Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Systematische Analyse von Feature-Interaktionen in Softwareproduktlinien

Verfasser:

Andreas Schulze

15. Oktober 2009

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Christian Kästner,
Dipl.-Wirtsch.-Inf. Thomas Leich

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Schulze, Andreas:
Systematische Analyse von Feature-Interaktionen in Softwareproduktlinien
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2009.

Danksagung

Ich möchte mich für die Hilfe und Unterstützung sowie für das aufgebrachte Verständnis bei allen bedanken, die mich auf meinem Weg begleitet haben.
Besonderer Dank gilt:

- Dipl.-Wirtsch.-Inf. Christian Kästner und Dipl.-Wirtsch.-Inf. Thomas Leich für die Betreuung der Arbeit durch Ideen, Kritik und immer wieder neue Anregungen.
- Maik Siegemund, Thomas Raasch und Matthias Ritter, meinen Kommilitonen, ohne die der Verlauf meines Studiums sicher ein anderer gewesen wäre.
- meiner Familie für die fortwährende vor allem moralische Unterstützung.

Inhaltsverzeichnis

Inhaltsverzeichnis iii

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

Verzeichnis der Abkürzungen xi

1 Einleitung 1

1.1 Überblick 1

1.2 Zielstellung 2

1.3 Gliederung 2

2 Grundlagen 5

2.1 Entwicklung der Softwaretechnik 5

2.1.1 Entstehung der Softwaretechnik 5

2.1.2 Implementierungstechniken der Softwaretechnik 6

2.2 Softwareproduktlinien 9

2.2.1 Übersicht 9

2.2.2 Domänenanalyse 11

2.2.3 Variabilität in SPLs 12

2.2.4 Das Optional-Featureproblem 15

3 Analyse der Optional-Featureprobleme 17

3.1 Motivation 17

3.2 Überblick 18

3.3 Probleme 20

3.3.1 P1: Benutzerschnittstelle (UI) 20
3.3.2 P2: Betriebssystemabstraktion 21
3.3.3 P3: Authentifikation 21
3.3.4 P4: Rechtemanagement 22
3.3.5 P5: Logging 23
3.3.6 P6: Sichere Verbindung 23
3.3.7 P7: Accountmanagement 24
3.3.8 P8: Nachrichtenerweiterungen 25
3.3.9 P9: Statistiken 26
3.4 Lösungsstrategien 26
3.4.1 Designanpassung 27
3.4.2 Quellcodeaustausch 28
3.4.3 Bedingte Kompilierung (Präprocessor) 29
3.4.4 Auslagern der Interaktion (Derivate) 29
3.4.5 Optionales Weben 30
3.4.6 Kapselung 30
3.5 Untersuchung der Lösungen 31
3.5.1 Nachrichtenerweiterungen 31
3.5.2 Accountmanagement 32
3.5.3 Rechtemanagement 33
3.5.4 Benutzerschnittstelle 34
3.5.5 Betriebssystemabstraktion 35
3.5.6 Authentifikation 37
3.5.7 Sichere Verbindung 37
3.5.8 Logging / Statistiken 38

4 Auswertung **43**
4.1 Klassifizierung 43
4.1.1 Kriterien 43
4.1.2 Klassen 48
4.2 Bewertung der Lösungsstrategien 52
4.2.1 Bewertungskriterien 52
4.2.2 Designanpassung 54

4.2.3 Quellcodeaustausch . . . 55
4.2.4 Bedingte Kompilierung . . . 56
4.2.5 Auslagern der Interaktion . . . 57
4.2.6 Optionales Weben . . . 58
4.2.7 Kapselung . . . 58
4.3 Richtlinie . . . 60

5 Zusammenfassung und Ausblick 63
5.1 Zusammenfassung . . . 63
5.2 Ausblick . . . 64

Literaturverzeichnis 67

Abbildungsverzeichnis

2.1 Traditionelle versus frameworkbasierte Entwicklung 8
2.2 Dimensionen einer Produklinie 10
2.3 Beziehungspfade im Merkmalsdiagramm 12
2.4 Variationspunkt: Optionale Feature 13
2.5 Variationspunkt: Alternative (obligatorisch) 13
2.6 Variationspunkt: Alternative (optional) 14
2.7 Variationspunkt: ODER 14
2.8 Variationspunkt: Kombination 15

3.1 Gesamtübersicht des SPL 19
3.2 Übersicht von Problem 1 im FD 20
3.3 Übersicht von Problem 2 im FD 21
3.4 Übersicht von Problem 3 im FD 22
3.5 Übersicht von Problem 4 im FD 23
3.6 Übersicht von Problem 6 im FD 24
3.7 Übersicht von Problem 7 im FD 24
3.8 Übersicht von Problem 8 im FD 25
3.9 Optional-Featureproblem 26
3.10 Designanpassung 27
3.11 Quellcodeaustausch 28
3.12 Auslagern der Interaktion (Derivate) 29
3.13 Lösung mit Interface 30

Tabellenverzeichnis

4.1 Zeitpunkt der Interaktion 44
4.2 Richtung der Interaktion 46
4.3 Beziehung der Feature 47
4.4 Anzahl der Feature 47
4.5 Art der Feature 48
4.6 Gesamtübersicht der Kriterien 49
4.7 Bewertung der Designanpassung 55
4.8 Bewertung des Quellcodeaustausches 56
4.9 Bewertung der bedingten Kompilierung 57
4.10 Bewertung des Auslagerns der Interaktion 58
4.11 Bewertung der Kapselung 59
4.12 Richtlinie (Teil 1) 60
4.13 Richtlinie (Teil 2) 61

Verzeichnis der Abkürzungen

AOP	Aspectorientierte Programmierung
API	Application Programming Interface
FODA	Featureorientierte Domänenanalyse
FOP	Featureorientierte Programmierung
GUI	Graphical User Interface
OFP	Optional-Featureproblem
OOP	Objektorientierte Programmierung
SPL	Softwareproduktlinie
SWD	Stepwise Refinment Development
UI	Userinterface
VP	Variationspunkt

Kapitel 1

Einleitung

1.1 Überblick

Modularität ist ein Grundkonzept der Softwareentwicklung. Identische Umsetzungen werden vermieden durch die Bildung von wiederverwendbaren Modulen. Das Ziel dieser Modularisierungen von Software ist eine Verminderung der Komplexität des Quellcodes, um die Wartbarkeit und die Qualität zu verbessern.

Aktuelle Entwicklungen des Software Engineering stellen eine neue Ebene der Modularisierung heraus. Die Einheiten (Feature) sind unabhängig von Implementierungsmodulen definiert und beschreiben eine Erweiterung der Programmfunktionalität [BSR03]. Aufgrund moderner Programmierkonzepte lassen sich Feature in der Implementierung in modulare Einheiten abbilden [BLS03, TOHJ99].

Die Aufteilung einer Software in Feature ermöglicht eine variable Konfiguration der enthaltenen Funktionalität der Software. Die Entwicklung eines einzelnen Sofwareproduktes wandelt sich somit zu der Entwicklung einer Produktlinie. Die Softwareproduktlinien (SPL) fassen die Entwicklung von Produkten einer Domäne unter einer gemeinsamen Architektur zusammen [Bos00]. Eine SPL kann optionale und obligatorische Feature enthalten. Die Produkte einer SPL bestehen aus einer Auswahl der Feature ihrer SPL. Ausgehend von einer gemeinsamen Codebasis erzeugen die Feature die Gemeinsamkeiten und Unterschiede der Produkte (Varianten) einer Produktlinie [CEC00]. Die Anzahl möglicher Produkte einer SPL ergibt sich aus der Anzahl der möglichen Kombinationen der vorhandenen Feature. Diese Variabilität entsteht durch die domainspezifische Optionalität der Feature.

Es bestehen zwei unterschiedliche Arten von Abhängigkeiten, die zwischen den Featuren einer SPL auftreten können. Jede dieser Abhängigkeiten reduziert die Menge der Varianten einer SPL. Die technischen bzw. domänenspezifischen Abhängigkeiten ergeben sich aus der Domäne selbst und aus speziellen Anforderungen an die Feature. Die andere Art der Abhängigkeit ergibt sich aus der Implementierung der Feature und ist aus diesem Grund unabhängig von der Domäne der jeweiligen SPL. Die domänenspezifischen Abhängigkeiten können bereits in der Konzeptionsphase einer Produktlinienentwicklung erkannt und erfasst werden.

Die Abhängigkeiten, die aus der Implementierung der einzelnen Feature resultieren, werden als Optional-Featureproblem (OFP) bezeichnet. Die Erkennung und die Auflösung der Interaktionen von optionalen Featuren stellen wichtige Forschungsansätze

im Bereich der Softwareproduktlinien dar [LARS05]. Eine geeignete Auflösung der Interaktionen eines Optional-Featureproblems ist notwendig, um die Erstellung zahlreicher Varianten einer SPL zu erstellen.

Aus jeder möglichen Variante einer SPL kann ein Produkt werden, wenn die jeweilige Variante mit seinen Anforderungen die Anforderungen eines Absatzmarktes erfüllt. Die Anzahl der Produkte die aus einer Produktlinie hervorgehen bestimmt die Kosteneffizenz der Entwicklung der Produkte in einer SPL gegenüber der einzelnen Entwicklung jedes Produktes [PBVDL05]. Die Lösung und Erkennung von Optional-Featureproblemen beeinflusst somit direkt den Nutzen von Softwareproduktlinien.

1.2 Zielstellung

Die Variabilität von vielen Softwareproduktlinien wird durch das Auftreten von Optional-Featureproblemen beeinflusst. Es existiert eine Vielzahl von Techniken zur Lösung und Erkennung dieser Probleme. Die Eignung dieser Techniken wurde bisher nur bei der Betrachtung spezieller Problemfälle oder für die Gesamtheit aller Optional-Featureprobleme einer Domäne untersucht.

Die Aufstellung einer Richtlinie, um die Entwicklung von Softwareproduktlinien zu unterstützen, ist das Ziel dieser Arbeit. Die Richtlinie soll den Umgang mit Optional-Featureproblemen für Entwickler und Designer einer Softwareproduktlinie erleichtern.

Die Klassifizierung von Optional-Featureproblemen bildet die Grundlage dieser Richtlinie. Die Klassen ermöglichen eine Beschreibung von Optional-Featureproblemen unabhängig von einer bestimmten SPL, Domäne oder Programmiersprache. Die Zuordnung jedes Optional-Featureproblems soll durch die Informationen der Richtlinie möglich sein. Ausgehend davon können somit die unterschiedlichen Vorkommen von Optional-Featureproblemen während der Entwicklung der SPL unterschieden werden.

Zusätzlich zu der Klassifizierung soll die Richtlinie geeignete Lösungsvorschläge für die Feature-Interaktionen der einzelnen Klassen geben.

1.3 Gliederung

Die in der Arbeit durchgeführte systematische Untersuchung von Feature-Interaktionen spiegelt die benutze Systematik in dem Aufbau der wieder. Am Aufbau der Arbeit lässt sich somit das Vorgehen der Analyse chronologisch nachverfolgen.

Das erste Kapitel enthält neben dieser Gliederung eine einführende Motivation in die Problematik der Aufgabenstellung und eine Zielstellung, welche die Ziele dieser Arbeit zusammenfasst.

Im zweiten Kapitel werden die nötigen Grundlagen zum Verständnis der behandelten Thematik erläutert. Eine Einführung in die Entwicklungen der Softwaretechnik schafft einen Überblick über die verwendeten Programmierkonzepte. Der zweite Teil des Grundlagenkapitels befasst sich mit dem Themengebiet der Softwareproduktlinien und definiert die in der Arbeit untersuchten Interaktionen.

Die Umsetzung und Analyse einer eigenen Softwareproduktlinie bildet den ersten Teil des dritten Kapitels. Ausgehend von den erkannten und untersuchten Problemfällen

werden im zweiten Teil des Kapitels Lösungsansätze vorgestellt und umgesetzt.

Das vierte Kapitel widmet sich der Auswertung der vorangegangenen Analysen. Die untersuchten Problemfälle werden klassifiziert und verglichen. Die Lösungsansätze werden bezüglich ihrer Eignung für die aufgestellten Klassen untersucht. Abschließend werden die Ergebnisse zu einer Richtlinie zusammengefasst.

Im letzten Kapitel erfolgt eine Zusammenfassung der Betrachtungen und Ergebnisse der gesamten Arbeit und mögliche weitere Entwicklungen des Themengebietes anhand der Ergebnisse werden diskutiert.

Kapitel 2

Grundlagen

In diesem Kapitel wird ein Überblick über die Entwicklungen der Softwaretechnik gegeben, um eine Hinführung zu der behandelten Thematik zu schaffen.

Des weiteren werden wichtige Grundlagen zum Verständnis der Arbeit geschaffen. Die Einführung in die Thematik der Softwareproduktlinien schafft nötige Grundlagen zum allgemeinen Verständnis der betrachteten Problematik dieser Arbeit.

2.1 Entwicklung der Softwaretechnik

In der Pionierzeit der Softwareentwicklung existierten feste Grenzen für die Komplexität und Leistungsfähigkeit von Software. Die Möglichkeiten der verfügbaren Hardware bestimmten diese Grenzen eindeutig. Die Weiterentwicklung von Software umfasste somit hauptsächlich die Anpassung auf neuere Hardware und die Ausnutzung der erweiterten Möglichkeiten dieser Hardware. Techniken und Methoden, die sich in der Softwareindustrie durchgesetzt haben, werden im Folgenden erläutert. Dabei ist ein spezieller Gesichtspunkt die erreichbare Modularität der einzelnen Techniken.

2.1.1 Entstehung der Softwaretechnik

Die sprunghafte Weiterentwicklung der Hardware zeigte der Softwareentwicklung Grenzen auf. Um das Leistungspotential der Hardware vollständig zu nutzen, entstanden viele Softwareprojekte deren Komplexität nicht mehr zu überschauen waren. Durch diese starke Zunahme der Komplexität der Softwareprojekte, entstanden die ersten Ansätze des Softwareengeneering, da die bisher genutzten Techniken zur Planung und Umsetzung für den ansteigenden Umfang und die zunehmende Komplexität nicht mehr geeignet waren. Im Jahre 1968 wurde auf einer NATO-Konferenz [NR69] erstmals der Begriff des Software Engineering geprägt.

Diese Entwicklung fand zu einer Zeit statt, in der Software selbst noch nicht als eigenständiges Produkt existierte. Software wurde nur mit der Hardware vertrieben für die sie entwickelt wurden war. Eine sogenannte Softwareindustrie existierte somit noch nicht. Software wurde entwickelt, um für die jeweilige Hardware einen Absatzmarkt zu schaffen [PD93].

Der 1968 provokativ gewählte Titel „Software Engineering“ stellte den Anspruch die Softwareentwicklung als einen ingenieurtechnichen Prozess anzusehen. Somit sind die Ansätze und Methoden zur Verbesserung die aus diesem Zweig der Informatik vorgegangen sind, fast immer eine Adaption aus anderen ingeneurtechnichen Bereichen. Obwohl bereits 1968 der Begriff geprägt wurde, fand erst 1975 die „Conference of Software Engineering“ als internationale Konferenz statt.

Die Gründe für diese Entwicklung waren, dass viele Softwareprojekte wiederholt drei große Probleme aufwiesen [Fai86]. Die Herstellung der Software war teurer als geplant, da die tatsächliche Entwicklungszeit oft deutlich die geplante Zeit überstieg [GJM91]. Somit erhöhten sich die Kosten für die jeweilige Software. Aufgrund des Umfangs der Projekte konnte zu dem die Korrektheit der Software nicht sichergestellt werden [Wir72]. Dafür gab es zwei maßgebliche Ursachen, zum Einen waren strukturierte Testmechanismen noch nicht bekannt und zum Anderen erfüllten die Projekte nur hinreichend die tatsächlich geforderten Anforderungen. Das dritte Hauptproblem der Softwareentwicklung ist die Effizienz des Produktes [Som04, PP04].

2.1.2 Implementierungstechniken der Softwaretechnik

Die Prinzipien und Fortschritte des Software Engineering erstrecken sich über den gesamten Softwarelebenszyklus. Im Folgenden werden die Entwicklungen auf dem Bereich der Umsetzung, der Implementierung, betrachtet. Diese Entwicklungen geben eine gute Einleitung und Zuordnung in das Themengebiet der Arbeit. Spezieller Gesichtspunkt der Betrachtungen ist das Abstraktionslevel der vorgestellten Techniken. Die Möglichkeiten der Modularisierung innerhalb einer Software legen den Grundstein für die Ausarbeitungen der Arbeit.

Strukturierte Programmierung

Der bekannteste Ansatz, um ein großes Problem zu lösen, ist der Versuch es in kleinere Teilprobleme zu zerlegen. Auf dem Gebiet der Implementierungstechniken ist die strukturierte Programmierung ein erstes Paradigma welches diesem Ansatz folgt. Bei dem Übergang von Assemblerprogrammen zu höheren Programmiersprachen änderten sich vorerst nicht die Methoden zur Umsetzung von Programmen. Die Folge waren stark monolithische Programme. Die Idee eines strukturierten, hierarchischen Programmablaufes trennt diese bloßen Folgen von Anweisungen auf [DDH72]. Es entstehen somit mehrere kleine logische Pogrammeinheiten [Dij79]. Zusätzlich ermöglicht die Einführung von Methoden und Funktionen eine gezielte Abtrennung von Geltungsbereichen. Eine in einem monolithisch gehaltenem Programm benutzte Variable hat hingegen einen globalen Geltungbereich.

Zur Umsetzung eines Programms in strukturierter Form ist vorher eine Designphase notwendig. Das verwendete Verfahren zur Erstellung des Designs ist die Top-Down-Methode. Dabei wird die hierarchische Abfolge des Programms in Schritten entwickelt, beginnend mit einem Gesamtüberblick der Funktionalität bis hin zu kleinen Funktionseinheiten. Das Ziel dieser Methode ist den Verständnisgrad des Quellcodes zu erhöhen. Die mögliche Wiederverwendung einzelner Komponenten wird hingegen nicht berücksichtigt.

Objektorientierte Programmierung (OOP)

Der objektorientierte Ansatz nutzt die Möglichkeiten der strukturierten Programmierung und erweitert sie um Objekte und Klassen. Zur Laufzeit des Programms bildet jedes Objekt eine Sammlung von Methoden und Daten ab. Eine Klasse ist die Abstraktion einer Objektmenge und definiert die Methoden und Datentypen ihrer Objekte [Zam98, Bru02, EFB01]. Im Gegensatz zur strukturierten Programmierung erfolgt somit keine Trennung zwischen Daten und den zugehörigen Funktionen. Die Interaktion von Objekten bildet den Programmablauf eines objektorientierten Programms ab. Die Komplexität der einzelnen Klassen respektive ihrer Objekte, während dieser Interaktion, wird durch Kapselung der Klassen versteckt [Sny86, Zam98]. In der Klasse ist genau festgelegt, welche Funktionen von außerhalb der Klasse erreichbar sind und welche nicht. Die Auszeichnung der öffentlichen Funktionen einer Klasse werden in ihrem Interface definiert.

Die Kapselung erhöht dadurch die Möglichkeiten der Wiederverwendung der Klassen. Bei der Verwendung einer Klasse interessiert nicht ihr interner Aufbau, sondern nur die nach Außen sichtbaren Funktionen. Zusätzlich ermöglicht die Kapselung den einfachen Austausch einer verwendeten Klasse durch eine Andere, wenn die Interfaces der beiden Klassen identisch sind. Die interne Implementierung der Klassen hat auf die Austauschbarkeit keinen Einfluss.

Die Modularisierungselemente der OOP bilden somit eine höhere Abstraktionsebene ab. Für die Wiederverwendbarkeit stehen die zusätzlichen Mechanismen Vererbungen, Aggregationen, Templates und Polymorphismen zur Verfügung [Weg86, Mey88].

Komponentenbasierte Entwicklung

Eine in sich abgeschlossene Problemlösung wird in der Softwaretechnik als Komponente bezeichnet [KKST79]. Das Ziel der komponentenbasierten Entwicklung ist somit die Erstellung gekapselter Einheiten die in beliebigen Zusammenhängen genutzt werden können. Die Komponenten umfassen dabei mehrere kollaborierende Klassen der OOP und stellen eine rein funktionale Schnittstelle zur Verfügung [JBR99]. Die komponentenbasierten Entwicklung gliedert sich in zwei Teilgebiete aufgrund der unterschiedlichen Verwendung der Komponenten im jeweiligen Projekt.

Komponenten stellen eine für Entwickler nutzbare modulare Lösung dar. Die Auswahl von Komponenten ist ein Teil des Entwicklungsprozesses. Die Funktionalität der Komponenten benötigt daher immer eine Anpassung (glue code) des Codes der Software. Es existieren zwei Möglichkeiten die Funktionalität von Komponenten zu nutzen, indem Objekte der Komponente instantiiert werden und durch die Vererbung von Klassen der Komponente auf Klassen der Anwendung. Die Wiederverwendung von Klassen steht im Vordergrund bei Komponenten, was einen zusätzlichen wirtschaftlichen Faktor einbringt: Der Entwickler kann für jede Komponente den besten/billigsten Anbieter auswählen [NT95].

Das zweite Teilgebiet der komponentenbasierten Entwicklung bilden Frameworks (Entwicklungsgerüste). Ein Framework definiert das Design der entstehenden Software und enthält grundlegende Funktionalität [BMMB00]. Erweiterbar ist ein Framework durch definierte Platzhalter und Schnittstellen [MN96]. Die sogenannten Platzhalter bie-

ten Stellen zur Erweiterung des Frameworks um interne Funktionalität. Die Schnittstellen hingegen fungieren als Ansatzpunkte zur Datenmanipulation. Das bedeutet Daten werden durch eine Outputschnittstelle dem Programmfluss entnommen, verarbeitet und mit Hilfe einer Inputschnittstelle wieder in den Programmfluss integriert. Auf dieser Basis erstellte Erweiterungen werden als Plugins bezeichnet. Die Verwendung von Frameworks erstreckt sich über mehr als eine einzelne Anwendung, sie umfasst eine Menge von Anwendungen innerhalb einer Domäne (vgl. Abbildung 2.1). Frameworks ermöglichen somit eine Wiederverwendung von Quellcode und zusätzlich von Konzepten auf der Ebene des Designs [JF88, GHJV93].

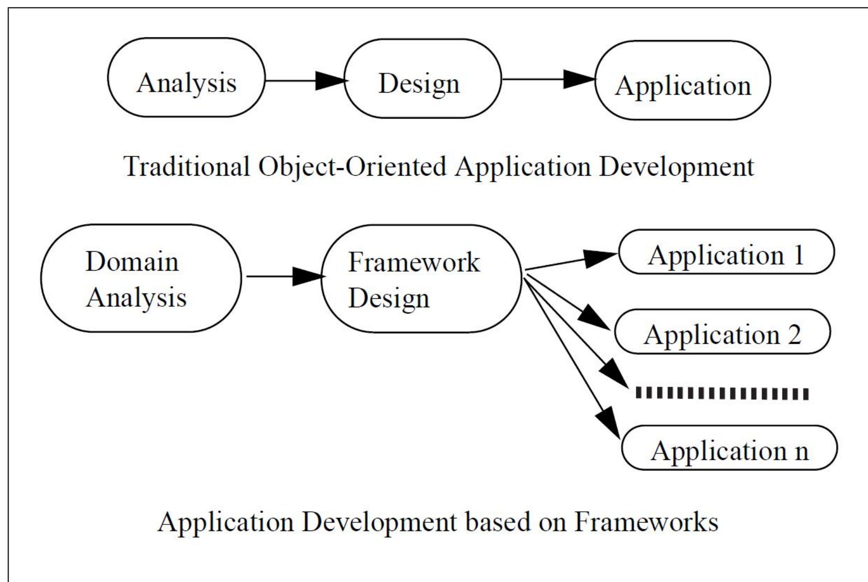

Abbildung 2.1: Traditionelle versus frameworkbasierte Entwicklung [Mic96]

Frameworks bilden einen Rumpf zur domänenspezifischen Entwicklung und bilden jeweils das Grundgerüst, während Komponenten die Software modular erweitern [Szy02]. Somit ist die Festlegung auf ein Framework eine bindende Entscheidung und Komponenten hingegen bleiben als modulare Einheiten erhalten.

