Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Bachelorarbeit

Anwendungsspezifische Generierung von Quelltext-Dokumentationen für die Feature-Orientierte Programmierung

Autor:

Sebastian Krieter

23.04.2014

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Reimar Schröter

Institut für Technische und Betriebliche Informationssysteme

Krieter, Sebastian:
Anwendungsspezifische Generierung von Quelltext-Dokumentationen für die Feature-Orientierte Programmierung
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2014.

Inhaltsverzeichnis

Abbildungsverzeichnis vi

Tabellenverzeichnis vii

Abkürzungsverzeichnis ix

1 Einführung 1

2 Grundlagen 5
- 2.1 Software-Dokumentation . . . 5
  - 2.1.1 Arten der Software-Dokumentation . . . 5
  - 2.1.2 Dokumentationsgeneratoren . . . 6
- 2.2 Software-Produktlinien . . . 9
  - 2.2.1 Grundlagen der Software-Produktlinien . . . 9
  - 2.2.2 Feature-Modellierung . . . 10
  - 2.2.3 Feature-Orientierte Programmierung . . . 13
- 2.3 Zusammenfassung . . . 17

3 Anforderungsanalyse 19
- 3.1 Anwendungsfälle für SPL-Dokumentationen . . . 19
  - 3.1.1 Produkt-Dokumentation . . . 20
  - 3.1.2 SPL-Dokumentation . . . 21
  - 3.1.3 Kontext-Dokumentation . . . 22
  - 3.1.4 Feature-Modul-Dokumentation . . . 23
- 3.2 Probleme bei der Verwendung trivialer Ansätze . . . 24
  - 3.2.1 Produkt-Ansatz . . . 25
  - 3.2.2 SPL-Ansatz . . . 26
  - 3.2.3 Feature-Modul-Ansatz . . . 27
- 3.3 Zusammenfassung . . . 28

4 Lösungsansatz 31
- 4.1 Grundidee des Lösungsansatzes . . . 31
- 4.2 Erstellung von Modul-Kommentaren . . . 33
  - 4.2.1 Trennung allgemeiner und Feature-spezifischer Informationen . 33
  - 4.2.2 Priorisierung der Informationen . . . 34
  - 4.2.3 Übersicht über die neuen Schlüsselwörter . . . 34
- 4.3 Ablauf des Vereinigungsverfahrens . . . 35
  - 4.3.1 Vereinigung von gleichen Tags . . . 38

4.3.2 Generierung von Dokumentationskommentaren 41
4.4 Zusammenfassung 42

5 Implementierung 45
5.1 Umsetzung der Modul-Kommentare 45
5.2 Implementierung des Prototypen SPLDoctor 50
5.2.1 Integration von SPLDoctor in FeatureIDE 50
5.2.2 Details zur Implementierung von SPLDoctor 51
5.3 Zusammenfassung 52

6 Evaluierung 55
6.1 Ablauf der Evaluierung 55
6.1.1 Qualitative Evaluierung 55
6.1.2 Quantitative Evaluierung 56
6.2 Verwendete Produktlinien 58
6.2.1 Chat-Produktlinie 58
6.2.2 Snake-Produktlinie 59
6.3 Erhaltene Ergebnisse 60
6.3.1 Chat-Produktlinie 60
6.3.2 Snake-Produktlinie 61
6.3.3 Interpretation der Ergebnisse 62
6.3.4 Aussagekraft der Ergebnisse 64
6.4 Zusammenfassung 65

7 Verwandte Arbeiten 67

8 Zusammenfassung und Fazit 69

9 Zukünftige Arbeiten 73

Literaturverzeichnis 77

Abbildungsverzeichnis

2.1 JavaDoc Beispiel anhand eines Ausschnitts der Klasse `Encryption` in der Chat-Produktlinie . . . 7
2.2 Vergleich der Projektkosten zwischen Produkten einer SPL und einzelnen Systemen . . . 10
2.3 Vergleich der Zeiten bis zur Markteinführung zwischen Produkten einer SPL und einzelnen Systemen . . . 11
2.4 Feature-Modell einer einfachen Chat-Produktlinie . . . 12
2.5 Aussagenlogischer Ausdruck für das Feature-Modell der Chat-Produktlinie . . . 12
2.6 Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie 16
2.7 Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie . 17

3.1 Kommentare des Produkt-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie . . . 21
3.2 Kommentare des SPL-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie . . . 23
3.3 Kommentare des Feature-Modul-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie . . . 25
3.4 Kommentare des Produkt-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie . . . 27
3.5 Kommentare des SPL-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie . . . 27
3.6 Kommentare des Feature-Modul-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie . . . 28

4.1 Grundsätzlicher Ablauf des Konzepts . . . 32
4.2 Modul-Kommentare für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie . . . 36
4.3 Flussdiagramm über den grundsätzlichen Ablauf des Vereinigungsverfahren . . . 37

4.4 Flussdiagramm für die Vereinigung von gleichen Tags 39
4.5 Flussdiagramm für die Generierung von Dokumentationskommentaren 41
4.6 Pseudo-Quelltext für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie 42

5.1 Modul-Kommentare mit HTML-Tags anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie) 47
5.2 Modul-Kommentare mit JavaDoc-Tags anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie) 48
5.3 Modul-Kommentare mit eigenständigen Kommentaren anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie) . 49
5.4 Erstellen einer Dokumentationen 53

6.1 Feature-Modell der Snake-Produktlinie 59
6.2 Grafische Repräsentation der quantitativen Evaluierung für die Chat-Produktlinie 61
6.3 Grafische Repräsentation der quantitativen Evaluierung für die Snake-Produktlinie 63

Tabellenverzeichnis

2.1 Übersicht über die Verwendung der JavaDoc-Standard-Tags 8
2.2 Ausschnitt des Kollaborationsdiagramms der Chatproduktlinie 14

3.1 Übersicht über die Probleme bei verschiedenen Ansätzen 29

4.1 Übersicht über die Regeln für das Vereinigen von Tags der Modul-Kommentare 40

5.1 Übersicht über mögliche Formen eines neuen Schlüsselwortes 46

6.1 Übersicht über die Modul-Kommentare der Chat-Produktlinie 60
6.2 Übersicht über die Ergebnisse der quantitativen Evaluierung für die Chat-Produktlinie 61
6.3 Übersicht über die Modul-Kommentare der Snake-Produktlinie 62
6.4 Übersicht über die Ergebnisse der quantitativen Evaluierung für die Snake-Produktlinie 62

Abkürzungsverzeichnis

AOP	Aspekt-orientierte Programmierung
FOP	Feature-orientierte Programmierung
IDE	Entwicklungsumgebung (Integrated Development Environment)
JDK	Java Development Kit
OOP	Objekt-orientierte Programmierung
SPL	Software-Produktlinie

1. Einführung

In der Software-Entwicklung werden häufig Software-Produkte erstellt, die große Ähnlichkeit zu bereits vorhandenen Software-Produkten haben und auch ähnliche Funktionalität bieten. Das liegt zumeist daran, dass die Benutzer der Software-Produkte unterschiedliche Anforderungen daran stellen. Somit gibt es für eine bestimmte Funktionalität oft viele ähnliche Produkte und Varianten dieser Produkte. Für das Erstellen solcher Produkte greifen die Entwickler oft auf die Wiederverwendung von Quelltext zurück. Dafür kann alter Quelltext einfach geändert, erweitert oder auch vollständig beziehungsweise teilweise in neuen Software-Produkte übernommen werden. Im Kontrast zur vollständigen Neuentwicklung eines Software-Produkts, erspart diese Methode oft viel Zeit bei der Entwicklung eines neuen Software-Produkts und senkt damit auch die Entwicklungskosten [Kru92]. Dennoch muss hier beachtet werden, dass nicht jeder Quelltext einfach wiederverwendet werden kann. Ist der Quelltext speziell für ein bestimmtes Anwendungsszenario geschrieben, so kann die Anpassung an ein Anderes durchaus sehr zeitaufwendig sein.

Für eine Wiederverwendung von Quelltext ist es notwendig, dass dieser gut strukturiert ist. Dafür ist es nötigt, die Besonderheiten der verwendeten Programmiersprache auszunutzen. Viele der heutzutage populären Programmiersprachen, wie Java, C++, C# oder PHP benutzen die Objekt-orientierte Programmierung (OOP) als Paradigma. OOP weist einige Merkmale auf, die eine leichtere Wiederverwendung des Quelltextes unterstützten. Gerade die Möglichkeiten der Vererbung und die Kapselung von Objekten eignet sich dafür. Durch das Nutzen von Design-Pattern kann der Quelltext so gut strukturiert werden, dass eine Wiederverwendung stark vereinfacht wird [GHJV94]. Dennoch ist OOP nicht optimal zur Wiederverwendung des Quelltextes geeignet [Cza98]. Oft ist der Quelltext eines OOP-Projekts an eine Situation angepasst. Wird der Quelltext für eine andere Situation wiederverwendet, so kann dies zu Problemen führen, wie zum Beispiel Programmierfehlern oder Performanceeinbußen.

Um eine bessere Wiederverwendung von Quelltexten zu erreichen, werden neue Konzepte, wie Software-Produktlinien (SPLs) benötigt. Eines der Hauptziele von SPLs ist eine einfachere Wiederverwendung von Quelltexten. Eine Möglichkeit eine SPL zu implementieren ist das Paradigma der Feature-orientierte Programmierung

(FOP). Die Idee von FOP ist die Zerlegung des gesamten Quelltextes der SPL in Feature [BSR04, Pre01]. Software-Produktlinien werden immer häufiger genutzt, da Sie neben der einfacheren Wiederverwendung von Quelltext auch einige andere Vorteile gegenüber herkömmlich entwickelter Software bieten [PBvdL05]. Der größte Vorteil von SPLs liegt in der großen Variabilität innerhalb ihrer Domäne, sodass die generierten Produkte einer SPL genau an die Bedürfnisse der Anwender angepasst werden können.
Abgesehen von der guten Strukturierung des Quelltextes, gibt es noch eine andere wesentliche Voraussetzung für eine Wiederverwendung. Die Funktionalität von einmal geschriebenem Quelltext muss für die Entwickler auch noch im Nachhinein verständlich sein. Dies ist absolut notwendig, um den Quelltext editieren und erweitern zu können. Das Problem dabei ist jedoch, dass Quelltext ab einem gewissen Grad an Größe oder Komplexität schnell unübersichtlich und schwer verständlich wird. Insbesondere ist dies der Fall, wenn dieser von fremden Entwicklern stammt oder die Erstellung schon eine längere Zeit zurückliegt. An dieser Stelle helfen Quelltext-Dokumentationen. Diese beschreiben die Funktionalität und die besonderen Merkmale von bestimmten Quelltextabschnitten. Entwickler sind mithilfe von gut geschriebenen Dokumentationen in der Lage den Quelltext schnell zu erfassen und zu verstehen. Es fällt ihnen so wesentlich leichter den Quelltext zu editieren oder zu erweitern. Damit wird Programmierfehlern vorgebeugt, was letztendlich die Zeit zum Erstellen und Warten der Software sehr positiv beeinflusst. Bei größeren Software-Projekten stellen Dokumentationen des Quelltextes daher eine Notwendigkeit dar und sind somit auch Standard bei der herkömmlichen Software-Entwicklung.
In OOP es gibt verschiedene Methoden Quelltext-Dokumentationen anzufertigen. Eine der am häufigsten genutzten Methoden sind Dokumentationsgeneratoren [FL02], welche aus speziellen Kommentaren im Quelltext eine Quelltext-Dokumentation des ganzen Projekts erzeugen. Allerdings ist es in FOP bisher nicht möglich Dokumentationsgeneratoren einzusetzen. Momentan fehlt FOP-Projekten die Unterstützung für Dokumentationsgeneratoren, da diese nicht die besondere Struktur von FOP berücksichtigen. Dabei ist der Quelltext von FOP-Projekten noch deutlich komplexer als bei OOP-Projekten. Der Quelltext einer FOP-Produktlinie kann gerade bei größeren Projekten mit mehreren Entwicklern schnell unübersichtlich werden. Eine gute Quelltext-Dokumentation ist daher auch dort absolut notwendig.
Die Kombination von FOP und Dokumentationsgeneratoren für Quelltext-Dokumentationen ist somit äußerst wünschenswert. Daher befasst sich diese Bachelorarbeit mit dem Thema, wie der Quelltext von FOP effizient und intuitiv dokumentiert und gleichzeitig für verschiedenen Sichtweisen angepasst werden kann.

Zielstellung der Arbeit

Das Ziel der Arbeit ist es, sowohl für die Anwender von SPL-Produkten, als auch für die Entwickler dieser SPLs effizient Dokumentationen bereitzustellen, damit die Anwender die Funktionalität eines Produkts nachvollziehen und die Entwickler der SPL den Überblick über ihren Quelltext behalten können. Im Rahmen dieser Arbeit bedeutet effizient, dass wenig Mehraufwand für die Entwickler bei der Dokumentation entsteht. Die Dokumentationen sollen außerdem maßgeschneidert für den Anwendungsfall und die jeweilige Zielgruppe sein, damit weder zu viel, noch zu wenig

Informationen in der Dokumentation enthalten sind. Es sollen die Dokumentationen für alle Produkte einer SPL generiert werden können, um den Anwender zu helfen die Produkte in anderen Projekten wiederzuverwenden. Weiterhin sollen auch Dokumentationen über eine gesamte SPL und Teile einer SPL erstellt werden können, sodass sich die Entwickler der SPL besser im Quelltext zurecht finden. Außerdem sollen auch einzelne Feature-Module dokumentiert werden, damit sie leichter in anderen SPLs wiederverwendet werden können, was vor allem für Entwickler dieser Produktlinien hilfreich ist. Im Zuge der Arbeit sollen zwei wichtige Forschungsfragen beantwortet werden.

- Ist es möglich maßgeschneiderte Dokumentationen für verschiedene Anwendungsfälle mit einem einzigen Verfahren zu erzeugen?
- Sind Entwickler mithilfe eines solchen Verfahrens in der Lage effizient Dokumentationen zu erzeugen?

Gliederung der Arbeit

Die Arbeit ist folgendermaßen aufgebaut. In Kapitel 2 werden die Grundlagen über Software-Dokumentationen und SPLs vermittelt. Die Grundlagen der Software-Dokumentationen beinhalten dabei insbesondere die Quelltext-Dokumentation. Im Zusammenhang mit den Quelltext-Dokumentationen wird das Prinzip der Dokumentationsgeneratoren erklärt. Dabei wird insbesondere auf den Dokumentationsgenerator JavaDoc eingegangen. Die SPL-Grundlagen vermitteln das grundsätzliche Prinzip des Aufbaus und der Funktionalität von SPLs. Da diese Arbeit hauptsächlich FOP-Produktlinien betrachtet, wird vor allem FOP als Paradigma zur Implementierung einer SPL näher erläutert. Nach den Grundlagen wird in Kapitel 3 eine Anforderungsanalyse für die Dokumentation einer FOP-Produktlinie durchgeführt. Dabei wird einzeln auf die verschiedenen Anwendungsfälle der Dokumentation eingegangen. Die Anforderungsanalyse zeigt auf, welche Probleme derzeit bei der Verwendung von trivialen Ansätzen bestehen. In Kapitel 4 wird deshalb ein Konzept entwickelt, mit dem sich Dokumentationen für alle Anwendungsfälle erzeugen lassen. Um das Konzept auf seine Eignung und Effizienz zu prüfen, wird in Kapitel 5 beschrieben, wie es in einer prototypischen Implementierung umgesetzt wurde. Anschließend beschreibt Kapitel 6 die Evaluierung der prototypischen Implementierung mittels zweier Beispiel-Produktlinien. Die Evaluierung umfasst dabei sowohl die qualitativen, als auch die quantitativen Merkmale des Konzepts. Damit ist es möglich die beiden zu Beginn gestellten Fragen zu beantworten. In Kapitel 7 werden verwandte Arbeit und Themengebiete vorgestellt. Abschließend werden die Ergebnisse der Arbeit in Kapitel 8 zusammengefasst und auf Themen für zukünftige Arbeiten eingegangen.

2. Grundlagen

Dieses Kapitel vermittelt die Grundlagen der Software-Dokumentation. Dabei wird vor allem auf die Dokumentation des Quelltextes eingegangen. Des Weiteren werden Software-Produktlinien (SPLs) vorgestellt und insbesondere die Feature-orientierte Programmierung (FOP) als Paradigma zur Erstellung von SPLs beschrieben.

2.1 Software-Dokumentation

In diesem Abschnitt wird beschrieben, was unter der Dokumentation einer Software verstanden wird. Es werden die verschiedenen Arten von Software-Dokumentationen kurz vorgestellt und erklärt mit welcher Art sich diese Arbeit auseinandersetzt. In diesem Zusammenhang wird die allgemeine Funktionsweise von Dokumentationsgeneratoren vorgestellt und dabei insbesondere auf den Dokumentationsgenerator JavaDoc eingegangen.

2.1.1 Arten der Software-Dokumentation

Unter einer Software-Dokumentation versteht man die Bündelung aller relevanten Informationen zu einer bestimmten Software [IEE90]. Dazu zählen insbesondere der Zweck und die besonderen Eigenschaften der Software, die Details zur korrekten Bedienung, die Details zur Implementierung und Informationen über den Prozess der Software-Entwicklung [PB96].
Die in einer Dokumentation enthaltenen Informationen richten sich vorrangig an drei verschiedene Zielgruppen. Diese Zielgruppen bestehen aus den Anwendern und den Entwicklern der Software und aus den Managern des Projekts. Da für jede Zielgruppe unterschiedliche Informationen von Interesse sind, wird die gesamte Dokumentation auf die drei Zielgruppen aufgeteilt. Die unterschiedlichen Arten der Dokumentation umfassen die Benutzerdokumentation, die Systemdokumentation und die Projekt- beziehungsweise Prozessdokumentation [PB96, Som92]. Während Benutzer- und Systemdokumentation Informationen über das fertige Software-Produkt enthalten, gibt die Projektdokumentation Auskunft über die Entstehung des Software-Produkts.

