature-basierte* Ansatz genutzt werden. Dazu werden für jedes Feature Beweise geschrieben, die durch eine Komposition komplexer gemacht werden, aber nicht erneut bewiesen werden müssen [TSKA11]. Ein weiterer Ansatz für eine effizientere Verifikation wird von Thüm et al. [TSAH12] als *Familien-basierte* Verifikation vorgestellt. Die Idee dabei ist ein Metaprodukt zu generieren, welches alle Features der Produktlinie enthält. Dadurch kann jede mögliche Variante simuliert werden. Der *Familien-basierte* Ansatz ist ebenfalls effizienter als die *Produkt-basierte* Verifikation. Durch die Verwendung von Schlüsselwörtern zur Angabe der Kompositionstechnik könnten weitere Effizienzen für die Verifikation geschaffen werden, indem diese in die Verifikationstechniken mit einbezogen werden.

Automatisierung sprachunabhängiger Komposition - FeatureHouse

Die Möglichkeit für die automatische Komposition neuer Sprachen wurde von Apel et al. [AKL13] geschaffen. Mit dem Kommandozeilen-Werkzeug *FeatureHouse* können beliebige Sprachen komponiert werden. Zur Komposition wird die *Superimposition* verwendet. Dabei werden die Feature Structure Trees der zu komponierenden Software-Artefakte zusammengefügt. Um für beliebige Software-Artefakte automatisch Feature Structure Trees erzeugen zu können, nutzt FeatureHouse eine Grammatik, in der die Struktur definiert wird. Auf Basis der angegeben Grammatik erzeugt ein Generator die notwendigen Hilfsmittel, um die Eingabedateien aus einer Sprache automatisch zu komponierten. Für die Komposition wird die Komponente *FSTComposer* verwendet. Diese komponiert entweder mit einer Standardkompositionstechnik oder eigens definierten Regeln zwei Feature Structure Trees. Mit Hilfe von FeatureHouse können beliebige Sprachen automatisiert komponiert werden. Es wurden bereits unterschiedliche Sprachen aus verschiedenen Paradigmen, wie objektorientierte, funktionale oder Feature-orientierte Sprachen, umgesetzt. In dieser Arbeit wurde die Automatisierung der Methoden-basierten Komposition von Kontrakten in FeatureHouse realisiert. Diese Kombination der Sprachen Java und Java Modeling Language war in diesem Ausmaß zuvor nicht möglich, sodass dies als neue Sprache angesehen werden kann. Die Möglichkeit der automatisierten Komposition konnte erst durch die vorherige Arbeit in dieser Form umgesetzt werden.

7. Zusammenfassung

Um den Bedarf nach komplexer Software zu decken und effizient die Entwicklung solcher Systeme zu ermöglichen, wurden in den vergangen Jahrzehnten unterschiedliche Programmierparadigmen entwickelt. Feature-orientierte Programmierung eignet sich besonders für die Entwicklung variabler Software, also Software-Systeme, die eine gemeinsame Basis besitzen, aber sich anhand unterschiedlicher Features unterscheiden. Die Features werden in Feature-Modulen entwickelt. Die Erstellung eines Produkts erfolgt durch die automatische Komposition von Feature-Modulen.

Weiterhin rückt bei der Entwicklung von Software die Sicherheit von Software-Komponenten in den Fokus. In dieser Arbeit bezieht sich der Begriff der Sicherheit auf die korrekte Ausführung. Dazu eignet sich die Spezifikation von Software. Eine Möglichkeit ist in der objektorientierten Programmierung die Angabe von Kontrakten. Diese Kontrakte werden zwischen der Methode und dessen Aufrufer geschlossen und gewährleisten Garantien auf die sich beide vor bzw. nach der Methodenausführung verlassen können.