Aspekt- und Featureorientierte Programmierung

Das Ziel des objektorientierten Designs ist die Trennung von Belangen [Par72, Dij76]. Viele Projekte haben jedoch gezeigt das sich nicht alle Arten von Belangen mit den Mitteln objektorientierte Programmierung modellieren lassen [EFB01]. Bei der Umsetzung querschneidender Belange mit den Mechanismen der OOP ergeben sich Codestreuungen (Code Scattering), d.h. über mehrere Module verteilter Code eines Belanges, und Codemischungen (Code Tangling), d.h. mehrere Belange innerhalb eines Modules [KHH+01, FECA05]. Mit reiner OOP ist somit keine modulare Umsetzung von querschneidenen Belangen möglich. Im Rahmen dieser Arbeit werden zwei Techniken betrachtet die sich querschneidenen Belangen widmen.

Bei der featureorientierte Programmierung (FOP) stellen Feature die zentrale modulare Einheit dar [KHH+01]. Die Grundidee eines Features entspricht der Umsetzung eines Belanges. Innerhalb eines Fetures können neue Klassen erstellt werden und bestehende Klassen erweitert werden. Somit können querschneidene Belange ebenfalls in einer

modularen Einheit umgesetzt werden. Ein weiterer Vorteil des FOP Ansatzes entsteht durch eine strikte Trennung von Konfiguration und Umsetzung. Feature können somit variabel zur Erzeugung von Produktvariationen eingesetzt werden.

Die aspektorientierte Programmierung (AOP) widmet sich ausschließlich der Implementierung von querschneidenen Belangen [KLM+97, GH05, BSR03]. Die Modularisierung wird durch das Auslagern der verteilten Codesegmente in zusammenhängende Einheiten realisiert. Die Zuordnung dieser sogenannten Aspekte erfolgt mit Hilfe der Definition von Einfügepunkten(Joinpoints). Das Erstellen der Anwendung wird durch einen zusätzlichen Schritt im Compilevorgang realisiert. Diese Codetransformationen brechen das OOP Prinzip der Kapselung und können so zu fehlerhaften Erweiterungen führen [CL03, MS06].

Präprozessoren

Präprozessoren sind ursprünglich eine Besonderheit weniger Programmiersprachen und stellen keine direkte Implementierungstechnik dar. Mit Hilfe von Präprozessoranweisungen kann der Quellcode dynamisch vor dem Compileraufruf transformiert werden. Quellcodeabschnitte die mit einem *#ifdef*-Statement annotiert werden, können in Abhängigkeit von der Bedingung ihrer Statements aus dem Quellcode der Anwendung entfernt werden.

Die Anweisungen können bis auf die Ebene einzelner Zeichen den Quellcode modifizieren. Die Vor- und Nachteile von bedingter Kompilierung werden ausführlich in [SC92] diskutiert.

2.2 Softwareproduktlinien

Die im vorigen Abschnitt 2.1 vorgestellten Implementierungstechniken bilden die zunehmende Abstraktion im Softwareentwicklungsprozess ab. Aufgrund dieses erhöhten Abstraktionslevels ergibt sich eine erweiterte Form der Softwareerstellung. Angelegt an die Bezeichnungen der Industrie werden nicht mehr einzelne Anwendungen entwickelt, sondern komplexe domänenabhängige Softwareproduktlinien erstellt. Das folgenden Kapitel gibt eine Übersicht über die Ideen Funktionsweisen, Vorteile und Probleme dieses Enwicklungsansatzes.

2.2.1 Übersicht

Eine Software definiert sich durch die Anforderungen die an sie gestellt werden und wie sie diese umsetzt. Jede Software kann somit als eine Kombination bestimmter Belange angesehen werden. Die im Abschnitt 2.1 vorgestellten Techniken ermöglichen eine Modularisierung von jedem dieser Belange. Somit können die einzelnen Komponenten als Bausteine innerhalb der Software und für andere Anwendungen modular verwendet werden. Der SPL-Ansatz vereint die Entwicklung mehrerer Anwendungen in einem gemeinsamen Softwareentwicklungsprozess. Die Idee der SPL-Entwicklung geht dabei auf das Prinzip des Stepwise Refinment Development (SWD) [Dij76] zurück.

Produktlinien decken naturgemäß den gesamten Softwarelebenszyklus ab, sie integrieren andere Themenbereiche wie Requirements Engineering, Softwarearchitekturen und Reengineering.

Die Entwicklung einer Produktlinie hat auf verschiedene Bereiche des Prozesses Einfluss. Bosch gliedert diese Bereiche in drei Dimensionen einer SPL (vgl. Abbildung 2.2) [Bos00].

In der ersten Dimension befindet sich die Software oder Plattform der Produkte selbst. Architektur und Komponenten müssen alle geplanten und möglichen Anforderungen der SPL abdecken. Das Management(2.Dimension) bei der Entwicklung einer SPL muss ständig den gesamten Umfang der Produkte überblicken um die Wirtschaftlichkeit und eine funktionierende Organisationsstruktur der SPL zu gewährleisten. Die Konzeptbildung und der Entwurf der SPL bilden die dritte Dimension. Der Bereich den die Produkte der SPL aufspannen wird im Allgemeinen als Domäne bezeichnet.

Die Analyse und Festlegung (Scoping) dieser Domäne ist der Mittelpunkt des Konzeptes von SPL. Vergleichbar mit Bosch definieren Clement und Northrop [CN02] die Bereiche Plattformentwicklung, Produktentwicklung und Management.

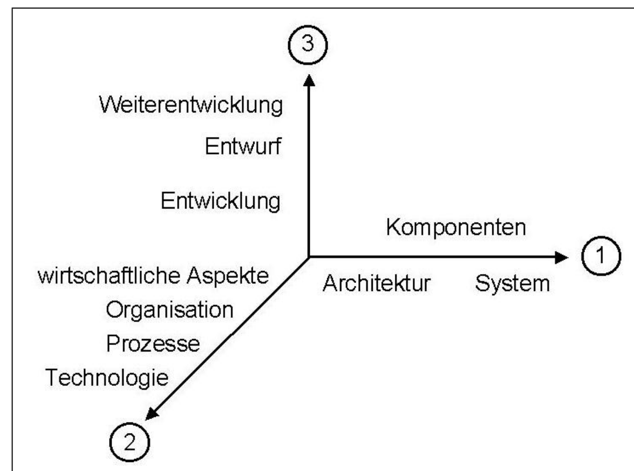

Abbildung 2.2: Dimensionen einer Produklinie [Bos00]

Ein großer Unterschied zum allgemeinen Software Engineering tritt in der Phase der Anforderungsanalyse auf. In dieser initialen Phase des Entwicklungsprozesses erfolgt eine Abgrenzung eines Bereiches für alle weiteren Entwicklungsschritte. Die so genannte Domäne der SPL legt fest welche Anforderungen und dazu gehörenden Produkte Teil der Produktlinie sind. Die Entscheidung welche Softwareprodukte sich zu einer SPL zusammenfassen lassen, hängt hauptsächlich von wirtschaftlichen Gesichtspunkten ab. Die Definition der Domäne erfolgt durch die Einteilung der möglichen Produkte oder deren Funktionalität in drei Gruppen. Produkte können in der Domäne enthalten sein, eventuell enthalten sein oder stellen keinen relevanten Teil der SPL dar. Das entscheidende Merkmal von SPL was diese Einteilung aufweist ist, das die Domäne für eine Evolution ausgelegt ist. So können im Entwicklungsprozess der SPL neue Produkte integriert werden, durch eine Anpassung der Domäne.

Die Architektur von SPL unterteilt sich in zwei grundlegende Einheiten. Bei der Domänenanalyse wird die statische Funktionalität einem Kern und die dynamischen Teile verschiedenen Einheiten zugeordnet. Statische Funktionalität umfasst dabei alle

Eigenschaften die in jedem Produkt der SPL übereinstimmen. Die Umsetzung dieser Basiseinheiten bildet die Grundlage der Implementierung einer SPL. Den Übergang zu einer variablen Produktpalette innerhalb einer SPL realisieren dynamisch auswählbare Eigenschaften. Die Methoden der Domänenanalyse werden im folgenden Abschnitt erläutert.

2.2.2 Domänenanalyse

Die Wiederverwendung von Komponenten verspricht viele Verbesserungen für den Entwicklungsprozess, wenn sie systematisch eingesetzt wird. Im Rahmen einer SPL ist dazu die Erkennung gemeinsamer Eigenschaften und Fähigkeiten der Produkte innerhalb der Domäne der SPL notwendig. Eine Domänenanalyse ist die systematische Untersuchung von Softwaresystemen die ihre Gemeinsamkeiten und Unterschiede definiert und ausnutzt. Daher sind Domänenanalysemethoden ein wichtiger Faktor, um den Softwareentwicklungsprozess zu verbessern. Der Analyseprozess unterteilt sich in drei Phasen [GIL89, MPC+86, PD90].

1. Kontextanalyse (Scoping: Quellen für weitere Phasen)
   Ergebnis: Strukturdiagramm, Kontextdiagramm

2. Domänenmodellierung (Erstellen eines Modells das die Anforderungen der Domäne abbildet)
   Ergebnis: Entity Relationship Modell, Merkmal Modell, Funktionales Modell, Domänen Terminologie

3. Architekturmodellierung (Umsetzung der Anforderungen)
   Ergebnis: Prozess Interaktion Modell, Klassenstrukturdiagramm

Das Ergebnis jeder dieser Phasen dient als Ausgangspunkt für die anderen Phasen und für weitere Schritte im Entwicklungsprozess. Somit stellt die Domänenanalyse einen evolutionären Prozess dar, und erlaubt eine fortlaufende Anpassung der Domäne selbst.

Featureorientierte Domänenanalyse (FODA)

Für die Abbildung von Funktionen und Eigenschaften hat sich im Bereich der Domänenanalyse der Begriff des Features (deut.: Merkmal) durchgesetzt. Die genaue Definition eines Features hängt von der benutzten Domänenanalysemethode ab. Unabhängig von der genauen Bedeutung eines Features stellt es eine Erweiterung der Funktionalität aus Sicht der Stakeholder dar. In der featureorientierten Domänenanalyse bildet ein Merkmal eine markante für den Benutzer sichtbare Eigenschaft eines Systems [KCH+90]. Merkmale sind somit ein Mittel, um die Gemeinsamkeiten und Unterschiede von Softwaresystemen zu kommunizieren [KCH+90, CEC00].

Das Zentrale Ergebnis der FODA ist ein Merkmalsdiagramm (engl. Feature Diagram (FD)), welches die entsprechende Domäne durch die enthaltenen Feature und deren Beziehung zueinander beschreibt [KCH+90, CEC00, Bat05]. Die grundlegende Struktur eines FD entspricht einem n-ären Baum. Dabei stellt die Wurzel den Kern der SPL dar

und die Tiefe eines Merkmals gibt den Zeitpunkt seiner Auswahl an. Die Beziehungen zwischen den einzelnen Featuren werden in Abbildung 2.3 dargestellt. Feature können als verbindliche oder optionale Knoten auftreten. Wenn ein Feature mehrere Kindknoten besitzt, können drei verschiedene Beziehungstypen zwischen den Kindknoten auftreten. Bei der Und-Beziehung ist die Auswahl aller Kindknoten erforderlich. Die Alternative-Beziehung erlaubt ausschließlich die Auswahl von einem der Kindknoten. Die Auswahl mindestens eines Kindknotens zeichnet die Oder-Beziehung aus. Für zusätzliche Bedingungen die unabhängig von der Baumstruktur liegen, können zusätzliche Bedingungen angegeben werden. Diese Bedingungen sind unabhängig von der Baumstruktur. Aus der unterschiedlichen Zusammenstellung der Feature ergeben sich im weiteren Prozess die Produkte der SPL [KCH+90]. Jeder Blattknoten im FD stellt eine modular zu implementierende Komponente dar.

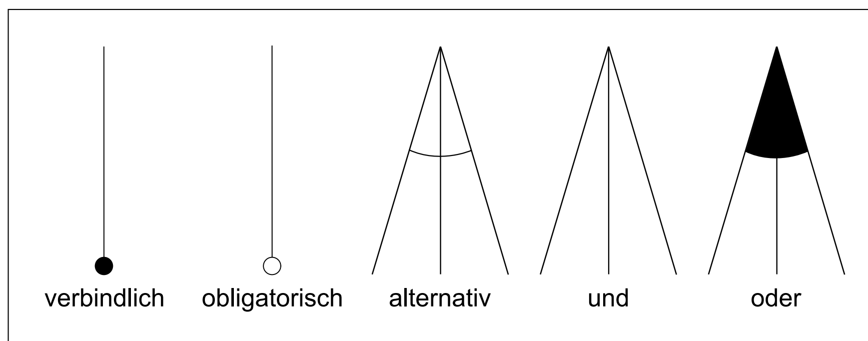

Abbildung 2.3: Beziehungspfade im Merkmalsdiagramm [Bat05]

2.2.3 Variabilität in SPLs

Der gesamte Entwicklungsprozess einer SPL, bietet Möglichkeiten Variabilität zu erzeugen. Die Variabilität einer SPL findet in ihren optionalen Featuren Ausdruck. Die Auswahl optionaler Feature stellt die Unterschiede zwischen den Produkten einer SPL dar. Knoten innerhalb eines FDs, welche optionale Kindknoten oder spezielle Beziehungen enthalten, stellen die Variationspunkte einer SPL dar [CEC00]. Unter Variabilität wird im Allgemeinen die Möglichkeit verstanden, Systeme oder Komponenten zu ändern und an individuelle Bedürfnisse anzupassen.

Im Design einer SPL können unterschiedliche Variationspunkte auftreten. Die Art dieser Verzweigungen ergibt sich aus den Elementen aus denen sie bestehen. Ein Variationspunkt dessen Kindknoten alle vom gleichen Knotentyp sind, ist ein homogener Variationspunkt. Die Kombination von optionalen und obligatorischen Featuren als Kindknoten eines Variationspunktes zeichnet diesen hingegen als inhomogener VP bezeichnet.

Eine weitere Unterscheidung der möglichen Variationspunkte ergibt sich aus der Anzahl der gleichzeitig auswählbaren Feature des VPs. Kann lediglich jeweils ein Feature ausgewählt werden, handelt es sich um einen singularen VP. Variationspunkte deren Feature gleichzeitig ausgewählt werden unterscheiden sich von dieser Gruppe.

Die folgenden Abbildungen zeigen die Variationspunkte, die in einem FD auftreten können. Das Feature bezeichnet mit C entspricht dabei einem Feature oder Konzeptknoten einer SPL. In unterschiedlichen Beziehungen besitzt dieser Knoten jeweils drei ihm

untergeordnete Subfeature (f1,f2,f3). Die Kenntnis dieser Punkte bildet die Grundlage für eine Analyse von Interaktionen zwischen optionalen Featuren. Ähnliche Variationspunkte werden dabei vergleichend betrachtet.

Optionale Feature Den einfachsten Weg, um in einer SPL Variabilität zu erzeugen, stellen optionale Feature dar. Ein beliebiges Feature aufgrund seiner Anforderungen als optionalen Knoten zu deklarieren und zu implementieren, stellt die Grundidee von SPLs dar. Für jedes optionale Feature ist die Auswahl nur von der Auswahl der im FD übergeordneten Feature abhängig. Enthält die Auswahl Feature C so können die Subfeature fn unabhängig voneinander ausgewählt werden.

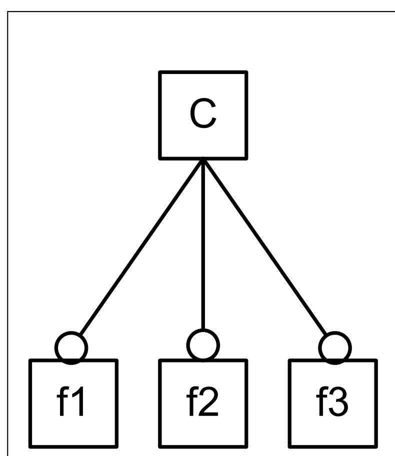

Abbildung 2.4: Variationspunkt: Optionale Feature [CEC00]

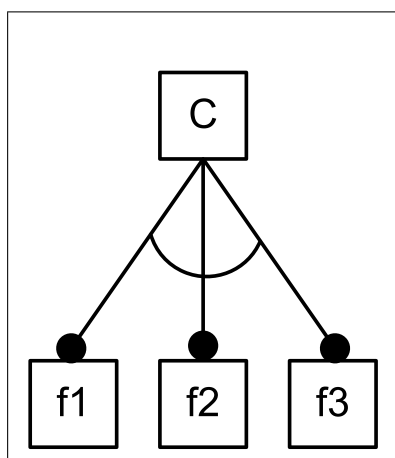

Abbildung 2.5: Variationspunkt: Alternative (obligatorisch) [CEC00]

Alternativen Die Abbildungen 2.5 und 2.6 zeigen die beiden unterschiedlichen Variationspunkte, die durch eine Alternative-Beziehung entstehen können. Beide Variationspunkte sind homogen und singular. Die homogenen Kindknoten in Abbildung 2.5 repräsentieren obligatorische Feature. Aufgrund der Alternativ-Beziehung des dargestellten Variationspunktes ist die Auswahl genau eines der Feature erforderlich. Die Anzahl

der Varianten der SPL erhöht sich somit durch diesen Typ eines VPs um die Anzahl seiner Feature. Innerhalb dieses VPs können keine Interaktionen zwischen den Subfeaturen auftreten.

Die Subfeature in Abbildung 2.6 sind optionale Knoten. Eine Alternativ-Beziehung mit optionalen Featuren erfordert die Auswahl genau eines Features oder von keinem der vorhandenen Subfeature. Ein optionaler Kindknoten einer Alternativ-Beziehung zieht eine Transformation aller Kindknoten der Beziehung in optionale Knoten nach sich. Eine inhomogene Alternativ-Beziehung entspricht somit der homogenen optionalen Alternativ-Beziehung aus Abbildung 2.6.

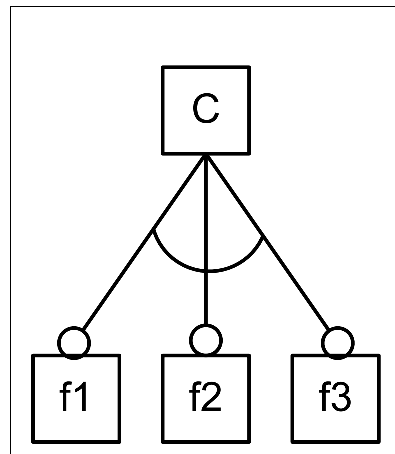

Abbildung 2.6: Variationspunkt: Alternative (optional) [CEC00]

Oder Einen Variationspunkt einer Oder-Beziehung mit homogenen obligatorischen Subfeaturen zeigt Abbildung 2.7. Definitionsmäßig erfordert dieser Beziehungstyp die Auswahl mindestens eines seiner Subfeature. Im Gegensatz zu dem Variationspunkt aus Abbildung 2.5 ermöglicht die Oder-Beziehung die Auswahl mehrerer Subfeature in einer Variante.

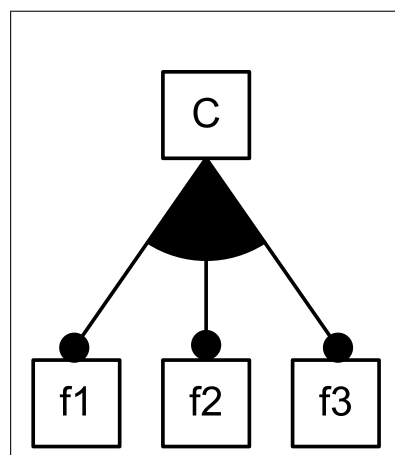

Abbildung 2.7: Variationspunkt: ODER [CEC00]

Kombination Die Kombination der vorgestellten Variationspunkte ist eine weitere Möglichkeit Variabilität einer SPL zu modellieren. Abbildung 2.8 zeigt die Kombination eines einzelnen optionalen Features mit zwei weiteren Featuren die in einer Oder-Beziehung zueinander stehen. Für die Auswahl der einzelnen Feature treten nur die Bedingungen ihrer jeweiligen Beziehungen auf. Zusätzliche Bedingungen zur Auswahl von Featuren an kombinierten Variationspunkten exitieren daher nicht.

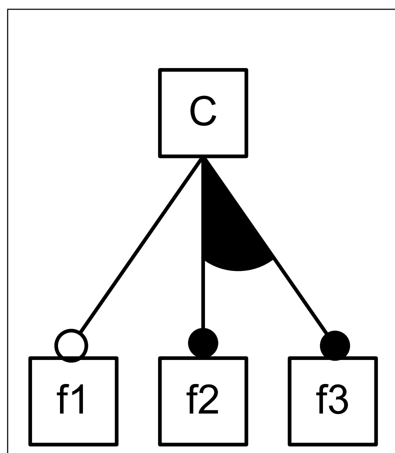

Abbildung 2.8: Variationspunkt: Kombination [CEC00]

2.2.4 Das Optional-Featureproblem

Variabilität in Kombination mit systematischer Wiederverwendung ist das Potential von Softwareentwicklungen in der Form von Softwareproduktlinien. Die Modellierung der Variabilität in der Designphase der Domäne betrachtet keine Interaktionen, die sich aus den Implementierungen einzelner Feature ergeben können. Die Interaktionen zwischen Featuren werden in Implementierungs- und Domänabhängigkeiten unterschieden. Domänspezifische Abhängigkeiten ergeben sich aus der Definition der Domäne einer SPL und der Anforderungen an die in ihr enthaltenen Feature. Die aus der Implementierung resultierenden Abhängigkeiten entstehen bei der Interaktion von Featuren die Abhängigkeiten zwischen den Quellcodes der Feature erzeugen.

Implementierungsabhängigkeiten für die im Design der SPL keine Abhängigkeiten existieren, führen bei optionalen Featuren zu einem Optional-Featureproblem [Pre97, LARS05].

Die Folge dieser Problematik ist die Verminderung der Variabilität der SPL. Betroffene Produkte können aufgrund der Abhängigkeiten nicht erstellt werden oder die Erstellung erzeugt technisch fehlerhafte Produkte.

Kapitel 3

Analyse der Optional-Featureprobleme

In diesem Kapitel werden anhand einer Fallstudie die unterschiedlichen Vorkommen von *Optional-Featureproblemen* erläutert und miteinander verglichen. Die betrachteten Problemfälle werden hinsichtlich möglicher Lösungsstrategien untersucht und bewertet. Ziel dieser Charakterisierung ist die Klassifizierung der gefundenen Probleme.

Da es keine allgemeingültige Lösung für die Vielzahl der Ausprägungen des Optional-Featureproblem gibt, werden in diesem Kapitel mehrere bekannte Lösungsansätze vorgestellt. Die Ergebnisse und Probleme der Umsetzungen der Lösungen für die Problemfälle werden erläutert und ausgewertet. Aus den Ergebnissen dieser Betrachtungen ergibt sich die Grundlage für die Erarbeitung einer Richtlinie zum Lösen von Optional-Featureproblemen.