Die Projektdokumentation stellt Informationen für die Manager des Software-Projekts bereit. Sie beschreibt den Prozess den Software-Entwicklung und enthält hauptsächlich organisatorische Informationen. Darunter fallen der Projektplan, der Organisationsplan und das Projektlogbuch. Mithilfe diese Dokumentationsart lassen sich die Ziele und Aufgaben des Projekts, sowie der Projektfortschritt festhalten [PB96, Som92].
Die Benutzerdokumentation richtet sich vor allem an die Anwender des Software-Produkts. Diese enthält alle Informationen, die nötig sind, damit das Software-Produkt von Personen ohne Hintergrundwissen darüber ordnungsgemäß verwendet werden kann. Unter diese Dokumentationsart fällt unter anderem die Installations- und die Bedienungsanleitung [PB96, Som92].
Die Systemdokumentation enthält wesentliche Informationen für die Entwickler des Software-Produkts, wie die Systemspezifikationen und die Quelltext-Dokumentation [PB96, Som92], mit welcher sich diese Arbeit hauptsächlich beschäftigt. Die Quelltext-Dokumentationen dienen dazu den Quelltext von Software-Produkten und deren Funktionsweisen zu beschreiben. Sie können außerdem noch zusätzliche Informationen, wie zum Beispiel den Autor und die Version des dokumentierten Quelltext enthalten. Die Quelltext-Dokumentation richtet sich an die Entwickler des Software-Produkts und an Entwickler, welche das Produkt für andere Projekte benutzen möchten. Das Ziel einer solchen Dokumentation ist das leichtere Verständnis des Quelltextes. Ein Entwickler kann mit Hilfe einer guten Quelltext-Dokumentation den Sinn und die Struktur des Quelltextes verstehen ohne diesen komplett zu lesen. Dieses Umstand erspart wiederum viel Zeit und beugt Programmierfehlern vor.

2.1.2 Dokumentationsgeneratoren

Ein viel genutzter Weg, um Quelltext-Dokumentationen zu erstellen sind Dokumentationsgeneratoren [FL02]. Diese Werkzeuge erzeugen aus der Programmstruktur und speziellen Dokumentationskommentaren im Quelltext den Dokumentationstext. Zudem können abhängig vom verwendeten Tool auch verschiedene Ausgabeformate erzeugt werden. So kann die Dokumentation unter anderem als reiner Text, HTML oder PDF generiert werden. Bekannte Dokumentationsgeneratoren sind unter anderem JavaDoc für die Sprache Java [Ora14] und Doxygen für die Sprachen C++, Java, C#, PHP oder Python [vH14]. Beide Generatoren sind sich konzeptuell sehr ähnlich und generieren auch ähnliche Dokumentationen.
Im späteren Verlauf der Arbeit zeigt sich, dass sich das vorgestellte Konzept prototypisch auf Software-Produktlinien, welche mit Java implementiert wurden stützt. Da JavaDoc direkt für Java entwickelt wurde, wird im weiteren Verlauf der Arbeit deshalb der Fokus auf diesen Dokumentationsgenerator gelegt.
Bei JavaDoc handelt es sich um einen in Java geschriebenen Dokumentationsgenerator. Dieser befindet sich momentan in der Version 1.5 und ist im Java Development Kit (JDK) von Oracle enthalten [Ora14]. JavaDoc ist das Standardwerkzeug zur Dokumentation von Java-Quelltext. Es wird in populäreren Entwicklungsumgebungen (Integrated Development Environment) (IDEs) wie Eclipse[1] und Netbeans[2] unterstützt.

[1] https://www.eclipse.org/
[2] https://netbeans.org/

Der Quelltext wird dabei direkt mit JavaDoc-Kommentaren annotiert, welche jeweils die Funktionalität eines Quelltext-Elements beschreiben, wobei es sich um Pakete, Klassen, Methoden oder auch Felder handeln kann. Die Kommentare können in Ausnahmefällen aber auch in separaten Dateien enthalten sein, dies zum Beispiel der Fall bei den Kommentaren für Pakete, welche in die Datei package.html oder package-info.java geschrieben werden.
JavaDoc-Kommentare beginnen mit der Zeichenfolge `/**` und enden mit `*/`. Im folgenden Java Beispiel in Abbildung 2.1 wird die Klasse `Encryption` mit ihren Methoden `decrypt` und `encrypt` durch einen JavaDoc-Kommentar beschrieben. Da der gezeigte Quelltext nicht sonderlich komplex ist, kann die Funktionalität leicht nachvollzogen werden. Es lässt sich jedoch auch erkennen, dass die Funktionalität durch die Kommentare wesentlich schneller erfasst werden kann.

```
1  /**
2   * Provides methods for encrypting and decrypting strings.
3   *
4   * @author Autor 1
5   * @author Autor 2
6   * @version 1.0
7   */
8  public class Encryption {
9    /**
10    * Decrypts a string, which is encrypted
11    * by the ROT13-algorithm.
12    *
13    * @param code the encrypted string
14    * @return the decrypted string
15    * @see Encryption#encrypt(String)
16    */
17   public static String decrypt(String code) {
18     char[] chars = code.toCharArray();
19     for (int i = 0; i < chars.length; ++i) {
20       chars[i] = (char) (chars[i] - 13);
21     }
22     return String.valueOf(chars);
23   }
24   public static String encrypt(String text) { ... }
25 }
```

Abbildung 2.1: JavaDoc Beispiel anhand eines Ausschnitts der Klasse `Encryption` in der Chat-Produktlinie

Zu Beginn eines JavaDoc-Kommentars steht ein allgemeiner Beschreibungstext (Zeile 2 und 10–11), um die Funktion des Quelltext-Elements zu beschreiben. Unter dem Beschreibungstext folgen die JavaDoc-Block-Tags, die spezielle Eigenschaften des Elements beschreiben. Alle Tags werden mit einem `@`-Symbol eingeleitet. Im Kommentar der Klasse `Encryption` erscheinen zwei verschiedene Tags. Der `@author`-Tag (Zeile 4 und 5) gibt jeweils einen Entwickler der Klasse an. Die Version der Klasse

wird durch den `@version`-Tag (Zeile 6) angegeben. In den Kommentaren zur Methode `decrypt` finden sich noch weitere Tags. Für Methoden insbesondere der `@param`- und der `@return`-Tag interessant. Jeweils ein Parameter einer Methode wird durch den `@param`-Tag beschrieben. So beschreibt der `@param`-Tag im Beispiel den Parameter `code` (Zeile 13). Hat die Methode einen Rückgabewert, so kann dieser durch den `@return`-Tag näher erklärt werden, wie im Beispiel in Zeile 14 zu sehen ist. Der `@see`-Tag referenziert ein anderes Element im Quelltext. Im Falle des Beispiels referenziert der `@see`-Tag in Zeile 15 die Methode `encrypt` derselben Klasse. In der fertigen Dokumentation verweist an dieser Stelle ein Link auf den Kommentar zu diesem Element. Auch die von einer Methode geworfenen Ausnahmen (Exceptions) können mittels JavaDoc-Tags beschrieben werden. Dies kann mithilfe des `@throws`-Tag geschehen, der jeweils eine Ausnahme einer Methode beschreibt. In früheren JavaDoc-Version wurde statt des `@throws`- der `@exception`-Tag verwendet.
Neben den oben vorgestellten Tags gibt es noch verschiedene weitere, welche ebenfalls eingesetzt werden können. Der `@since`-Tag macht Angaben darüber, seit welcher Version das Element im Quelltext existiert. Bei der Verwendung der `@Deprecated`-Annotation können durch den `@deprecated`-Tag Angaben gemacht werden, welches Element in Zukunft verwendet werden soll. Die Tags `@serial`, `@serialData` und `@serialField` beschreiben jeweils serialisierbare Objekte.
Wie leicht zu erkennen ist, eignen sich nicht alle Tags für alle Element des Quelltextes. So macht es beispielsweise keinen Sinn einen `@param`-Tag für ein Feld oder eine Klasse zu verwenden. Alle Standard-Tags und bei welchen Elementen sie verwendet werden können, sind in der Tabelle 2.1 aufgelistet. Für diese Arbeit sind dabei nur die Block-Tags von JavaDoc interessant. Die Inline-Tags, wie `{@link}` oder `{@inheritDoc}`, welche zur weiteren Strukturierung und Formatierung der JavaDoc-Kommentare genutzt werden können, werden nicht betrachtet, da diese das in dieser Arbeit entwickelte Konzept nicht beeinflussen.

Tag	**Paket**	**Klasse**	**Konstruktor**	**Methode**	**Feld**
`@author`	•	•			
`@deprecated`		•	•	•	•
`@param`			•	•	
`@return`				•	
`@see`	•	•	•	•	•
`@serial`	•	•			•
`@serialData`			•	•	
`@serialField`					•
`@since`	•	•	•	•	•
`@throws`			•	•	
`@version`	•	•	•	•	

Tabelle 2.1: Übersicht über die Verwendung der JavaDoc-Standard-Tags
(• = Kann in dem entsprechenden JavaDoc-Kommentar verwendet werden)

JavaDoc erzeugt aus den JavaDoc-Kommentaren die Quelltext-Dokumentation einer Java-Anwendung. Dafür werden Doclets benutzt, welche aus einer Menge von Java-Klassen bestehen und die Aufgabe haben, die JavaDoc-Kommentare zu inter-

pretieren [Ora14]. Die Doclets bestimmen, wie mit den einzelnen Tags umgegangen wird, wie die Ausgabe aussieht und auch welches Ausgabeformat verwendet wird. In der Regel wird das direkt in JavaDoc integrierte Standard-Doclet verwendet. Dieses unterstützt alle Standard-Tags (siehe Tabelle 2.1) und erzeugt eine aus mehreren HTML-Dokumenten bestehende Dokumentation. Es existieren zahlreiche weitere Doclets, sowohl von Oracle selbst, als auch von dritten Anbietern [Ora14].

2.2 Software-Produktlinien

Dieser Abschnitt vermittelt die Grundlagen von Software-Produktlinien (SPLs). Dazu zählen die Motivation, die Struktur und die verschieden Paradigmen zur Umsetzung von SPLs [ABKS13, PBvdL05]. Insbesondere wird dabei auf das Paradigma der Feature-orientierten Programmierung (FOP) eingegangen, da sich das Konzept dieser Arbeit auf derartige Produktlinien bezieht.

2.2.1 Grundlagen der Software-Produktlinien

Für den Begriff der SPL existieren viele unterschiedliche Definitionen, die jedoch im Kern das Gleiche ausdrücken. Clements und Northrop definieren SPLs folgendermaßen [CN06]:
„A software product line is a set of software-intensive systems sharing a common mangaged set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“
Zusammengefasst besteht eine SPL aus einer Menge von Software-Produkten mit bestimmten Eigenschaften. Alle Produkte bauen auf einer gemeinsamen Quelltext-Basis auf und sind alle auf die gleiche Weise mit derselben Technik entwickelt wurden. Weiterhin zielt ihr Zweck auf eine bestimmte Domäne, also eine konkrete Aufgabe oder ein bestimmtes Marktsegment ab.
SPLs wurden entwickelt, um eine ausreichend große Variabilität innerhalb einer Domäne zu erreichen und somit allen Anforderungen von potentiellen Anwendern gerecht zu werden. Dabei sollen die Kosten und der Aufwand, um diese Variabilität zu erreichen möglichst gering ausfallen. Zur Realisierung dieses Ziels werden einzelne Elemente der SPL unabhängig voneinander entwickelt. Diese Software-Elemente werden als Feature bezeichnet und miteinander kombiniert, um ein bestimmtes Produkt zu generieren. Kang et al. definieren ein Feature folgendermaßen [KCH+90]:
„Features are user-visible aspects or characteristics of the domain.“
Feature bilden die Grundlage der SPLs, da auf ihnen die gesamte Variabilität beruht. Jedes Produkt einer SPL besteht aus einer Teilmenge aller Feature. Für die Nutzer von Produkten, die mittels einer SPL erstellt wurden sind Feature wahrnehmbare Merkmale in der Domäne der SPL.
Der Aufwand beim Entwickeln einer Produktlinie besteht also darin die einzelnen Feature zu entwickeln und dafür zu sorgen, dass diese dynamisch miteinander kombiniert werden können. Daher muss zu Beginn der Entwicklung einer SPL mehr Aufwand betrieben werden, als bei der Entwicklung eines einzigen Software-Produkts. Langfristig ist jedoch ein SPL der konventionellen Entwicklung von mehreren Software-Produkten überlegen. Sowohl die Kosten für einzelne Produkte können gesenkt werden, als auch die Zeit, welche die neuen Produkte zur Markteinführung

benötigen. Dies zeigt sich im Vergleich von SPLs und einzelnen Systemen in den folgenden Diagrammen. Die summierten Kosten für die Entwicklung neuer Software-Systeme beginnen bei einer SPL, bedingt durch die anfänglichen Entwicklungskosten zwar an einem höheren Punkt, haben aber einen wesentlich geringeren Anstieg, wie in Abbildung 2.2 zu sehen ist. In Abbildung 2.3 ist zu erkennen, dass die Zeit bis zur Markteinführung neuer Produkte für eine SPL im Laufe der Zeit deutlich geringer ist, als bei der Entwicklung einzelner Systeme. Gibt es eine neue Anforderungen, die an die SPL gestellt wird, so muss lediglich ein neues Feature entwickelt werden.
Weitere Vorteile einer SPL liegen in der großen Variabilität innerhalb der Domäne, wodurch sich die Anwender der Produkte, ihr individuelles Produkt maßschneidern können. Durch die unterschiedliche Kombination der Feature wird eine hohe Produktvariabilität erreicht und die Produkte der SPL können sehr einfach an die Anforderungen der Anwender angepasst werden. Somit können sich die Anwender genau die Funktionalität zusammenstellen, die sie benötigen und ihre Produkte nach gewissen Gesichtspunkten optimieren. Ziele für die Anwender können dabei Performancegewinn, Speicherplatzeinsparung oder Komplexitätsreduktion ihres Produkts sein.

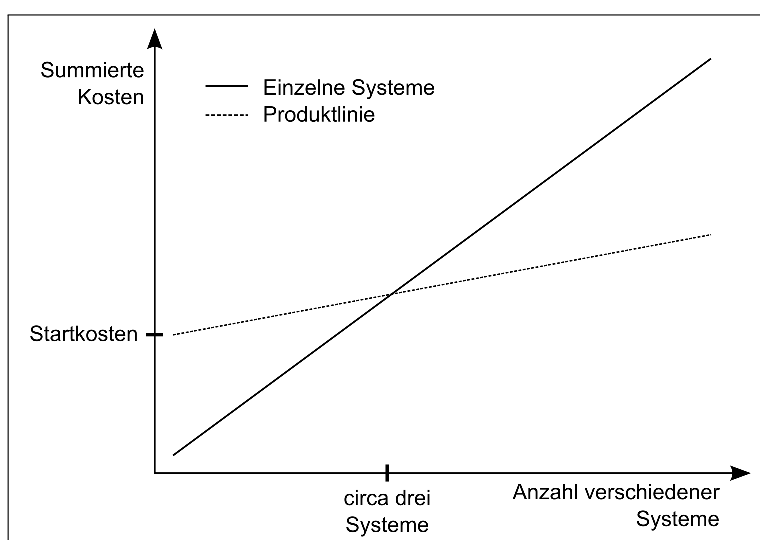

Abbildung 2.2: Vergleich der Projektkosten zwischen Produkten einer SPL und einzelnen Systemen [PBvdL05]

2.2.2 Feature-Modellierung

Feature sind in den meisten Fällen abhängig von einander. Ihre Beziehungen untereinander müssen daher bei der Entwicklung einer SPL und der Erstellung derer Produkte berücksichtigt werden. Die Feature-Modellierung bezeichnet den Prozess der Erstellung eines Feature-Modells zu einer SPL. Sinn und Zweck des Feature-Modells ist die Darstellung der Beziehungen der Feature untereinander und die Beschreibung der Variabilität der Produktlinie. Feature-Modelle bieten die Möglichkeit die Abhängigkeiten der Feature abstrakt zu beschreiben, ohne dabei Implementierungsdetails berücksichtigen zu müssen [CE05].
Um ein Produkt einer SPL zu erzeugen, wird wie in Abschnitt 2.2.1 erwähnt eine Auswahl von Features benötigt. Eine solche Teilmenge von allen Featuren wird als

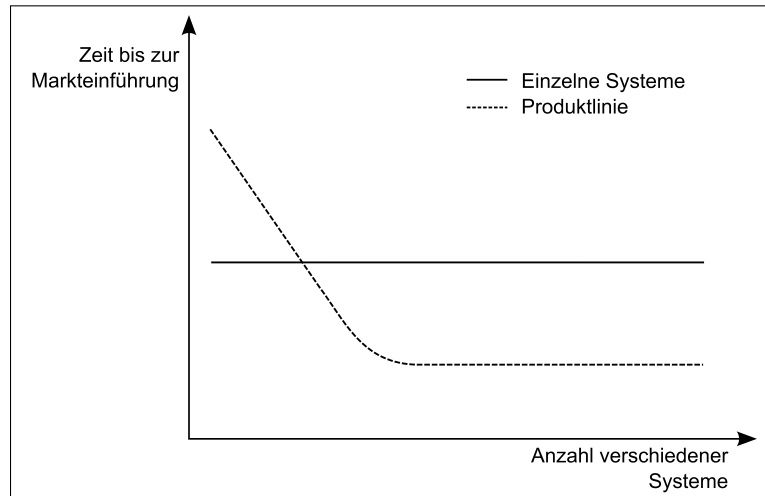

Abbildung 2.3: Vergleich der Zeiten bis zur Markteinführung zwischen Produkten einer SPL und einzelnen Systemen [PBvdL05]

Konfiguration bezeichnet. Wenn eine Konfiguration allen im Feature-Modell dargestellten Abhängigkeiten genügt, wird sie als gültig bezeichnet. Eine gültige Konfiguration entspricht somit genau einem Produkt der SPL.
Das Feature-Diagramm ist eine grafische Darstellung des Feature-Modells und stellt die Feature-Beziehungen als gerichteten Baum dar [CE05]. Die Feature werden durch die Knoten des Baumes repräsentiert, wobei der Wurzelknoten die gemeinsame Basis aller Produkte der SPL darstellt.
Durch die Position der Feature innerhalb des Baumes und durch die verschiedenen Kantenarten werden die Beziehungen dargestellt. Grundsätzlich gilt, dass die Feature in tieferen Ebenen des Baumes abhängig sind von ihren Eltern. Das bedeutet, dass ein Feature nur ausgewählt werden kann, wenn auch der Elternknoten gewählt ist. Die genaue Art der Abhängigkeit ist durch die verschiedenen Kantenarten festgelegt. So können durch verschiedenen Kantenarten obligatorische Feature, optionale Feature, Alternativen und ODER-Gruppen dargestellt werden. Im Nachfolgenden werden die verschiedenen Kantenarten und ihre Bedeutung erklärt. Dazu wird das in Abbildung 2.4 dargestellte Feature-Modell einer Chat-Produktlinie verwendet, welche in dieser Arbeit noch mehrfach als Beispiel dienen wird. Die Chat-Produktlinie besteht aus ihrem Basis-Feature *Chat* und den 12 weiteren Featuren *History*, *Spam*, *Display*, *Security*, *UserCommands*, *GUI*, *Console*, *Encryption*, *Authentication*, *Color*, *ROT13* und *SWL*.
Obligatorische Feature müssen genau dann ausgewählt werden, wenn auch ihr übergeordnetes Feature ausgewählt wird. *Display* ist ein obligatorisches Feature und muss in jedem Produkt verwendet werden, da das übergeordnetes Feature *Chat* die Basis der Produktlinie ist.
Die Feature *History*, *Spam*, *Security*, *UserCommands* und *Color* sind optionale Feature. Diese Feature können nur ausgewählt werden, wenn auch deren übergeordnetes Feature ausgewählt ist. Allerdings besteht kein Zwang optionale Feature auszuwählen.
Von allen Featuren in einer Alternative muss genau ein Feature ausgewählt werden. Die Feature *GUI* und *Console* befinden sich in einer Alternative, sowie auch die