Da in Feature-orientierter Programmierung ein Produkt aus mehreren Features bestehen kann, können Methoden und auch dessen Kontrakte durch mehrere Features implementiert werden. Deshalb muss bei der Verwendung von Kontrakten in Feature-orientierter Programmierung auch auf die Komposition von Kontrakten eingegangen werden. Bislang standen sieben Kompositionstechniken für Kontrakte zur Auswahl: Explicit Contract Refinement, Contract Overriding, Conjunctive Contract Refinement, Consecutive Contract Refinement, Cumulative Contract Refinement und Plain Contracting. Die sprachabhängige Kompositionstechnik Pure Methode Refinement kann dabei mit jeder anderen Kompositionstechnik verwendet werden.

Die sechs sprachunabhängigen Techniken unterscheiden sich anhand ihrer Erweiterbarkeit und Garantien, die sie dem Aufrufer einer Methode gewährleisten. So bietet Explicit Contract Refinement eine flexible Erweiterbarkeit, gewährleistet im Gegenzug aber keine Garantien. Auf der anderen Seite ist bei der Kompositionstechnik Plain Contracting eine Erweiterung von Methodenkontrakten nicht möglich. Dadurch werden jedoch größtmögliche Garantien erreicht, was die Verständlichkeit des

Kontrakts verbessert. Bislang konnte ausschließlich eine dieser Techniken für eine gesamte Software-Produktlinie angeben werden, sodass alle Kontrakte mit dieser Technik komponiert wurden.

In dieser Arbeit wurde untersucht, ob es notwendig ist, die Kompositionstechnik für jede Methode bzw. dessen Verfeinerung individuell zu bestimmen. Dadurch können flexibel entweder starke Garantien erzeugt oder eine gute Erweiterbarkeit erreicht werden. Dazu mussten zuerst Konzepte zur Anwendung erstellt werden. Um Methoden-basiert eine Kompositionstechnik anzugeben, wurden Schlüsselwörter definiert: `conjunctive_contract` für Conjunctive Contract Refinement, `consecutive_contract` für Consecutive Contract Refinement, `cumulative_contract` für Cumulative Contract Refinement, `final_contract` für Plain Contract und `final_method`, um Kontrakt- und Methodenverfeinerungen z.B. von `pure` gekennzeichneten Methoden zu verbieten. Um den Aufwand zur Angabe eines Schlüsselworts so gering wie möglich zu halten, wurde festgelegt, dass Explicit Contract Refinement als Standardkompositionstechnik genutzt werden soll. Des Weiteren soll Contract Overriding durch Explicit Contract Refinement abgebildet werden, indem dass Schlüsselwort `original` nicht verwendet wird.

Damit die Kompositionstechniken ähnlich wie auch Methoden bzw. Kontrakte in Feature-orientierter Programmierung verfeinert werden können, soll ein Überschreiben der Schlüsselwörter für die Kompositionstechnik erlaubt werden. Zur Sicherung der Garantien, die durch eine bestimmte Kompositionstechnik gewährleistet werden, wurde eine Rangfolge (s. Abbildung 3.14) festgelegt, wie Kompositionstechniken überschrieben werden dürfen. So kann der Entwickler Schrittweise starke Garantien erzeugen und dennoch die Erweiterbarkeit eines Kontrakts erhalten.

Mit diesen Konzepten ist es möglich, Kompositionstechniken Methoden-basiert zu verwenden. Damit auch die Komposition anhand der definierten Schlüsselwörter automatisch durchgeführt wird, wurde das Kommandozeilen-basierte Kompositionswerkzeug FeatureHouse und die für Feature-orientierte Programmierung ausgelegte Entwicklungsumgebung FeatureIDE um die Methoden-basierte Komposition erweitert. Dadurch wird der Aufwand für die Komposition von Kontrakten und potentielle Fehler verringert. Zusätzlich werden Hinweise bei fehlerhafter Verwendung der Schlüsselwörter bzw. fehlerhafter Komposition gegeben, wodurch eine bessere und benutzerfreundliche Verwendung ermöglicht wird.

Die Implementierung der Methoden-basierten Komposition wurde anschließend an 13 bereits spezifizierten Fallstudien getestet. Dabei wurde gezeigt, dass die Produkte der Projekt-basierten Komposition der Produkte der Methoden-basierten Komposition entsprechen.