3.1 Motivation

Das Ziel dieser Arbeit *Feature-Interaktionen* zu identifizieren und zu klassifizieren, wurde innerhalb einer eigenen SPL umgesetzt. Die Domäne dieser SPL bildet ein einfach gestalteter Chat-Client. Dieses Umfeld bietet zahlreiche optionale Anforderungen und besteht gleichzeitig aus leicht nachvollziehbaren Abläufen. Zahlreiche Anwendungen existieren bereits in dieser Domäne und zeigen vielfältige Möglichkeiten durch ihren Variantenreichtum auf. Diese Produkte unterstützen die Domänenanalyse und zeigen grundlegende Anforderungen wie z.B. verschlüsselte Übertragungen auf.

Trotz der guten Grundeigenschaften dieser Domäne wurden einige spezielle Problemfälle speziell konstruiert, d.h. das einige Designentscheidungen absichtlich mit höheren Anforderungen erneut getroffen wurden.

So entstand beispielsweise aus der Anforderung der Betriebssystemabstraktion die weitere Anforderung mehrerer Benutzerschnittstellen. In der ersten Ausarbeitung der Anforderungen bestand ein direkter Zusammenhang zwischen dem verwendeten Betriebssystem und der Benutzerschnittstelle. In diesem Fall hätte die Anwendung mit der Anforderung unter Windows zu arbeiten, Windows Forms als Grundlage für die Benutzerschnittstelle verwendet und die für Linux umgesetzte Version eine entsprechende API (application programming interface).

Ziel der Umsetzung war nicht die Konstruktion vieler untypischer *Feature-Interaktionen*, sondern eine einfache und repräsentative SPL zu schaffen. Es wurden keine Designentscheidungen unabhängig konstruiert, ferner folgte jede Abänderung des Designs explizit einer vorhergehenden Änderung der Anforderungen. Somit wurde der vorgeschriebene Zyklus der FODA strickt eingehalten.

3.2 Überblick

Aus der Domänenanalyse, welche hauptsächlich aus der Analyse von bestehenden Lösungen bestand, ergaben sich sieben Anforderungen für die Chat-SPL. Diese Anforderungen beziehen sich auf den allgemeinen Aufbau der Chatfunktionalität. Ein in allen untersuchten bestehenden Produkten enthaltenes Feature, ist die Möglichkeit verschlüsselte Verbindungen zur Datenübertragung zu nutzen. Dieses Feature schützt zum einen die Daten während der Übertragung und stellt zusätzlich die Identität des Chatpartners sicher. Die weiteren Feature beziehen sich auf spezielle Funktionen, welche das Chatten unterstützen und lassen sich in zwei Hauptkategorien unterteilen.

Narichtenverlauf, Nachrichtenformatierungen, Nachrichtenverschlüsselung und die Unterstützung von Nachrichtenprotokollen bilden die Gruppe der Anforderungen, die direkt den Nachrichtenaustausch erweitern. Der Nachrichtenverlauf zeichnet sämtlichen Nachrichtenverkehr auf und stellt ihn aufbereitet zur Ansicht bereit. Smileys, Schriftfarben und -größen werden durch die Anforderung der Nachrichtenformatierung umgesetzt. Zusätzliche Sicherheit für den Nachrichtenverkehr erfordert die Nachrichtenverschlüsselung durch eine Verschlüsselung der übertragenen Daten. Da bereits eine Vielzahl von Komponenten der bewährten Protokollen zum Nachrichtenaustausch existieren, soll die Benutzung dieser Module unterstützt werden.

Die Anforderungen der zweiten Gruppe beschreiben Funktionalität im Umfeld einer Authentifizierung. Ein Rechtemanagement steuert und verwaltet die Rechte anderer Nutzer mit dem Nutzer in Kontakt zu treten, seine Anwesenheit zu erkennen und andere netzwerkabhängige Rechte. Des weiteren ermöglicht eine Authentifizierung personalisierte Einstellungen zu verwalten.

Weitere Anforderungen wurden aus anderen SPL Entwicklungen anderer Domänen übernommen, um bereits bekannte Probleme ebenfalls zu untersuchen [KAR+09, LARS05]. Die Durchführung eines Loggings und das Aufzeichnen von Statistiken erfüllen in einer Chat Anwendung einen praktischen nutzen und sind weit verbreitete Beispiele für querschneidene Belange. Die Unterstützung unterschiedlicher Betriebssystemarchitekturen entstammt ebenfalls der Analyse anderer SPL Projekte. Um die Anzahl der Interaktionen zusätzlich zu erhöhen entstand die Anforderung mehrere Benutzerschnittstellen zu unterstützen. Diese Schnittstellen bilden das Bindeglied zwischen der Anwendung und ihrem Benutzer.

Die endgültige in dieser Arbeit verwendete Version der Chat-SPL enthält 26 Feature. Das umgesetzte Design der Chat-SPL mit allen erläuterten Featuren zeigt Abbildung 3.1. Die Umsetzung der SPL erfolgte in der Programmiersprache C++. Die einzelnen Varianten werden in der Basisimplementierung mit Hilfe von C-Präprozessoranweisungen erzeugt.

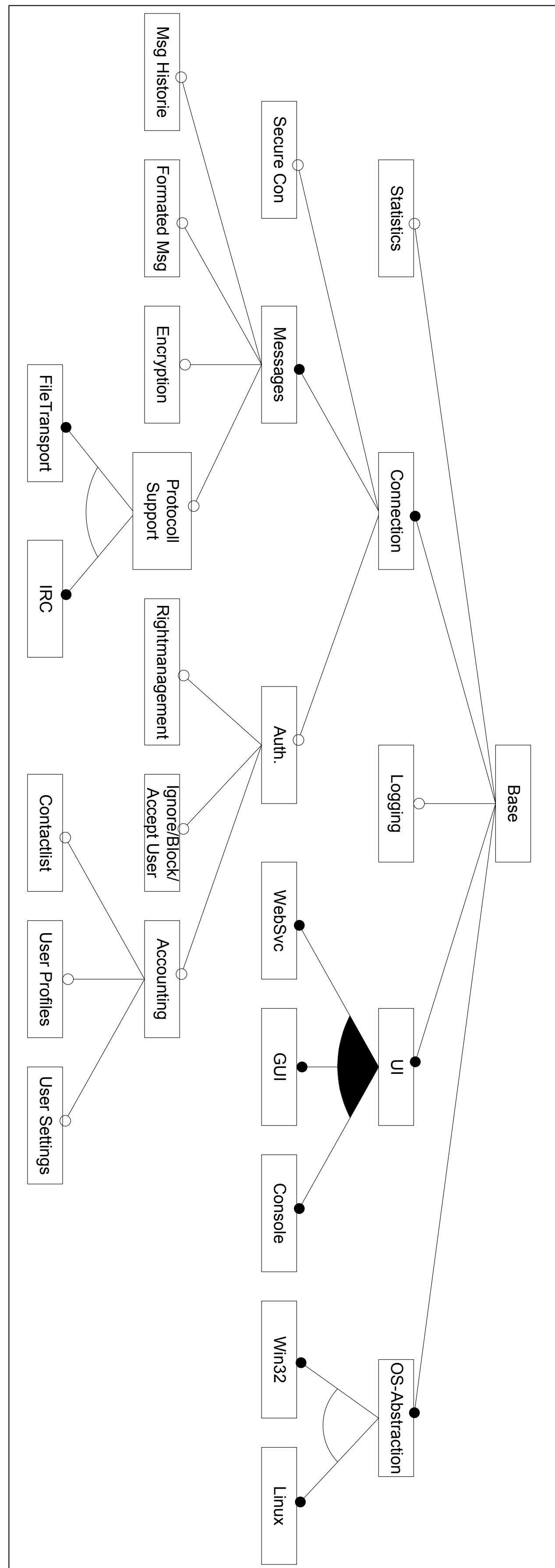

Abbildung 3.1: Gesamtübersicht des SPL

3.3 Probleme

In diesem Abschnitt werden die gefundenen Feature und ihre Interaktionen erläutert. Die einzelnen Abschnitte bestehen aus einer Beschreibung der Funktionalität der betroffenen Feature, einer Zusammenfassung der Designabhängigkeiten der Feature im Featurebaum und einer genauen Beschreibung der Interaktion selbst. Zusätzlich wird der gesamte Sachverhalt abstrakt und unabhängig von der Beispiel-SPL dargestellt.

Die folgenden Erläuterungen bilden die Grundlage für die weiteren Schritte der Arbeit. Sie schaffen ein Grundverständnis für alle in der Fallstudie erkannten Problemfälle.

3.3.1 P1: Benutzerschnittstelle (UI)

Beschreibung: Die Produkte der Chat-SPL unterscheiden sich im Punkt der Benutzerinteraktion. Jedes Produkt enthält mindestens eine Schnittstelle. Die in der Beispielanwendung vorgesehenen Schnittstellen sind eine Konsole, eine graphische Oberfläche (GUI) und die Ansteuerung der Anwendung über Webservices. Diese unterscheiden sich durch die jeweiligen verwendeten Elemente für Ein und Ausgabeoperationen.

Design: Im Featurediagramm erfolgt die Modellierung des Problemfalls mit Hilfe eines Konzeptknotens (UI) unter dem die Implementierungen in einer Oder-Beziehung auftreten. Die Einführung dieses Konzeptknotens erhöht hauptsächlich die Lesbarkeit des Diagramms, das Konzept selber repräsentiert nur den allgemeinen Belang, dass in der SPL eine Implementierung enthalten sein muss.

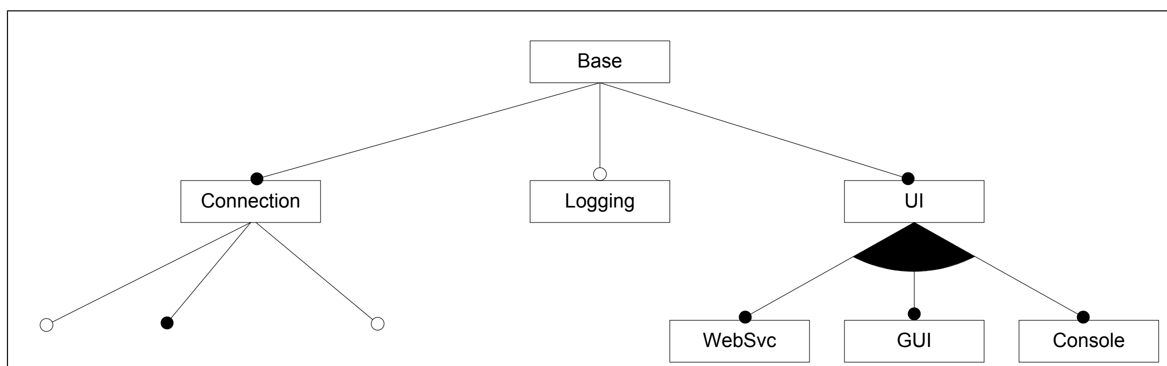

Abbildung 3.2: Übersicht von Problem 1 im FD

Interaktion: Aus dem Featurediagramm ergibt sich die klassische Aufteilung zwischen Benutzerschnittstelle und Businesslogik. Im Detail betrachtet sind diese Teilbäume mit einer Vielzahl von Interaktionen versehen. Viele der Businesslogik Features benötigen Eingaben vom Benutzer oder stellen spezielle Ansichten bereit. Das heißt jedes dieser Feature muss diese Schnittstelle erweitern. Das Feature, welches die Kontaktliste implementiert benötigt eine Möglichkeit diese dem Nutzer anzuzeigen und definierte Operation darauf auszuführen. Zu einem Feature optionality Problem werden diese Erweiterungen durch die alternativ vorhandenen Schnittstellen.

Abstraktion: Ein Konzeptknoten der durch mindestens eine Featureauswahl implementiert wird (Oder). Die Funktionalität des Konzeptes ist ein Hauptbestandteil der Anwendung und wird von vielen Featuren benutzt. Diese Feature benötigen jeweils unterschiedliche Funktionalität.

3.3.2 P2: Betriebssystemabstraktion

Beschreibung: Die Betriebssystemabstraktion ermöglicht die Auslieferung der Anwendung auf unterschiedlichen Client-Betriebssystemen. Anpassungen sind für alle Ein- und Ausgabeoperationen innerhalb der Anwendung nötig. Dies betrifft Dateisystemzugriffe (z.B. Logging) sowie Socketverbindungen (Connection) und alle Benutzerschnittstellen (UI).

Design: Im Featurediagramm tritt die Abstraktion als Konzept auf, dem die möglichen Betriebssysteme in Form von einzelnen Feature untergeordnetet sind. Da eine bestimmte Variation der Anwendung auf genau einem Betriebssystem ausgeführt wird, stehen die einzelnen Betriebssystemfeature in einer Alternativen-Beziehung zueinander.

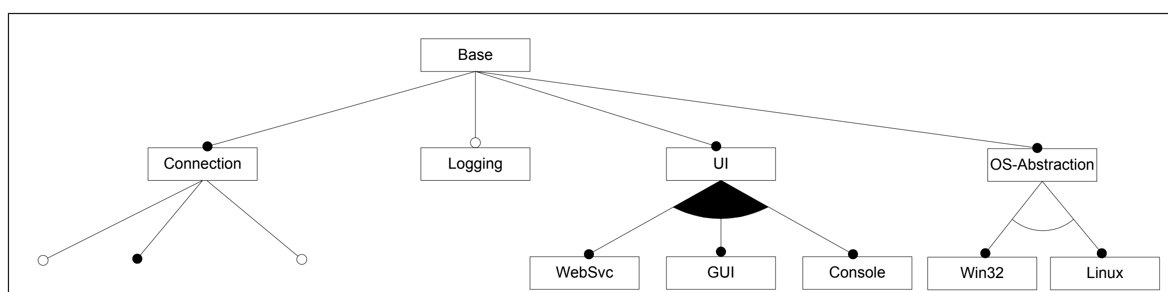

Abbildung 3.3: Übersicht von Problem 2 im FD

Interaktion: Die Interaktion folgt bei der Betriebssystemabstraktion einer Richtung. Alle Komponenten die eine Anpassung aus der Sicht des Betriebssystems benötigen, müssen vom jeweiligen Betriebssystemsfeature angepasst oder beeinflusst werden. Im Connection Feature müssen beispielsweise die verwendeten Sockets in Abhängigkeit vom Betriebssystem unterschiedlich initialisiert und verwendet werden.

Abstraktion: Grundlegende Funktionalität anderer Feature muss speziell auf unterschiedliche Bedingungen angepasst werden. Die Anpassungen werden durch Feature realisiert.

3.3.3 P3: Authentifikation

Beschreibung: Die Authentifizierung erweitert die Fähigkeiten des Nachrichtenfeatures um anonyme und personalisierte Dienste. Zu diesen Diensten gehören zum Beispiel ein Rechtemanagement und eine Kontaktliste.

Design: Bei dem Knoten Authentifizierung handelt es sich um einen reinen Konzeptknoten, der erst durch die Auswahl von seinen Kindknoten Funktionalität repräsentiert. Die Kindknoten sind direkte Feature und weiter verzweigende Konzepte.

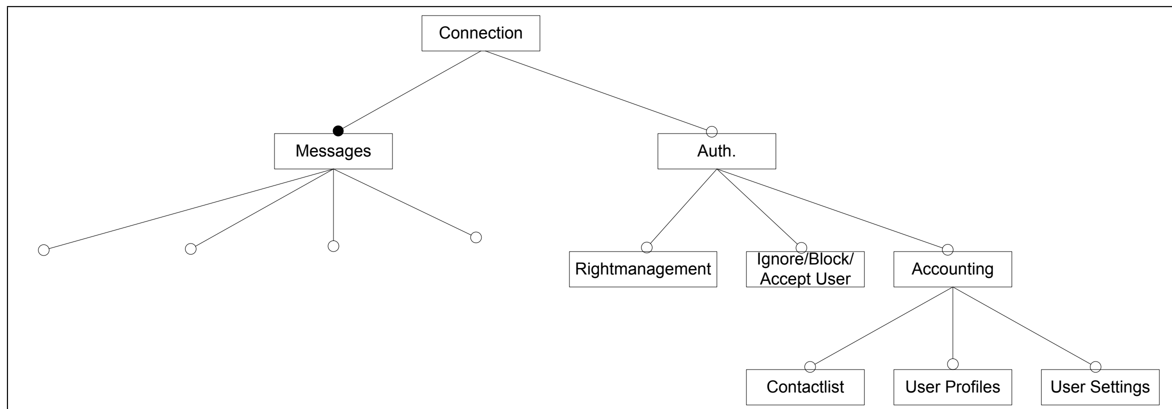

Abbildung 3.4: Übersicht von Problem 3 im FD

Interaktion: Das Konzept ist in diesem Fall der Auslöser der Interaktion, da unabhängig von den ausgewählten Kindknoten ein identisches Problem auftritt. Das Nachrichtenfeature muss einen Syntax oder eine Schnittstelle zur Authentifizierung unterstützen. Dem Rechtemanagement muss es zum Beispiel möglich sein eine Identität für eingehende Nachrichten festzustellen, um diese gegebenenfalls zu blockieren.

Abstraktion: Ein Ast des Baumes benötigt Erweiterungen der Funktionalität eines Features im Baum. Der Wurzelknoten des Astes entspricht einem allgemeinem Konzept der Feature seiner Kindknoten.

3.3.4 P4: Rechtemanagement

Beschreibung: Das Rechtemanagement ermöglicht dem Nutzer für Nachrichtensitzungen, vorgegebene Privilegien und Verbote zu verwalten. Diese Funktionalität teilt sich auf zwei modulare Feature auf. Das Feature Rightmanagement ermöglicht die personalisierte Pflege und Vergabe von Rechten gegenüber anderen authentifizierten Nutzern. Das zweite Feature setzt analoge Methoden zum anonymen Verwalten von Rechten um.

Design: Die zwei interagierenden Feature sind optionale Nachbarknoten.

Interaktion: Die Auswirkungen der Feature überlagern sich zur Laufzeit der Anwendung. Beide Feature setzen ähnliche Funktionalität unter anderen Anforderungen um. Ein Nutzer der anonym auf der Liste der geblockten Nutzer steht kann gleichzeitig einen gegensätzlichen Status im Rechtemanagement haben.

Abstraktion: Mehrere Nachbarknoten implementieren ähnliche (oder ergänzende) Funktionalität und erzeugen somit eine Überschneidung im Ablauf der Anwendung.

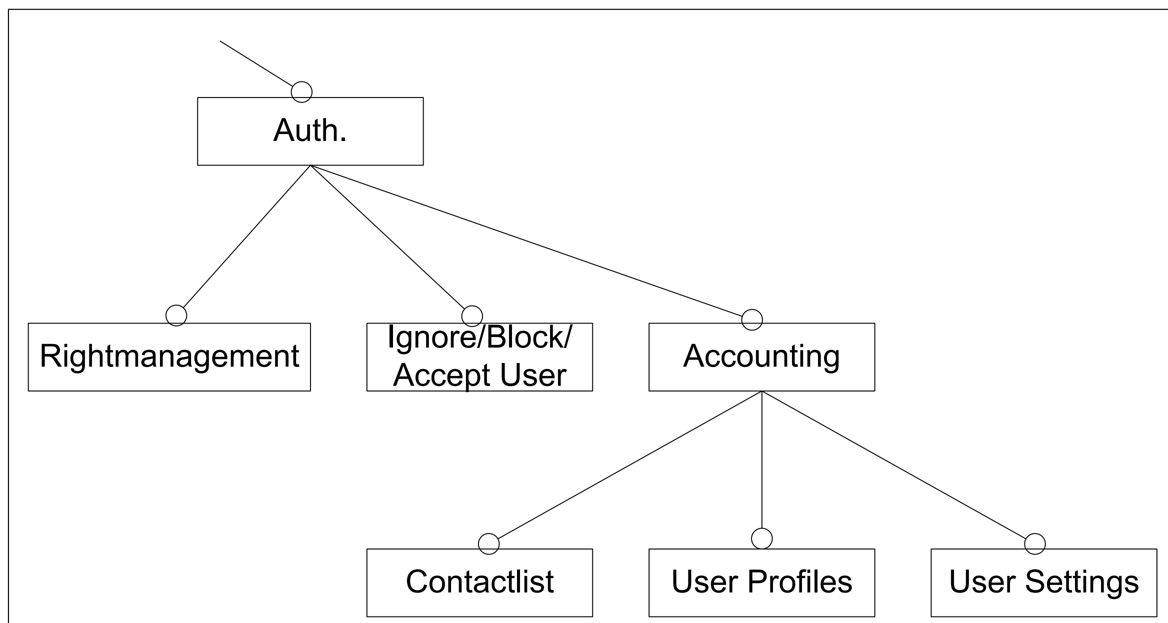

Abbildung 3.5: Übersicht von Problem 4 im FD

3.3.5 P5: Logging

Beschreibung: Die Aufgabe des Loggings ist, die Arbeitsweise anderer Komponenten nachvollziehbar zu protokollieren.

Design: Im Baum ist das Logging als unabhängiges optionales Feature modelliert. Das Feature ist direkt dem Basisfeature (Base) untergeordnet. Die Interaktionen treten in Verbindung mit anderen über den gesamten Baum verteilten Feature auf.

Interaktion: Ein systemweites Logging erzeugt natürlich Interaktionen mit jeder betroffenen Komponente. Jedes „um Loggingfunktionalität" erweiterte Feature hat eine direkte einseitige Interaktion mit dem Loggingfeature.

Abstraktion: Ein optionaler querschneidener Belang erweitert andere Feature um seine eigene Funktionalität.

3.3.6 P6: Sichere Verbindung

Beschreibung: Nachrichtenverbindungen über eine verschlüsselte Verbindung herzustellen ist die Funktionalität des Features Secure Con.

Design: Die sichere Verbindung (Secure Con) ist ein optionaler Kindknoten des Features Connection. Die von der Interaktion betroffenen Feature Messages und Secure Con sind Nachbarknoten.

Interaktion: Das Nachrichtenfeature kann über die sichere Verbindung kommunizieren, wenn dieses optionale Feature ausgewählt ist.

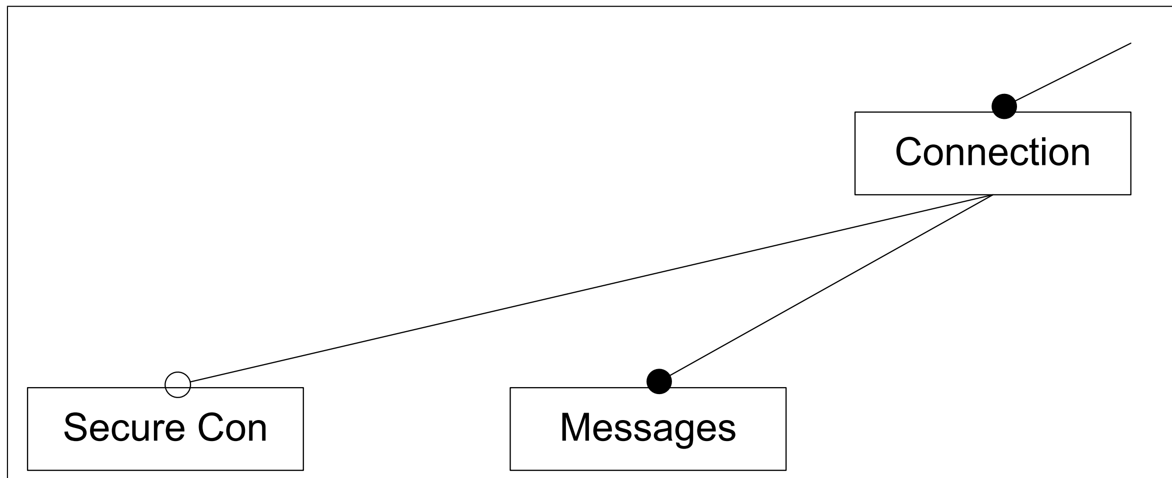

Abbildung 3.6: Übersicht von Problem 6 im FD

Abstraktion: Ein optionales Feature erweitert die Funktionalität eines Feature, das eine zentrale Rolle in dem Ablauf der Anwendung besitzt. Die optionale Funktionalität muss von anderen Featuren im Ablauf genutzt werden können.