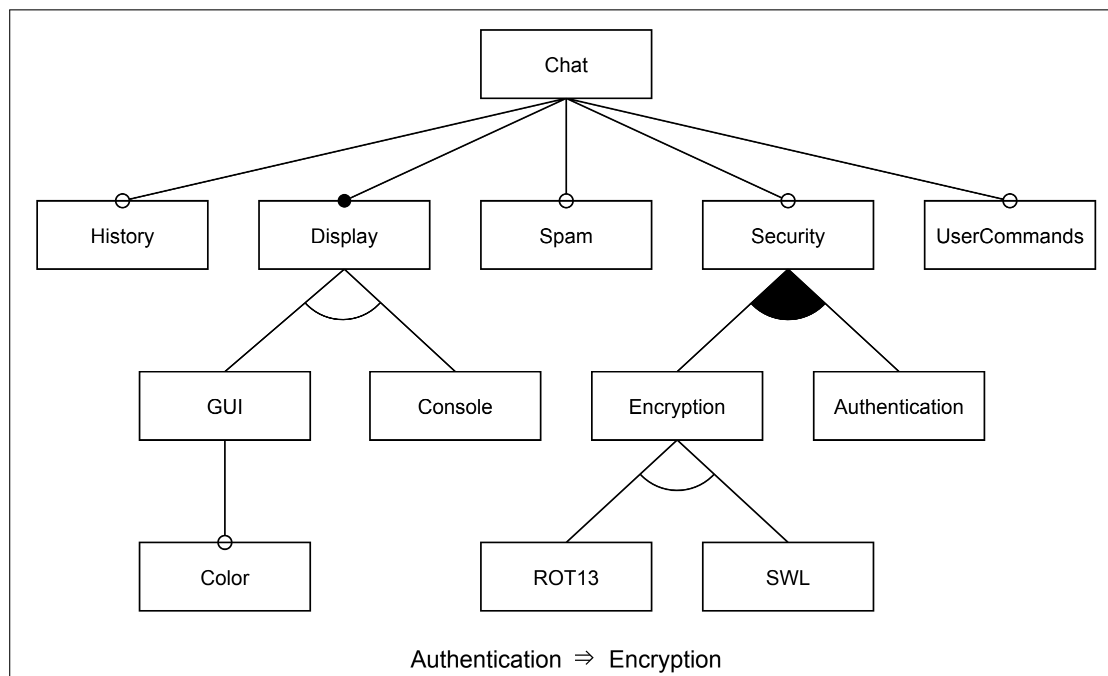

Abbildung 2.4: Feature-Modell einer einfachen Chat-Produktlinie

Feature *ROT13* und *SWL*. *Display* ist ein obligatorisches Feature und muss, da es ein Kind des Basis-Features ist immer gewählt werden. Daraus folgt, dass immer auch genau eines der beiden Feature *GUI* oder *Console* gewählt werden muss.
Die Feature *Encryption* und *Authentication* befinden sich in einer ODER-Gruppe. Ist das übergeordnete Feature einer ODER-Gruppe gewählt, so muss auch mindestens eines der Feature in der Gruppe gewählt werden. Es können aber auch beliebig viele Weitere ausgewählt werden. Wird also das optionale Feature *Security* ausgewählt, so muss auch mindestens eines der beiden Feature *Encryption* oder *Authentication* ausgewählt werden.
Ein weiteres Element von Feature-Modellen sind zusätzliche aussagenlogische Ausdrücke, um ein einfaches Feature-Modell weiter zu beschränken. Im Feature-Modell der Chat-Produktlinie wird der aussagenlogische Ausdruck `Authentication` ⇒ `Encryption` verwendet, welcher besagt, dass wenn das Feature *Authentication* ausgewählt wurde, auch das Feature *Encryption* ausgewählt werden muss.

```
1  Chat ∧ Display
2   ∧ (GUI ⊻ Console)
3   ∧ (Color ⇒ GUI)
4   ∧ (Security ⇔ (Encryption ∨ Authentication))
5   ∧ (Encryption ⇔ (ROT13 ⊻ SWL))
6   ∧ (Authentication ⇒ Encryption)
```

Abbildung 2.5: Aussagenlogischer Ausdruck für das Feature-Modell der Chat-Produktlinie

Aussagenlogische Ausdrücke sind eine andere Form der Feature-Modellierung. Ein Feature-Modell kann auch durch einen einzigen Aussagenlogischen Ausdruck dargestellt werden. Dafür werden die einzelnen Feature als Variablen betrachtet und eine

gültige Belegung der Variablen als gültige Konfiguration angesehen. Für die Chat-Produktlinie ist ein zum Feature-Modell äquivalenter aussagenlogischer Ausdruck in Abbildung 2.5 dargestellt. Jeder aussagenlogische Ausdruck kann als konjunktive Normalform dargestellt werden. Diese Darstellungsform hat den Vorteil, dass sie leicht vom Computer erfasst werden kann. Daher werden aussagenlogische Ausdrücke für Analysen der Variabilität einer Produktlinie verwendet.

2.2.3 Feature-Orientierte Programmierung

Wie in Abschnitt 2.2.1 erwähnt, existieren verschiedene Paradigmen, die eine Implementierung von SPLs ermöglichen [ABKS13, PBvdL05]. Im Allgemeinen gibt es annotative und kompositionale Paradigmen [KAK08]. Zu den annotativen Paradigmen zählen alle Präprozessor-Techniken, wie der C/C++-Präprozessor, Antenna[3] oder Munge[4]. Diese ermöglichen durch Annotationen im Quelltext einer Produktlinie die Aufteilung der Funktionalität in mehrere Feature. Ein Merkmal dafür ist, dass der Quelltext für ein Feature nicht in einer eigenen Datei separiert ist, sondern alle Annotationen einer Klasse in der selben Datei enthalten sind. Hingegen zählen die Aspekt-orientierte Programmierung (AOP) [KLM+97] und die Feature-orientierte Programmierung (FOP) zu den kompositionalen Paradigmen [ABKS13]. Diese zeichnen sich dadurch aus, dass sie den Quelltext auf die einzelnen Feature aufteilen und so zu einer Trennung der Funktionalität führen. Der kompositionale Ansatz FOP bietet zur Zeit die meisten Vorteile für die Erstellung eine SPL, im Vergleich zu den anderen Paradigmen. So bietet FOP eine feine Granularität und ist damit in der Lage den Quelltext flexibel zu erweitern. Bei FOP gibt es eine klare Trennung der Features, wodurch die Übersichtlichkeit steigt. Dies hilft sowohl beim Entwickeln, als auch bei der Fehlersuche und Wartung der SPL. Außerdem besitzt FOP viele Parallelen zu der Objekt-orientierten Programmierung (OOP), da die Verfeinerung von Klassen und Methoden stark der Vererbung in OOP gleichen. Daher findet FOP eine große Verbreitung unter den SPL-Paradigmen. Diese Arbeit beschäftigt sich deshalb mit Produktlinien, die mit FOP entwickelt wurden.
Die Grundidee von FOP ist die Zerlegung des gesamten Quelltextes der SPL in Feature-Module, welche jeglichen Quelltext kapseln, der zur Implementierung eines Features gehört [Pre01]. Weiterhin beinhaltet ein Feature-Modul auch alle anderen Ressourcen, die zu einem Feature gehören, wie zum Beispiel Bild-, Ton- oder Text-Dateien. Bei der Generierung eines Produkts werden die Feature-Module aller in der Konfiguration gewählten Feature in einer bestimmten Reihenfolge wieder zusammengesetzt.
Der Aufbau einer FOP-Produktlinie folgt dem so genannten Kollaborations-Design [AR92]. Kollaborationen bestehen aus einer Menge von Klassen, in denen die Funktionalität eines Feature implementiert ist [ABKS13]. Damit entsprechen die Kollaborationen den Feature-Modulen. Die Feature verlaufen orthogonal zu den Klassen einer SPL. Die Überschneidung eines Feature mit einer Klasse wird Rolle genannt. Alle Rollen eines Feature zusammen ergeben ein ganzes Feature-Modul beziehungsweise eine ganze Kollaboration. Dieses Prinzip lässt sich am leichtesten in einem Kollaborationsdiagramm, welches eine grafische Repräsentation des

[3] http://antenna.sourceforge.net/index.php
[4] https://sonatype.github.io/munge-maven-plugin/

Kollaborations-Design einer Produktlinie darstellt veranschaulichen. Tabelle 2.2 zeigt einen Ausschnitt des Kollaborationsdiagramms für die Chat-Produktlinie (Abbildung 2.4). Dem Kollaborationsdiagramm kann beispielsweise entnommen werden,

	Klassen			
Feature	`Client`	`Server`	`TextMessage`	`Encryption`
Chat	•	•	•	
Display	•			
UserCommands	•	•	•	
Encryption	•		•	•
ROT13				•
SWL				•

Tabelle 2.2: Ausschnitt des Kollaborationsdiagramms der Chatproduktlinie
(• = Dieses Feature besitzt eine Rolle für die entsprechende Klasse)

dass die Kollaboration für das Feature *Chat* drei Rollen besitzt, jeweils für die Klasse `Client`, `Server` und `TextMessage`. Außerdem lässt sich ablesen, dass die Klasse `Server` von den beiden Featuren *Chat* und *UserCommands* benutzt wird.
Ein Feature kann also mehrere Klassen verwenden, um seine Funktionalität zu implementieren. Dabei wird jede Rolle in einer eigenen Datei abgelegt und ist damit separiert von anderen Klassen und Featuren.
In einer Rolle kann es Einführungen und Verfeinerungen von Klassen, Feldern und Methoden geben. Die Einführung einer Klasse, eines Feldes oder einer Methode unterscheidet sich nicht von der in OOP-Projekten üblichen Deklaration und Implementierung. Verfeinerungen erweitern die Funktionalität von vorher eingeführten Klassen, Feldern und Methoden. Bei einer Verfeinerung wird eine Klasse, ein Feld oder eine Methode in einer weiteren Rolle nochmals deklariert. Dieses Prinzip wird dazu genutzt neue Funktionalität hinzuzufügen und hat je nachdem welches Element verfeinert wird andere Möglichkeiten.
Für Klassen besteht die Möglichkeit neue Methoden, Felder und innere Klassen hinzuzufügen. Bei der Verfeinerung eines Feldes kann der initiale Werte verändert werden. Bei einer Methodenverfeinerung kann ein Feature eine Methode vollständig neu implementieren und damit die Methode neu einführen oder die Funktionalität der alten Methode benutzen und erweitern. Für die Erweiterung einer Methode wird in der verfeinernden Methode ein Schlüsselwort wie `original` oder `Super` verwendet. Diese Schlüsselwort drückt aus, dass an dieser Stelle die Funktion der verfeinerten Methode ausgeführt wird. Findet eine Neueinführung durch zwei Feature statt, bei welchen die Auswahl des einen Features nicht die Auswahl des anderen Features bedingt und umgekehrt, so wird dies als parallele Einführung bezeichnet.
Um Produkte einer FOP-Produktlinie zu generieren, müssen die Feature-Module beziehungsweise die Rollen wieder zusammengefügt werden. Dies geschieht in FOP durch eine schrittweise Komposition der Rollen, mittels des Prinzips der Verfeinerung. Für die praktische Umsetzung der Verfeinerungen existieren mehrere Alternativen. Eine Möglichkeit ist die Nutzung der Vererbungshierarchie von OOP, wobei für jede Rolle eine neue Klassen erstellt wird, welche Rollen, die diese verfeinern beerbt. Eine andere Möglichkeit ist das Inlining, bei dem für alle Rollen zu einer

Klasse auch nur eine einzige Klasse angelegt wird und alle Felder und Methoden der Rollen in diese Klasse übernommen werden. Bei Methodenverfeinerungen wird das entsprechende Schlüsselwort, wie zum Beispiel `original` durch die jeweilige originale Methode ersetzt.
Natürlich spielt bei der Komposition der Feature-Module die Reihenfolge eine entscheidende Rolle. Der Entwickler, kann selbst entscheiden, in welcher Reihenfolge seine Feature-Module komponiert werden sollen und muss daher darauf achten, dass die von ihm gewählte Feature-Reihenfolge auf die Feature-Module abgestimmt ist.
Im weiteren Verlauf der Arbeit wird für die Begriffe Klasse, Methode und Feld zusammenfassend der Begriff der Signatur verwendet. Signaturen bestehen aus den Deklarationen von Klassen, Feldern und Methoden und sind eindeutig innerhalb einer SPL. Durch die Verfeinerungen in den Feature-Modulen ist es möglich bei jeder Deklaration auch einen entsprechenden JavaDoc-Kommentar anzugegeben. Das bedeutet, dass es mehrere Kommentare zu einer Signatur geben kann.
Für FOP existieren schon eine Reihe von ausgereiften Modellen für verschiedene Objekt-orientierte Sprachen, wie AHEAD [Bat04] und FeatureHouse [AKL13] für die Sprache Java und FeatureC++ [ALRS05] für die Sprache C++. FOP lässt sich neben Objekt-orientierten Sprachen aber auch auf andere Sprachparadigmen, wie etwa auf die funktionale Programmierung anwenden [AKL13].
Es wurde bereits erwähnt, dass Feature-Module mehr als nur Quelltext-Dateien enthalten können. Bei der Komposition der Feature-Module greift daher das Prinzip der Uniformität. Es besagt, dass die Implementierung eines Features nicht nur aus Quelltext besteht, sondern sich aus diversen Artefakten einer SPL zusammensetzen kann [BSR04]. Das bedeutet, dass jedes beliebige Artefakt einer SPL durch Feature verfeinert werden kann. Unter anderem stellen die Dokumentationskommentare in einer SPL solch ein Artefakt dar. Weshalb die Anwendung des Uniformitätsprinzips dennoch nicht für die Dokumentation einer SPL ausreicht, wird in Kapitel 3 eingehend erläutert.
Ein neuer Aspekt von FOP-Produktlinien sind die so genannten Kontext Interfaces [SSTS14]. Ein Kontext-Interface ist eine spezielle Sicht auf die SPL und zeigt alle Signaturen, die in einer partiellen Konfiguration der SPL enthalten sind. Partielle Konfigurationen entstehen, indem das Feature-Modell schrittweise spezialisiert wird [CHE05]. Das bedeutet, es wird zunächst das Basis-Feature ausgewählt und alle weiteren, sich direkt daraus ergebenen Feature. Im Falle der Chat-Produktlinie wird demnach das Basis-Feature *Chat* gewählt und auch *Display*, da dies ein obligatorisches Feature und ein direktes Kind von *Chat* ist. Nun können der partiellen Konfiguration weitere Feature hinzugefügt werden, solange dies nicht zu einem Widerspruch mit den Abhängigkeiten des Feature-Modell führt. Zum Beispiel stellt die Auswahl der Feature *Chat*, *Display* und *Authentication* eine partielle Konfiguration der Chat-Produktlinie dar. Das Kontext-Interface kann unter anderem dazu verwendet werden, alle Signaturen anzuzeigen, die von einer bestimmten Rolle aus erreichbar sind. Dafür wird ein Kontext-Interface mit einer partiellen Konfiguration verwendet, in der neben dem Basis-Feature und allen daraus folgenden Featuren noch das einzelne Feature für die entsprechende Rolle ausgewählt wurde.