Die Kompositionstechnik der 13 Fallstudien wurden so verändert, dass größtmögliche Garantien erreicht werden. Zusätzlich wurde eine Software-Produktlinie - die Graph-Produktlinie - entwickelt, die von Anfang an die Methoden-basierte Komposition verwendet. Nur eine der 14 Fallstudien nutzt ausschließlich eine Kompositionstechnik. In allen anderen Fällen können mindestens zwei verschiedene Kompositionstechniken genutzt werden. Das zeigt die Notwendigkeit der eingeführten Schlüsselwörter zur Methoden-basierten Angabe der Kompositionstechnik.

Cumulative Contract Refinement kann in allen Fällen in den hier verwendeten Fällen durch Conjunctive Contract Refinement bzw. teilweise durch Consecutive Contract Refinement ersetzt werden. Es werden die gleichen Garantien erhalten. Daraus stellt sich die Frage, ob Cumulative Contract Refinement tatsächlich notwendig ist.

Insgesamt steht mit der Methoden-basierten Komposition in Feature-orientierter Programmierung ein Konzept für eine flexible Spezifikation von Methoden bereit. Es kann individuell auf die Erweiterbarkeit oder starke Garantien von Kontrakten eingegangen werden. Dennoch besteht weiterhin die Möglichkeit eine Kompositionstechnik für das gesamte Projekt festzulegen. Durch die Werkzeugunterstützung können der Aufwand und die Fehleranfälligkeit bei der Komposition für den Entwickler reduziert werden.

8. Zukünftige Arbeiten

Das in dieser Arbeit vorgestellte Konzept zur Methoden-basierten Komposition von Kontrakten in Feature-orientierter Programmierung und dessen Implementierung in FeatureHouse und FeatureIDE bietet Ansatzpunkte, die in zukünftigen Arbeiten betrachtet werden können. In diesem Kapitel werden offene Fragen zum Konzept, der Werkzeugunterstützung und der Evaluierung vorgestellt.

Ansatzpunkte zum Konzept

Zur Reduzierung des Aufwands sollte untersucht werden, ob es sinnvoll ist eine Kompositionstechnik für gesamte Module wie Pakete, Klassen und Interfaces anzugeben, welche dann für alle Methoden in diesem Modul gelten soll. Aus der Erfahrung mit der Evaluierung kann es denkbar sein, dass dies für einige Software-Produktlinien gelten könnte. Die Verständlichkeit der Kontrakte könnte dadurch ebenfalls für Entwickler verbessert werden.

Außerdem wurde bislang noch nicht die Notwendigkeit der Komposition von Invarianten bzw. Spezifikationen innerhalb von Methodenkörpern (z.B. `assert`) untersucht. Es muss geprüft werden, ob solche Spezifikationen ebenfalls komponiert werden müssen und wenn ja wie. Diese Komposition sollte dann mit FeatureHouse und FeatureIDE automatisiert werden.

Die in Kapitel 4 vorgestellte Erweiterung von FeatureHouse und FeatureIDE unterstützt den Entwickler bei der Komposition von Methodenkontrakten und gibt Hinweise bei fehlerhafter Verwendung der Schlüsselwörter für die Komposition von Kontrakten. Wünschenswert könnte jedoch zusätzliche Werkzeugunterstützung bei Entwicklung von Software-Produktlinien in Feature-orientierter Programmierung sein. So wird das ungültige Überschreiben von Schlüsselwörtern zur Angabe der Kompositionstechnik erst bei der Komposition zweier Kontrakte geprüft. Für die Prüfung müssten theoretisch alle Programmvarianten erzeugt werden, um festzustellen, ob die Verwendung der Schlüsselwörter für die gesamte Software-Produktlinie korrekt ist. Dies kann unter Umständen großen Aufwand erzeugen. Daher könnte eine Überprüfung auf Basis des Feature-Modells stattfinden, sodass die Korrektheit ohne das

Erzeugen von Programmvarianten stattfindet könnte. Wie viel schneller diese Überprüfung ist und ob alle Fälle damit abgedeckt werden, sollte ebenfalls überprüft und ggf. implementiert werden.