3.3.7 P7: Accountmanagement

Beschreibung: Aus der Authentifizierung ergeben sich weitere mögliche Funktionalitäten. Das Abspeichern und Verwalten von benutzerspezifischen Einstellungen ist ein Beispiel dieser Funktionen, welches von dem Feature Accounting umgesetzt wird.

Design: Das Feature Accounting selbst ist ein Konzeptknoten dessen Funktionalität über die Kindknoten repräsentiert wird. Diese drei Knoten stehen in einer Und-Beziehung zueinander.

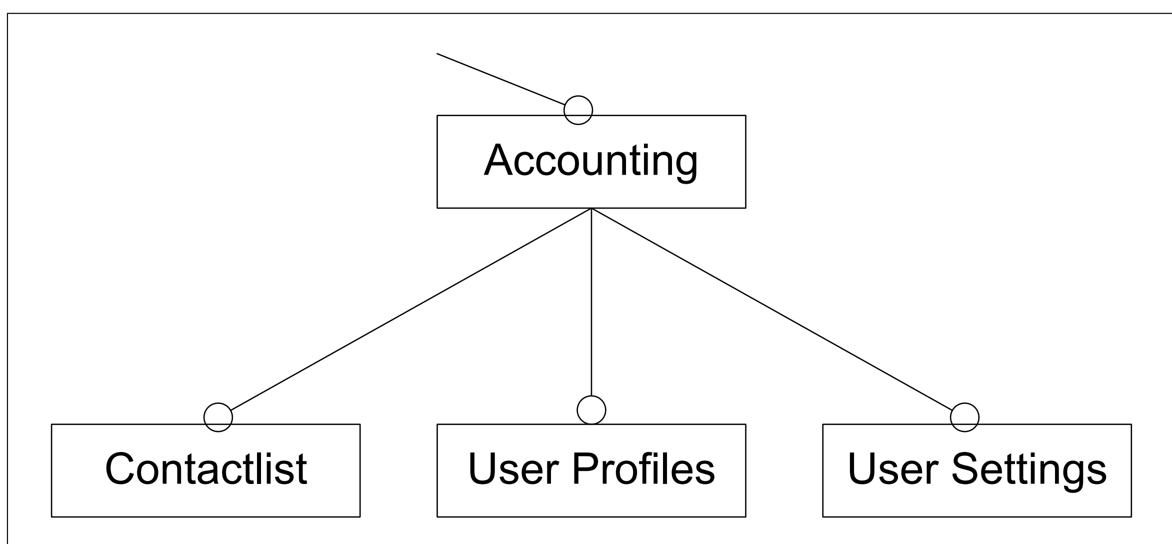

Abbildung 3.7: Übersicht von Problem 7 im FD

Interaktion: Jedes der optionalen Feature des Accountmanagements enthält überschneidende Funktionalität mit den anderen Featuren. Bei der Auswahl von zwei der betroffenen Featuren ergibt sich somit mehr Funktionalität als die Summe der Funktionen der jeweils einzeln ausgewählten Feature.

Abstraktion: Optionale Feature ergänzen gegenseitig ihre Funktionalität und müssen auf die Auswahl der anderen Feature reagieren. Die Kombination mehrerer Feature ermöglicht die Nutzung aufeinander aufbauender Funktionalitäten.

3.3.8 P8: Nachrichtenerweiterungen

Beschreibung: Formatierte Nachrichten werden vor der Ausgabe formatiert und können bei der Eingabe formatiert werden. Die Verschlüsselung stellt sicher das alle Nachrichten verschlüsselt übertragen werden und empfangene Nachrichten entschlüsselt werden. Das Verlaufsfeature zeichnet alle empfangenen und gesendeten Nachrichten auf.

Design: Die Feature Message Historie, Formated Messages und Encryption sind optionale Kindknoten des Features Messages.

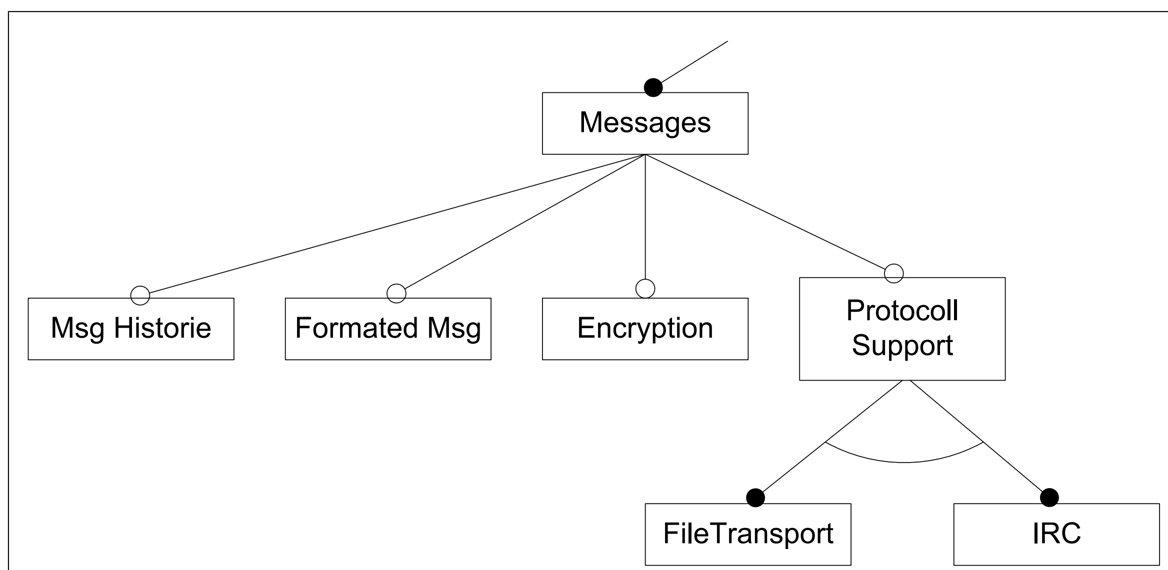

Abbildung 3.8: Übersicht von Problem 8 im FD

Interaktion: Aus Nutzersicht folgt aus einem verschlüsselten Nachrichtenversand das der Verlauf der Nachrichten verschlüsselt gespeichert wird. Bei der Auswahl beider Feature muss somit das Verschlüsselungsfeature zusätzlich das Verlaufsfeature erweitern. Den gleichen Sachverhalt erzeugt das Formatierungsfeature im Zusammenspiel mit dem Verlauf.

Abstraktion: Optionale Feature erweitern die Funktionalität eines obligatorischen Features. Zusätzlich müssen die optionalen Feature jeweils die anderen Feature um diese Funktionalität erweitern.

3.3.9 P9: Statistiken

Beschreibung: Die Statistiken stellen einen mit dem Problem Logging 3.3.5 vergleichbaren Sachverhalt dar. Der Unterschied besteht in dem Ziel der Datenerhebung. Während beim Logging der genaue zeitliche Ablauf der einzelnen Aktionen interessiert, zählen bei der Erhebung von Statistiken ausschließlich die Häufigkeit und Dauer der einzelnen Aktionen.

Design: Ein dem Basisfeature untergeordnetes unabhängiges optionales Feature interagiert mit anderen unabhängig verteilten Featuren.

Interaktion: Das Feature Statistics muss so in alle betreffenden Feature integriert werden, dass keine Beeinflussung der Messungen auftreten. Für jedes betroffene Feature müssen exakte Punkte definiert werden an denen Messungen erhoben werden sollen.

Abstraktion: Die Art der Integration eines Features stellt Ansprüche an die Performance der Anwendung.

3.4 Lösungsstrategien

In diesem Abschnitt werden die Lösungsstrategien für das Optional-Featureproblem erläutert, um einen Überblick über mögliche Techniken zu geben bevor diese direkt auf ihre Eignung bezüglich der gefundenen Probleme hin untersucht werden. Die Lösungen unterscheiden sich in den verwendeten Implementierungstechniken und der Art der Lösungen im Allgemeinen.

Ein Optional-Featureproblem besteht im Allgemeinen aus unterschiedlichen Seiten zwischen denen eine Interaktion auftritt. Die Besonderheit dabei ist, dass mindestens eine der beteiligten Seiten eine optionale Komponente darstellt und somit die Interaktionen nicht in jeder Variante der SPL auftreten. Die Lösungsstrategien müssen somit den korrekten und performanten Ablauf der Anwendung in allen Variationen der SPL sicherstellen.

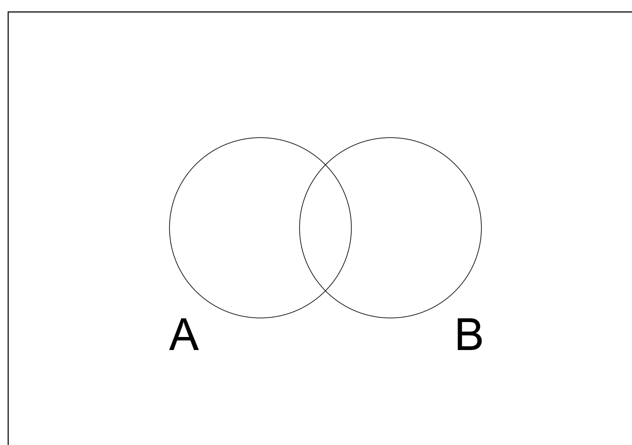

Abbildung 3.9: Optional-Featureproblem[Käs07]

Zur vereinfachten Übersicht über die beschriebenen Lösungsansätze dient ein einfaches abstraktes Beispiel. In diesem Beispiel besteht die Interaktion zwischen zwei Feature. In Abbildung 3.9 sind die beiden Feature dargestellt und ihre Überschneidung repräsentiert den Teil der Feature in dem sie miteinander interagieren.

3.4.1 Designanpassung

Bei der Anpassung des Designs wird die bestehende Interaktion so im Design abgebildet, dass keine Problemfälle mehr ausgewählt werden können. Eine Ausprägung der SPL bei der die Interaktionen auftreten, werden explizit im Design der SPL ausgeschlossen. Wenn beispielsweise zwei Feature eine Interaktion aufweisen, wird bei dieser Lösungsstrategie dem Design die Bedingung hinzugefügt, das jeweils nur genau eines der beiden betroffenen Feature ausgewählt sein darf. Das Design der SPL wird um ausschließende Bedingungen zwischen Featuren ergänzt oder durch Umwandlungen von Beziehungen umgeformt. Eine Oder-Beziehung zwischen interagierenden Featuren, verhindert wenn sie in eine Alternative-Beziehung überführt wird das gemeinsame Auftreten der Feature. Die Folge der Anpassungen ist der Verlust von Varianten in der SPL, da kritische Varianten generell ausgeschlossen werden. Der große Vorteil dieser Lösung ist, dass sie keine Anpassungen am Quellcode der Anwendung benötigt. Zur Vermeidung von Optional-Featureproblemen müssen nur die Interaktionen erkannt werden, um die entsprechenden Abhängigkeiten dem Design hinzuzufügen. Vorausgesetzt das die Interaktionen korrekt erkannt wurden, verhindert dieser Lösungsansatz zuverlässig ihr Auftreten.

In den Spezialfällen des Optional-Featureproblems, bei denen eine der beteiligten Seiten keinen optionalen Teil der SPL darstellt, hat die Anwendung dieser Lösung besondere Folgen. Die benötigten Designabhängigkeiten verhindern die Auswahl des optionalen Teiles in allen Varianten der SPL. Der Ausschluss eines Teiles der Interaktion stellt somit die Lösung in diesem Fall dar.

Nicht nur in diesem speziellen Szenario ist die Folge dieser Lösungsstrategie ein Verlust von Varianten der SPL. Alle Varianten der SPL in denen die beteiligten Feature gemeinsam vorkommen, werden bei dieser Lösung aus den möglichen Varianten der SPL eliminiert.

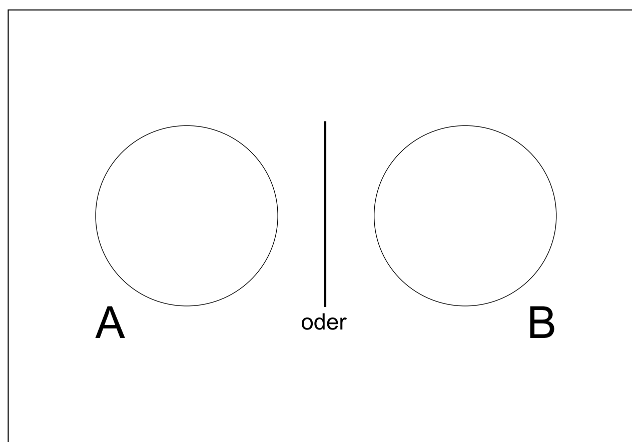

Abbildung 3.10: Designanpassung

Abbildung 3.10 zeigt, dass es durch die Anpassung des Designs zu keiner Interaktion

(Überschneidung) der beiden Feature kommt, da immer nur eines ausgewählt werden kann.

Die Anpassung des Designs wurde von Kästner et. al. unter der Bezeichnung Ignorieren der Abhängigkeit in abstrakterer Form betrachtet [KAR+09].

3.4.2 Quellcodeaustausch

Eine Grundidee der modularen Programmierung ist der Austausch von Komponenten, welche identische Funktionalität auf unterschiedliche Art und Weise umsetzten. Im Fall des Optional-Featureproblems stellen somit zwei Implementierungen eines Features eine mögliche Lösung dar, wenn eine der Implementierungen die Auswahl der anderen Feature berücksichtigt und die andere Implementierung vom Fehlen dieser Feature ausgeht. Im einfachen Beispiel (Abbildung 3.11) existiert eine Implementierung (B2) von Feature B, welche mit Feature A interagiert und eine Implementierung (B1) die keine Interaktion mit Feature A aufweist. Abgesehen von dem Code der Interaktion unterscheiden sich die Implementierungen von Feature B nicht.

In Abhängigkeit von der Auswahl der Feature für eine Variante der SPL muss bei diesem Lösungsansatz zusätzlich die korrekte Implementierung der Feature ausgewählt werden. Die benötigten Schritte zur Erstellung einer Variante der SPL wird, um den Schritt der Auswahl der richtigen Implementierungen erweitert.

Ein gravierenderer Nachteil dieser Lösung ist der hohe Grad der Replikation der unterschiedlichen Implementierungen. Der Aufwand für die Weiterentwicklung und Wartung des betroffenen Features steigt an, da alle Implementierungen bearbeitet und getestet werden müssen.

Eine besondere Art dieser Lösung ist möglich, wenn die Interaktion des OFP auf eine Erweiterung der Funktionalität eines der betroffenen Feature zurückgeht. In diesem Fall ermöglicht ein optionales Feature einem anderen Feature eine andere Funktionsweise, in dem es die Rahmenbedingungen verändert oder neue Funktionalität bereitstellt. Die alternative Implementierung kann in diesem Fall zu einem neuen Feature der SPL werden, welches vom anderen Featuren abhängig ist.

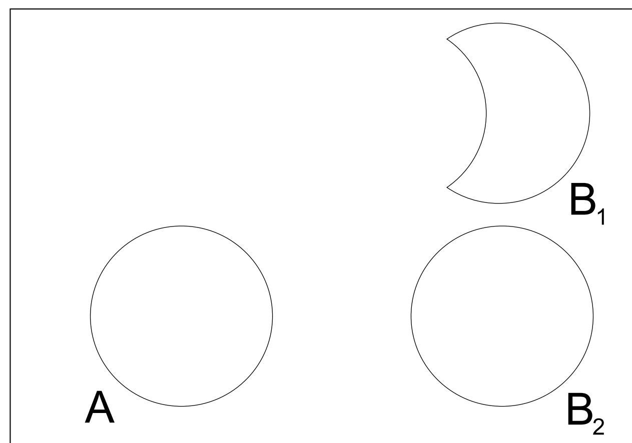

Abbildung 3.11: Quellcodeaustausch

3.4.3 Bedingte Kompilierung (Präprocessor)

Durch eine Vorverarbeitung des Quellcodes eines oder mehrerer der betroffenen Feature bietet die bedingte Kompilierung eine Möglichkeit nötige Anpassungen vorzunehmen, um die Interaktion der Feature zu ermöglichen. Indirekt enthält somit jedes bearbeitete Feature mehrere Ausprägungen, die speziellen Varianten der SPL zugeordnet sind. Die Ausprägungen entsprechen dabei den alternativen Implementierungen der vorherigen Lösung. Aufgrund der Vorverarbeitung kann der Anteil an replizierten Code bei diesem Ansatz stark vermindert werden. Es besteht dabei ein direkter Zusammenhang zwischen dem Grad der Replikation und der Anzahl der betroffenen Codestellen innerhalb eines Features. Je feingranularer die Präprozessoranweisungen in dem Quellcode auftreten, desto weniger identischer Code ist in den Modulen enthalten.

Mit der zunehmender Granularität der Anweisungen verschlechtert sich die Lesbarkeit des Quellcodes, wodurch sich der Aufwand für Wartung und Weiterentwicklung des Features erhöht.

3.4.4 Auslagern der Interaktion (Derivate)

Die theoretische Grundlage der Auslagerung der Interaktion geht davon aus, dass die Interaktion zwischen zwei Featuren modularisiert werden kann. Der Quellcode bzw. die Veränderungen des Quellcodes, die für eine korrekte Interaktion der Feature nötig ist, wird bei dieser Lösung Teil eines neuen Features. Diese Derivate genannten Feature werden nur bei der Auswahl beider interagierenden Feature zusätzlich ausgewählt. Derivate stellen keine eigenständigen Feature dar und sind direkt den betroffenen Featuren zugeordnet.

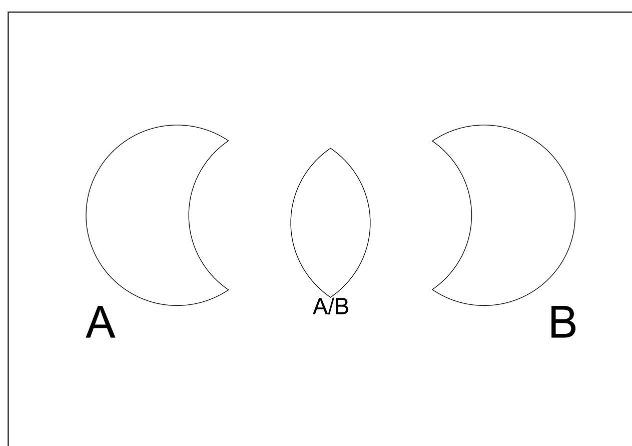

Abbildung 3.12: Auslagern der Interaktion (Derivate) [Käs07]

Abbildung 3.12 zeigt die Feature A und B, welche jeweils ohne die interagierende Teile implementiert sind. Das Derivat A/B enthält diese Teile der beiden beteiligten Feature und erzeugt die Interaktion von Feature A und Feature B in dem es sie, um die interagierenden Teile erweitert. Entscheidend ist, dass das Derivat A/B nur dann ausgewählt werden darf, wenn Feature A und Feature B ausgewählt sind.

Die Grundidee der Derivate geht davon aus, dass die Interaktion auf das Zusammenwirken von zwei Featuren begrenzt ist. In diesem Fall wird für jede Interaktion

ein zusätzliches Feature benötigt. Dies bedeutet im schlechtesten Fall, einer Interaktion zwischen allen Featuren, das n(n-1)/2 zusätzliche Feature benötigt werden. Theoretisch sind allerdings auch Interaktionen zwischen Gruppen von Featuren möglich. Bei einer vollständigen Interaktion von drei Featuren werden somit bereits vier Derivate benötigt.

Die Entscheidung, welche Derivate zu einer gegebenen Auswahl von Featuren ausgewählt werden müssen, ist trivial, wenn die Abhängigkeiten bekannt sind. Derivat A/B aus Abbildung 3.12 muss mit der Bedingung das Feature A und Feature B vorhanden sind verknüpft werden. Da diese Regeln in einem vom Computer lesbaren Format abgelegt werden können, ist eine Automatisierung der Derivatauswahl möglich. Eine automatisierte Auswahl ermöglicht es ferner, die durch die Derivate entstandene zusätzliche Komplexität vor dem Nutzer zu verstecken. Der Nutzer wählt somit weiterhin nur die reinen Feature aus und wird von der Existenz von Derivaten nicht tangiert.

3.4.5 Optionales Weben

Die AOP modifiziert Quellcode an definierten Stellen, wie im Abschnitt 2.1.2 beschrieben ist. Die Idee um Optional-Featureprobleme zu vermeiden geht davon aus, dass diese Veränderung des Quellcodes nur dann vollzogen wird, wenn die definierten Stellen im Quellcode enthalten sind. Ein Aspekt der ein optionales Feature erweitert, kann dieses nur in den Fällen erweitern in denen es ausgewählt ist.

3.4.6 Kapselung

Feature die in der Beziehung einer Alternative zueinander stehen, entsprechen dem klassischen OO-Konzept des Austauschs der Implementierungen eines Moduls. Werden die alternativen Feature somit in der SPL über ein gemeinsames Interface angesprochen, können sie einfach gegeneinander ausgetauscht werden. Die Trennung von Schnittstelle und Implementierung lässt sich am Design der SPL nachvollziehen. Der Konzeptknoten der die Alternative aufspannt, entspricht dem gemeinsamen Interface seiner Kindknoten. Die unterschiedlichen Implementierungen des Interfaces stellen die alternativen Feature da.

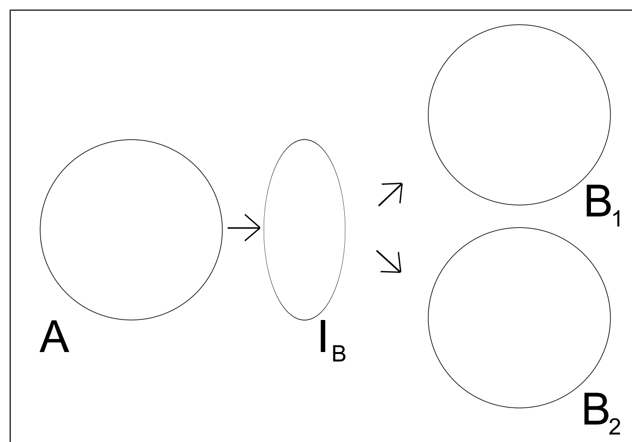

Abbildung 3.13: Lösung mit Interface

3.5 Untersuchung der Lösungen

Die im Abschnitt 3.4 beschriebenen Lösungsansätze wurden zur Lösung der heraus gearbeiteten Optional-Featureprobleme der Chat-SPL implementiert. Die Analyse der Eignung der einzelnen Lösungen für die speziellen Problemfälle wird in diesem Abschnitt beschrieben.

Bisherige Untersuchungen von Lösungen des Optional-Featureproblems betrachteten diese allgemein ohne eine Unterscheidung der einzelnen Problemfälle [KAR+09]. Im Rahmen dieser Arbeit stellte sich heraus, dass erst in Bezug auf die Problemfälle eine verwertbare Bewertung der Lösungsansätze erfolgen kann.

Eine Unterscheidung der Bewertungskriterien in zwei unterschiedliche Klassen verdeutlicht den subjektiven Charakter einiger Kriterien. Der Aufwand zur Umsetzung einer beliebigen Lösung für einen Problemfall lässt sich leicht anhand der veränderten bzw. neuimplementierten Quellcodezeilen messen. Entscheidend ist dabei jedoch hauptsächlich die Komplexität der benötigten Anpassungen. Die stupide und gleichartige Anpassung von mehreren Quellcodeabschnitten kann einen geringeren Aufwand aufweisen, wie die Modifikation weniger bestimmter, eventuell schwer zu identifizierenden Quellcodezeilen.