Beispiele für Quelltext in Feature-Modulen

Um das Prinzip von FOP noch einmal zu veranschaulichen, werden nachfolgend zwei Quelltextausschnitte aus der Chat-Produktlinie vorgestellt. Diese werden im weiteren Verlauf der Arbeit noch mehrfach als Beispiele aufgegriffen, um die Verwendung von JavaDoc-Kommentaren zu verdeutlichen. Die Chat-Produktlinie ist mit FeatureHouse erstellt wurden und daher in der Sprache Java implementiert. Bei den beiden Quelltextausschnitten handelt es sich um die beiden Methoden `incomingAction` und `sendMessage` aus der Klasse `Client` in jeweils mehren verschiedenen Feature-Modulen. Beide Beispiele sind keine trivialen Fälle für eine Quelltext-Dokumentation und eignen sich daher gut, um die in dieser Arbeit vorgestellten Probleme und Lösungsansätze zu veranschaulichen.
Die Methode `incomingAction` ist in den drei verschiedenen Featuren *Chat*, *Encryption* und *Spam* implementiert. In Abbildung 2.6 ist für jedes dieser drei Feature der entsprechende Quelltextausschnitt abgebildet. Die Methode wird in *Chat* ein-

```
1  private Message incomingAction(Message msg) {            Chat
2    return msg;
3  }
4  private Message incomingAction(Message msg) {      Encryption
5    if (msg.isEncoded()) {
6      msg.setContent(Encryption.decrypt(msg.getContent()));
7      msg.setEncoded(false);
8    }
9    return original(msg);
10 }
11 private Message incomingAction(Message msg) {            Spam
12   if (SpamFilter.filter(msg)) {
13     return null;
14   } else {
15     return original(msg);
16   }
17 }
```

Abbildung 2.6: Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie

geführt (Zeile 1–3) und dient hier als Hook-Methode. Das bedeutet, dass diese bei der Einführung keine praktische Funktion erfüllt, sondern nur dem Zweck dient von anderen Featuren verfeinert zu werden. Dieser Trick wird genutzt, um Featuren die Möglichkeit zu geben die Funktionalität des Quelltextes an einer bestimmten Stelle zu verändern oder zu erweitern. In diesem Falle, verfeinert das Feature *Encryption* die Methode `incomingAction`, sodass diese alle eingehenden, verschlüsselten Nachricht entschlüsselt (Zeile 4–10). Das Feature *Spam* verfeinert die Methode ebenfalls, sodass diese alle eingehenden Nachrichten auf Spam überprüft und gegebenenfalls verwirft (Zeile 11–17). Eine Verfeinerung lässt sich in FeatureHouse leicht an der Benutzung des Schlüsselwortes `original` erkennen (Zeile 9 und 15). Das Beispiel verdeutlicht ebenso gut, dass die Feature-Reihenfolge eine große Rolle spielt. Werden die Feature so komponiert, dass nach der Einführung in *Chat* zunächst *Encryption*

und danach *Spam* verfeinert, so könnte der Spam-Filter nicht richtig arbeiten, da Nachrichten zum Teil noch verschlüsselt sein können.
Bei dem zweiten Beispiel handelt es sich um die Methode `sendMessage`, welche in Abbildung 2.7 in den Featuren *Chat* und *UserCommands* dargestellt ist. Das Feature *Chat* führt die Methode ein und implementiert die Funktion, dass eine gegebene Zeichenfolge in ein Nachrichtenobjekt umgewandelt und an den Chat-Server weiter geschickt wird. Im Feature *UserCommands* wird die Funktion um die Möglichkeit erweitert Befehle an den Server zu schicken, indem ein `/`-Zeichen an den Anfang der Nachricht gesetzt wird.

```
1  public static void sendMessage(String line) {            Chat
2    if (canSend()) {
3      sendObject(toTextMessage(line));
4    }
5  }
6  public static void sendMessage(String line) { UserCommands
7    if (line.startsWith("/")) {
8      ...
9    } else {
10     original(line);
11   }
12 }
```

Abbildung 2.7: Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

2.3 Zusammenfassung

In diesem Kapitel wurden zwei Konzepte des Software-Engineering vorgestellt. Zum einen die Software-Dokumentationen und zum anderen die Feature-orientierte Programmierung (FOP). Beide Konzepte haben auf den ersten Blick wenig miteinander zu tun. Jedoch ist eine gute Software-Dokumentation absolut notwendig um gute Software-Produkte zu erstellen und FOP wird immer häufiger genutzt um Software-Produkte zu entwickeln. Bislang existiert aber noch keine ausreichende Unterstützung von Software-Dokumentationen bei der Verwendung von FOP. Daher ist es ein Ziel dieser Arbeit ein Konzept zu entwickeln, sodass beide Konzepte zusammen verwendet werden können.
FOP ist ein gutes Konzept, um innerhalb einer Domäne große Variabilität zu erreichen. Es ergeben sich durch die Verwendung von FOP einige wesentliche Vorteile. Langfristig senkt die Entwicklung einer SPL die Kosten und die Zeit zur Entwicklung neuer maßgeschneiderter Produkte. Weiterhin profitieren die Nutzer von SPL-Produkten von der maßgeschneiderten Anpassung an ihre Bedürfnisse. Diese Vorteile machen FOP in vielen Fällen überlegen gegenüber konventionell entwickelter Software.
Eine Software-Dokumentation ist bei Projekten mit mehren Entwicklern absolut notwendig. Insbesondere die Dokumentation des Quelltextes stellt für viele Entwickler eine große Hilfe dar. Durch die zahlreichen Dokumentationsgeneratoren können in vielen Sprachen Dokumentationen einfach und effizient erzeugt werden.

Wie auch in herkömmlichen Software-Systemen ist es bei SPLs sinnvoll die Vorteile von dokumentiertem Quelltext zu nutzen. Gerade bei FOP-Projekten ist der gesamte Quelltext zumeist deutlich größer als bei einem herkömmlichen Software-Produkt. Es ist so für die Entwickler noch schwieriger den Überblick über alle Aspekte der Implementierung zu behalten. Dies macht eine gute Dokumentation für die Entwickler umso wichtiger.
Doch gerade durch die besondere Struktur einer mit FOP erstellten SPL, gestaltet sich die Dokumentation des Quelltext mit herkömmlichen Mitteln als schwierig. FOP bietet bislang keine besondere Unterstützung für Quelltext-Dokumentationen an. Wünschenswert wäre daher eine Kombination beider Konzepte, damit auch die Entwickler von FOP-Projekten von den Vorteilen einer guten Dokumentation profitieren können. Die sich aus dieser Kombination ergebenen Probleme werden im nächsten Kapitel besprochen.

3. Anforderungsanalyse

Dieses Kapitel setzt mit den Problemen auseinander, die bei der Dokumentation von Software-Produktlinien, die durch die Feature-orientierte Programmierung (FOP) erstellt wurden auftreten. Eine Anforderung an die Dokumentation einer Produktlinie ist die Unterstützung mehrerer Anwendungsfälle für unterschiedliche Zielgruppen. Die Anwendungsfällen werden in diesem Kapitel im Einzelnen vorgestellt und die auftretenden Probleme bei der Erstellung einer entsprechenden Dokumentation untersucht.

3.1 Anwendungsfälle für SPL-Dokumentationen

In dieser Arbeit werden die Möglichkeiten zur Dokumentation einer Software-Produktlinie (SPL) für vier verschiedene Anwendungsfälle untersucht. Wie bei der Einteilung der Software-Dokumentationen in Abschnitt 2.1 existieren für unterschiedlichen Nutzer auch unterschiedliche Anwendungsfälle. Obwohl Quelltext-Dokumentationen im Allgemeinen auf Entwickler ausgerichtet sind, ist eine detaillierte Unterscheidung notwendig, da verschiedene Entwickler auch verschiedene Blickwinkel auf den Quelltext benötigen. Diese Anforderung ergibt sich aus den verschiedenen Anwendungsszenarien für SPLs und deren Komponenten, welche im Folgenden detailliert betrachtet werden.

Die Anwendungsfälle umfassen die Dokumentation eines Produkts, eines einzelnen Feature-Moduls, einer gesamten SPL und eines Kontext-Interfaces für eine SPL. In den nachfolgenden Abschnitten werden die vier Fälle, sowie deren Anwendungsszenarien und Zielgruppen beschrieben. Wie bereits in Kapitel 1 geschildert, ist es ein Ziel der Arbeit, dass Dokumentationen für alle Anwendungsfälle mithilfe eines einheitlichen Konzepts erstellt werden können. Um die Schwierigkeiten bei der Verwendung eines einheitliches Konzept zu vermitteln, werden zunächst für die einzelnen Anwendungsfälle triviale Lösungsansätze aufgezeigt, mit denen sich jeweils eine für diesen Fall passende Dokumentation erstellen lässt. Die Lösungsansätze umfassen jeweils die Strukturierung der Kommentare und die Methode, mit der aus diesen Kommentaren eine Dokumentation generiert werden kann. Alle trivialen Lösungsansätze zeichnen sich dadurch aus, dass diese für den jeweiligen Anwendungsfall eine

passende und vollständige Dokumentation generieren. Weiterhin nehmen sie wenig bis gar keine Änderungen an der Syntax der JavaDoc-Kommentare vor und lassen sich einfach umsetzen.
Bei den trivialen Lösungsansätzen findet eine Unterscheidung in statische und dynamische Methoden statt. Der Unterschied bei beiden Methoden liegt in der Behandlung der Kommentare vor der Generierung der Dokumentation. Bei einer dynamischen Methode werden die Kommentare je nach Situation verändert und angepasst, bei einer statischen Methode bleiben die Kommentare unverändert.
Des Weiteren werden alle Lösungsansätze an Beispielen verdeutlicht. Dazu dient der schon in Kapitel 2 vorgestellte Quelltextabschnitt für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie (siehe Abbildung 2.6). Dieser Abschnitt stellt die Einführung der Methode `incomingAction` durch das Feature *Chat* und Verfeinerung der Methode durch die Feature *Encryption* und *Spam* dar. Anhand dieses Abschnitts wird die Struktur der Kommentare für den Lösungsansatz verdeutlicht.

3.1.1 Produkt-Dokumentation

Bei der Produkt-Dokumentation handelt es sich um die Dokumentation eines vollständig konfigurierten Produkts einer SPL. Es wird also von einem konkreten Produkt die Dokumentation generiert. Dafür umfasst die Produkt-Dokumentation alle wichtigen Informationen zu denen in dem Produkt enthaltenen Signaturen. Dabei müssen alle Informationen aus jedem enthaltenen Feature des Produkts berücksichtigt werden.
Die Produkt-Dokumentation richtet sich vor allem an andere Entwickler, die ein bestimmtes Produkt der SPL nutzen möchten, um diese beispielsweise in einem ihrer Projekte zu verwenden. In der Regel ist dies der Fall, wenn es sich bei dem Produkt um eine Programmbibliothek handelt.

Trivialer Ansatz

Der trivialste Ansatz zur Erstellung einer Dokumentation für ein Produkt richtet sich nach dem Uniformitätsprinzip, welches besagt, dass nicht nur Quelltext Ziel einer Verfeinerung sein kann, sondern jedes Artefakt einer SPL (siehe Abschnitt 2.2.3).
Bei diesem Ansatz werden die Kommentare auf die Feature-Module aufgeteilt und anschließend bei der Generierung einer Dokumentation für ein bestimmtes Produkt wieder zusammengesetzt. Das bedeutet, um die Kommentare für die Dokumentation zu erstellen, folgt die Methode dem Uniformitätsprinzip und verfeinert schrittweise die Kommentare an den Signaturen. Es handelt sich daher um eine dynamische Methode. Da die Generierung der Produkte exakt diesem Prinzip folgt, ist der Produkt-Ansatz am besten zur Erstellung einer Produkt-Dokumentation geeignet.
Im Fall der Produkt-Dokumentation ist die Anwendung einer dynamischen Methode erforderlich, da die Zahl der Produkte exponentiell mit der Anzahl der Feature steigt. Im Höchstfall gibt es bei n Featuren 2^n Produkte in einer Produktlinie. Daher existieren in der Regel zu viele Produkte, als dass man zu jeder davon vorgefertigte Kommentare anlegen könnte.
Bei der Zusammensetzung der Kommentare wird davon ausgegangen, dass eine echte Verfeinerung der Kommentare stattfindet. Das heißt die Kommentare werden nicht

nur von Feature zu Feature überschrieben. Stattdessen können sich die Kommentare sowohl überschreiben, als auch gegenseitig ergänzen. Dieses Verhalten setzt voraus, dass Schlüsselwörter wie beispielsweise `original`, welches bei der Verfeinerung von Quelltext Verwendung findet, innerhalb der Kommentare verwendet werden.
Die Struktur der Produkt-Kommentare für den trivialen Ansatz ist in Abbildung 3.1 dargestellt. Laut der Feature-Reihenfolge der Chat-Produktlinie wird zuerst das Feature *Chat*, danach das Feature *Spam* und zum Schluss das Feature *Encryption* zu einem Produkt komponiert. Bei der Einführung der Methode im Feature *Chat* wird die Methode vollständig mit Kommentaren annotiert (Zeile 1–7). In jedem weiteren Feature kommen jeweils Informationen dazu. Im Feature *Encryption* wird die Beschreibung des Kommentars erweitert (Zeile 9–12). Dazu wird das Schlüsselwort `original` verwendet, um anzuzeigen, dass die Beschreibung erweitert und nicht überschrieben werden soll. Im Feature *Spam* wird ebenfalls mit `original` der `@return`-Tag erweitert (Zeile 14–17). Wird nun ein Produkt dokumentiert, in dem alle drei Feature enthalten sind, so wird der Kommentar aus *Chat* zunächst mit dem Kommentar aus *Encryption* und schließlich mit dem Kommentar aus *Spam* verfeinert.

```
1  /**                                                        Chat
2   * Edits incoming messages.
3   * Hook method.
4   *
5   * @param msg the message to edit
6   * @return the edited message
7   */
8  private Message incomingAction(Message msg) { ... }
9  /**                                                  Encryption
10  * {@original}
11  * Decodes the given message.
12  */
13 private Message incomingAction(Message msg) { ... }
14 /**                                                        Spam
15  * @return {@original}</br>null if message
16  *   is identified as spam
17  */
18 private Message incomingAction(Message msg) { ... }
```

Abbildung 3.1: Kommentare des Produkt-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie

3.1.2 SPL-Dokumentation

Bei dem Anwendungsfall der SPL-Dokumentation handelt es sich um die Dokumentation einer gesamten Produktlinie, in der alle Signaturen, die in einem beliebigen Feature definiert sein können, einmalig dokumentiert werden. Hierzu werden alle Informationen zu allen enthaltenen Signaturen der SPL benötigt. Die Dokumentation der Signaturen enthält sowohl allgemeinen Informationen, als auch spezielle Informationen zu der konkreten Implementierung in einem bestimmten Feature.

Die Dokumentation einer gesamten SPL ist in einigen Fällen sehr hilfreich. Entwickler der SPL können so einfacher und schneller die Funktion des Quelltextes anderer Entwickler nachvollziehen. Insbesondere neue Entwickler, die sich in einen Großteil des Quelltextes einlesen müssen, können davon profitieren. Die SPL-Dokumentation ermöglicht einen Überblick über die Produktlinie. Der Entwickler bekommt eine Übersicht über die Funktion der vorhandenen Klassen, Felder und Methoden in der gesamten Produktlinie. Die SPL-Dokumentation ist dabei nicht abhängig von einer bestimmten Konfiguration beziehungsweise einem Produkt, sondern zeigt jede Signatur. Damit bietet die SPL-Dokumentation eine gute Ergänzung zum Feature-Modell, um einen Überblick über die Produktlinie zu erhalten. Entwickler, die eine SPL verwenden möchten sind folglich in der Lage zu erkennen, welche Funktionalität diese bietet. Neue Entwickler eines SPL-Projekts, können sich einen Überblick über die vorhandenen Klassen, Felder und Methoden verschaffen.
Die SPL-Dokumentation ähnelt einer Produkt-Dokumentation, in der alle Feature ausgewählt sind. In der Regel ist dies bei komplexeren Produktlinien jedoch nicht möglich, da dies durch Alternativen und anderen Restriktionen im Feature-Modell untersagt ist. Somit zeigt die SPL-Dokumentation auch Kombinationen von Featuren, die laut Feature-Modell in keinem Produkt enthalten sein können.

Trivialer Ansatz

Für die Dokumentation einer SPL lässt sich ein trivialer Ansatz finden, der ganz ohne Änderungen der Syntax von JavaDoc-Kommentaren auskommt. Es handelt sich um eine statische Methode, bei der die Kommentare nicht aufgeteilt werden und auch keinerlei Verfeinerung der Kommentare stattfindet. Jede Signatur in der Produktlinie wird vollständig von einem einzigen Kommentar beschrieben. Dabei spielt es keine Rolle in welchem Feature-Modul sich der Kommentar für eine Signatur befindet, solange für die jeweilige Signatur maximal ein Kommentar existiert.
Die statische Methode eignet sich am besten für die SPL-Dokumentationen, da genau eine SPL-Dokumentation für eine Produktlinie existiert. In Abbildung 3.2 wird das Prinzip der vollständigen Kommentare verdeutlicht. Nur die Methode im Feature *Chat* ist annotiert (Zeile 1–6). Alle für die SPL-Dokumentation relevanten Informationen sind in diesem Kommentar enthalten. Die Verfeinerung der Methoden in den Featuren *Encryption* und *Spam* (Zeile 9 und 11) enthalten keinen Kommentar. Im Vergleich zum Lösungsansatz für Produkt-Dokumentationen sind wesentlich weniger Informationen zu sehen (siehe Abbildung 3.2).
Wird die SPL-Dokumentation erstellt, so können die Kommentare unverändert genutzt werden. Es müssen keine Kommentare zusammengesetzt werden, da jede Signatur höchstens einen Kommentar besitzt.

3.1.3 Kontext-Dokumentation

In Kapitel 2 wurde bereits das Prinzip vom Kontext-Interface erklärt. Es bildet eine Teilansicht auf die SPL und enthält alle Signaturen, die für eine bestimmte partielle Konfiguration vorhanden sind. Der Sinn des Kontext-Interface ist es den Entwickler beim Implementieren der SPL zu unterstützen. Es zeigt dem Entwickler alle verwendbaren Signaturen aus der Sicht des aktuell bearbeiteten Quelltextabschnitts. Demnach dokumentiert die Kontext-Dokumentation alle Signaturen eines gegeben

```
/**                                                              Chat
 * Edits incoming messages.
 *
 * @param msg the message to edit
 * @return the edited message
 */
private Message incomingAction(Message msg) { ... }
                                                           Encryption
private Message incomingAction(Message msg) { ... }
                                                                 Spam
private Message incomingAction(Message msg) { ... }
```

Abbildung 3.2: Kommentare des SPL-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie

Kontext-Interfaces.
Die Kontext-Dokumentation ähnelt einer eingeschränkten SPL-Dokumentation. Alle dokumentierten Signaturen befinden sich im Kontext-Interface. Gerade in diesem Fall sind dokumentierte Klassen und Methoden wichtig, damit das Interface den Entwickler bestmöglich bei der Wiederverwendung von Methoden aus anderen Feature-Modulen unterstützten kann. Daher ist bei diesem Anwendungsfall auch ein höherer Detaillierungsgrad, als bei der SPL-Dokumentation nötig. Neben den allgemeinen Informationen werden auch alle Feature-spezifischen Informationen angezeigt.

Trivialer Ansatz

Die Kontext-Dokumentation ist der einzige Anwendungsfall für den kein trivialer Ansatz existiert. Zwar ähnelt die Kontext- der SPL-Dokumentation, dennoch besteht ein maßgeblicher Unterschied. Für eine Produktlinie existiert nur eine einzige SPL-Dokumentation. Im Gegensatz dazu ist die Zahl der Kontext-Dokumentationen für eine Produktlinie im Regelfall wesentlich größer. Das theoretische Maximum an verschiedenen Kontext-Interfaces für eine Produktlinie ist 2^n bei n verschiedenen Featuren. Deshalb ist es, wie auch bei der Produkt-Dokumentation, hier nicht möglich für jeden Kontext vorgefertigte Kommentare anzulegen. Für die Kontext-Dokumentation lassen sich zwar auch dynamische Lösungsansätze finden, jedoch sind diese nicht mehr einfach umsetzbar oder benötigen eine zu starke Anpassung der JavaDoc-Syntax. Daher lässt sich für diesen Anwendungsfall kein trivialer Ansatz finden.