In dieser Arbeit wurde zwar auf die Validierung der Implementierung der Methoden-basierten Komposition eingegangen, allerdings nicht auf die Verifikation der Kontrakte. Durch eine zukünftige Prüfung, ob eine Methode tatsächlich ihren Kontrakt erfüllt, würde die Korrektheit der Implementierung weiter gestärkt werden. Zusätzlich sollte überprüft werden, ob Verifikationstechniken angepasst werden müssen, damit sie für die Spezifikation mit den Schlüsselwörtern der Methoden-basierten Komposition anwendbar sind. Ggf. lassen sich Effizienzen durch die Schlüsselwörter bei der Verifikation steigern.

Ansatzpunkte zur Werkzeugunterstützung

Eine Erweiterung des Kollaborationsdiagramms einer Software-Produktlinie würde den Entwickler in der Verwendung der Methoden-basierten Komposition unterstützen. Im Kollaborationsdiagramm könnten die verwendeten Kompositionstechniken für die Methoden angezeigt werden, ähnlich wie in Abbildung 3.11. Um die Übersichtlichkeit und Verständlichkeit zu erhöhen, sollten auch die verwendeten Kompositionstechniken und Überschreibungen im Kollaborationsdiagramm visualisiert werden. Zusätzlich könnten im Kollaborationsdiagramm auch Hinweise bei fehlerhafter Verwendung angezeigt und Statistiken der verwendeten Schlüsselwörter berechnet werden.

Für eine bessere Benutzbarkeit der Schlüsselwörter sollte dem Entwickler eine Hilfe durch die Entwicklungsumgebung angeboten werden. In Eclipse können mithilfe des Content Assists Vorschläge des zu verwendenen Schlüsselworts gemacht werden. Ggf. könnten dabei Abhängigkeiten zu anderen Features aufzeigt werden, um die Verständlichkeit zu verbessern.

Ansatzpunkte zur Evaluierung

Die in dieser Arbeit diskutierten Konzepte und die entwickelte Werkzeugunterstützung erzielen ihren Zweck nur, wenn sie auch verwendet werden. Die Akzeptanz beim Entwickler wurde hierbei jedoch noch nicht betrachtet. Es wurde gezeigt, dass generell eine Entwicklung von Feature-orientierter Software mit Methoden-basierter Komposition von Kontrakten funktioniert und genutzt werden kann. Zukünftig sollte z.B. anhand von Nutzerstudien die Verbreitung der Methoden-basierten Komposition bei Entwicklern überprüft werden.

Durch eine breitere Verwendung der Feature-orientierten Programmierung mit der Methoden-basierten Komposition von Kontrakten könnte auch evaluiert werden, wie man die Extrema der Erweiterbarkeit bzw. Garantien auf Kontrakten besser verbinden kann. Entwickler können sehr flexibel diese Eigenschaften durch Verwendung unterschiedlicher Schlüsselwörter auszunutzen. Welches Schlüsselwort zur Angabe der Kompositionstechnik in welcher Situation angebracht ist, sollte in der Zukunft geprüft werden.

Literaturverzeichnis

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013. To appear. (zitiert auf Seite 2, 3, 9, 16 und 17)

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the 21st international conference on Software engineering*, pages 221–231, Washington, DC, USA, May 2009. IEEE. (zitiert auf Seite 49)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Engineering (TSE)*, 39(1):63–79, January 2013. (zitiert auf Seite 4, 9 und 85)

[ALB+07] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, 2007. (zitiert auf Seite 9 und 11)

[AP98] Vangalur S. Alagar and Kasi Periyasamy. *Specification of Software Systems*. Springer, New York, Berlin, Heidelberg, 1998. (zitiert auf Seite 3 und 12)

[Ari07] Doga Arinir. *Multidimensionale Separierung der Belange in der Softwareentwicklung durch Feature-Komponenten: Ein Ansatz zur dynamischen Komposition von Anwendungen*. PhD thesis, Ruhr University Bochum, 2007. (zitiert auf Seite 8)