3.5.1 Nachrichtenerweiterungen

Die Nachrichtenerweiterungen sind drei Feature, welche die Funktionalität eines obligatorischen Features erweitern. Zusätzlich schafft jedes der optionalen Feature Erweiterungsmöglichkeiten für das optionale Verlaufsfeature. Das Verlaufsfeature ist der zentrale Punkt der Interaktion. In der umgesetzten SPL existieren somit vier mögliche Ausprägungen des Verlaufsfeatures, der reine Verlauf, ein verschlüsselter Verlauf, ein formatierter Verlauf und ein verschlüsselter und formatierter Verlauf.

Designanpassung: Eine Designanpassung kann in diesem Fall über ausschließende oder erfordernde Bedingungen realisiert werden. In Abhängigkeit von der Version des Verlaufsfeatures, welche in der SPL erhalten werden soll. Für den verschlüsselten Verlauf müsste beispielsweise sichergestellt werden, dass der Verlauf nur dann ausgewählt werden kann, wenn das Feature zur Verschlüsselung ausgewählt ist und das Feature der Formatierung nicht in der Auswahl enthalten ist. Somit kann durch eine Designanpassung genau eine der vier denkbaren Ausprägungen des Verlaufs erhalten werden. Der nötige Aufwand für die Designanpassung hängt dabei von dem Zeitpunkt ab, in dem die Interaktion bemerkt wird. Wenn bereits eine der möglichen Ausprägungen umgesetzt ist, kann ausschließlich diese Variante ohne Anpassungen des Quellcodes in der SPL erhalten werden. Der Erhalt einer anderen Variante verlangt zusätzlich zu der Anpassung des Designs Das Hinzufügen oder Entfernen von Interaktionen aus dem Quellcode. Das mehrere Ausprägungen für eine Designanpassung zur Auswahl stehen, geht darauf zurück das die Interaktionen zwischen den Featuren nur dann notwendig sind, wenn die zusätzliche Funktionalität die sie implementieren benötigt wird.

Quellcodeaustausch: Für den Quellcodeaustausch mussten die vier Ausprägungen des Verlaufes einzeln implementiert werden. Die Grundfunktionalität des Nachrichtenver-

laufs ist in allen Implementierungen in nahezu identischer Art enthalten. Unterschiede bestehen zwischen den Implementierungen in der Speicherfunktionalität des Verlaufs. Beim normalen Verlauf müssen die gespeicherten Verlaufsdaten nicht vorverarbeitet werden, während bei dem verschlüsselten Verlauf eine Entschlüsselung der Daten beim Auslesen erfolgen muss. Das Umsetzen der Formatierungssyntax stellt einen ähnlichen Vorverarbeitungsschritt dar. Die zusätzlichen Funktionalitäten benötigen jeweils die Anpassung der Implementierung an zwei Stellen im Quellcode. Für einen verschlüsselten Verlauf müssen Speichern und Laden angepasst werden und für den formatierten Verlauf die Ein und Ausgabe.

Bedingte Kompilierung: Bei der Lösung mit bedingter Kompilierung mussten adäquat zur vorherigen Lösung vier Stellen angepasst werden. Im Detail zwei Blöcke für den verschlüsselten Verlauf und zwei Blöcke für den formatierten Verlauf. Die Blöcke sind nicht ineinander geschachtelt und jeweils zusammenhängend, was die Übersichtlichkeit im Quellcode nur geringfügig verschlechtert.

Auslagern der Interaktion: Die Auslagerung der Interaktionen erfolgt in zwei unabhängige Derivate. Das Derivat für den verschlüsselten Verlauf, sowie das Derivat für den formatierten Verlauf bestehen aus jeweils einem Aspekt, welcher den Quellcode an zwei Stellen verändert. Die Bedingungen, wann die Derivate ausgewählt werden müssen, sind in diesem Fall einfach und ermöglichen eine Automatisierung der Auswahl.

Die Derivate erzeugen bei diesem Problemfall neue Funktionalität für ein optionales Feature und können als Feature interpretiert werden. Durch diese Integration der beiden Derivate in das Design der SPL erhöht sich durch diese Lösung die Anzahl der möglichen Varianten. Bei der Auswahl kann beispielsweise trotz der Auswahl des Verschlüsselungsfeatures für Nachrichten auf einen verschlüsselten Verlauf verzichtet werden.

Optionales Weben: Das optionale Weben entspricht der Lösung der Derivate. In diesem Fall erfolgt die Zuordnung der Aspekte direkt zu den entsprechenden funktionalen Featuren. Der Aspekt der einen verschlüsselten Verlauf erzeugt, ist so beispielsweise in den Aspekt der Verschlüsselung integriert.

Kapselung: Das Prinzip der Kapselung und andere Prinzipien der OOP bieten keine geeignete Möglichkeit zur Auflösung oder Vereinfachung der Interaktionen dieses Problemfalls.

3.5.2 Accountmanagement

Das Accountmanagement stellt einen den Nachrichtenerweiterungen identischen Problemfall dar. Insgesamt treten drei Interaktionen auf, die wie im vorherigen Abschnitt beschrieben gelöst werden können. Eine detailierte Beschreibung der umgesetzten Lösungen ist daher nicht notwendig.

3.5.3 Rechtemanagement

Das Rechtemanagement stellt einen Sonderfall dar. Die Interaktion beruht bei den beteiligten Featuren darauf, dass jedes eine eigene Datenverwaltung implementiert. Die Auswertung der unterschiedlichen Daten ergibt Seiteneffekte der Funktionalität der einzelnen Feature, die jeweils ein Rechtemanagement implementieren.

Designanpassung: Eine Designanpassung zur Auflösung dieser Interaktion stellt die beste Lösungsstrategie für diesen Problemfall dar, wenn man davon ausgeht das ein Fehler im Design die Ursache der überschneidenen Funktionaliltät ist. In dem betrachteten Problemfall widersprechen sich die Anforderungen der beiden Feature nicht, was eine Koexistenz in Varianten der SPL erfordern kann. Die Anpassung des Designs vermindert somit die Zahl der Produkte die aus der SPL generiert werden können. Die Art der Anpassung stellt jedoch einen Spezialfall dar, weil sie für diesen Problemfall eine gleichzeitige Auswahl der Feature nicht erfordert sondern für alle Varianten der SPL verbietet.

Quellcodeaustausch: Die Interaktionen dieses Problemfalls spiegeln sich ausschließlich in der Funktionaliltät der beteiligten Feature wieder. Eine Neuimplementierung der Feature erfordert die Auswahl einer Methodik, um die Überschneidung der Funktionalität zu verhindern. Im Rahmen dieser Arbeit erfüllt eine Sperrkontrolle diese Aufgabe. Um ein Recht an einen Kontakt zu vergeben, muss jedes der Feature vorher selbstständig sicherstellen, dass der betroffene Kontakt nicht von dem anderen Feature verwaltet wird. Somit wird der gegenseitige Ausschluß der beiden Feature von der Designebene auf eine andere Ebene verschoben. Im Gegensatz zur Designanpassung können durch die Reimplementierung der Feature beide Feature gemeinsam Teil einer Variante der SPL sein. Die gleichzeitige Benutzung der beiden Feature für die gleiche Instanz eines Kontakts wird jedoch von der Programmlogik verhindert.

Bedingte Kompilierung: Mit Hilfe von Präprozessoranweisungen konnten die für den Quellcodeaustausch neuimplementierten Überprüfungen in die bestehenden Feature integriert werden. Der Aufwand der Implementierung dieser Lösung entspricht somit der vorherigen Lösung. Der Vorteil dieser Umsetzung das keine Duplikate von identischem Quellcode entstehen, zeigt sich bei dem Rechtemangement besonders deutlich. Der durch den Präprozessor hinzugefügte Quellcode ergänzt die Grundimplementierungen der interagierenden Feature. Die ursprüngliche Umsetzung der Feature bleibt vollständig erhalten.

Auslagern der Interaktion: Die bereits für die beiden vorher betrachteten Lösungen umgesetzte Implementierung muss für eine Auslagerung in Derivate aufgeteilt werden. Um eine Funktionsüberschneidung der Feature sicher auszuschließen müssen beide Feature erweitert werden. Somit zeigt sich auch bei der Auslagerung der Interaktion die Besonderheit dieses Problemfalls. Um die auftretende Interaktion auszulagern sind zwei vom Aufbau her identische Derivate notwendig. Jedes der Derivate erweitert gleichzeitig beide Feature. Jedes Feature muss einerseits um die Abfragen erweitert, ob der betroffene

Kontakt bereits mit Rechten versehen ist, und muss gleichzeitig eine Möglichkeit bieten, um diesen Status für einen beliebigen Kontakt abzufragen.

Optionales Weben: Ein eindeutiges Zuordnen des interagierenden Teils des Quellcodes zu einem der Feature ist wie bereits bei den Derivaten beschrieben nicht möglich. Ein optionales Weben dieses Quellcodes funktioniert somit nicht mit dem vorher beschriebenen Aufbau der Derivate. Um ein korrektes optionales Weben zu implementieren, musste die Funktionalität zur Abfrage, ob ein Kontakt gesperrt ist, in den Quellcode der Feature übernommen werden. Die Komplexität der Feature umfasst somit Funktionalität die ausschließlich in Varianten mit beiden Featuren des Rechtemanegements benötigt wird. Das optionale Weben erweitert die Funktionalität der Feature nur um die Abfragemechanismen selbst.

Kapselung: Die Interaktionen des Rechtemanagements könnten durch eine komplette Umgestaltung des Problemfalls vereinfacht werden. Eine gemeinsame zentrale Verwaltung der Daten fasst die auftretenden Interaktionen in einem Modul zusammen. Die genauen Auswirkungen dieser Umgestaltungen wurden im Rahmen dieser Arbeit nicht untersucht.

3.5.4 Benutzerschnittstelle

Die Alternativen der Benutzerschnittstelle werden von optionalen und obligatorischen Featuren zur Aus- und Eingabe benutzt. Einige Feature benötigen dafür spezielle Konstrukte, die alle Benutzerschnittstellen unterstützen müssen.

Designanpassung: Eine Designanpassung um alle Interaktionen der Benutzerschnittstellen mit anderen Featuren auszuschließen hat einen großen Umfang aufgrund der Vielzahl an beteiligten Featuren. Die Anpassung ist nur dann möglich wenn genau eine der Alternativen jeweils die Funktionalität für das jeweilige Feature bereitstellt. Der Ausgangspunkt für die Designanpassung in dieser Arbeit war, dass die grafische Schnittstelle (GUI) alle benötigten Konstrukte bereitstellt. Die anderen beiden alternativen Schnittstellen stellten nur die von obligatorischen Featuren benötigten Konstrukte bereit. Die Anpassung des Designs besteht somit aus einer Bedingung für jedes optionale Feature, welches auf Ein- oder Ausgabeoperationen zugreift. Die Schnittstellen Konsole und Webservice können bei dieser Lösung ausschließlich ohne eine Auswahl dieser Feature gewählt werden.

Quellcodeaustausch: Der Lösungsansatz des Quellcodeaustausches wurde für diesen Problemfall aufgrund seines hohen Umfangs nur theoretisch betrachtet. Anhand der Erkenntnisse der weiteren Lösungen sind für den Quellcodeaustausch vielfältige Implementierungen nötig. Für jedes optionale Feature und alle möglichen Kombinationen dieser Feature wäre je eine Implementierung der drei Schnittstellen notwendig.

Bedingte Kompilierung: Die bedingte Kompilierung durchzieht alle drei Schnittstellen mit drei bis 6 Anweisungsblöcken pro interagierendem Feature. In den einfachen

Fällen handelt es sich um abgeschlossene, nicht verschachtelte, Abschnitte die neue Funktionen vollständig in die Schnittstellen integrieren. Bei Überschneidungen der Funktionalität der optionalen Feature untereinander, kommt es zusätzlich zu verschachtelten Blöcken. Ein Beispiel dafür ist die Möglichkeit Benutzerprofile von Kontakten anzeigen zu lassen. Der Menüeintrag, der diese Funktionalität aufruft, muss in die Kontaktliste integriert werden. Diese Verschachtelung der Präprozessoranweisungen realisiert somit die Abbildung der Kombinationen der optionalen Feature. Aus der Sicht der Modularität werden bei dieser Lösung Teile der optionalen Feature in den Quellcode der Schnittstellen verstreut. Das Auffinden aller Teilstücke eines Features ist ohne Werkzeugunterstützung für die Programmierung sehr zeitaufwändig.

Auslagern der Interaktion: Für diesen Problemfall erfolgte die Umsetzung der Derivate mit Hilfe von AOP. Aufgrund der Richtung der Interaktionen modifizieren die Derivate ausschließlich den Quellcode der Benutzerschnittstellen. Da die Umsetzungen von Steuerelementen für einen Webservice, eine grafischen Oberfläche und einer Konsole sich unterscheiden, muss auch bei den Derivaten für jede unterstützte Schnittstelle ein Derivat erstellt werden. Für jede Interaktion der Bernutzerschnittstelle mit einem optionalen Feature sind somit drei unterschiedliche Derivate erforderlich. Zusätzlich benötigt jede Interaktion zwischen zwei optionalen Featuren drei weitere Derivate, wenn die Interaktion Auswirkungen auf die Benutzerschnittstelle hat.

Optionales Weben: Die Umsetzung der Lösung mit optionalem Weben verläuft bei den Benutzerschnittstellen nach einem abgeänderten Prinzip. Jedes optionale Feature wurde um einen Aspekt erweitert der so aufgebaut ist, dass er alle drei Schnittstellen erweitert. Die Aspekte enthalten drei unabhängige Bereiche mit Joinpoints für jeweils eine der Schnittstellen. Welche der Schnittstellen durch den Aspekt erweitert wird, ergibt sich aus der Auswahl der SPL. In jedem Fall findet somit ein Drittel der Aspekte die Stellen die er modifizieren soll. Zur Abbildung der Interaktionen der optionalen Feature müsste der Aspekt eines der beteiligten Feature auf die Auswirkungen des anderen Features reagieren.

Kapselung: Das Prinzip der Kapselung stellt für diesen Problemfall eine Teillösung dar. Mit Hilfe einer identischen Zugriffsmöglichkeit auf alle Alternativen der Benutzerschnittstellen vermindert sich die Komplexität dieses Optional-Featureproblems. Wenn jede Benutzerschnittstelle eigene Funktionen bereitstellt, müssten zur Lösung der Interaktionen auf beiden beteiligten Seiten der Interaktion von der Featureauswahl abhängige Veränderungen vorgenommen werden. Durch eine einheitliche Zugriffsstruktur werden die Benutzerschnittstellen abstrahiert. Die Feature die auf die Funktionalität einer Schnittstelle zugreifen, agieren mit dem abstrakten Interface und nicht direkt mit den einzelnen Alternativen. Die Problemmatik eines Optional-Featureproblems wird dadurch lediglich vereinfacht und nicht vollständig gelöst.

3.5.5 Betriebssystemabstraktion

Die Betriebssystemabstraktion ist notwendig, um bestimmte Implementierungen anderer Feature an das Umfeld unterschiedlicher Betriebssysteme anzupassen. Die alternativen

Feature repräsentieren jeweils ein Betriebsystem und müssen andere Feature anpassen. Die Implementierung der Feature erfolgte auf der Grundlage eines der Betriebssysteme, wodurch das Betriebssystemfeature Win32 keine Anpassungen vornehmen muss.

Designanpassung: Da eine Grundimplementierung aller Feature vorliegt, kann eine Designanpassung die Interaktionen der Betriebssystemabstraktion lösen. Optionale Feature die eine Anpassung auf die Gegebenheiten des Betriebssystems benötigen, werden durch eine Bedingung an das Ausgangsbetriebssystem (Win32) gebunden. Für die alternativ unterstützten Betriebssysteme der SPL stehen diese Feature bei dieser Lösung nicht zur Verfügung. Insgesamt mussten für diese Designanpassung sieben optionale Feature an das Ausgangsbetriebssystem gebunden werden.

Quellcodeaustausch: Der Quellcodeaustausch erfordert zusätzliche Implementierungen für jedes unterstützte Betriebssystem. Alle betroffenen optionalen Feature müssen mehrfach implementiert werden, um die Abstraktion auf unterschiedliche Betriebssysteme abzubilden. Die Unterschiede zwischen den Implementierungen eines Features fallen sehr gering aus und betreffen lediglich einzelne Funktionen. Bei fünf der betroffenen sieben Featuren stellten ausschließlich Zugriffe auf das Dateisystem ein Problem dar. Die Anpassungen dieser Quellcodeabschnitte gestaltete sich somit recht einfach.

Bedingte Kompilierung: Bei der Umsetzung der Betriebssystemabstraktion mit bedingter Kompilierung sind hauptsächlich zusätzliche Initialisierungen notwendig. Die Anpassungen betreffen in der gesamten SPL vier unterschiedliche Konstrukte, welche in vielen optionalen und obligatorischen Featuren wiederholt auftreten. Eine Modularisierung der Konstrukte (z.B. Dateizugriffe) könnte den Aufwand der Lösung deutlich vermindern.

Ein Betriebssystemfeature enthält in diesem Fall keinen Quellcode. Zur Auswahl eines Betriebssystems müssen lediglich die entsprechenden Definitionen für die zugehörigen Präprozessoranweisungen gesetzt werden, damit diese die nötige Funktionalität in die Feature integrieren.

Auslagern der Interaktion: Die Deklaration der Stellen, auf die ein Aspekt wirken soll, bietet vielfältige Möglichkeiten. Für die Lösung dieser Featureinteraktionen ist es von Vorteil, dass sich mehrere Funktionen mit einer Deklaration gemeinsam adressieren lassen. Die Mächtigkeit dieses Vorteils ergibt sich aus den identisch gestalteten Funktionen, die betriebssystemabhängigen Logik ausführen. In der Chat-SPL lassen sich alle Funktionen, die einen Dateizugriff realisieren, mit Hilfe eines Derivats an das ausgewählte Betriebssystem anpassen. Die erstellten Derivate sind keinem Feature eindeutig zugeordnet. Die Zuordnung erfolgt wenn ein Feature eine von den Derivate betroffene Funktion implementiert.

Zur Realisierung der Betriebssystemabstraktion besteht jedes Betriebssytemfeature aus insgesamt vier einfachen Derivaten. Die Anpassungen betreffen für jedes betrachtete Betriebssystem die gleichen Funktionen.

Optionales Weben: Bei diesem Problemfall entspricht bereits die vorher beschriebene Auslagerung der Interaktion einem optionalem Weben. Die erstellten Derivate erweitern alle Funktionen, welche als betriebssystemabhängig deklariert wurden.

Kapselung: Eine Kapselung der Betriebssystemfeature ergibt keinen Sinn, da keines der Feature von anderen Featuren benutzt wird. Die Betriebssystemfeature erweitern bestimmte Funktionalität anderer Feature.

Die betriebssystemabhängige Funktionalität kann mit Hilfe OOP in Klassen ausgelagert werden. Diese Klassen können dann von jedem Feature benutzt werden. Dieses Vorgehen entspricht einer Art der Kapselung, da jedes Betriebssystem seine Implementierung dieser Klassen in die SPL integriert.

3.5.6 Authentifikation

Die Authentifikation fasst eine Vielzahl gleichartiger Interaktionen zusammen. Das Problem besteht in der dynamischen Zuordnung von Daten anhand ihres syntaktischen Aufbaus. Für jedes Datenpaket muss das Feature Conection eindeutig eine Zuordnung zu dem richtigen Feature treffen.

Die Funktionalität zur Zuordnung der Daten existiert im Conection Feature und wird mit der Auswahl der optionalen Feature erweitert. Eine fehlerhafte Interaktion tritt dann auf, wenn optionale Feature eine ähnliche oder identische Syntax für ihre Daten verwenden.

Die Vermeidung von fehlerhaftem Verhalten einzelner Varianten muss somit in der Designphase vollzogen werden. Eine eindeutige Definition der für den Nachrichtenverkehr erlaubten Syntax verhindert das beschriebene Fehlverhalten.

3.5.7 Sichere Verbindung

Die sichere Verbindung interagiert mit dem obligatorischen Feature Connection, welches die Verbindungen zwischen Chat-Clients aufbaut und verwaltet. Bei der sicheren Verbindung besteht somit kein Optional-Featureproblem, da das Feature Connection in jeder Variante der SPL enthalten ist.

Eine Untersuchung der Interaktionen dieser beiden Feature soll einen Vergleich der Lösungsstrategien gegenüber dieser Art von Featureinteraktionen ermöglichen. Um einen theoretischen Ansatz für eine spezielle Art des Optional-Featureproblems zu haben, wird davon ausgegangen, dass zwei Implementierungen des Features Connection vorliegen. Ein Austausch der Implementierung eines obligatorischen Features ist im Vergleich zur Erstellung von Varianten der SPL ein einmaliger Vorgang. Die Besonderheit des Austausches ist die Tatsache, dass die alternative Implementierung des Features korrekt mit dem optionalen Feature interagieren muss.

Designanpassung: Eine Designanpassung wird notwendig wenn die alternative Implementierung des obligatorischen Features die Interaktionen des vorhandenen optionalen Features nicht unterstützt. In dem betrachteten Problemfall eliminiert die Alternative das optionale Feature vollständig aus der SPL.

Quellcodeaustausch: Bei dieser Lösungsstrategie wird die Ursache des Problems gleichzeitig zum Lösungsmechanismus. Die Idee behält beide Implementierungen der Connection bei und legt für die Varianten der SPL fest, welche sie verwenden. In Varianten, die das optionale Feature enthalten, wird somit auf die ursprüngliche Implementierung des Features Connection zurückgegriffen. Für alle anderen Varianten kann die alternative Implementierung genutzt werden.

Bedingte Kompilierung: Präprozessoranweisungen stellen für diesen Problemfall eine bekannte Methode zur Implementierung von Featuren dar. Mit Hilfe der bedingten Kompilierung muss das optionale Feature in die alternative Implementierung der Connection integriert werden.

Bei der Verwendung von bedingter Kompilierung muss somit jedes interagierende optionale Feature zusätzlich alternativ implementiert werden.

Auslagern der Interaktion: Zur Untersuchung dieser Lösungsstrategie muss davon ausgegangen werden, dass das optionale Feature mit Hilfe von AOP umgesetzt wurde. Die ursprüngliche Implementierung der Connection wird durch das optionale Feature so modifiziert, das sichere Verbindungen aufgebaut werden. Die alternative Implementierung muss in ihrem funktionalem Aufbau an den Stellen ihrem Vorgänger gleichen, die vom Aspekt betroffen sind.

Im betrachteten Problemfall waren nur geringfügige Anpassungen an dem Quellcode des optionalen Features notwendig. Die Komplexität dieser Anpassungen hängen von den technischen Unterschieden der alternativen Implementierungen ab.

Optionales Weben: Die Übereinstimmung des Aufbaus der alternativen Umsetzungen ist für eine Lösung der Interaktion mit Hilfe des optionalen Webens eine Voraussetzung. Wenn der Quellcode des optionalen Features keine technischen Anpassungen an die alternative Implementierung des anderen Features benötigt, ermöglicht ein identischer Aufbau der vom Aspekt betroffenen Teile die weitere Nutzung der Implementierung des optionalen Features.