3.1.4 Feature-Modul-Dokumentation

Die Feature-Modul-Dokumentation ist die Dokumentation eines einzelnen Feature-Moduls. Diese dokumentiert alle Signaturen, die in einem bestimmten Feature-Modul enthalten sind. Dabei spielt es keine Rolle, ob eine Signatur in dem Modul eingeführt oder nur verfeinert wird.
In einigen Fällen ist es hilfreich nur die Dokumentation für ein bestimmtes Feature-Modul zu generieren. Ein Anwendungsszenario ist die Wiederverwendung eines Feature-Moduls in einer anderen SPL. Durch die Dokumentation wird eine leichtere Wiederverwendung des Moduls ermöglicht. Dieser Anwendungsfall richtet sich daher

an Entwickler einer neuen SPL, welche eine Wiederverwendung von bereits vorhanden Feature-Modulen anstreben. Die Feature-Modul-Dokumentation ist dabei im Vergleich zur SPL-Dokumentation detailreicher, da sie auch Informationen zu dem entsprechenden Featuren enthält.

Trivialer Ansatz

Die Feature-Modul-Dokumentation lässt sich, wie auch die SPL-Dokumentation durch eine statische Methode erzeugen. Das heißt die Kommentare werden nicht aufgeteilt und es ist auch keine Änderung der JacaDoc-Syntax nötig. Es findet daher auch keinerlei Verfeinerung der Kommentare statt. Der Unterschied zur SPL-Dokumentation besteht darin, dass jede Signatur nicht nur einen Kommentar erhält, sondern dass in jedem Feature-Modul jede Signatur mit einem vollständigen Kommentar annotiert wird.
Für die Feature-Modul-Dokumentation eignet sich diese statische Methode am besten, da die Anzahl der Feature-Modul-Dokumentationen stark begrenzt ist. Diese entspricht der Anzahl der Feature in der SPL. In Abbildung 3.3 werden die vollständigen Kommentare für jedes abgebildete Feature-Modul dargestellt. Jedes Feature enthält einen vollständigen Kommentar für die *incomingAction* Methode (Zeile 1–7, 9–15 und 17–23). Wird eine Feature-Modul-Dokumentation erzeugt, so werden nur die Kommentare des entsprechenden Feature-Moduls betrachtet. Alle anderen Kommentare der Produktlinie werden ignoriert.

3.2 Probleme bei der Verwendung trivialer Ansätze

In Abschnitt 3.1 wurde gezeigt, dass die Anwendungsfälle für Quelltext-Dokumentationen in FOP recht unterschiedlich sind. Dementsprechend unterscheidet sich auch die trivialen Ansätze zur Generierung einer Dokumentation stark. Dennoch wäre es wünschenswert angepasste Dokumentation für jeden Anwendungsfall aus den selben Kommentaren generieren zu können, um den Aufwand für die Entwickler möglichst gering zu halten. Aus diesem Grund wurde die Verwendung der trivialen Ansätze, bezogen auf alle Anwendungsszenarien betrachtet und deren Vor- und Nachteile analysiert.
Wie bereits erwähnt ist ein einheitlicher Lösungsansatz das Ziel dieser Arbeit. Daher wird nachfolgend untersucht, wie sich die drei vorgestellten Lösungsansätze für die Dokumentation der andere Anwwendungsfälle verhalten. Es wird beschrieben, welche Probleme auftreten, wenn die Lösungsansätze für andere Anwendungsfälle verwendet werden. Dafür wird jeweils die Kommentarstruktur eines bestimmten trivialen Ansatzes verwendet und für den jeweiligen Anwendungsfall die eigene Methode zur Komposition dieser Kommentare angewandt.
Um die auftretenden Probleme verdeutlichen zu können, wird das Beispiel `sendMessage` aus Kapitel 2 verwendet (siehe Abbildung 2.7). Es zeigt die Methode `sendMessage` aus der Klasse `Client` der Chat-Produktlinie. Die Methode wird im Feature *Chat* eingeführt und im Feature *UserCommands* verfeinert. Die Funktion der Methode in den beiden Featuren wurde in Kapitel 2 bereits ausführlicher erklärt. Anhand dieses Beispiels wird die Struktur der Kommentare für die drei verschiedenen Ansätze gezeigt und beschrieben welche Probleme aus der jeweiligen Struktur hervorgehen.

```
1  /**                                                          Chat
2   * Edits incoming messages.
3   * Hook method.
4   *
5   * @param msg the message to edit
6   * @return the edited message
7   */
8  private Message incomingAction(Message msg) { ... }
9  /**                                                    Encryption
10  * Edits incoming messages.
11  * Decodes the given message.
12  *
13  * @param msg the message to edit
14  * @return the edited message
15  */
16 private Message incomingAction(Message msg) { ... }
17 /**                                                          Spam
18  * Edits incoming messages.
19  *
20  * @param msg the message to edit
21  * @return the edited message</br>
22  *   null if message is identified as spam
23  */
24 private Message incomingAction(Message msg) { ... }
```

Abbildung 3.3: Kommentare des Feature-Modul-Ansatzes für die Methode `incomingAction` der Klasse `Client` in der Chat-Produktlinie

3.2.1 Produkt-Ansatz

Der Lösungsansatz für Produkt-Dokumentation ist der interessanteste, da er konform zum Uniformitätsprinzip und daher sehr intuitiv für die Entwickler einer auf FOP-basierenden SPL ist. Trotzdem treten bei der Anwendung dieses Ansatzes mehrere Probleme für die jeweils anderen Anwendungsfälle auf. Die sich ergebenen Probleme lassen sich in drei Arten untergliedern. So können die Informationen innerhalb der Kommentare einer Signatur doppelt vorhanden sein, die Informationen können sich widersprechen und es ist auch möglich, dass Informationen in den Kommentaren fehlen. Diese drei Problemarten werden nachfolgend im Details erklärt und anhand von Abbildung 3.4 verdeutlicht.

Doppelte Informationen

Enthalten mehrere Kommentare für eine Signatur, in unterschiedlichen Feature-Modulen die gleichen Informationen, so entstehen dadurch verschiedene Probleme. Die Kommentare verbrauchen insgesamt mehr Platz und belasten dadurch die Übersichtlichkeit des Quelltextes. Änderungen an den Kommentaren müssen immer konsistent gehalten werden und die generierte Dokumentation würde doppelte Texte enthalten.

Im Hinblick auf die Anwendung des Produkt-Ansatzes können doppelte Informationen sowohl bei der Generierung der SPL-, als auch bei der Kontext-Dokumentation auftreten. In Abbildung 3.4 lässt sich dieses Problem anhand des `@param`-Tags erkennen. Der `@param`-Tag enthält redundante Informationen im Feature *Chat* (Zeile 6) und im Feature *UserCommands* (Zeile 14). Bei der Bildung der SPL- oder der Kontext-Dokumentation kommt es infolgedessen zu den oben genannten Problemen.

Widersprüchliche Informationen

Widersprechen sich Informationen in den Kommentaren einer Signatur, so ist es schwierig die Kommentare richtig zu komponieren. Widersprüche finden sich zum Beispiel in der Beschreibung eines Kommentars, wenn sich die Funktionalität oder Bedeutung einer Signatur durch ein Feature stark verändert. Werden in den Beschreibungen von Parametern und Rückgabewert einer Methode Beschränkungen angegeben, so können sich bei einer Verfeinerung der Methode auch diese Informationen widersprechen. Widersprüche können aber auch in fast jedem anderen Teil des Kommentars auftreten.
Wie auch bei den Problemen mit doppelten Informationen sind bei Anwendung des Produkt-Ansatzes die SPL- und die Kontext-Dokumentation von Problemen mit widersprüchlichen Informationen betroffen. Auch dieses Problem kann mithilfe des Beispiels in Abbildung 3.4 gezeigt werden. Die Informationen des `@param`-Tags widersprechen sich. Der Tag in Feature *Chat* sagt aus, dass in einer Nachricht jedes Zeichen verwendet werden darf (Zeile 7). Dagegen wird im Tag in Feature *UserCommands* spezifiziert, dass eine mit dem `/`-Zeichen anfangende Nachricht als Befehl an den Server interpretiert wird (Zeile 15–16).

Fehlende Informationen

Des Weiteren kann es vorkommen, dass in den Kommentaren Informationen fehlen oder dass Informationen nicht zugeordnet werden kann, ob diese eine Signatur allgemein beschreiben oder spezifisch für ein bestimmtes Feature sind.
Bei der Anwendung des Produkt-Ansatzes tritt das Fehlen von Informationen bei der Generierung der Feature-Modul-Dokumentation auf. Abermals kann dies am Beispiel in Abbildung 3.4 verdeutlicht werden. Der Kommentar im Feature *Chat* enthält einige allgemeine Informationen in der Beschreibung des Kommentars (Zeile 2–4). Im Feature *UserCommands* fehlen diese Informationen im Kommentar (Zeile 11).

3.2.2 SPL-Ansatz

Der Lösungsansatz für die SPL-Dokumentation zeichnet sich dadurch aus, dass er für jede Signatur höchstens einen Kommentar bereitstellt, welcher allgemeine Informationen über die Signatur enthält. Das Hauptproblem für alle anderen Anwendungsfälle ist das Fehlen von Informationen. In den Kommentaren des SPL-Ansatzes finden sich keine Informationen zu den einzelnen Featuren. Doch selbst falls in den Kommentaren Informationen zu den einzelnen Featuren wären, so ließen sich diese nicht zuordnen, da für jede Signatur in höchstens einem Feature-Modul ein Kommentar vorhanden ist.
Abbildung 3.5 stellt wieder das *sendMessage* Beispiel dar und zeigt die Probleme

```
 1  /**                                                        Chat
 2   * Sends a Message to the Server.
 3   * Creates a new {@link TextMessage} and
 4   * sends it to the server.
 5   *
 6   * @param line the message content.
 7   *   The message can contain any character.
 8   */
 9  public static void sendMessage(String line) { ... }
10  /**                                                UserCommands
11   * {@original}
12   * Can be used to trigger user commands.
13   *
14   * @param line the message content.
15   *   If the message starts with a /,
16   *   the whole line is interpreted as user command.
17   */
18  public static void sendMessage(String line) { ... }
```

Abbildung 3.4: Kommentare des Produkt-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

mit fehlenden Informationen auf. In diesem Fall ist der Quelltext mit den Kommentaren für den SPL Ansatz annotiert. Es ist zu sehen, dass keine näheren Informationen über die Feature *Chat* und *UserCommands* vorhanden sind. Die Informationen werden allerdings bei der Generierung der Produkt-, Kontext- und Feature-Modul-Dokumentation benötigt.

```
1  /**                                                        Chat
2   * Sends a Message to the Server.
3   *
4   * @param line the message content.
5   */
6  public static void sendMessage(String line) { ... }
7                                                     UserCommands
8  public static void sendMessage(String line) { ... }
```

Abbildung 3.5: Kommentare des SPL-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

3.2.3 Feature-Modul-Ansatz

Der Lösungsansatz für die Feature-Modul-Dokumentation geht davon aus, dass jede Signatur mit einem eigenen vollständigen Kommentar annotiert ist. Für die Erstellung einer Feature-Modul-Dokumentation müssen lediglich die Signaturen eines einzelnen Feature-Moduls und deren Kommentare betrachtet werden. Bei der Verwendung des Ansatzes für andere Anwendungsfälle treten jedoch erhebliche Probleme auf, da dieser das gegenteilige Problem zum Ansatz für die SPL-Dokumentation

aufweist. In diesem Fall fehlen keine Informationen, sondern es werden zu viele Informationen angegeben. Daher ergeben sich bei der Produkt-, SPL- und Kontext-Dokumentation Probleme mit doppelten und widersprüchlichen Informationen.
Ein Beispiel dafür ist in Abbildung 3.6 dargestellt. Die Beschreibung beider Kommentare enthalten doppelte Informationen (Zeile 2 und 11). Auch der `@param`-Tag enthält doppelte Informationen (Zeile 6 und 14). Der `@param`-Tag enthält jedoch auch widersprüchliche Informationen. Die Informationen in Zeile 7 und 15–16 sind konträr zu einander, wie schon im Beispiel zu den widersprüchlichen Informationen gezeigt. Wie leicht zu erkennen ist, treffen diese Probleme auf die Produkt-, SPL- und Kontext-Dokumentation zu.

```
1  /**                                                          Chat
2   * Sends a Message to the Server.
3   * Creates a new {@link TextMessage} and
4   * sends it to the server.
5   *
6   * @param line the message content.
7   *   The message can contain any character.
8   */
9  public static void sendMessage(String line) { ... }
10 /**                                                  UserCommands
11  * Sends a message to the server.
12  * Can be used to trigger user commands.
13  *
14  * @param line the message content.
15  *   If the message starts with a /,
16  *   the whole line is interpreted as user command.
17  */
18 public static void sendMessage(String line) { ... }
```

Abbildung 3.6: Kommentare des Feature-Modul-Ansatzes für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

3.3 Zusammenfassung

Ziel der Arbeit ist eine maßgeschneiderte Quelltext-Dokumentation für mehrere Anwendungsfälle einer SPL zu generieren. Dabei handelt es sich um die Anwendungsfälle Produkt-, SPL-, Kontext- und Feature-Modul-Dokumentation. Ein Ziel aus Kapitel 1 ist, dass alle Dokumentationen mit demselben Verfahren und aus denselben Kommentaren generiert werden, um den Dokumentationsaufwand für die Entwickler möglichst gering zu halten. Für die meisten Anwendungsfälle existiert zwar jeweils ein trivialer Ansatz, jedoch sind die Ansätze nicht kompatibel zu einander, weshalb für jeden Anwendungsfall eigene Kommentare zur Generierung der Dokumentation notwendig sind. Es ergeben sich jeweils bei der Verwendung eines trivialen Ansatzes einige Probleme mit doppelten, widersprüchlichen und fehlenden Informationen für die jeweiligen anderen Anwendungsfälle. Die Probleme sind in der Tabelle 3.1

	Triviale Ansätze		
Anwendungsfall	**Produkt**	**SPL**	**Feature-Modul**
Produkt	- - -	F - -	- D W
SPL	- D W	- - -	- D W
Kontext	- D W	F - -	- D W
Feature-Modul	F - -	F - -	- - -

Tabelle 3.1: Übersicht über die Probleme bei verschiedenen Ansätzen (F = fehlende, D = doppelte, W = widersprüchliche Informationen)

für jeden trivialen Ansatz noch einmal zusammengefasst. In der Tabelle ist zu sehen, dass das Hauptproblem des SPL-Ansatzes die fehlenden Informationen sind. Im Gegensatz dazu treten beim Feature-Modul-Ansatz Probleme mit doppelten und widersprüchlichen Informationen auf. Der Produkt-Ansatz ist zwar der intuitivste der drei trivialen Ansätze, trotzdem treten bei diesem alle drei Problemarten auf.
Aus der Untersuchung der trivialen Lösungsansätze ist zu erkennen, dass sich keiner der drei Lösungsansätze für die Dokumentation aller Anwendungsfälle eignet. Daher wird ein neuer Ansatz zur Strukturierung der Kommentare und Generierung der Dokumentation benötigt.

4. Lösungsansatz

In diesem Kapitel wird der Lösungsansatz für die Probleme des vorherigen Kapitels beschrieben. Diese Probleme bestanden aus doppelten, widersprüchlichen und fehlenden Informationen.
In Abschnitt 2.2.3 wurde über das Prinzip der Uniformität gesprochen. Es liegt nahe dies auch auf die JavaDoc-Kommentare zum Erstellen der Dokumentation anzuwenden. Dieses Prinzip funktioniert jedoch nur für die Dokumentation eines Produkts. Für die Software-Produktlinie (SPL)-, Kontext- und Feature-Modul-Dokumentation ist die Anwendung des Uniformitätsprinzips nicht ausreichend genug. Es ist nötigt die Kommentare für die Dokumentation auf eine andere Art zu strukturieren und eine Methode zu entwickeln, welche die Informationen aus den Kommentaren für jeden Anwendungsfall individuell zusammensetzt. Wie eine solche Strukturierung der Kommentare aussieht und wie daraus die Dokumentationen für die einzelnen Anwendungsfälle generiert werden können, wird in den nächsten Abschnitten beschrieben.