[ASLK10] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170, Washington, DC, USA, 2010. IEEE. (zitiert auf Seite 18)

[Bat06] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Proc. Generative and Transformational Techniques in Software Engineering*, pages 3–35, Berlin, Heidelberg, New York, London, 2006. Springer. (zitiert auf Seite 9 und 49)

[Bau06] Friedrich L. Bauer. Frühe Zeugnisse der „Software“. *Informatik Spektrum*, 29(6):433–441, 2006. (zitiert auf Seite 1)

[Ben12] Fabian Benduhn. Contract-Aware Feature Composition. Master's thesis, University of Magdeburg, October 2012. (zitiert auf Seite xii, 4, 11, 19, 26, 30, 32, 34, 40, 50, 51 und 83)

[BFL+11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: the Spec# Experience. *Comm. ACM*, 54(5):81–91, June 2011. (zitiert auf Seite 3 und 14)

[BSR04] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371, 2004. (zitiert auf Seite 11)

[CGK+] Paul Clements, Birgit Geppert, Kyo Kang, Linda Northrop, John McGregor, Dirk Muthig, Klaus Pohl, and Frank van der Linden. Product Line Hall of Fame. Website. Available online at http://www.splc.net/fame.html; visited on April 5th, 2013. (zitiert auf Seite 2)

[Das05] Jürgen Dassow. *Logik für Informatiker*. B. G. Teubner Verlag, Wiesbaden, 1st edition, April 2005. (zitiert auf Seite 36 und 38)

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, 1993. Springer. (zitiert auf Seite 2)

[Gri01] Frank Griffel. *Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine: ein Komponenten-basierter Ansatz zur Generativen Softwarekonstruktion*. PhD thesis, Universität Hamburg, Von-Melle-Park 3, 20146 Hamburg, 2001. (zitiert auf Seite 9)

[HGD03] Manfred Hauswirth, Harald Gall, and Schahram Dustdar. *Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software*. Springer, Berlin, Heidelberg, 1st edition, July 2003. (zitiert auf Seite 1)

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral Interface Specification Languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012. (zitiert auf Seite 15, 16 und 84)

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, October 1969. (zitiert auf Seite 15)

[HS12] Reiner Hähnle and Ina Schaefer. A Liskov Principle for Delta-Oriented Programming. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 7609 of *LNCS*, pages 32–46, Berlin, Heidelberg, 2012. Springer. (zitiert auf Seite 84)

[JHF+12] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A Technique for Agile and Automatic Interaction Testing for Product Lines. In *Proc. Int'l Conf. Testing Software and Systems (ICTSS)*, volume 7641 of *LNCS*, pages 39–54, Berlin, Heidelberg, 2012. Springer. (zitiert auf Seite 68)

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, January 1997. (zitiert auf Seite 3 und 12)

[Kar95] Even-André Karlsson, editor. *Software Reuse: A Holistic Approach*. John Wiley & Sons, New York, NY, USA, 1995. (zitiert auf Seite 2 und 8)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, Berlin, Heidelberg, New York, London, 1997. Springer. (zitiert auf Seite 84)

[Kna01] Dr. Peter Knauber. Planung und Realisierung von Software-Produktlinien. In *Component Developer's & User's Forum 2001*, volume 10, 2001. (zitiert auf Seite 17)

[Kor85] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 1:97–109, 1985. (zitiert auf Seite 20)

[LC06] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, September 2006. (zitiert auf Seite 3, 12, 14 und 83)

[LHKS91] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An Empirical Study of the Object-Oriented Paradigm and Software Reuse. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 184–196, New York, NY, USA, 1991. ACM. (zitiert auf Seite 2)

[LPC+11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, July 2011. (zitiert auf Seite 51)

[McI69] M.D. McIlroy. Mass-Produced Software Components. In *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, pages 138–155, Garmisch, Germany, 1969. (zitiert auf Seite 2)

[Mey85] Bertrand Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, January 1985. (zitiert auf Seite 12)

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. (zitiert auf Seite 34)

[Mey92a] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992. (zitiert auf Seite 3, 12, 14 und 83)