Kapselung: Im betrachteten Fall wurde die technische Grundlage über die Chat-Clients kommunizieren durch eine alternative Implementierung verändert. Die Art der Datenübertragung ist unabhängig vom optionalen Feature. Zwischen dem Quellcode der reinen Datenübertragung und dem optionalen Feature bestehen somit keine Interaktionen. Eine Abstraktion der technischen Übertragung in ein eigenständiges Feature vereinfacht diesen Problemfall. Das neu gebildete Feature besitzt in der SPL keine Interaktionen mit optionalen Featuren.

3.5.8 Logging / Statistiken

Die Aufgabe des Loggings ist die Protokollierung des technischen Ablaufs einer Anwendung. In der betrachteten Produktlinie interagiert dieses Feature mit obligatorischen und optionalen Featuren. Ein Loggingfeature stellt ein klassisches Vorkommen eines querschneidenen Belanges dar.

Charakterisierend für die Interaktionen eines querschneidenen Belanges ist ihre eindeutige Richtung. Im Falle des Loggings benutzt jedes betroffene Feature ein vom Logging bereitgestelltes Objekt, um seine Fortschritte zu vermelden. Welche Fortschritte zur Protokollierung relevant sind ergibt sich aus der Spezifikation der Produktlinie.

Der Problemfall der Statistiken unterscheidet sich nur geringfügig vom Logging. Ausschließlich die jeweils integrierte Funktionalität fällt bei dem Problemfall der Statistiken komplexer aus. Die Bedingungen zur Lösung der Problemfälle sind identisch und werden daher nur für den Problemfall des Loggings erläutert.

Designanpassung: Die Anpassung des Designs verknüpft alle beteiligten optionalen Feature mit dem Feature Logging. Die Auswahl eines mit Loggingmechanismen versehenen optionalen Features erfordert aufgrund der Zugriffsabhängigkeit die gleichzeitige Auswahl des Logging Features. Die optionalen Feature der SPL, welche mit Loggingmechanismen implementiert wurden, stehen durch die Designanpassung ausschließlich mit dieser Funktionalität zur Auswahl. Eine Verwendung eines Features des Rechtemanagement ohne seine zugehörigen Ausgaben an das Logging ist bei dieser Lösung nicht möglich.

Zusätzlich erweitert die bedingte Auswahl des Features Logging die obligatorischen Feature, um ihre zugehörigen Loggingmechanismen.

Quellcodeaustausch: Zur Umsetzung des Quellcodeaustausches bestand die Notwendigkeit alle betroffenen optionalen Feature doppelt zu implementieren. Die zum Austausch vorgesehenen Replikate unterscheiden sich in Bezug auf die Loggingmechanismen. Eine Implementierung mit der zusätzlichen Funktionalität und eine ohne diese abhängigen Funktionen werden für diese Lösung benötigt. Die Auswahl des Features Logging entscheidet bei der Erstellung einer Variante welche der Implementierungen der optionalen Feature erforderlich sind.

Zusätzlich zu der beschriebenen festgelegten Entscheidung für alle betroffenen optionalen Feature kann die Auswahl der Implementierungen im Auswahlprozess einer Variante erfolgen. Die unterschiedlichen Implementierungen der einzelnen optionalen Feature werden somit als eigenständige Feature aufgefasst. Die Zahl der Varianten der SPL wird durch diese Veränderungen im Design erhöht. Der Nutzen der Möglichkeit für einzelne optionale Feature unabhängig zu entscheiden, ob diese Loggingmechanismen enthalten, kann durch eine erneute Analyse der Anforderungen untersucht werden.

Bedingte Kompilierung: Die Besonderheit bei der Implementierung von Loggingmechanismen ist, dass diese zusätzliche Funktionalität keine Veränderungen an der Funktionsweise der erweiterten Feature erfordert und nicht erfordern darf. Die Funktionen und Quellcodeabschnitte, die erweitert werden müssen, definieren sich durch den logischen Ablauf des jeweils betroffenen Features.

Die Positionen im Quellcode eines Features, an denen die benötigten Präprozessoranweisungen eingefügt werden müssen, sind genau spezifiziert. Die Daten, welche an diesen jeweiligen Positionen an das Logging übermittelt werden müssen, ergeben sich ebenfalls aus der Spezifikation der Produktlinie.

Die Umsetzung des Loggings in optionalen Featuren mit Hilfe von bedingter Kompilierung entspricht somit einer klar definierten Aufgabe.

Auslagern der Interaktion: Die klar definierten Anforderungen an die Loggingfunktionalität jedes Features erleichtert die Auslagerung der Interaktion. Um die Anforderungen eindeutig zu erfüllen muss für jede festgelegte Position eine zutreffende Deklaration erzeugt werden. Die genaue Definition der erforderlichen Daten verhindert eine Zusammenfassung der Quellcodeabschnitte eines oder mehrerer Feature. Für jede Position ist somit die Implementierung einer vollständigen Anweisung (Advice) zu ihrer Erweiterung notwendig.

Die Möglichkeiten der Werkzeuge zur Umsetzung von AOP bieten dynamische Strukturen zum Zugriff auf Eigenschaften der erweiterten Quellcodeabschnitte. Der Zugriff auf derartige Strukturen ermöglicht eine erleichterte Implementierung von Loggingmechanismen mit Hilfe von AOP. Die Ausgaben des Loggings beruhen bei dieser Implementierung des Loggings ausschließlich auf dynamischen Daten. Aus der definierten Ausgabe das eine Verbindung erfolgreich aufgebaut wurde, wird zu der Ausgabe das die Funktion connect der Klasse CConnection mit dem Rückgabewert true endete.

Die beschriebene Abstraktion der einzelnen Loggingmechanismen ermöglicht eine weitere Möglichkeit die Auslagerung umzusetzen. Bisher ist der Vorgang der Implementierung der Funktionalität getrennt von der Implementierung der zugehörigen Loggingmechanismen erfolgt. Wenn die Art der Ausgaben mit Hilfe der vorgestellten Abstraktion für alle Stellen vereinheitlicht werden kann, verändert sich die Implementierung des Loggings.

Zur Umsetzung müssen lediglich die betroffenen Stellen deklariert werden. Eine weitere Besonderheit der AOP ist es das Funktionen im Quellcode anhand ihrer Bezeichnung durch reguläre Ausdrücke deklariert werden können. Das Listing 3.1 zeigt die Deklaration eines allgemeinen Pointcuts der alle Funktionen auszeichnet, welche auf log enden. Ausgehend von dieser Deklaration können die für das Logging erforderlichen Daten ermittelt werden.

Listing 3.1: LoggingAspect.ah

```
1  aspect LoggingAspect
2  {
3    //alle Memberfunktionen aller Klassen die auf log enden
4    pointcut logs() = execution("% %::%(%log)");
5
6    advice logs() : before()
7    {
8        // loggingausgaben
9        ...
10   }
11   // weitere Adviceblöcke
12   ...
13 };
```

Der Vorteil dieser Lösung ist, dass die Zuordnung der der Loggingmechansimen während der Umsetzung der Funktionalität der Feature erfolgen kann. Die Abbildung der Abhängigkeit eines Features vom Feature Logging über die Bezeichnung seiner Funktionen erleichtert zusätzlich die Entwicklung weiterer Feature für die SPL. Die Implementierung eines Features mit Loggingmechanismen benötigt keine Anpassungen des Features Logging und lediglich die Beachtung einer Namenskonvention für die Bezeichnung von

Funktionen.

Die vorgestellte Lösung ist bei Einhaltung der Namenskonvention für alle Feature der SPL nutzbar. Das erstellte Derivate des Logging erweitert optionale und obligatorische Feature.

Optionales Weben: Das vorgestellte dem Logging zugeordnete Derivat unterstützt bereits die Idee des optionalen Webens.

Kapselung: Zur Umsetzung von querschneidenen Belangen bieten die Konzepte der OOP keine geeigneten Mittel.

Kapitel 4

Auswertung

Dieses Kapitel behandelt die Auswertung der untersuchten Problemstellung. Die erkannten Problemfälle werden anhand unterschiedlicher Kriterien untersucht und verglichen. Das Ziel dieser Aufarbeitung der Problemfälle ist eine Klassifizierung von Optional-Featureproblemen in eindeutige Typen.

Aus der Auswertung der Eignung der umgesetzten Lösungsstrategien für die gebildeten Klassen ergibt sich der Nutzen der aufgestellten Klassifizierung.

Ziel dieses Abschnittes ist eine Richtlinie die den aufgestellten Problemklassen die besten Lösungsansätze zuordnet. Diese Richtlinie kann den gesamten Entwicklungsprozess unterstützen. In der Designphase können bekannte Interaktionen auf andere Weise modelliert werden um die verfügbaren Lösungen ausnutzen zu können. Während der Implementierung kann die beste Lösung für die auftretenden Probleme gewählt werden.

4.1 Klassifizierung

In diesem Abschnitt werden die gefundenen Probleme in geeignete Klassen eingeteilt. Dazu werden mögliche Kriterien zur Klassifizierung erläutert, ausgewertet und verglichen.

Eine Klassifizierung ist die Einteilung von Objekten in eindeutig abgegrenzte Klassen. Um diese Einteilung durchzuführen müssen Kriterien gefunden werden, die eine abstrakte Abgrenzung der betrachteten Objekte in allgemeine Klassen ermöglichen. Im Rahmen dieser Arbeit erfolgt die Klassifizierung beispielhafter Problemfälle in allgemeingültige Klassen, um diese weitergehend zu untersuchen. Ziel dieser Abstraktion ist es, die Ergebnisse dieser Arbeit einfach auf andere Projekte übertragen zu können.

4.1.1 Kriterien

In diesem Abschnitt werden mögliche Kriterien erläutert und anhand der Problemfälle ausgewertet. Anschließend erfolgt eine Bewertung der vorgestellten Kriterien und die Auswahl der besten Kriterien für die Klassifizierung der Probleme.

Zeitpunkt der Interaktion

Ein Unterscheidungskriterium der Problemfälle stellt der Zeitpunkt bzw. die Zeitpunkte der Interaktionen dar. Dies betrifft die Zeitpunkte im Softwarelebenszyklus bei denen die Interaktion auftritt bzw. erkannt wird.

- Designphase:
  Der frühest mögliche Zeitpunkt der Interaktion tritt in der Designphase auf. Bei der Erstellung des Featurebaums kann in diesen Fällen bereits durch den logischen Zusammenhang eine Interaktion erkannt werden.

- Entwicklungsphase:
  In der Entwicklungsphase zeigen sich Interaktionen auf zwei unterschiedlichen Wegen. Kompilerfehler in bestimmten Varianten der SPL treten durch direkte Abhängigkeiten gegenüber optionalen Featuren auf und Abhängigkeitsverletzungen zeigen sich wenn im Ablauf benötigte Methoden Teile der optionalen Feature sind.

- Laufzeit:
  Der späteste Zeitpunkt der Interaktion ist während der Laufzeit der Anwendung. Ursachen für Interaktionen zu diesem Zeitpunkt können nicht initialisierte Objekte oder Variablen, sich widersprechende Funktionalitäten oder andere vom Ablauf abhängige Konstrukte sein.

	Userinterface	Betriebssystemabstraktion	Authentifikation	Rechtemanagement	Logging	Sichere Verbindung	Accountmanagement	Nachrichtenerweiterungen	Statistiken
	P1	P2	P3	P4	P5	P6	P7	P8	P9
Designphase					x		x	x	x
Entwicklungsphase	x	x				x			
Laufzeit	x		x	x					

Tabelle 4.1: Zeitpunkt der Interaktion

In der Tabelle 4.1 werden die beschriebenen Zeitpunkte mit den Problemfällen in Beziehung gesetzt. Doppelte Zuordnungen zeigen, dass das Kriterium des Zeitpunktes der Interaktion nicht eindeutig für jeden Problemfall zu ermitteln ist.

Eine Erkennung von Feature-Interaktionen in der Designphase der Entwicklung beruht auf erforderlichen oder verlangten logischen Zusammenhängen der Feature. Der Problemfall der Nachrichtenerweiterungen entsteht aus der Anforderung das der Nachrichtenverlauf ein dem Nachrichtenfeature identisches Verhalten aufweisen soll. Aus dieser

Anforderung geht hervor, dass die optionalen Feature, die das Verhalten einer Nachrichtensitzung verändern zusätzlich das optionale Feature des Nachrichtenverlaufs erweitern müssen.

In der Entwicklungsphase können bestimmte Feature-Interaktionen durch die Art der Umsetzung einzelner Anforderungen hervorgerufen oder vermieden werden. Die Verwendung eines geeigneten Frameworks kann beispielsweise die Abstraktion auf unterschiedliche Betriebssysteme realisieren.

Feature-Interaktionen die erst zur Laufzeit einer betroffenen Variante der SPL erkannt werden, haben dynamische Abhängigkeiten zwischen den Featuren als Ursache. Die für den Nutzer nicht erkennbare Überlagerung der Auswirkungen der Feature des Rechtemanagement sind ein Beispiel für diese dynamischen Abhängigkeiten.

Richtung der Interaktion

Die Richtung der Interaktion beschreibt die Abhängigkeit zwischen den an der Interaktion beteiligten Featuren. Dieses Kriterium ist abhängig von der ausgewählten Lösungsstrategie. Die unterschiedlichen Lösungsstrategien können eine Veränderung der Richtung der Interaktion ermöglichen oder benötigen. Die Betrachtung der Richtung erfolgt aus der Sicht des optionalen Features in dem die Interaktion auftritt. Treten in allen beteiligten optionalen Featuren eines Problemfalls Interaktionen auf, ergibt die Betrachtung der Richtung aus jeder Sicht die identische Zuordnung des Problemfalls.

Die Richtung der Interaktion gliedert sich in vier verschiedene Ausprägungen. Die folgende Aufzählung erläutert ihre Unterschiede. Die Auswertung des beschriebenen Kriteriums ist in Tabelle 4.2 dargestellt.

- Kein funktioneller Zusammenhang:
  - Benutztrelation (==>)
    Ein optionales Feature erweitert andere Feature, um seine eigene Funktionalität umzusetzen. Eine Veränderung der Ausgangsfunktionalität der betroffenen Feature erfolgt dabei nicht.
  - Wird-Benutztrelation (<==)
    Ein optionales Feature benötigt die Erweiterung durch andere Feature für seine korrekte Funktionsweise. Die Interaktion geht meist von alternativ vorhandenen Featuren aus. Die Art und Weise der Zusammenarbeit der interagierenden Feature kann sich von Alternative zu Alternative unterscheiden.
- Funktionelle Abhängigkeit:
  - Integration (+=>)
    Die Funktionalität eines optionalen Features ergänzt ein anderes Feature. Die Notwendigkeit der Kombination der Funktionalität der betroffenen Feature ergibt sich aus den Anforderungen an die Logik der Feature.
  - Reaktion (<=+)
    Ein korrektes Verhalten der Anwendung wird durch die Reaktion eines optionalen Features auf andere Feature sichergestellt.

	Userinterface P1	Betriebssystemabstr. P2	Authentifikation P3	Rechtemanagement P4	Logging P5	Sichere Verbindung P6	Accountmanagement P7	Nachrichtenerweiterungen P8	Statistiken P9
==>					x				x
<==	x	x							
+=>						x	x	x	
<=+	x		x	x					

Tabelle 4.2: Richtung der Interaktion

Die querschneidenen Belange Logging und Statistiken sind die Vertreter einer Benutztrelation in der betrachteten SPL. Ihre Funktionalität muss direkt in alle dafür vorgesehenen Feature integriert werden.

Alle Feature die eine Interaktion mit dem Nutzer der Anwendung erfordern, müssen die bereitgestellte Funktionalität der Feature, die eine Benutzerschnittstelle umsetzen, verwenden. Der Problemfall des Userinterface stellt damit einen Vertreter für die Wird-Benutztrelation dar.

Die Integration optionaler Feature in andere optionale Feature erzeugt die Kombination der Funktionalität der betroffenen Feature. Die von der Auswahl der Feature bestimmten unterschiedlichen Ausprägungen des Nachrichtenverlaufs entsprechen den Auswirkungen mehrerer Integrationen des Problemfalls Nachrichtenerweiterungen.

Die Reaktion betrifft die Umsetzung der Koordination der überlagernden Funktionalität optionaler Feature. Der Problemfall der Authentifikation hat gezeigt, dass zur Auflösung dieser Interaktionen nicht in jedem Fall eine Implementierung von zusätzlicher Logik notwendig ist.

Design der SPL

Die konzeptionelle Darstellung der Feature im FD enthält Informationen über die Art der betroffenen Feature und ihre Beziehungen untereinander. Beispielsweise kann zwischen erforderlichen und optionalen Featuren unterschieden werden und eine zusätzliche Einteilung in die Beziehungen der Feature ist möglich.

Die Art der Beziehung, die Feature in einem FD zueinander besitzen können, bilden ein Kriterium zur Bewertung der Problemfälle. Ausschlaggebend für die Bewertung dieses Kriteriums ist entweder die besondere Beziehung einer der interagierenden Seiten des Problemfalls oder die Beziehung aller betroffenen Feature. Das Auftreten einer Alternative- oder Oder-Beziehung wird bei der Auswertung zu einer Kategorie zusammengefasst. Bei einem Abstand der Feature über mehrere Ebenen des FDs besitzen die Feature keine direkte Beziehung.

	Userinterface	Betriebssystemabstr.	Authentifikation	Rechtemanagement	Logging	Sichere Verbindung	Accountmanagement	Nachrichtenerweiterungen	Statistiken
	P1	P2	P3	P4	P5	P6	P7	P8	P9
Alternative/Oder-Bez.	x	x				x			
Und-Beziehung				x			x	x	
keine direkte Beziehung			x		x				x

Tabelle 4.3: Beziehung der Feature

Die in Tabelle 4.3 ausgewerteten Beziehungen der beteiligten optionalen Feature der Problemfälle ergeben sich direkt aus dem FD der Produktlinie.

Eine weitere Möglichkeit Optional-Featureprobleme zu unterscheiden, bietet die Anzahl der an einem Problemfall beteiligten Feature. Entscheidend ist die Anzahl der Feature jeder der zwei interagierenden Seiten. Im einfachsten Fall eines Optional-Featureproblems besteht die Interaktion zwischen zwei Featuren *(1:1)*. Ein komplexerer Fall ergibt sich, wenn eine Seite der Interaktion aus mehreren betroffenen Featuren *(1:n)* besteht. Der häufigste Fall in der untersuchten SPL beschreibt Problemfälle, die auf beiden Seiten der Interaktion eine Menge von Featuren *(n:n)* besitzen.

	Userinterface	Betriebssystemabstr.	Authentifikation	Rechtemanagement	Logging	Sichere Verbindung	Accountmanagement	Nachrichtenerweiterungen	Statistiken
	P1	P2	P3	P4	P5	P6	P7	P8	P9
1 : 1			x	x					
1 : n		x			x	x		x	x
n : n	x						x		

Tabelle 4.4: Anzahl der Feature

Zur Auswertung des Kriteriums der Anzahl (vgl. Tabelle 4.4) wurden jeweils die Varianten der Problemfälle betrachtet in denen die maximale Anzahl der beteiligten Feature ausgewählt ist. Eine Interpretationsfreiheit bei der Auswertung entsteht bei der Zuordnung der beteiligten Featuren zu den Seiten einer Interaktion.

Die Art eines Features ist eine weitere Information die aus dem Design der SPL

abgelesen werden kann. Für die Unterscheidung von Feature-Interaktionen stellen die Arten der von der Interaktion betroffenen Feature ein geeignetes Kriterium dar. Im besonderen Fall der Untersuchung von Optional-Featureproblemen ergeben sich zwei Kategorien. Von der Interaktion sind ausschließlich optionale Feature betroffen oder eine Kombination von optionalen und obligatorischen Featuren. In Tabelle 4.5 sind die Arten der beteiligten Feature der Problemfälle ausgewertet.

	Userinterface P1	Betriebssystemabstr. P2	Authentifikation P3	Rechtemanagement P4	Logging P5	Sichere Verbindung P6	Accountmanagement P7	Nachrichtenerweiterungen P8	Statistiken P9
gemischte F.	x	x			x	x			x
Nur opt. F.			x	x			x	x	

Tabelle 4.5: Art der Feature

Zusätzlich zu den Informationen über die einzelnen Feature aus dem FD ist eine genau Kenntnis der Interaktionen der Problemfälle eine Voraussetzung zur Auswertung dieses Kriteriums. Das Wissen über die Zugehörigkeit eines Features zu einem bestimmten Problemfall ist entscheidend für die korrekte Einteilung der Problemfälle.

4.1.2 Klassen

Aus der Analyse und Auswertung der Kriterien zur Klassifizierung der gefundenen Problemfälle wurden in diesem Abschnitt geeignete Klassen gebildet. Die folgende Übersicht über die erarbeiteten Klassen von Optional-Featureproblemen berücksichtigt nur die im vorherigen Abschnitt betrachteten Kriterien. Die Klassen bilden den Ausgangspunkt für die weiteren Betrachtungen dieser Arbeit.

Die Abgrenzung der einzelnen Klassen sollte möglichst eindeutig sein und eine einfache Zuordnung der Problemfälle ermöglichen. Die im vorherigen Abschnitt betrachteten Kriterien ergeben eine logische Aufteilung der untersuchten Problemfälle in vier unterschiedliche Klassen. Die Zuordnung der einzelnen Problemfälle zu einer Klasse ist bis auf einige Ausnahmen eindeutig durch die Auswertung der betrachteten Kriterien möglich. Tabelle 4.6 zeigt die Gesamtübersicht der Ergebnisse der Untersuchung der Probleme. Im Gegensatz zur Auswertung der Kriterien (vgl Tabelle 4.1 - 4.5) wurden die Reihenfolgen der Problemfälle und der Kategorien verändert. Die Anordnung der Problemfälle entspricht ihrer Zugehörigkeit zu gemeinsamen Klassen. Die veränderte Reihenfolge der unterschiedlichen Kategorien der Kriterien verdeutlicht die gemeinsamen Merkmale der Problemfälle.

Das Kriterium der Anzahl der beteiligten Feature (vgl. Tabelle 4.4) wird in Tabelle 4.6 nicht aufgeführt. Die Ergebnisse der Auswertung dieses Kriteriums ergeben eine

kontroverse Einteilung der Problemfälle gegenüber der anderen Kriterien.

	Userinterface P1	Betriebssystemabstr. P2	Sichere Verbindung P6	Logging P5	Statistiken P9	Accountmanagement P7	Nachrichtenerweiterungen P8	Authentifikation P3	Rechtemanagement P4
Alternative/Oder-Bez.	x	x	x						
Entwicklungsphase	x	x	x						
<==	x	x							
gemischte Feature	x	x	x	x	x				
keine direkte Beziehung				x	x			x	
==>				x	x				
Designphase				x	x	x	x		
Und-Beziehung						x	x		x
+=>			x			x	x		
Nur opt. Feature						x	x	x	x
<=+	x							x	x
Laufzeit	x							x	x

Tabelle 4.6: Gesamtübersicht der Kriterien

Die aus der Auswertung der Kriterien entstandenen Klassen werden in den folgenden Abschnitten im Einzelnen genauer betrachtet.

Klasse 1: Alternativen

Die Klasse der *Alternativen* umfasst alle Optional-Featureprobleme bei denen die Feature-Interaktion durch alternative Vorkommen von Featuren verursacht werden. Alternative Feature ergeben sich aus Anforderungen die in Alternativen- oder Oder-Beziehungen umgesetzt werden. Eine geforderte Funktionalität der SPL bietet in diesen Fällen mehrere Möglichkeiten der Umsetzung oder für die Umsetzung einer Anforderung existieren alternative Ansätze.