4.1 Grundidee des Lösungsansatzes

Der Lösungsansatz zur Generierung eine Dokumentation besteht aus drei aufeinander aufbauenden Phasen. In Abbildung 4.1 ist ein grafischer Ablauf des der Lösungsansatzes dargestellt. Die drei Phasen sind die Erstellung von Modul-Kommentaren, die Erzeugung von Pseudo-Quelltext und Dokumentationskommentaren und die Generierung der eigentlichen Dokumentation. Die Phasen werden von unterschiedlichen Personen ausgeführt. Diese Personen bestehen aus den Entwicklern der Produktlinie, dem in dieser Arbeit implementierten Prototypen und dem JavaDoc-Tool.
Die erste Phase ist das Erstellen von speziellen *Modul-Kommentaren*, die in Abschnitt 4.2 eingeführt werden. Diese Phase wird von den Entwicklern, während der Entwicklung der Produktlinie übernommen. Das Erstellen der Modul-Kommentare geschieht nur einmalig, da die Informationen in den Modul-Kommentare ausreichen, um für jeden Anwendungsfall die entsprechende Dokumentation zu erzeugen.
In der zweiten Phase wird Pseudo-Quelltext erzeugt, welcher mit *Dokumentationskommentaren* annotiert ist. Diese Phase übernimmt ein Vereinigungsverfahren, welches in einem, im Rahmen dieser Arbeit implementierten Prototypen umgesetzt

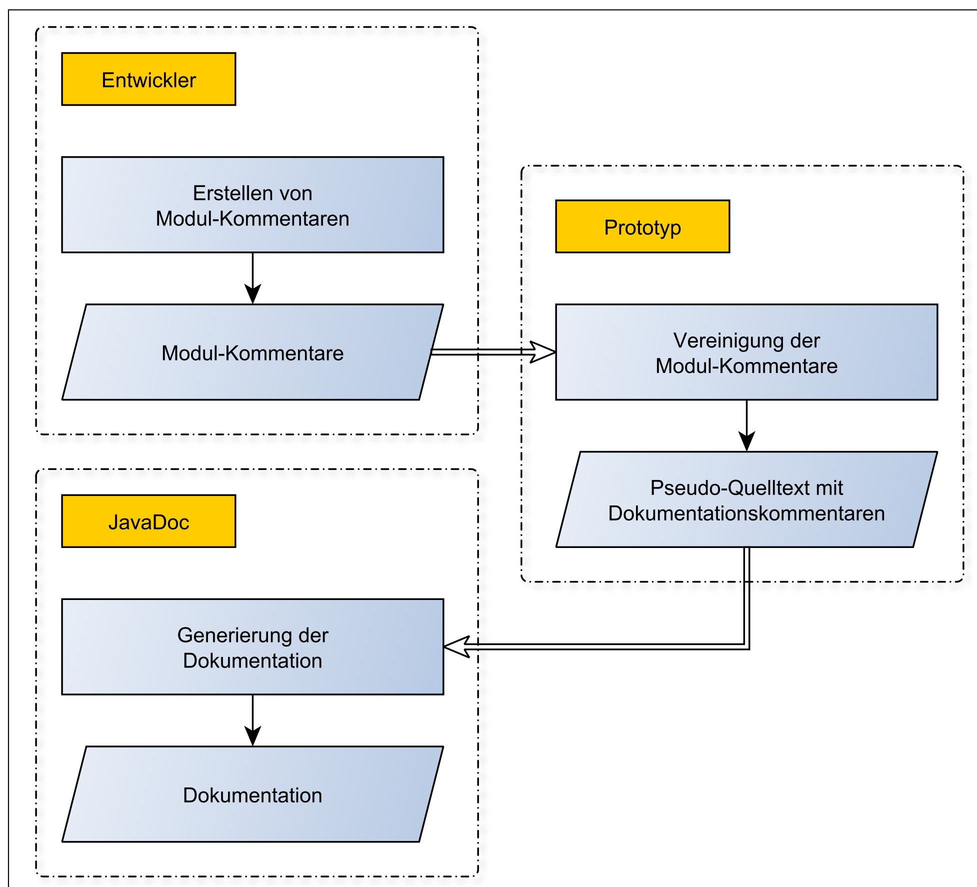

Abbildung 4.1: Grundsätzlicher Ablauf des Konzepts

ist. Als Dokumentationskommentare werden die vom Vereinigungsverfahren erzeugten JavaDoc-Kommentare bezeichnet. Es handelt sich dabei um normale JavaDoc-Kommentare, die problemlos vom JavaDoc-Tool eingelesen werden können.
In der dritten Phase wird eine Dokumentation aus dem erzeugten Pseudo-Quelltext generiert. Diese Aufgabe wird vom normalen JavaDoc-Tool übernommen.
Die Aufteilung des Konzepts in die drei Phasen bietet einige Vorteile. Die Kombination aus den Modul-Kommentaren und dem Vereinigungsverfahren beseitigt alle in Abschnitt 3.2 beschriebenen Probleme. Das heißt, dass dieses Konzept es ermöglicht für Produkte, SPLs, Kontext-Interfaces und Feature-Module Dokumentationen zu erzeugen. Weiterhin bedeutet es, dass die einmalig erstellten Modul-Kommentare für alle vier Anwendungsfälle genutzt werden können. Ein weiterer großer Vorteil des Konzepts besteht darin, dass das JavaDoc-Tool von dem Vereinigungsverfahren und den Modul-Kommentaren unabhängig ist. Das bedeutet, dass das JavaDoc-Tool in keiner Weise eingeschränkt wird. Es ist möglich alle üblichen Einstellungen des Tools zu nutzen. Insbesondere können benutzerdefinierte Doclets verwendet werden, welche das Ausgabeformat einer durch das JavaDoc-Tool erzeugten Dokumentation bestimmten [Ora14]. Außerdem ist es möglich auch andere Versionen des JavaDoc-Tools zu verwenden.
Um das Konzept zu veranschaulichen, wird das in Kapitel 2 einführte Beispiel `sendMessage` verwendet (siehe Abbildung 2.7) Das Beispiel zeigt einen Ausschnitt aus der Chat-Produktlinie in welchem die Methode `sendMessage` der Klasse `Client` in

den Featuren *Chat* und *UserCommands* zu sehen ist. Die Methode wird im Feature *Chat* eingeführt und in *UserCommands* verfeinert.

4.2 Erstellung von Modul-Kommentaren

Um die Probleme aus Abschnitt 3.2 zu beheben werden die Modul-Kommentare eingeführt. Bei den Modul-Kommentaren handelt es sich um leicht angepasste JavaDoc-Kommentare, welche zusätzliche Informationen für das Vereinigungsverfahren bereitstellen. Diese zusätzlichen Informationen beziehen sich auf die Tags in den Modul-Kommentaren und ordnen diesen einerseits einen Informationstyp (*allgemein* oder *Feature-spezifisch*) und andererseits eine Priorität zu.

4.2.1 Trennung allgemeiner und Feature-spezifischer Informationen

Zunächst muss die Bedeutung von allgemeinen und Feature-spezifischen Informationen erklärt werden. Daher werden diese folgendermaßen definiert.

Definition 1. *Als allgemeine Informationen werden alle Informationen bezeichnet, die eine Signatur grundlegend beschreiben und unabhängig von Verfeinerungen durch Feature sind.*

Definition 2. *Als Feature-spezifische Informationen werden alle Informationen bezeichnet, die an ein bestimmtes Feature gebunden sind.*

Die allgemeinen Informationen dienen dazu die prinzipielle Funktion einer Signatur zu beschreiben. Wird die Signatur durch weitere Feature verfeinert, so können die Erweiterungen oder Änderungen mit Feature-spezifischen Kommentaren dokumentiert werden. Ein Problem mit mehreren Kommentaren für eine Signatur ist, dass sie sowohl allgemeine Informationen zu der Signatur, als auch Feature-spezifische Informationen enthalten. Es daher notwendig die Informationen der Kommentare in allgemeine und Feature-spezifische aufzuteilen. Dabei werden allgemeine Informationen immer bei der Einführung einer Signatur bereitgestellt. Gibt es eine parallele Einführung einer Signatur durch zwei oder mehr Feature, so können auch mehrere allgemeine Informationen angegeben werden. Im Zuge der Trennung nach allgemeinen und Feature-spezifischen Informationen können auch alle Redundanzen innerhalb der Modul-Kommentare vermieden werden, da alle Informationen entweder an ein bestimmtes Feature gebunden oder allgemeiner Natur sind und einmalig bei der Einführung der Signatur angegeben werden.
Die Trennung der Informationen wird über neue Schlüsselworte in den Modul-Kommentare realisiert. Es werden unter anderem die Schlüsselwörter `general` und `feature` eingeführt. Allgemeine Informationen werden durch das Schlüsselwort `general` und Feature-spezifische Informationen durch das Schlüsselwort `feature` angezeigt. Ein weiteres neues Schlüsselwort ist `new`, welches für einen speziellen Zweck eingeführt wird. Es dient dazu mit Neueinführungen von Signaturen durch Feature umzugehen, indem es alle vorherigen Feature-spezifischen Informationen verwirft. Bei der Neueinführungen einer Signatur geht die bisherige Funktion und damit auch alle Feature-spezifischen Informationen für die Signatur verloren. Daher ist auch das Schlüsselwort `new` wichtig für die Modul-Kommentare.

4.2.2 Priorisierung der Informationen

In Kapitel 3 wurde aufgezeigt, dass sich die Informationen in den Kommentaren eine SPL widersprechen können. Da es sich bei den Kommentaren um natürlichsprachliche Informationen handelt, ist es für einen Computer nicht möglich die richtige Entscheidung bei widersprüchlichen Informationen zu treffen. Die Lösung ist daher eine Priorisierung der Informationen in den Modul-Kommentaren, sodass die Entwickler die Entscheidung treffen, welcher Information der Vorrang gilt. Dabei überschreiben Informationen mit einer hohen Priorität Informationen mit einer niedrigeren. Haben zwei Informationen die gleiche Priorität, so werden sie durch das Vereinigungsverfahren zusammengefasst.
Die Priorisierung der Informationen bedeutet einigen Mehraufwand für die Entwickler, da sie den Überblick über die Kommentare behalten müssen. Sie müssen dafür sorgen, dass alle Informationen, die gleich wichtig sind und sich nicht widersprechen auch die gleiche Priorität besitzen. Ansonsten könnten bei der Vereinigung der Modul-Kommentare Informationen verloren gehen. Dafür löst sich allerdings das Problem der sich widersprechenden Informationen und die Entwickler erhalten mehr Flexibilität und Kontrolle bei der Generierung der Dokumentation.
Auch die Priorisierung der Informationen in den Modul-Kommentaren wird durch die neuen Schlüsselwörter ermöglicht. Da jede Information in den Modul-Kommentaren sowohl einen Informationstyp als auch eine Priorität benötigt, wird die Angabe der Priorität mit der Angabe des Informationstypen verbunden. Das bedeutet, dass die Priorität zusammen mit einem der Schlüsselwörter `general`, `feature` oder `new` angegeben wird.

4.2.3 Übersicht über die neuen Schlüsselwörter

Im Folgenden wird eine Übersicht über die drei neu eingeführten Schlüsselwörter `general`, `feature` und `new` für die Modul-Kommentare gegeben. Wie dabei die genaue Realisierung der Schlüsselwörter aussieht wird in Kapitel 5 diskutiert.
In den meisten Fällen sollte die Verwendung von `general` und `feature` ausreichen. Auch die Priorisierung von `general`-Kommentaren ist eher selten der Fall, da sich allgemeine Informationen per Definition nicht ändern sollen. Allerdings besteht immer die Möglichkeit, dass bei der Erarbeitung des Konzepts ein spezieller Fall nicht berücksichtigt wurde. Daher wurden die Modul-Kommentare so entworfen, dass mit diesen möglichst flexibel auf alle eventuellen Umstände reagiert werden kann.

General

Das Schlüsselwort `general` wird verwendet um allgemeine Informationen zu kennzeichnen. Zusätzlich wird für die Informationen eine Priorität mit angegeben, welche für alle allgemeinen Informationen einer Signatur gilt. Das bedeutet, falls für eine Signatur mehrere Modul-Kommentare mit dem Schlüsselwort `general` und verschiedenen Prioritäten existieren, so überschreibt derjenige Modul-Kommentar mit der höchsten Priorität vollständig alle anderen.

Feature

Das Schlüsselwort `feature` zeigt an, dass sich die nachfolgenden Informationen nur auf dieses Feature beziehen. Auch bei diesem Schlüsselwort wird eine Priori-

tät mit angegeben, welche allerdings nur für die Informationen, in diesem Modul-Kommentar gelten.

New

Das Schlüsselwort `new` verhält sich bis auf eine zusätzliche Funktion genauso, wie das Schlüsselwort `feature`. Bei der Verwendung des Schlüsselwortes `new` werden alle vorherigen Feature-spezifischen Informationen zu einer Signatur verworfen. Allgemeine Informationen aus den Modul-Kommentare werden davon nicht beeinflusst.

Beispiel

Wie oben bereits erwähnt, wird zur Veranschaulichung das Beispiel `sendMessage` betrachtet. Der Quelltextausschnitt in Abbildung 4.2 zeigt die Methode `sendMessage` in den beiden Featuren *Chat* und *UserCommands* mit den neuen Modul-Kommentaren. Da die genaue Syntax der Modul-Kommentare erst in Kapitel 5 beschrieben wird, wird hier lediglich mithilfe der beiden Schlüsselwörter `general` und `feature` der Informationstyp und die Priorität der Informationen angedeutet. In beiden Feature-Modulen ist die Methode jeweils mit einem Modul-Kommentar annotiert. Der erste Modul-Kommentar in Zeile 1–9 stellt sowohl allgemeine Informationen mit der Priorität 0 (Zeile 2–4), als auch Feature-spezifische Informationen mit der Priorität 0 (Zeile 5–8) bereit. Im zweiten Modul-Kommentar in Zeile 11–17 sind Feature-spezifische Informationen mit der Priorität 0 (Zeile 12–13) und der Priorität 1 (Zeile 14–16) zu sehen.
Wie leicht zu erkennen ist, enthalten die Modul-Kommentare keine redundanten Informationen. Weiterhin sind die widersprüchlichen Informationen des `@param`-Tags priorisiert wurden, sodass die Einschränkung im Feature *UserCommands* (Zeile 15–16) ein höheres Gewicht als die Einschränkung im Feature *Chat* (Zeile 8) hat. Es lässt sich außerdem zeigen, dass für jeden Anwendungsfall die richtigen Informationen bereitgestellt werden können, wenn die Informationen aus den Modul-Kommentaren richtig vereinigt werden. Wie genau die Vereinigung der beiden Modul-Kommentare stattfindet, wird im nächsten Abschnitt näher erläutert.

4.3 Ablauf des Vereinigungsverfahrens

Der zweite zentrale Aspekt des Konzepts ist das richtige Vereinigen der Modul-Kommentare. Das Vereinigungsverfahren erzeugt mit Dokumentationskommentaren annotierten Pseudo-Quelltext. Der Pseudo-Quelltext enthält Klassen, Felder, Methodendeklarationen und die entsprechenden Dokumentationskommentare, jedoch keinen Inhalt für die Methoden. Mithilfe des Pseudo-Quelltextes ist das JavaDoc-Tool in der Lage die finale Dokumentation zu generieren. Als Input verwendet das Vereinigungsverfahren die Signaturen mit annotierten Modul-Kommentaren und eine Liste von Featuren für den jeweiligen Anwendungsfall. Das Vereinigungsverfahren orientiert sich in seinem Ablauf an dem Prinzip der Uniformität. Der grundsätzliche Ablauf des Vereinigungsverfahrens wird nachfolgend näher erklärt und ist in Abbildung 4.3 als Flussdiagramm abgebildet.
Das Vereinigungsverfahren erhält als Input eine Liste mit ausgewählten Signaturen und Featuren für den konkreten Anwendungsfall. Es ist nicht nötig immer alle Signaturen der SPL zu verarbeiten. Dies ist nur der Fall, wenn für die gesamte SPL

```
1  /**                                                     Chat
2   * general 0:
3   *   Sends a message to the Server.
4   *   @param line the message content.
5   * feature 0:
6   *   Creates a new {@link TextMessage} and
7   *   sends it to the server.
8   *   @param line The message can contain any character.
9   */
10 public static void sendMessage(String line) { ... }
11 /**                                             UserCommands
12  * feature 0:
13  *   Can be used to trigger user commands.
14  * feature 1:
15  *   @param line If the message starts with a /,
16  *   the whole line is interpreted as user command.
17  */
18 public static void sendMessage(String line) { ... }
```

Abbildung 4.2: Modul-Kommentare für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

die Dokumentation erzeugt werden soll. Bei der Dokumentation eines Produkts, eines Kontext-Interfaces oder eines Feature-Moduls ist es ausreichend nur diejenigen Signaturen zu betrachten, welche auch wirklich in dem jeweiligen Anwendungsfall enthalten sind. Die Signaturliste wird nun sequentiell abgearbeitet und jede Signatur einzeln betrachtet.
Jede Signatur besitzt eine Menge von Modul-Kommentaren, welche den einzelnen Featuren zugeordnet sind. Anhand der Feature-Liste wird ermittelt, welche Modul-Kommentare für den aktuellen Anwendungsfall betrachtet werden müssen. Im Folgenden werden alle für den Anwendungsfall benötigten Modul-Kommentare der Signatur in ihre Tags aufgeteilt und so eine Liste von Tags erstellt. Dabei wird auf die Feature-Reihenfolge der Produktlinie geachtet, sodass alle Tags in der Liste in der richtigen Reihenfolge der Verfeinerung stehen. Die Aufteilung in die einzelnen Tags ist notwendig, um die Modul-Kommentare sinnvoll zu vereinigen. Ein JavaDoc-Kommentar besteht wie in Abschnitt 2.1.2 beschrieben aus einer Beschreibung und einer Menge von Tags. Sowohl die Beschreibung als auch jeder einzelne Tag stellt einen Teil des ganzen Kommentars dar. Die Beschreibung wird daher in diesem Vereinigungsverfahren genauso wie ein einzelner Tag behandelt.
Zu jedem Tag sind nach der Zerlegung des Modul-Kommentars der Name, der Inhalt, die Priorität und der Informationstyp bekannt. Die Priorität eines Tags gibt wie in Abschnitt 4.2.2 angegeben die Relevanz eines Tags an. Der Informationstyp eines Tags gibt an, ob es sich um allgemeine oder Feature-spezifische Informationen handelt. Der Tag-Inhalt besteht aus dem Text hinter dem jeweiligen Tag-Schlüsselwort. Der Name eines Tags ist eindeutig innerhalb eines Modul-Kommentars und setzt sich aus dem Tag-Typen (z.B. `@return`) und einem Teil des Tag-Inhalts zusammen. Der Tag-Typ allein reicht in manchen Fällen nicht aus, um einen Tag eindeutig zu