[Mey92b] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Upper Saddle River, NJ, USA, 1992. (zitiert auf Seite 3 und 14)

[Ora] Oracle. Javadoc Tool. Website. Available online at http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html; visited on June 14th, 2013. (zitiert auf Seite 14)

[Par72] David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, December 1972. (zitiert auf Seite 1 und 7)

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer New York, Secaucus, NJ, USA, January 2005. (zitiert auf Seite 2)

[PH00] Arnd Poetzsch-Heffter. *Konzepte Objektorientierter Programmierung: Mit Einer Einführung in Java*. Springer, Berlin, Heidelberg, 2000. (zitiert auf Seite 2)

[PP04] Gustav Pomberger and Wolfgang Pree. *Software-Engineering: Architektur-Design und Prozessorientierung*. Carl Hanser, München, Wien, 3rd edition, 2004. (zitiert auf Seite 1, 2 und 8)

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443, Berlin, Heidelberg, New York, London, 1997. Springer. (zitiert auf Seite 9)

[RL01] Arun D. Raghavan and Gary T. Leavens. Desugaring JML Method Specifications. Technical Report #00-03c, Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040, USA, August 2001. (zitiert auf Seite 16)

[SB86] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4):40–62, January 1986. (zitiert auf Seite 8)

[SB02] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. *Trans. Software Engineering and Methodology (TOSEM)*, 11(2):215–255, April 2002. (zitiert auf Seite 8 und 9)

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 77–91, Berlin, Heidelberg, New York, London, 2010. Springer. (zitiert auf Seite 84)

[SDW+10] Alexander Schatten, Markus Demolsky, Dietmar Winkler, Stefan Biffl, Erik Gostischa-Franta, and Thomas östreicher. *Best-Practice Software-Engineering : Eine praxisorientierte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Spektrum Akademischer Verlag, Heidelberg, 2010. (zitiert auf Seite 7 und 8)

[SGCH01] Kevin Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The Structure and Value of Modularity in Software Design. Technical report, University of Virginia, Charlottesville, VA, USA, 2001. (zitiert auf Seite 7)

[SM12] Detlef Streitferdt and Arfan Mansoor. Requirements on Tailoring Product Line Development Processes. *Softwaretechnik-Trends*, 32, 2012. (zitiert auf Seite 16)

[SPK06] Vijayan Sugumaran, Sooyong Park, and Kyo C Kang. Software product line engineering. *Comm. ACM*, 49(12):28–32, 2006. (zitiert auf Seite 16)

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, April 2012. (zitiert auf Seite 3, 4, 12 und 17)

[TAZ+13] Thomas Thüm, Sven Apel, Andreas Zelend, Reimar Schröter, and Bernhard Möller. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *Proc. Workshop MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 1–8, New York, NY, USA, July 2013. ACM. (zitiert auf Seite 84)

[TBAS13] Thomas Thüm, Fabian Benduhn, Sven Apel, and Gunter Saake. Feature-Oriented Contract Composition, 2013. Unpublished Manuscript. (zitiert auf Seite xv, 19, 20, 22, 25, 26, 30, 31, 66 und 83)

[Thü] Thomas Thüm. SPL2go. Website. Available online at http://spl2go.cs.ovgu.de/projects/; visited on April 5th, 2013. (zitiert auf Seite 4 und 65)

[TKB+13] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 2013. To appear. (zitiert auf Seite 4, 49 und 60)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM. (zitiert auf Seite 1, 7 und 8)

[TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA, September 2012. ACM. (zitiert auf Seite 84 und 85)

[TSK+12] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 255–269, Berlin, Heidelberg, New York, London, March 2012. Springer. (zitiert auf Seite 4, 19, 20, 22, 26, 32 und 83)

[TSKA11] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, DC, USA, March 2011. IEEE. (zitiert auf Seite 84 und 85)

[ZR03] Jianjun Zhao and Martin C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 2621 of *LNCS*, pages 150–165, Berlin, Heidelberg, New York, London, 2003. Springer. (zitiert auf Seite 84)

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

Magdeburg, den 12.08.2013