Die Beziehungen der Feature werden in der Designphase der Entwicklung einer SPL festgelegt. Für die Zugehörigkeit von alternativen Featuren zu einem Optional-Featureproblemmuss zusätzlich eine Interaktion zwischen den Alternativen und anderen optionalen Featuren auftreten. Diese Interaktionen ergeben sich aus den Anforderungen an die einzelnen Feature der SPL. Das tatsächliche Auftreten von Feature-Interaktionen ergibt sich bei Problemfällen dieser Klasse aus der Implementierung der einzelnen Feature. Die Erkennung eines vorhandenen Optional-Featureproblems liegt somit in der Entwicklungsphase der SPL.

Der Ursprung der Interaktionen der Optional-Featureprobleme dieser Klasse zeichnet sich dadurch aus, dass andere Feature die Funktionalität der alternativ vertretenen Feature benutzen. Eine Veränderung dieser Funktionalität durch die Auswahl eines alternativen Features benötigt somit die Anpassung des Zugriffs der anderen beteiligten Feature.

In der untersuchten Beispiel-Produktlinie befinden sich drei Vertreter, welche die Kriterien dieser Klasse erfüllen (vgl. Tabelle 4.6). Das Userinterface und die Betriebssystemabstraktion bilden vollständig die beschriebenen Kriterien ab. Der Problemfall der sicheren Verbindung weist eine abweichende funktionale Abhängigkeit gegenüber der anderen Vertreter der Klasse auf. Die Zuordnung des Problemfalls ist trotzdem eindeutig da drei der vier Entscheidungskriterien zutreffen.

Klasse 2: Globale Anforderungen

Die zweite Klasse von Optional-Featureproblemen ergibt sich aus Anforderungen, die eine bestimmte Funktionalität für mehrere Feature der Anwendung erfordern. Ein Optional-Featureproblembesteht dann wenn diese Anforderung optionalen Charakter aufweist und unter den betroffenen Featuren ebenfalls optionale Feature vorkommen. Ziel der Anforderung sind somit optionale Feature und können zusätzlich obligatorische Feature sein. Eine Zuordnung des Features der „globalen“ Anforderung und der betroffenen Feature ist aus dem Design der SPL meist nicht zu erkennen.

Zur Umsetzung der übergreifenden Anforderung müssen die betroffenen Feature, um die Funktionalität der Anforderung erweitert werden. Diese Erweiterungen stellen eine Benutztrelation dar. Die ursprüngliche Funktionalität der betroffenen Feature wird durch die vollzogenen Veränderungen nicht beeinflusst.

Die Erkennung eines bestehenden Optional-Featureproblems und dessen Zuordnung zu dieser Klasse ergibt sich aus der genauen Spezifikation der globalen Anforderung. Eine Einteilung von Problemfällen zu dieser Klasse ist daher bereits in der Designphase der Produktlinie möglich.

Die Feature Logging und Statistiken ergeben die zwei in der Chat-SPL untersuchten Problemfälle dieser Klasse. Beide Problemfälle erfüllen eindeutig die Kriterien der Klasse (vgl. Tabelle 4.6).

Klasse 3: Gleichrangige Feature

Die Problemfälle der Klasse der *gleichrangigen Feature* enthalten optionale Feature die auf einer Ebene des FDs in Beziehung stehen. Die Besonderheit der Feature besteht darin, dass sie gleichrangig in der SPL auftreten. Das bedeutetet das für ihre Auswahl die gleichen Bedingungen gelten und sie ähnliche Auswirkungen auf das Verhalten der SPL nach sich ziehen. Dem entsprechende Übereinstimmungen sind in den Spezifikationen der Anforderungen der Feature zu finden.

Die untersuchten Problemfälle wiesen eine Und-Beziehung zwischen ihren beteiligten Featuren auf. Die Interaktionen dieser Feature ergibt sich aus der Anforderung, das mindestens eines der optionalen Feature um zusätzlich um die Funktionalität der anderen beteiligten optionalen Feature erweitert werden muss. Die Richtung der Interaktion entspricht somit einer Integration optionaler Funktionalität in ein anderes optionales

Feature. Das hauptsächliche Verhalten der Feature ist die Erweiterung der von ihnen repräsentierten Funktionalität in ein anderes Feature. Bei den untersuchten Problemfällen war jeweils der Elternknoten der Feature das Ziel dieser Integration.

Ein weiterer Grund für die Bezeichnung der Klasse ist die Art der beteiligten Feature der Problemfälle. Eine weitere Gemeinsamkeit der Feature ergibt sich aus der Tatsache das alle Feature optionale Komponenten der SPL sind.

Die Notwendigkeit zur zusätzlichen Integration der Funktionalität eines optionalen Features in ein anderes optionales Feature geht aus den Anforderungen der betroffenen Feature hervor. Das Vorkommen eines Optional-Featureproblems dieser Klasse kann somit bereits in der Designphase der Produktlinie erkannt werden.

Die Chat-SPL enthält zwei Beispiele für Problemfälle dieser Klasse. Die Nachrichtenerweiterungen erweitern zusätzlich zu der allgemeinen Nachrichtensitzung das optionale Feature des Nachrichtenverlaufs. Der Problemfall des Accountmanegements weist eine Vielzahl von gegenseitigen Integrationen der Teilfunktionalitäten seiner Feature auf.

Klasse 4: Dynamische Interaktionen

Die Klasse der *dynamischen Interaktionen* unterscheidet sich von bisher betrachteten Klassen. Die Interaktionen der Optional-Featureprobleme dieser Klasse treten durch die dynamische Zusammenarbeit oder Überlagerung ihrer Feature auf. Dynamisch bedeutet in diesem Zusammenhang, dass die Auswirkungen der Funktionalität der beteiligten Feature während der Ausführung der Anwendung Wechselwirkungen aufzeigen. Zur Lösung dieser Überschneidungen sind gegenseitige Reaktionen im Ablauf der wechselwirkenden Funktionalitäten erforderlich.

Die Problematik eines Optional-Featureproblems entsteht, weil die Notwendigkeit dieser Reaktionen von der Auswahl von optionalen Featuren der SPL abhängig ist. Eine zusätzliche Beteiligung eines obligatorischen Features trat bei den in der Chat-SPL untersuchten Problemfällen dieser Klasse nicht auf.

Das Auftreten der Wechselwirkungen von optionalen Featuren findet, in Varianten in denen mehr als ein Feature eines Problemfalls dieser Klasse ausgewählt ist, statt. Die Erkennung der Optional-Featureprobleme erfolgt zur Laufzeit der Anwendung in speziellen Varianten der SPL, wenn eine betroffene Abfolge im Ablauf der Anwendung ausgeführt wird. Das Auffinden von Feature-Interaktionen die Problemfällen dieser Klasse angehören, erfolgt daher meist erst in der Testphase eines speziellen Produkts der Produktlinie.

Eine Besonderheit der Optional-Featureproblem dieser Klasse ist, dass sie bei der Auswahl und nicht bei dem Fehlen eines optionalen Features auftreten.

Die zwei untersuchten Problemfälle der Chat-SPL weisen unterschiedliche Beziehungen unter ihren beteiligten Featuren auf. Das Kriterium der Beziehung (vgl. Tabelle 4.6) erlaubt keine Aussage über die Zugehörigkeit eines Problemfalls zu dieser Klasse.

Die Problemfälle Authentifikation und Rechtemanegment sind die Vertreter für die Klasse der *dynamischen Interaktionen*.

4.2 Bewertung der Lösungsstrategien

Ausgehend von der Klassifizierung der Problemfälle werden in diesem Abschnitt die vorgestellten und untersuchten Lösungsstrategien für Optional-Featureprobleme bewertet. Interessant für die Erstellung einer Richtlinie zur Unterstützung der Entwicklung von SPLs in Bezug auf das Optional-Featureproblemist die Eignung einer bestimmten Lösungsstrategie für die Problemfälle einer Klasse.

Zur einheitlichen Bewertung der Lösungsstrategien werden in diesem Abschnitt Berwertungskriterien aufgestellt und erläutert. Diese Kriterien werden einzeln für jede Lösungsstrategie in Bezug auf alle vier Klassen ausgewertet.

Das Ergebnis der Bewertung der Lösungsstrategien dient im folgenden Verlauf dieser Arbeit zur Erstellung der Richtlinie zur Lösung von Optional-Featureproblemen. Die detaillierten Ergebnisse dieses Abschnittes verdeutlichen zusätzlich die Eignung der aufgestellten Klassen von Optional-Featureproblemen.

4.2.1 Bewertungskriterien

Die folgenden Bewertungskriterien sollen die Vor- und Nachteile beim Einsatz der Lösungen verdeutlichen. Im Allgemeinen wird bei der Bewertung davon ausgegangen, dass das Ergebnis der Lösungsstrategie zu einer korrekten und in allen Varianten lauffähigen Version der SPL führt. Ausnahmen dieser Annahme werden extra in der Bewertung der einzelnen Strategien betrachtet.

Überschneidung einiger Kriterien (aufwand, lesbarkeit)

Variabilität: Eines der entscheidendsten Kriterien für die Bewertung einer Lösung ist die Beeinflussung der Variabilität der SPL, die aus ihrer Umsetzung resultiert. Ein Erhalt der geforderten Anzahl an Varianten, die aus der SPL erzeugt werden können, ist das normale Ziel bei der Auflösung der Feature-Interaktionen.

Bei der Bewertung der Variabilität werden folgende Abstufungen unterschieden:

- ++Erhöhung der möglichen Varianten
  Die Umsetzung der Lösungsstrategie erzeugt zusätzliche Modularität und ermöglicht die Erstellung von Varianten die vorher kein Teil der SPL waren. Die Vorteile der zusätzlich gewonnen Varianten können nur durch eine angepasste Anforderungsanalyse ermittelt werden.

- +Erhalt der Varianten
  Die Anzahl der Varianten der SPL werden durch die jeweilige Lösungsstrategie nicht beeinflusst.

- −Eliminierung einer oder weniger Varianten
  Das Ergebnis der Lösungsstrategie ist der Wegfall einiger Varianten des Problemfalls. Die Variabilität der interagierenden Feature wird nicht vollständig entfernt.

- −−Festlegung auf eine Variante
  Die Umsetzung der Lösungsstrategie eliminiert die Varianten eines Problemfalls vollständig und legt eine bestimmte Konfiguration fest.

Implementierungsaufwand: Zusätzlich zu der Umsetzung der Feature eines Problemfalls erfordern die notwendigen Maßnahmen einiger Lösungsstrategien weiteren Aufwand. Die Bewertung des benötigten Aufwands ergibt sich aus der Komplexität und der Anzahl der für die jeweilige Lösungsstrategie durchgeführten Veränderungen. Die Qualität der verfügbaren Werkzeuge um einzelne Lösungen umzusetzen hat keinen Einfluss auf diese Bewertung, da sich die Werkzeuge auf sehr unterschiedlichen Entwicklungsständen befinden.

- ++Kein zusätzlicher Aufwand
  Die Umsetzung der Lösungsstrategie erfolgt ohne eine Veränderung an der Implementierung der SPL.
- +Geringer zusätzlicher Aufwand
  Die Implementierungen der betroffenen Feature müssen nur geringfügig oder in einer einfachen Art modifiziert werden.
- −Erhöhter Aufwand
  Große Teile der Implementierungen der betroffenen Feature müssen verändert werden.
- −−Stark erhöhter Aufwand
  Der Aufwand der Umsetzung der Lösungsstrategie übersteigt den Aufwand für die Implementierung der betroffenen Feature.

Modularität: Das Kriterium der Modularität bezieht sich auf den Aufbau des Quellcodes eines Problemfalls. Das Auftreten von Codereplikationen und das Umgehen von Prinzipien der Kapselung und der Vererbung erzeugen zahlreiche Nachteile. Die Wartbarkeit und das Verständnis des Quellcodes stehen in einem direkten Zusammenhang mit dem betrachteten Kriterium der Modularität.

- ++Kein Einfluss
  Die vorhandene Trennung in Module bleibt bestehen. Die Veränderungen der Lösung beachten die bestehenden Modulgrenzen und führen Modifikationen nur innerhalb dieser Grenzen aus.
- +Zusätzliche Module
  Zur Einhaltung der Modularität müssen zur Umsetzung der Lösung zusätzliche Module erzeugt werden. Diese Module entsprechen der Kombination der Funktionalität mehrerer Feature.
- −xxx
  Die Umsetzung der Lösungsstrategie verstößt gegen die bestehende Trennung in Module. Teile der Lösung implementieren Funktionalität in Modulen, welche keinen Bezug zu dieser Funktionalität besitzen.
- −−Replikation
  Bestimmte Teile oder die Gesamte Implementierung der betroffenen Feature werden für Varianten der SPL dupliziert. Die Veränderungen in den zusätzlichen Implementierungen betreffen nur einige Teile des Quellcodes, und erzeugt somit Replikationen des unveränderten Quellcodes.

Erweiterbarkeit: Das Kriterium der Erweiterbarkeit bewertet den benötigten Aufwand, um neue Feature zu einem Problemfall hinzuzufügen. Die Erweiterbarkeit einer Lösungsstrategie ist entscheidend, um dem kontinuierlichen Entwicklungsprozess einer SPL zu unterstützen.

- ++Kein zusätzlicher Aufwand
  Die Erweiterung eines Problemfalls wird bereits von der Lösung unterstützt und benötigt keinen zusätzlichen Anpassungen.
- +Geringer Aufwand
  Das neue Feature des Problemfalls und die verwendete Lösungsstrategie müssen einander angepasst werden.
- −Erhöhter Aufwand
  Die gesamte Umsetzung der Lösung muss zur Integration neuer Feature überarbeitet werden.
- −−Nicht Möglich
  Die verwendete Lösung ist genau auf den vorliegenden Problemfall zugeschnitten und ermöglicht keine Veränderung des Problemfalls.

Performance: Die Durchführung von Messungen der Performance der einzelnen Lösungen war in dem zeitlichen Rahmen dieser Arbeit nicht möglich. Die Bewertung der Performance beruht somit auf rein subjektiven Beobachtungen.

- ++Kein Einfluss
  Die Umsetzung der Lösung zeigt keine Auswirkungen auf die Performance der Anwendung.
- +Geringer Einfluss
  Eine geringfügige kaum wahrnehmbare Einschränkung der Performance tritt bei der Verwendung der Lösungsstrategie auf.
- −Lokaler Verschlechterung
  Die Auswirkungen auf die Performance der veränderten Komponenten sind eindeutig wahrnehmbar. Der Ablauf der betroffenen Komponenten wird durch die verschlechterte Performance gestört.
- −−Globale Verschlechterung
  Die Lösungsstrategie wirkt sich negativ auf die Performance der gesamten Anwendung aus.

4.2.2 Designanpassung

Die Grundidee der Designanpassung ist es, die Auswahl von fehlerhaften Varianten zu verhindern (vgl. Abschnitt 3.4.1). Die Bedingungen der einzelnen Interaktionen werden dabei in das Design der SPL in Form von Abhängigkeits- und Ausschlussbedingungen überführt.

Ein Ausschluss von Varianten der SPL ist somit die Folge für alle Klassen von Optional-Featureproblemen. Die Anzahl der eliminierten Varianten hängt jedoch von der betroffenen Klasse ab. Bei Problemfällen der Klasse der gleichrangigen Feature (Klasse 3) erfolgt die Festlegung auf genau zwei Varianten. Die Auswahl aller beteiligten optionalen Feature oder die Auswahl keines dieser Feature sind die verbleibenden Varianten dieser Problemfälle. Bei Problemfällen der anderen Klassen wird die Variabilität durch die Bedingungen der Lösung eingeschränkt.

Die Bewertung der anderen Kriterien unterscheidet sich nicht für die Umsetzung der Designanpassung unterschiedlicher Klassen von Optional-Featureproblemen.

Zusätzlicher Implementierungsaufwand entsteht bei der Umsetzung einer Designanpassung nicht. Die vorhandenen Implementierungen der Feature müssen lediglich hinsichtlich ihrer Interaktionen untersucht werden. Aufgrund dieser Untersuchung der Interaktionen können fehlerhafte Konfigurationen der Produktlinie entfernt werden.

Die Bewertungskriterien Modularität, Erweiterbarkeit und Performance beurteilen die Auswirkungen von Veränderungen an der Implementierung der SPL. Da bei der Lösungsstrategie der Designanpassung keine Veränderungen am Quellcode vorgenommen werden, ergibt sich die gute Bewertung der übrigen Kriterien. Die zusammenfassende Auswertung der Designanpassung ist in Tabelle 4.7 dargestellt.

	Klasse 1	Klasse 2	Klasse 3	Klasse 4
Variabilität	−	−	−−	−
Implementierungsaufwand	++/+	++	++	++
Modularität	++	++	++	++
Erweiterbarkeit	++	++	++	++
Performance	++	++	++	++

Tabelle 4.7: Bewertung der Designanpassung

4.2.3 Quellcodeaustausch

Das Ziel der Lösungsstrategie des Quellcodeaustausches ist die jeweilige Implementierung von Featuren für die Verwendung in den unterschiedlichen Varianten eines Problemfalls (vgl. Abschnitt 3.4.2).

Die Implementierung spezifischer Umsetzungen jedes Features um Interaktionen zu vermeiden, ermöglicht als Ergebnis dieser Lösung die Nutzung aller Varianten des jeweiligen Problemfalls. Bei der Erstellung einer Variante müssen die zugehörigen Implementierungen der beteiligten Feature ausgewählt werden. Diese Möglichkeit der Auswahl einer entsprechenden Implementierung für jedes Feature erzeugt bei Problemfällen der Klassen 2 und 3 zusätzliche Varianten. Bei Klasse 2 kann die globale Anforderung für jedes betroffene Feature einzeln ausgewählt werden. Problemfälle der Klasse 3 erhalten zusätzliche Varianten da die gegenseitigen Erweiterungen der optionalen Feature untereinander für bestimmte Varianten beeinflusst werden kann.

Die Durchführung dieser Lösungsstrategie erfordert einen mittleren bis hohen Implementierungsaufwand. Bei Problemfällen der Klasse 2 sind unabhängig von der Anzahl der beteiligten Feature für jedes Feature nur zwei Implementierungen nötig. Der Unterschied zwischen diesen beiden alternativen Implementierungen umfasst nur wenige unabhängige

Konstrukte die dem Quellcode hinzugefügt werden müssen. Die Problemfälle der anderen Klassen benötigen multiple Implementierungen ihrer Feature zur Auflösung ihrer vielfältigen Interaktionen.

Die Notwendigkeit mehrerer Implementierungen eines Features erzeugt in allen Klassen von Optional-Featureproblemen Replikationen. Die unterschiedlichen Implementierungen stammen von einer gemeinsamen Grundimplementierung ab und weisen daher viele identische Quellcodeabschnitte auf. Die Veränderung eines Features aus Gründen der Wartung oder Weiterentwicklung erfordert somit die Anpassung aller alternativen Implementierungen des Features.

Auswirkungen auf die Performance treten bei der Lösungsstrategie des Quellcodeaustausches nicht auf. Die gezielten Implementierungen für die einzelnen Varianten ergeben eine performante Anwendung.

Die beschriebene Auswertung der Lösungsstrategie des Quellcodeaustausches ist zusammenfassend in Tabelle 4.8 dargestellt.

	Klasse 1	Klasse 2	Klasse 3	Klasse 4
Variabilität	+	++	++	+
Implementierungsaufwand	−−	−	−−	−−
Modularität	−−	−−	−−	−−
Erweiterbarkeit	−	+	−	−
Performance	++	++	++	++

Tabelle 4.8: Bewertung des Quellcodeaustausches

4.2.4 Bedingte Kompilierung

Die bedingte Kompilierung bietet Möglichkeiten den Quellcode eines Moduls in Abhängigkeit bestimmter Parameter zu modifizieren. Die Abbildung mehrerer Ausprägungen von Quellcodeabschnitten lassen sich damit in einem Modul umsetzen (vgl Abschnitt 3.4.3).

Mit Hilfe der bedingten Kompilierung lassen sich die Implementierungen der Feature eines Problemfalls an alle Varianten anpassen. Bei der Auswahl der beteiligten Feature werden bei dieser Lösung jeweils die richtigen Parameter für die Erstellung der Variante gesetzt. Aus diesen Parametern ergeben sich die benötigten Ausprägungen der Feature für die jeweiligen Varianten.

Der Aufwand für die Umsetzung dieser Lösungsstrategie variiert in Abhängigkeit von der verlangten Modularität des erzeugten Quellcodes. Die geringste Modularität entspricht der Umsetzung des Quellcodeaustausches in dem jeweils der gesamte Quellcode mit Hilfe einer Präprozessoranweisung dupliziert wird. Je detaillierter die Auswahl des betroffenen Quellcodes durchgeführt wird, desto höher kann der Aufwand zur Umsetzung der entsprechenden Lösungen ausfallen. Entscheidend für den Aufwand ist dabei zusätzlich die Anzahl der notwendigen Quellcodemodifikationen, die von der Komplexität der betroffenen Feature und ihrer Interaktionen abhängt.

Die Modularisierung erfolgt bei der Lösung der bedingten Kompilierung durch die Annotation der Veränderungen des Quellcodes. Die Erweiterbarkeit dieser Lösung in Bezug auf neue Feature erfordert das Verständnis für die vorhandenen Annotationen.

Das neue Feature muss je nach Art seiner Interaktionen andere Feature modifizieren und/oder von ihnen modifiziert werden. Bei Problemfällen der Klasse der dynamischen Interaktionen (Klasse 4) muss zusätzlich die Gültigkeit der bereits umgesetzten Lösung überprüft werden.

Die Performance der Anwendung wird von Lösungsvarianten der bedingten Kompilierung nicht beeinflusst. Lediglich die Erstellung von Varianten der SPL wird um einen weiteren Schritt erweitert.

Tabelle 4.9 zeigt die Ergebnisse der Auswertung der Kriterien für die Lösungsstrategie der bedingten Kompilierung.

	Klasse 1	Klasse 2	Klasse 3	Klasse 4
Variabilität	+	+	+/++	+
Implementierungsaufwand	−	+	−	−
Modularität	−	−	−	−
Erweiterbarkeit	+	+	+	−
Performance	++	++	++	++

Tabelle 4.9: Bewertung der bedingten Kompilierung

4.2.5 Auslagern der Interaktion

Das Auslagern der Interaktion in abhängige, eigenständige Module (Derivate) ist die Grundidee dieser Lösungsstrategie (vgl. Abschnitt 3.4.4). Die ausgelagerten Interaktionen können somit in Abhängigkeit von der Auswahl der Feature in die Varianten der SPL integriert werden. Die Möglichkeiten der AOP bieten in vielen Fällen die nötigen Ansätze, um die Auslagerung der einzelnen Interaktionen zu realisieren. Analog zu den beiden vorher betrachteten Lösungsstrategien ist für diese Lösung eine zusätzliche Auswahl der richtigen Komponenten in Abhängigkeit von den Varianten der SPL notwendig.

Mit Hilfe der Bildung von Derivaten lassen sich die Interaktionen der Problemfälle für alle Varianten der Produktlinie organisieren.

Der Implementierungsaufwand hängt bei der Auslagerung der Interaktion stark von der Anzahl der unterschiedlichen Interaktionen und Varianten eines Problemfalls ab. Für jede Interaktion zwischen zwei Featuren ist die Erstellung eines eigenen Derivates nötig. Identische Interaktionen hingegen lassen sich mit den Mitteln der AOP bei dieser Lösung über identische Derivate abbilden. Diese Eigenschaft vermindert den Implementierungsaufwand für die Problemfälle der Klasse der globalen Anforderungen (Klasse 2) sehr stark.

Die zusätzliche Modularität dieser Lösungsstrategie erhöht die Komplexität des Quellcodes. Die Lesbarkeit und das Verständnis wird durch die zusätzlichen Aufsplittungen vermindert. Die Auswirkungen dieser Lösung erzeugen einen erhöhten Grad an Modularität.