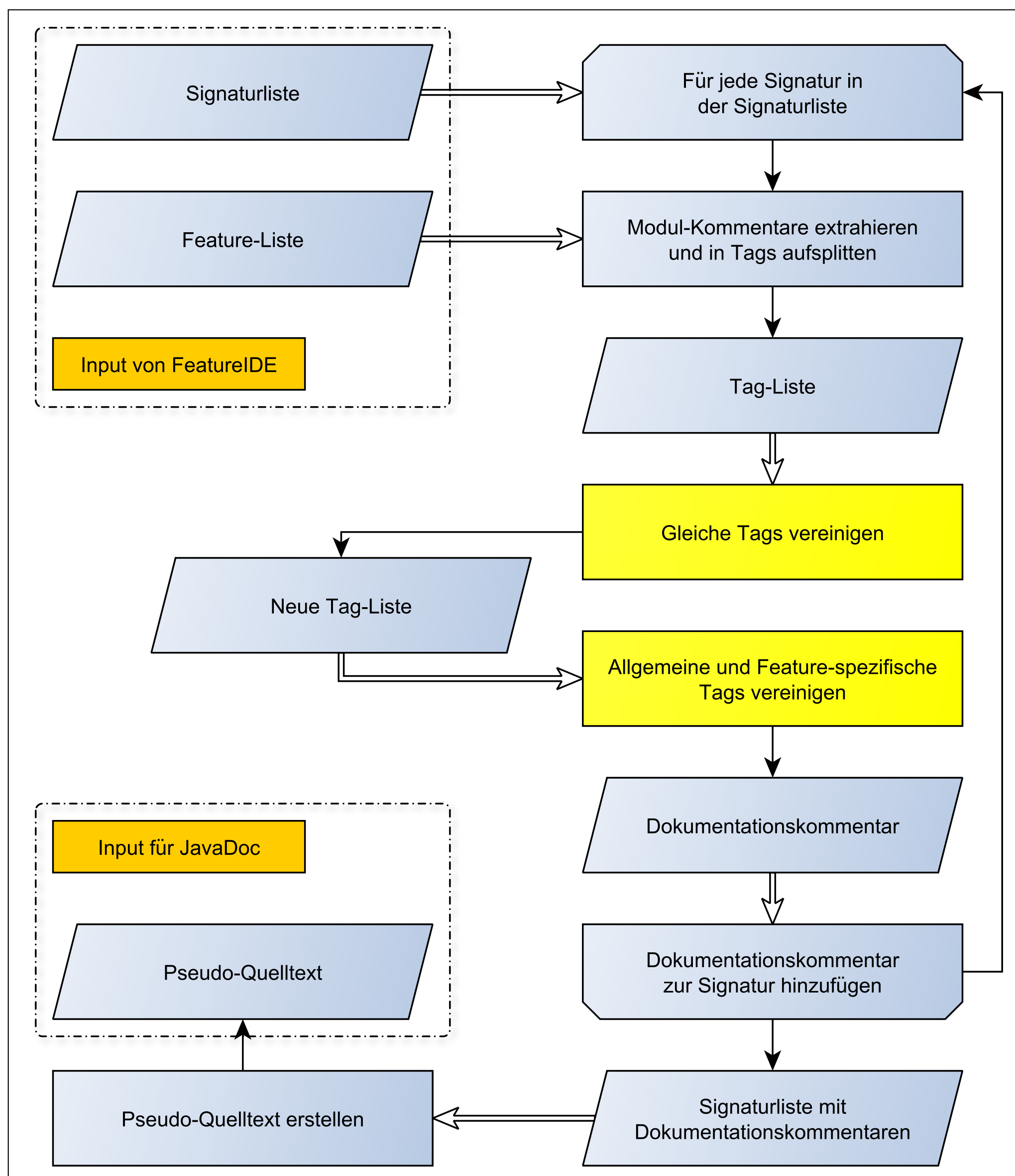

Abbildung 4.3: Flussdiagramm über den grundsätzlichen Ablauf des Vereinigungsverfahren

identifizieren, da einige Tags auch mehrfach in Kommentaren vorkommen dürfen. Dazu zählen die Tags `@author`, `@param`, `@see`, `@serialField` und `@throws` beziehungsweise `@exception`. Für diese Tags wird daher ein Teil des Tag-Inhalts für den Tag-Namen mitverwendet. Wie viel von dem Inhalt verwendet wird hängt von den einzelnen Tags ab. Bei dem `@author`- und dem `@see`-Tag wird der gesamte Inhalt verwendet. So ergibt sich beispielsweise für den Tag `@see Chat#sendMessage` der Tag-Name „*see Chat#sendMessage*“. Im Falle des `@param`- und des `@throws` beziehungsweise `@exception`-Tags wird jeweils das erste Wort des Tag-Inhalts benutzt. Bei dem `@param`-Tag entspricht dies dem Parameternamen und bei dem `@throws`-Tags dem Namen der geworfenen Exception. Der `@serialField`-Tag wird ähnlich dem `@param`-Tag behandelt, nur das die ersten zwei Worte nach dem Schlüsselwort verwendet werden. Dies geben den Namen und den Typ eines serialisierten Feldes an. Für alle anderen Tag reicht es aus den Typen des Tags als Tag-Namen zu ver-

wenden, so erhält beispielsweise der `@return`-Tag den Namen „`return`“.
Ist die Tag-Liste erstellt, so wird diese durch die zwei nachfolgenden Schritte in einen Dokumentationskommentar umgewandelt. Im Flussdiagramm sind diese Prozesse gelb unterlegt (siehe Abbildung 4.3) und werden in den nächsten Abschnitten noch im Detail erklärt.
Ist der Dokumentationskommentar erstellt, so wird dieser zur aktuell betrachteten Signatur hinzugefügt und die Signaturliste weiter abgearbeitet. Sobald alle Signaturen verarbeitet wurden, besitzen diese alle einen eindeutigen Dokumentationskommentar. Nachfolgend kann daher der Pseudo-Quelltext gebaut und mit den Dokumentationskommentaren annotiert werden. Mit diesem Schritt endet das Vereinigungsverfahren.
Der gebaute Pseudo-Quelltext dient in der dritten Phase als Grundlage für das JavaDoc-Tool. Dieses erstellt wie bei normalen Java-Projekten eine Dokumentation aus den gegebenen Kommentaren und der Struktur des Quelltextes. Bei diesem Schritt ist es möglich alle bekannten Parameter des JavaDoc-Tools zu verändern. So lässt sich insbesondere das Doclet anpassen, um eine benutzerdefinierte Dokumentation zu erstellen.

4.3.1 Vereinigung von gleichen Tags

In diesem Schritt werden gleiche Tags in der Tag-Liste zusammengefasst und dadurch eine neue modifizierte Tag-Liste erstellt. Dabei werden Tags als gleich angesehen, wenn sie sowohl den gleichen Namen, als auch den gleichen Informationstypen besitzen. Die Tags werden alle einzeln nacheinander betrachtet und gegebenenfalls in die neue Tag-Liste aufgenommen. Zudem werden die Tags in der von der SPL festgelegten Feature-Reihenfolge bearbeitet, da auch die Verfeinerung der Signaturen in dieser Reihenfolge stattfindet.
Für jeden Tag der Liste wird geprüft, ob bereits ein Tag mit dem gleichen Namen und Informationstypen in der Liste vorhanden ist. Ist dies nicht der Fall, so wird der aktuell betrachtete Tag in die neue Tag-Liste übernommen und der nächste Tag der Liste betrachtet. Existiert aber bereits ein solcher Tag, so ergeben sich drei Möglichkeiten, um mit dem doppelten Tag zu verfahren. Der Tag kann verworfen werden, den vorhandenen Tag überschreiben oder mit dem vorhandenen Tag zusammengefasst werden. Um zu entscheiden, welche der drei Methoden Verwendung findet, wird zunächst die Priorität des aktuellen Tags mit der des alten Tags verglichen. Ist die Priorität des aktuellen Tags kleiner, so wird er *verworfen* und nicht weiter betrachtet. Ist sie hingegen größer, so *überschreibt* der neue den bereits vorhandenen alten Tag. Wird der alte Tag überschrieben, so ersetzt der Inhalt des neuen Tags vollständig den Inhalt des alten Tags. Haben beide Tags die gleiche Priorität, so wird durch eine Regel entschieden, wie mit dem Tag verfahren wird. Welche Regel angewendete wird, hängt vom jeweiligen Tag ab. Eine Übersicht über alle Regeln für die bekannten Tags ist in Tabelle 4.1 abgebildet. Die Tabelle zeigt, dass die meisten Tags bei gleicher Priorität *zusammengefasst* werden. Ausnahmen bilden der `@author`- und der `@see`-Tag, welche jeweils verworfen werden, falls bereits ein gleicher Tag mit der selben Priorität vorhanden ist. Diese Verhalten liegt in der Tatsache begründet, dass für diese Tags, wie oben beschrieben, der gesamte Tag-Inhalt als Name verwendet wird. Daher unterscheiden sich zwei gleiche `@author`- oder `@see`-Tags auch nicht im Inhalt und es ergibt keinen Sinn diese Tags zusammenzufassen. Eine

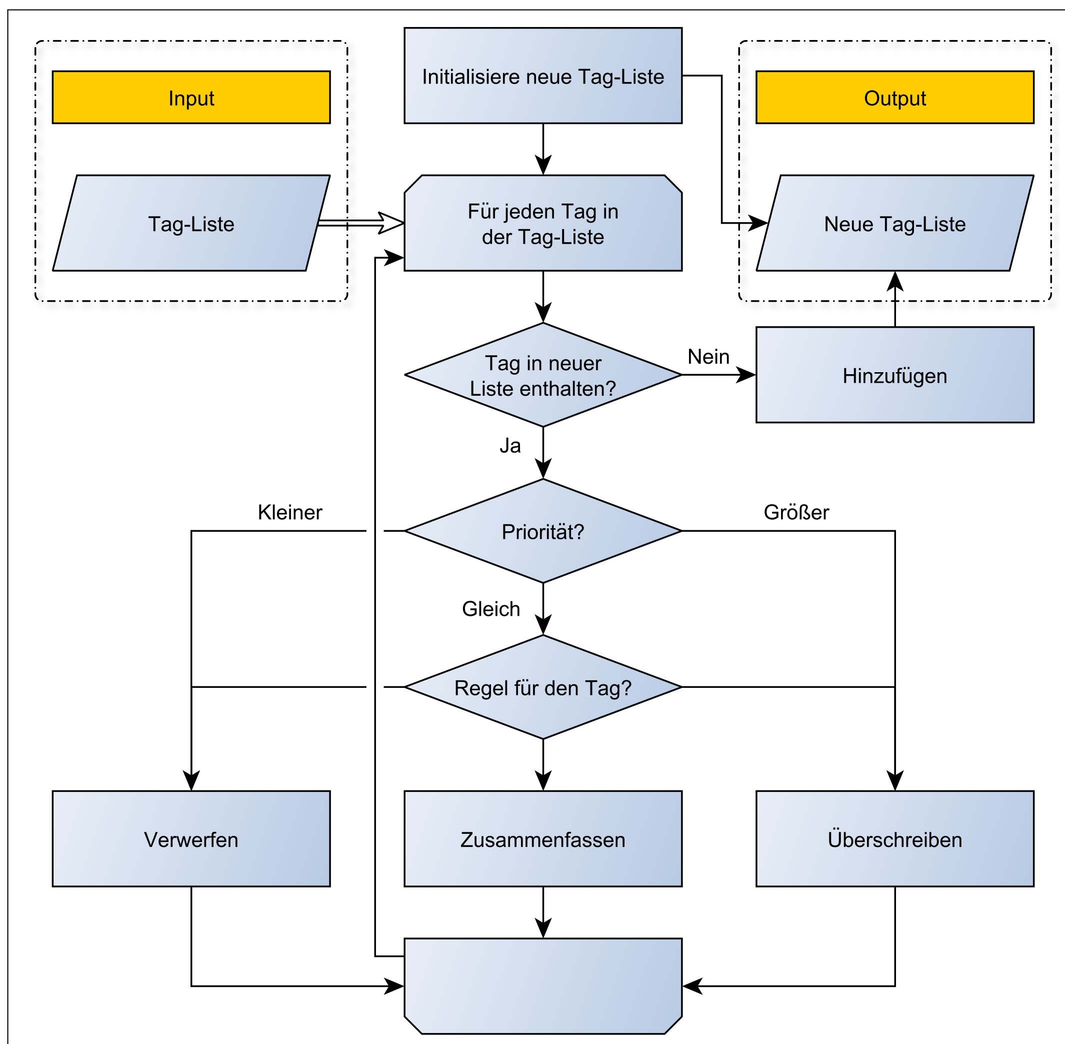

Abbildung 4.4: Flussdiagramm für die Vereinigung von gleichen Tags

weitere Ausnahmen bei den Regel für Tags mit gleicher Priorität sind der `@since`- und der `@version`-Tag, welche jeweils den bereits vorhandenen Tag überschreiben. Diese Tags geben jeweils eine Versionsnummer an und es würde keinen Sinn ergeben in der finalen Dokumentation an dieser Stelle mehrere Versionsnummern anzugeben. Daher werden gleiche `@since`- oder `@version`-Tags nicht zusammengefasst, sondern überschrieben. Demnach wird immer diejenige Versionsnummer verwendet, welche im letzte Tag in der Feature-Reihenfolge angegeben ist.
Nachdem alle Tags abgearbeitet wurden, ist die neue Tag-Liste vollständig und kann an den nächsten Prozess des Vereinigungsverfahrens übergeben werden, welcher aus den Tags der neuen Liste einen Dokumentationskommentar erzeugt.

Beispiel

Das Verarbeiten der Tags soll an dem Beispiel in Abbildung 4.2 nochmals verdeutlicht werden. Es wird beispielhaft angenommen, dass eine Produkt-Dokumentation generiert werden soll, in der beide Feature enthalten sind. Demnach gibt die Feature-Reihenfolge vor, dass zuerst das Feature *Chat* und danach das Feature *UserCommands* behandelt wird. Insgesamt sind sechs Tags in der Tag-Liste enthalten, welche

Tag	Zusammenfassen	Überschreiben	Verwerfen
Unbekannter Tag	•		
Beschreibung	•		
`@author`			•
`@deprecated`	•		
`@param`	•		
`@return`	•		
`@see`			•
`@serial`	•		
`@serialData`	•		
`@serialField`	•		
`@since`		•	
`@throws`	•		
`@version`		•	

Tabelle 4.1: Übersicht über die Regeln für das Vereinigen von Tags der Modul-Kommentare (• = Zutreffende Regel für den Tag)

in dieser Reihenfolge in den Modul-Kommentaren gefunden wurden. Die Liste wird nun sequentiell abgearbeitet und die Tags in die neue Tag-Liste übernommen. Der erste Tag der Liste ist der Beschreibungs-Tag (Zeile 3) mit der Priorität 0 und dem Informationstyp *allgemein*. Da noch kein solcher Tag in der neuen Liste enthalten ist (die Liste ist leer), wird der Tag zu dieser hinzugefügt. Es folgt der *allgemeine* `@param`-Tag mit der Priorität 0 (Zeile 4). Wie oben erwähnt, wird das erste Wort hinter einem `@param`-Tag zum Tag-Namen hinzu gezählt, sodass sich in diesem Fall der Name „`param line`" ergibt. In der neuen Tag-Liste ist noch kein Tag dieses Namens, sodass der Tag der Liste hinzugefügt wird. Bei dem nächsten Tag handelt es sich wieder um einen Beschreibungs-Tag (Zeile 6–7). Der Informationstyp ist *Feature-spezifisch* und die Priorität ist 0. In der Liste befindet sich bereits ein Beschreibungs-Tag, dieser hat jedoch einen anderen Informationstypen (*allgemein*). Daher wird der aktuelle Tag ebenfalls in die Liste übernommen. Der nächste Tag ist ein `@param`-Tag (Zeile 8) mit dem Informationstyp *Feature-spezifisch* und Priorität 0. Wie auch beim vorherigen Tag befindet sich in der Tag-Liste bereits ein Tag mit dem gleichen Namen aber einem anderen Informationstypen (*allgemein*), Somit wird auch dieser Tag in die Liste übernommen.
Beim nächste Tag handelt es sich um einen *Feature-spezifischen* Beschreibungs-Tag (Zeile 13) mit der Priorität 0. In der Liste befindet sich bereits ein Tag mit diesem Namen und dem gleichen Informationstypen. Die Prioritäten der beiden Tags sind gleich (0) und die Regel für Beschreibungs-Tags verlangt das Zusammenfassen der Kommentare. Daher werden beide Tags vereinigt, indem der Tag-Inhalt des aktuellen Beschreibungs-Tags (`Can be used to trigger user commands.`) an den Tag-Inhalt des bereits vorhanden Tags (`Creates a new {@link TextMessage} and sends it to the server`) angefügt wird. Der letzte Tag der Liste ist ein *Feature-spezifischer* `@param`-Tag (Zeile 15–16) mit der Priorität 1. Da der `@param`-Tag den Parameter `line` beschreibt, ergibt sich der Tag-Name „`param line`". In der Liste befindet sich bereits ein Tag mit diesem Namen und dem gleichen Infor-

mationstypen. Die Priorität des aktuellen Tags ist allerdings größer (1) als die des bereits vorhanden Tags (0). Folglich ersetzt der Inhalt des neuen Tags vollständig den Inhalt des alten Tags.
Am Ende enthält die neue Tag-Liste einen allgemeinen und einen Feature-spezifischen Beschreibungs-Tag und einen allgemeinen und einen Feature-spezifischen `@param`-Tag.

4.3.2 Generierung von Dokumentationskommentaren

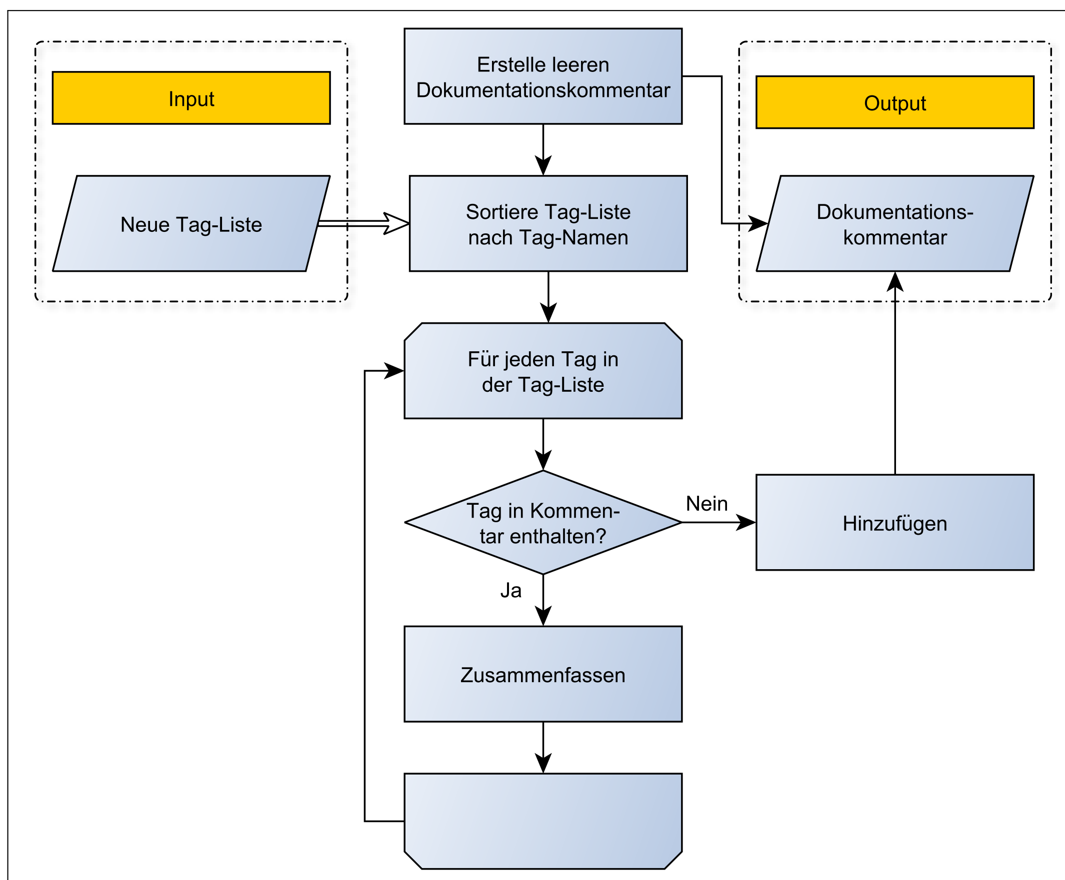

Abbildung 4.5: Flussdiagramm für die Generierung von Dokumentationskommentaren

In diesem Schritt des Vereinigungsverfahren werden die Tags in der neuen Tag-Liste zu einem Dokumentationskommentar zusammengefügt (siehe Abbildung 4.5). Dazu wird zunächst ein leerer Dokumentationskommentar erstellt und die neue Tag-Liste nach den Tag-Namen sortiert. Das Sortieren sorgt dafür, dass die Informationen in den Dokumentationskommentaren in der richtigen Reihenfolge stehen, damit Java-Doc diese später ohne Probleme verarbeiten kann.
Im vorherigen Schritt wurden Tags mit gleichen Namen und Informationstypen bereits zusammengefasst wurden. Sind daher in der Liste noch Tags mit gleichen Namen, so muss einer von ihnen ein allgemeiner und der andere ein Feature-spezifischer Tag sein. In diesem Fall werden die beiden Tags durch die Konkatenation ihrer Beschreibungen zusammengefasst und in den Dokumentationskommentar übernommen. Ist für einen bestimmten Tag-Namen nur ein Tag in der Liste, so wird dieser ohne weitere Bearbeitung in den Dokumentationskommentar übernommen. Das

Übernehmen der Tags in den Dokumentationskommentar hängt allerdings noch von dem jeweiligen Anwendungsfall ab. Bei der Produkt-, der Kontext- und der Feature-Modul-Dokumentation ist ein hoher Grad an Informationen vorhanden. Daher werden hier die allgemeinen zusammen mit allen Feature-spezifischen Informationen verwendet. Die SPL-Dokumentation bietet dagegen eher eine Übersicht und beschreibt nicht jede Signatur detailreich, sodass nur die Feature-spezifischen Informationen mit eine Priorität größer als 0 und die allgemeinen Informationen verwendet werden.