Die Erweiterung eines Problemfalls, um ein neues Feature, erzeugt Interaktionen mit einigen oder allen der vorher beteiligten Feature. Die neuentstandenen Interaktionen müssen in Derivate separiert werden und auf das Zusammenwirken mit den bestehenden Derivaten angepasst werden. Der Aufwand für die Erstellung eines neuen Features ist somit sehr umfangreich. Bei der vereinfachten Lösung für die Problemfälle der Klasse 2

ist nur die Einhaltung der mit der Lösung verknüpften Bedingungen bei der Implementierung eines neuen Features zu beachten (vgl. Abschnitt 3.5.8).

Die vorhandenen Werkzeuge zur Umsetzung von AOP im Bereich der Programmiersprache C++ erzeugen deutliche Beeinträchtigungen der Performance. Die Nutzbarkeit von der Lösungsstrategie wird somit durch eine schlechte Werkzeugunterstützung für die benutzten Techniken stark vermindert.

Die Bewertung der Lösungstrategie wird für alle Klassen von Optional-Featureproblemen in Tabelle 4.10 dargestellt.

	Klasse 1	Klasse 2	Klasse 3	Klasse 4
Variabilität	+	+	+	+
Implementierungsaufwand	−	+	−	−
Modularität	+	+	+	+
Erweiterbarkeit	−	++	−	−
Performance	−	−	−	−

Tabelle 4.10: Bewertung des Auslagerns der Interaktion

4.2.6 Optionales Weben

Eine Eigenschaft der AOP ermöglicht ein spezielles Verfahren zur Auswahl benötigter Derivate. Die notwendigen Veränderungen werden an optionalen Featuren in Abhängigkeit von ihrer Auswahl vorgenommen(vgl. Abschnitt 3.4.5). Die Umsetzung dieses Lösungsansatz benötigt für einige Problemfälle zusätzliche Überlegungen, entspricht in der Auswertung der betrachteten Kriterien der Lösungsstrategie der Auslagerung der Interaktion.

4.2.7 Kapselung

Die Lösungsstrategie der Kapselung stellt die Ausschöpfung der Möglichkeiten der OOP dar. Problemfälle können durch diesen Lösungsansatz aufgelöst oder zumindest vereinfacht werden (vgl. Abschnitt 3.4.6).

Für die Problemfälle der Klassen 2 und 3 bietet diese Lösungsstrategie keine Ansätze und die Auswertung kann somit nur über 2 Klassen von Optional-Featureproblemen erfolgen.

Es existiert nur ein einziger grundlegender Unterschied der Lösungsstrategie in Bezug auf die betrachteten Klassen. Bei den Problemfällen der Klasse der dynamischen Interaktionen (Klasse 4) werden die Interaktionen vollständig gelöst und bei den Problemfällen der Klasse der Alternativen (Klasse 1) ergibt sich nur eine Vereinfachung. Zur endgültigen Lösung dieser Problemfälle ist zusätzlich die Verwendung einer anderen Lösungsstrategie erforderlich.

Der initiale Implementierungsaufwand zur Umsetzung dieser Lösungsstrategie ist sehr hoch. Erst eine große Anzahl an beteiligten Featuren macht den Lösungsansatz praktikabel. Die deutliche Vereinfachung oder vollständige Auflösung des Optional-Featureproblems ohne die Verwendung zusätzlicher Mechanismen ist der Vorteil dieser Lösung.

Die Umsetzungen von Lösungen dieses Ansatzes erhöhen die Modularität der betroffenen Feature und vereinfachen gleichzeitig die Erweiterbarkeit des jeweiligen Problemfalls um neue Feature. Die vollständige Auswertung der Lösungsstrategie ist in Tabelle 4.11 dargestellt.

	Klasse 1	Klasse 4
Variabilität	+	+
Implementierungsaufwand	−	−
Modularität	+	+
Erweiterbarkeit	+	+
Performance	++	++

Tabelle 4.11: Bewertung der Kapselung

4.3 Richtlinie

In diesem Abschnitt ergibt sich aus den Ergebnissen dieser Arbeit eine Richtlinie. Die Analyse der Optional-Featureprobleme einer Beispiel-SPL und die Untersuchung der Eignung von Lösungsstrategien für die herausgearbeitete Klassen von Optional-Featureproblemen bilden die Grundlage für die Aufstellung geeigneter Anregungen und Hinweise. Das Ziel der Richtlinie ist es zukünftige Entwicklungsprozesse mit den Erkenntnissen dieser Arbeit zu unterstützen.

Klassen von Optional-Featureproblemen:	
Alternativen	
	Definition: Die Interaktionen werden von alternativ vorkommenden Featuren ausgelöst. Die Optionalität der alternativen Feature interagiert mit anderen optionalen Featuren. Merkmale: Ein Teil des Problems ist eine Alternative/Oder-Beziehung. Die Feature des Problems benutzen die Funktionalität anderer Feature des Problems. Beispiel: Betriebssystemabstraktion, unterschiedliche Zugriffsstrukturen einer Datenbank Lösungen: Keine der Lösungen zeichnet sich durch deutliche Vor- oder Nachteile aus.
Globale Anforderungen	
	Definition: Eine Anforderung erfordert die Anpassung mehrerer Feature der SPL. Merkmale: Die Umsetzung der Anforderung verändert die Funktionalität der betroffenen Feature nicht. Zwischen den Featuren besteht kein direkter Bezug. Beispiel: Logging, Statistiken Lösungen: Das Auslagern der Interaktionen in Derivate ist die beste Lösung. Die Erstellung weniger Derivate kann ausreichen, um alle Interaktionen abzubilden.

Tabelle 4.12: Richtlinie (Teil 1)

<table>
<tr><td colspan="2">Klassen von Optional-Featureproblemen:</td></tr>
<tr><td colspan="2">Gleichrangige Feature</td></tr>
<tr><td></td><td>Definition: Optionale Feature erweitern die Funktionalität anderer optionaler Feature.
Merkmale: Die Feature stehen ohne gegenseitiger domänespezifischer Abhängigkeiten in einer Und-Beziehung zueinander. Mindestens eines der Feature wird durch die Funktionalität eines anderen optionalen Features erweitert.
Beispiel: Nachrichtenerweiterungen
Lösungen: Die bedingte Kompilierung verfügt über die geeignetsten Eigenschaften der Lösungsansätze. Ihr Überblick über bereits vorhandene Erweiterungen der Feature erleichtert das Hinzufügen weiterer interagierender Feature.</td></tr>
<tr><td colspan="2">Dynamische Interaktionen</td></tr>
<tr><td></td><td>Definition: Die Interaktionen der Feature zeigen sich nur durch Wechselwirkungen im Ablauf der Anwendung. Die Auswirkungen reichen von schlecht nachvollziehbaren bis hin zu fehlerhaftem Verhalten.
Merkmale: Die Auswirkungen der Funktionalität optionaler Feature überschneiden sich und erzeugen ungewollte Seiteneffekte bei gemeinsamer Verwendung.
Beispiel: unterschiedliche Rechtemanagements
Lösungen: Das Auftreten der unterschiedlichen Funktionalität kann mit einer Designanpassung einfach verhindert werden. Erfordert die SPL die Variabilität des Problems erzielt eine Neugestaltung der Implementierungen des Problems die besten Ergebnisse.</td></tr>
</table>

Tabelle 4.13: Richtlinie (Teil 2)

Kapitel 5

Zusammenfassung und Ausblick

Im letzten Kapitel werden die erreichten Ergebnisse und gewonnenen Erkenntnisse dieser Arbeit zusammengefasst. Im Ausblick werden weiterführende Forschungsansätze der behandelten Thematik ausgehend von den Ergebnissen vorgestellt.

5.1 Zusammenfassung

Die Aufgabenstellung „Systematische Analyse von Feature-Interaktionen in Softwareproduktlinien“ wurde im Rahmen dieser Arbeit auf die spezielle Problematik der Optional-Featureprobleme eingegrenzt. Optional-Featureprobleme stellen eine bedeutende Untermenge der in einer SPL auftretenden Feature-Interaktionen dar. Die nutzbare Variabilität einer Softwareproduktlinie wird durch die Vorkommen von Optional-Featureproblemen stark vermindert. Die Behandlung und Auflösung der Feature-Interaktionen, die zu Optional-Featureproblemen führen, hat somit direkten Einfluss auf die Vielfalt der möglichen Produkte einer SPL. Ausgehend von der Zielstellung eine Richtlinie zur Unterstützung zukünftiger Entwicklungen von Softwareproduktlinien zu erarbeiten, wurden folgende Arbeitsschritte ausgeführt.

Die Einarbeitung in die Thematiken der Modularisierung und der Produktlinienentwicklung bildet die Grundvoraussetzung zum Verständnis des Themengebiets der gestellten Aufgabenstellung. Eine übergreifende Vorstellung bekannter Modularisierungsmethoden und -werkzeuge erläutert die grundlegenden Ansätze zur Implementierung von Softwareproduktlinien. Die Begriffsklärung der Softwareproduktlinien zeigt die Rahmenbedingungen und Voraussetzungen des betrachteten Typus von Entwicklungsprozessen auf.

Zur Analyse von Feature-Interaktionen wurde eine Produktlinie in der Domäne von Chatprogrammen entworfen und umgesetzt. Der Aufbau der Beispiel-SPL wurde dabei so bemessen, dass eine Vielzahl unterschiedlicher Interaktionen auftreten.

Die bei der Untersuchung der Chat-SPL erkannten Optional-Featureprobleme wurden im ersten Schritt der systematischen Analyse unabhängig von möglichen Lösungen betrachtet. Insgesamt wurden neun Problemfälle in der SPL ermittelt. Die Ergebnisse dieser ersten Analyse ergeben in Form genauer Beschreibungen der einzelnen Problemfälle die Grundlage für die weiteren Analysen dieser Arbeit.

Um weitergehende Betrachtungen an den untersuchten Problemfällen durchzuführen,

wurden geeignete Lösungsstrategien für Optional-Featureprobleme erarbeitet und vorgestellt. Die Umsetzung und Analyse dieser Strategien bildete den zweiten Schritt der systematischen Analyse. Für diesen Schritt waren intensivere Untersuchungen der Problemfälle und der Implementierungen der an ihnen beteiligten Feature notwendig.

Ausgehend von den Ergebnissen dieser durchgeführten Untersuchungen an einer SPL eines realen Anwendungsszenarios erfolgten mehrere Auswertungen. Die Resultate der vollzogenen Betrachtungen wurden für die getroffenen Auswertungen von unterschiedlichen Gesichtspunkten analysiert.

Die Nutzbarkeit der Ergebnisse dieser Arbeit für andere Entwicklungsprojekte von Softwareproduktlinien ergibt sich aus der Klassifizierung von Optional-Featureproblemen. Zur eindeutigen Einteilung der betrachteten Problemfälle wurden fünf Bewertungskriterien definiert. Die Auswertung dieser Kriterien wurde für jeden der Problemfälle durchgeführt und ergab vier verschiedene Klassen von Optional-Featureproblemen.

Darauf aufbauend wurden die in der Arbeit betrachteten Lösungsstrategien hingegen ihre Eignung für die gefundenen Klassen untersucht. Die Bewertung der Lösungsstrategien erfolgte anhand von fünf klar definierten Kriterien.

Das endgültige Ergebnis dieser Arbeit stellt die Formulierung einer Richtlinie zum Umgang mit Optional-Featureproblemen dar.

5.2 Ausblick

In dieser Arbeit wurde anhand der Untersuchung einer Softwareproduktlinie eine Richtlinie zur Klassifizierung von Optional-Featureproblemen erarbeitet. Die Klassifikation von Optional-Featureproblemen unterstützt das Verständnis und die Lösung der Probleme. Die Ergebnisse der Arbeit bilden eine solide Grundlage für den Aufbau einer Richtlinie umfassenden Klassifizierung. Die Bestandteile der Richtlinie können ausgehend von Erkenntnissen bei ihrer Verwendung in anderen Produktlinien erweitert und angepasst werden. Die stetige Verwendung steigert somit den Nutzen der Richtlinie.

Zur Erweiterung der Richtlinie sind zusätzlich folgende Ansätze für zukünftige Untersuchungen denkbar.

- Die Erforschung neuer Lösungsansätze kann besser geeignete Lösungen für einzelne Klassen von Optional-Featureproblemen aufzeigen. Für keine der erarbeiteten Klassen konnte im Rahmen dieser Arbeit eine ultimativer Ansatz zur Lösung ihrer Interaktionen vorgestellt werden. Ausgehend von den Eigenschaften einer Klasse können sich gezieltere Ansätze ergeben. Interessant ist dabei auch die Kombination der einzelnen Ansätze zur Lösung eines Problems.

- Die Möglichkeiten und das Verhalten der untersuchten Lösungsansätze können in Bezug auf die verwendeten Programmiersprachen variieren. Die Notwendigkeit der Spezialisierung der Richtlinie auf einzelne Programmiersprachen muss untersucht werden. Die Klassifizierung der Optional-Featureprobleme kann dabei in allen Sprachen verwendet werden.

- Für die Dokumentation von Feature-Interaktionen existieren keine geeigneten Mittel oder Methoden im Entwicklungsprozess einer SPL. Die Klassifizierung der

Optional-Featureprobleme kann als Grundlage die Erarbeitung einer grafische oder grammatikalischen Notation von Interaktionen unterstützen.

Eine zusätzliche Verbesserung der Lösungsansätze kann durch die Verbesserung der Werkzeugunterstützung erfolgen. Besonders im Bereich der Annotationen bestehen noch zahlreiche Ansätze zur Verbesserung. Die Ziele dieser Verbesserungen sind Überprüfungen der Richtigkeit durchgeführter Annotation und die Verbesserung ihrer Lesbarkeit.

Für die Lösungsansätze die auf Methoden der AOP aufbauen, existieren im Bereich C++ keine effektiven Werkzeuge. Die zur Verfügung stehenden Aspektweber vermindern die Performance der Anwendung deutlich. Außerdem existieren nur rudimentäre Ansätze, um die Lösungen zu debuggen.

Literaturverzeichnis

[Bat05] Batory, Don: Feature models, grammars, and propositional formulas. In: *Lecture notes in computer science* 3714 (2005), S. 7–20. – ISBN 3–540–28936–4

[BLS03] Batory, Don ; Liu, Jia ; Sarvela, Jacob N.: Refinements and Multi-Dimensional Separation of Concerns. In: *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM SIGSOFT, 2003. – ISBN 1–58113–743–5, S. 48–57

[BMMB00] Bosch, Jan ; Molin, Peter ; Mattsson, Michael ; Bengtsson, PerOlof: Object-oriented framework-based software development: problems and experiences. In: *ACM Computing Surveys (CSUR)* 32 (2000), Nr. 1es. – ISSN 0360–0300

[Bos00] Bosch, Jan: *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000. – ISBN 0–201–67494–7

[Bru02] Bruce, Kim B.: *Foundations of object-oriented languages: types and semantics*. Cambridge, MA, USA : MIT Press, 2002. – ISBN 0–262–02523–X

[BSR03] Batory, Don ; Sarvela, Jacob N. ; Rauschmayer, Axel: Scaling stepwise refinement. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* Bd. 30. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1877–X, S. 187–197

[CEC00] Czarnecki, Krzysztof ; Eisenecker, Ulrich ; Czarnecki, Krzysztof: *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA : Addison-Wesley Professional, 2000. – ISBN 0–201–30977–7

[CL03] Clifton, Curtis ; Leavens, Gary T.: Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy / Iowa State University Technical Report. 2003 (03-15). – Forschungsbericht. – Revised version of TR03-01

[CN02] Clements, Paul ; Northrop, Linda: *Software Product Lines: Practices and Patterns*. Boston, MA, USA : Addison-Wesley, 2002. – ISBN 0–2017–0332–7

[DDH72] DIJKSTRA, Edsger. W. ; DAHL, Ole-Johan ; HOARE, Charles Antony R.: *Structured programming*. London, UK : Academic Press, 1972. – ISBN 0–1220–0550–3

[Dij76] DIJKSTRA, Edsger W.: *A Discipline of Programming*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1976. – ISBN 0–1321–5871–X

[Dij79] DIJKSTRA, Edsger. W.: *The humble programmer*. New Jersey, NY, USA : Yourdon Press, 1979. – ISBN 0–321–21976–7

[EFB01] ELRAD, Tzilla ; FILMAN, Robert E. ; BADER, Atef: Aspect-oriented programming: Introduction. In: *Communications of the ACM (CACM)* 44 (2001), Nr. 10, S. 29–32. – ISSN 0001–0782

[Fai86] FAIRLEY, Richard E.: *Software Engineering Concepts*. New York, NY, USA : McGrawHill, Inc., 1986. – ISBN 0–07–019902–7

[FECA05] FILMAN, Robert E. ; ELRAD, Tzilla ; CLARKE, Siobhán ; AKŞIT, Mehmet: *Aspect-Oriented Software Development*. Boston, MA, USA : Addison-Wesley, 2005. – ISBN 0–3212–1976–7

[GH05] GRUNDY, John ; HOSKING, John: Developing Software Components with Aspects: Some Issues and Experiences. In: *A*spect-Oriented Software Development[FECA05], S. 585–604

[GHJV93] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E. ; VLISSIDES, John M.: Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*. London, UK : Springer-Verlag, 1993. – ISBN 3–540–57120–5, S. 406–431

[GIL89] GILROY, Kathleen A.: Impact of domain analysis on reuse methods / U.S. Army Communications-Electronics Command. 1989 (C04-087LD-0001-00). – Forschungsbericht. – NASA, Langley Research Center

[GJM91] GHEZZI, Carlo ; JAZAYERI, Mehdi ; MANDRIOLI, Dino: *Fundamentals of Software Engineering*. New Jersey, NY, USA : Prentice-Hall, Inc, 1991. – ISBN 0–13–820432

[JBR99] JACOBSON, Ivar ; BOOCH, Grady ; RUMBAUGH, James: *The Unified Software Development Process*. New York, NY, USA : Addison Wesley Professional, 1999. – ISBN 0–2015–7169–2

[JF88] JOHNSON, Ralph E. ; FOOTE, Brian: Designing reuseable classes. In: *Journal of Ob ject-Oriented Programming* 1 (1988), Nr. 2. – ISSN 0896–8438

[KAR+09] KÄSTNER, Christian ; APEL, Sven ; RAHMAN, Syed S. ; ROSENMÜLLER, Marko ; BATORY, Don ; SAAKE, Gunter: On the Impact of the Optional Feature Problem: Analysis and Case Studies. In: *Proceedings of the 13th International Software Product Line Conference (SPLC)* (2009)

[Käs07] Kästner, Christian: *Aspect-Oriented Refactoring of Berkeley DB.* Germany, University of Magdeburg, Diplomarbeit, 2007

[KCH+90] Kang, K.C. ; Cohen, S.G. ; Hess, J.A. ; Novak, W.E. ; Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. 1990 (CMU/SEI-90-TR-21). – Forschungsbericht. – Carnegie Mellon University

[KHH+01] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001).* London, UK : Springer-Verlag, 2001. – ISBN 3–540–42206–4, S. 327–353

[KKST79] Kimm, Reinhold ; Koch, W. ; Simonsmeier, Werner ; Tontsch, Friedrich: *Einfuehrung in Software Engineering.* Berlin, Germany : Walter de Gruyter, 1979. – ISBN 3–11–007836–8

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)* Bd. 1241. Jyvaskyla, Finland : Springer-Verlag, 1997, S. 220–242

[LARS05] Leich, Thomas ; Apel, Sven ; Rosenmüller, Marko ; Saake, Gunter: Handling Optional Features in Software Product Lines. In: *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code.* San Diego, CA, USA : ACM SIGPLAN, 2005

[Mey88] Meyer, Bertrand: *Object-oriented Software Construction.* Prentice-Hall, 1988

[Mic96] Michael, Mattson: *Object-Oriented Frameworks - A survey of methodological issues.* 1996

[MN96] Moser, Simon ; Nierstrasz, Oscar: The Effect of Object-Oriented Frameworks on Developer Productivity. In: *IEEE Computer* 29 (1996), Nr. 9, S. 45–51. – ISSN 0018–9162

[MPC+86] McNicholl, Daniel G. ; Palmer, Constance ; Cohen, Sandford G. ; Whitford, William H. ; Goeke, Gerard O.: *Common Ada missile packages–CAMP, Vol. I: Overview and commonality study results.* 1986

[MS06] Murphy, Gail C. ; Schwanninger, Christa: Guest Editors' Introduction: Aspect-Oriented Programming. In: *IEEE Software* 23 (2006), Nr. 1, S. 20–23. – ISSN 0740–7459

[NR69] Naur, Peter ; Randell, Brian: *Software Engineering Report of a conference sponsored by the NATO Science Committee.* Brussels, Belgium, 1969

[NT95] Nierstrasz, Oscar ; Tsichritzis, Dennis: *Object-oriented software composition.* Hemel Hempstead, Hertfordshire, UK : Prentice Hall International (UK) Ltd., 1995. – ISBN 0–13–220674–9

[Par72] PARNAS, David L.: On the criteria to be used in decomposing systems into modules. In: *Communications of the ACM* 5 (1972), Nr. 12, S. 1053–1058. – ISSN 0001–0782

[PBVDL05] POHL, Klaus ; BÖCKLE, Günter ; VAN DER LINDEN, Frank: *Software Product Line Engineering: Foundations, Principles, and Techniques*. New York, NY, USA : Springer-Verlag, 2005. – ISBN 3–5402–4372–0

[PD90] PRIETO DÍAZ, Rubén: Domain analysis: an introduction. In: *ACM SIGSOFT Software Engineering Notes* 15 (1990), Nr. 2, S. 47–54. – ISSN 0163–5948

[PD93] PRIETO DÍAZ, Rubén: Status Report: Software Reusability. In: *IEEE Software* 10 (1993), Nr. 3, S. 61–66. – ISSN 0740–7459

[PP04] PRESSMAN, Roger S. ; PRESSMAN, Roger: *Software Engineering: A Practitioner's Approach*. New York, NY, USA : McGraw-Hill Science/Engineering/Math, 2004. – ISBN 0–0730–1933–X

[Pre97] PREHOFER, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)* Bd. 1241. New York, NY, USA : Springer-Verlag, 1997, S. 419–443

[SC92] SPENCER, Henry ; COLLYER, Geoff: #ifdef Considered Harmful, or Portability Experience With C News. In: *Proceedings of the Usenix Summer 1992 Technical Conference*. Berkeley, CA, USA : Usenix Association, 1992, S. 185–198

[Sny86] SNYDER, Alan: Encapsulation and Inheritance in Object-Oriented Programming Languages. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* Bd. 21. New York, NY, USA : ACM Press, 1986, S. 38–45

[Som04] SOMMERVILLE, Ian: *Software Engineering*. Pearson Education, 2004. – ISBN 0–3212–1026–3

[Szy02] SZYPERSKI, Clemens A.: *Component Software — Beyond Object-Oriented Programming*. Addison Wesley, 2002. – ISBN 0–201–74572–0

[TOHJ99] TARR, Peri ; OSSHER, Harold ; HARRISON, William ; JR., Stanley M. S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. Los Angeles CA, USA : ACM Press, 1999, S. 107–119

[Weg86] WEGNER, Peter: Classification in Object-Oriented Systems. In: *ACM SIGPLAN Notices* 21 (1986), Nr. 10, S. 173–182. – ISSN 0362–1340

[Wir72] WIRTH, Niklaus: *Systematisches Programmieren*. Stuttgart, Germany : B. G. Teubner, 1972. – ISBN 3–5190–2375–X

[Zam98] Zamir, Saba: *Handbook of Object Technology.* Boca Raton, FL, USA : CRC Press, Inc., 1998. – ISBN 0–8493–3135–8

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 15. Oktober 2009

Andreas Schulze