Beispiel

Der finale Schritt in der Vereinigung der Modul-Kommentare wird verdeutlicht, indem das Beispiel aus Abbildung 4.2 fortgeführt wird. In der Liste befinden sich ein allgemeiner und ein Feature-spezifischer Beschreibungs-Tag und ein allgemeiner und ein Feature-spezifischer `@param`-Tag. Sowohl die Beschreibungs-Tags, als auch die `@param`-Tags werden zusammengefasst, indem jeweils ihre Tag-Beschreibungen konkateniert werden. Der durch dieses Beispiel entstandene Pseudo-Quelltextes ist in Abbildung 4.6 zu sehen. Es ist zuerkennen, dass alle relevanten Informationen für die Produkt-Dokumentation in den Dokumentationskommentar übernommen wurden. Alle allgemeinen Informationen wurden mit den Feature-spezifischen Informationen zusammengefasst und sind im Dokumentationskommentar enthalten. Weiterhin ist zu sehen, dass die höher priorisierten Feature-spezifischen Informationen aus dem Feature *UserCommands* die Feature-spezifischen Informationen aus dem Feature *Chat* überschrieben haben.

```
 1 /**                                                  Pseudo-Quelltext
 2  * Sends a message to the Server.</br>
 3  * Creates a new {@link TextMessage} and
 4  * sends it to the server.</br>
 5  * Can be used to trigger user commands.
 6  *
 7  * @param line
 8  *     the message content.</br>
 9  *     If the message starts with a /,
10  *     the whole line is interpreted as user command.
11  */
12 public static void sendMessage(String line) {}
```

Abbildung 4.6: Pseudo-Quelltext für die Methode `sendMessage` der Klasse `Client` in der Chat-Produktlinie

4.4 Zusammenfassung

In Kapitel 3 wurden vier Anwendungsfälle zur Dokumentation einer Produktlinie vorgestellt. Um ein einheitliches Verfahren zu schaffen, welches für alle Anwendungsfälle Dokumentationen erzeugen kann und dennoch keines der in Kapitel 3 beschriebenen Probleme aufweist, wurde ein Konzept mit drei sukzessiven Phasen entworfen. Die Entwickler annotieren ihrer Produktlinie in der ersten Phase mit den neu eingeführten Modul-Kommentaren. Diese erweitern leicht die JavaDoc-Syntax

und ermöglichen dadurch die Kategorisierung und Priorisierung der Informationen in den Kommentaren. Soll eine Dokumentation für einen Anwendungsfall generiert werden, so beginnt Phase zwei, in der die Modul-Kommentare durch ein Vereinigungsverfahren zusammengefasst werden. Das Vereinigungsverfahren erzeugt Dokumentationskommentare und Pseudo-Quelltext für den jeweiligen Anwendungsfall. Das JavaDoc-Tool kann aus den Dokumentationskommentaren und dem Pseudo-Quelltext eine Dokumentation generieren.
Durch den modularen Aufbau ergeben sich die Vorteile des Konzept. Die Modul-Kommentare müssen nur einmalig erstellt werden und können für jeden Anwendungsfall verwendet werden. Für jeden Anwendungsfall ist die Generierung einer angepassten Dokumentation möglich. Des Weiteren ist das JavaDoc-Tool in der Lage unabhängig vom restlichen Verfahren zu arbeiten und daher in seinem Funktionsumfang nicht eingeschränkt.

5. Implementierung

In Kapitel 4 wurde ein Konzept zur Generierung von Dokumentationen für die Feature-orientierte Programmierung (FOP) vorgestellt. Das Konzept baut auf den Modul-Kommentaren und dem Vereinigungsverfahren auf. Dieses Kapitel beschreibt die Umsetzung dieser beiden Komponenten und beleuchtet zum einen die Syntax der Modul-Kommentare und zum anderen die Implementierungsdetails des erstellten Prototypen *SPLDoctor* (**S**oftware **P**roduct **L**ine **Doc**umentation Genera**tor**).

5.1 Umsetzung der Modul-Kommentare

Das Konzept zur Generierung der Dokumentationen für Software-Produktlinien (SPLs) beruht auf der Nutzung der neu eingeführten Modul-Kommentare. Diese sorgen für eine Kategorisierung (allgemein oder Feature-spezifisch) und Priorisierung der Informationen zu einer Signatur. Um die Informationen in dieser Art verwalten zu können, wurden in Kapitel 4 die Verwendung der neuen Schlüsselwörter `general`, `feature` und `new` vorgeschlagen. Im Nachfolgenden wird gezeigt mit welcher Syntax die neuen Schlüsselwörter innerhalb der Modul-Kommentare umgesetzt werden.
Es gibt verschiedene Möglichkeiten die neuen Schlüsselwörter in die Kommentare einzubinden. In den nächsten Abschnitten werden drei Alternativen und deren Vor- und Nachteile betrachtet. Es ist möglich die Schlüsselwörter innerhalb eines Modul-Kommentars als HTML- oder JavaDoc-Tags zu verwenden. Eine weitere Möglichkeit besteht darin einen eigenständigen Modul-Kommentar für jedes Schlüsselwort zu verwenden. Die drei Möglichkeiten sind in Tabelle 5.1 dargestellt. Es wird beispielhaft das Schlüsselwort `general` mit Angabe der Priorität 1 verwendet. Die Verwendung der Schlüsselwörter `feature` und `new` verläuft genau analog dazu. Zur weiteren Verdeutlichung der drei Möglichkeiten wird das Beispiel in Abbildung 2.6 benutzt. Die in dem Beispiel enthaltenen JavaDoc-Kommentare wurde durch Verwendung der neuen Schlüsselwörter in Modul-Kommentare umgewandelt. Für jede mögliche Umsetzung der Modul-Kommentare wird nachfolgend ein Beispiel präsentiert.

HTML-Tag

HTML-Tags sind ein normaler Bestandteil von JavaDoc-Kommentaren, sie werden verwendet, um den Text in der Dokumentation zu strukturieren und zu formatie-

Form	Beispiel
HTML-Tag	<general p = 1> ... </general>
JavaDoc-Tag	{@general 1 ... }
Eigenständiger Kommentar	/**{@general 1} * ... */

Tabelle 5.1: Übersicht über mögliche Formen eines neuen Schlüsselwortes

ren. Es bietet sich daher an, die neuen Schlüsselwörter als HTML-Tags zu verwenden. Mithilfe der HTML-Tags lassen sich den Informationen innerhalb eines Modul-Kommentars ein Informationstyp (allgemein oder Feature-spezifisch) und eine Priorität zuordnen. Ein Beispiel, wie die Verwendung von HTML-Tags aussehen könnte ist in Abbildung 5.1 zu sehen. Der Modul-Kommentar im Feature *Chat* enthält allgemeine Informationen mit der Priorität 0 (Zeile 2–6) und Feature-spezifische Informationen mit der Priorität 0 für das Feature *Chat* (Zeile 7–9). Im Feature *Encryption* enthält der Modul-Kommentar ebenfalls Feature-spezifische Informationen, hier aber mit der höheren Priorität 1 (Zeile 12–17). Auch der Modul-Kommentar in dem Feature *Spam* enthält ausschließlich Feature-spezifische Informationen. Hier ist die Beschreibung des Kommentars mit der Priorität 1 und der `@return`-Tag mit der Priorität 2 angegeben.
HTML-Tags haben den Vorteil, dass sie ohne Probleme in JavaDoc-Kommentaren verwendet werden können. Der Entwickler muss sich bei ihrer Verwendung nicht an neue Sprachelemente gewöhnen. HTML-Tags haben zusätzlich den Vorteil, dass sie noch flexibler als JavaDoc-Tags innerhalb des ganzen Kommentars verwendet werden können. Allerdings gibt es auch einen Nachteil bei der Verwendung von bekannten Sprachelementen wie HTML-Tags. Bei der Verwendung von HTML-Tags besteht die Verwechslungsgefahr mit normalen HTML-Elementen zur Formatierung des Textes wie `<b>` oder `</br>`.

JavaDoc-Tag

JavaDoc-Tags bilden einen fundamentalen Bestandteil von JavaDoc-Kommentaren. Sie werden zur Organisation und Formatierung der Informationen in den JavaDoc-Kommentare verwendet und eignen sich daher gut zur Umsetzung der neuen Schlüsselwörter. Wie auch bei HMTL-Tags umschließen JavaDoc-Tags die Informationen innerhalb von Modul-Kommentaren mit einem bestimmten Informationstyp und einer bestimmten Priorität. In Abbildung 5.2 ist ein Beispiel für die Verwendung von JavaDoc-Tags gegeben. Die Informationstypen und Prioritäten sind dabei identisch zu dem Beispiel für HTML-Tags (siehe Abbildung 5.1). Im Feature *Chat* ist die Beschreibung des Modul-Kommentars in allgemeine Informationen mit der Priorität 0

```
1  /**                                                             Chat
2   * <general p=0>
3   *   Edits incoming messages.
4   *   @param msg the message to edit
5   *   @return the edited message
6   * </general>
7   * <feature p=0>
8   *   Hook method.
9   * </feature>
10  */
11 private Message incomingAction(Message msg) { return msg; }
12 /**                                                        Encryption
13  * <feature p=1>
14  *   Decodes the given message.
15  * </feature>
16  */
17 private Message incomingAction(Message msg) { ... }
18 /**                                                             Spam
19  * <feature p=1>
20  *   Checks the given message for spam.
21  * </feature>
22  * <feature p=2>
23  *   @return null if message is identified as spam
24  * </feature>
25  */
26 private Message incomingAction(Message msg) { ... }
```

Abbildung 5.1: Modul-Kommentare mit HTML-Tags anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie)

(Zeile 2) und Feature-spezifische Informationen mit der Priorität 0 (Zeile 3) aufgeteilt. Der `@param`- und der `@return`-Tag enthalten jeweils allgemeine Informationen mit der Priorität 0 (Zeile 4 und 5).
JavaDoc-Tags weisen im Bezug auf Vor- und Nachteile viele Parallelen zu HTML-Tags auf. Auch JavaDoc-Tags sind bekannte Sprachelemente von JavaDoc-Kommentaren. So ist der Entwickler an die Verwendung von JavaDoc-Tags gewöhnt, was die Erstellung der Modul-Kommentare erleichtert. Andererseits besteht jedoch auch hier wieder die Verwechslungsgefahr zu anderen JavaDoc-Tags, wie zum Beispiel `{@link ...}` oder `{@code ...}`. Im Gegensatz zu HTML-Tags werden die JavaDoc-Tags jedoch nur auf einzelne Abschnitte des Modul-Kommentars angewandt, da bei einer Überschneidung der neuen JavaDoc-Tags mit den herkömmlichen JavaDoc-Tags die Übersichtlichkeit der Modul-Kommentare zu stark leiden würde. Allerdings führt dieses Verhalten dazu, dass die Modul-Kommentare unnötig aufgebläht werden.
Ein interessanter Aspekt bei der Verwendung von selbst definierten JavaDoc-Tags, ist die Verwendung von angepassten Doclets. Ein Teil der Arbeit könnte vom Doclet und somit vom JavaDoc-Tool übernommen werden. Da dies allerdings eine Einschränkung in der Funktionalität des JavaDoc-Tools darstellt, da in diesem Falle

keine anderen Doclets verwendet werden könnten, ist diese Methode ungeeignet für den Prototypen. Außerdem birgt das Doclet zusätzlichen Wartungsaufwand und müsste bei einer neuen JavaDoc Version überarbeitet werden.

```
1  /**                                                            Chat
2   * {@general 0 Edits incoming messages.}
3   * {@feature 0 Hook method.}
4   * @param msg {@general 0 the message to edit}
5   * @return {@general 0 the edited message}
6   */
7  private Message incomingAction(Message msg) { return msg; }
8  /**                                                      Encryption
9   * {@feature 1 Decodes the given message.}
10  */
11 private Message incomingAction(Message msg) { ... }
12 /**                                                            Spam
13  * {@feature 1 Checks the given message for spam.}
14  * @return {@feature 2 null if message
15  *   is identified as spam}
16  */
17 private Message incomingAction(Message msg) { ... }
```

Abbildung 5.2: Modul-Kommentare mit JavaDoc-Tags anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie)

Eigenständiger Kommentar

Eine weitere Alternative, um die Informationen der Modul-Kommentare zu strukturieren, ist die Erstellung eines eigenständigen Kommentars pro Informationstyp und Priorität. Das bedeutet alle Informationen einer Signatur in einem Feature-Modul mit dem selben Informationstyp und der selben Priorität sind in einem eigenen Modul-Kommentar enthalten. Das Beispiel in Abbildung 5.3 veranschaulicht die Verwendung der eigenständigen Kommentare. Wiederum sind die Informationstypen und Prioritäten identisch zu denen in dem Beispiel für HTML-Tags (siehe Abbildung 5.1). Die Methode `incomingAction` im Feature *Chat* ist mit insgesamt zwei Modul-Kommentaren annotiert. Der erste Modul-Kommentar enthält allgemeine Informationen mit der Priorität 0 (Zeile 1–5), während der zweite die Feature-spezifischen Information mit der Priorität 0 für das Feature *Chat* enthält (Zeile 6–8). Der eigenständige Modul-Kommentar bietet den Vorteil die verschiedenen Informationen völlig unabhängig voneinander beschreiben zu können. Im eigenständigen Modul-Kommentar können sowohl Beschreibung also auch beliebig viele Tags verwendet werden. So steigt durch die Trennung in mehrere Kommentare die Übersichtlichkeit innerhalb des Quelltextes. Ein weiterer Vorteil im Vergleich zu HTML- und JavaDoc-Tags ist der geringere Schreibaufwand für die Schlüsselwörter. Auch lassen sich die eigenständigen Modul-Kommentaren leichter vom Computer einlesen, da dieser nicht innerhalb der Modul-Kommentare nach Schlüsselwörtern suchen muss.

```
1  /**{@general 0}                                                      Chat
2   * Edits incoming messages.
3   * @param msg the message to edit
4   * @return the edited message
5   */
6  /**{@feature 0}
7   * Hook method.
8   */
9  private Message incomingAction(Message msg) { return msg; }
10 /**{@feature 1}                                                Encryption
11  * Decodes the given message.
12  */
13 private Message incomingAction(Message msg) { ... }
14 /**{@feature 1}                                                      Spam
15  * Checks the given message for spam.
16  */
17 /**{@feature 2}
18  * @return null if message is identified as spam
19  */
20 private Message incomingAction(Message msg) { ... }
```

Abbildung 5.3: Modul-Kommentare mit eigenständigen Kommentaren anhand der Methode `incomingAction` in der Klasse `Client` (Chat-Produktlinie)

Auswahl der konkreten Syntax

Aufgrund der oben beschriebenen Vor- und Nachteile fiel die Wahl auf den eigenständigen Modul-Kommentar. Gerade durch die gute Strukturierung der Modul-Kommentare, die bei der Verwendung von eigenständigen Kommentaren entsteht, profitiert die Übersichtlichkeit.

Um die Arbeit der Entwickler zu erleichtern und die Kommentare nicht unnötig zu vergrößern, gelten zusätzlich noch die folgenden Vereinfachungen. Einerseits besteht die Möglichkeit die Priorität an einem Schlüsselwort nicht anzugeben. Ein Kommentar ohne Angabe einer Priorität erhält automatisch die niedrigste Priorität 0. Das Schlüsselwort `{@feature 0}` kann somit problemlos durch `{@feature}` ersetzt werden. Die Nichtangabe der Priorität hat vor allem den Vorteil, dass die Arbeit der Entwickler erleichtert wird. Sind die Priorität bei einer bestimmten Signatur nicht wichtig, so besteht für den Entwickler auch kein Grund sich darüber Gedanken zu machen.

Weiterhin besteht die Möglichkeit einen Modul-Kommentar ohne die Angabe eines Schlüsselwortes zu verwenden. Der Modul-Kommentar wird in dem Fall so interpretiert, als wenn das Schlüsselwort `{@general 0}` verwendet wurde. Wie auch bei der Nichtangabe einer Priorität greift hier der Vorteil, dass der Entwickler beim Erstellen der Modul-Kommentare entlastet wird. Die Vereinfachung hat aber noch einen weiteren Vorteil. Diese erlaubt SPLDoctor auch normale JavaDoc-Kommentare als Modul-Kommentare zu interpretieren. Das bedeutet, wenn ein Entwickler seine Produktlinie mit normalen JavaDoc-Kommentaren annotiert hat und diese vorerst nicht

in Modul-Kommentare umwandelt, so lassen sich trotzdem für jeden Anwendungsfall eingeschränkte Dokumentationen generieren.

5.2 Implementierung des Prototypen SPLDoctor

Das in Kapitel 4 vorgestellte Konzept wurde in dem Prototypen *SPLDoctor* (**S**oftware **P**roduct **L**ine **Doc**umentation Genera**tor**) umgesetzt. Die Grundlage für SPLDoctor bildet das Eclipse-Plugin FeatureIDE[1], welches im Nachfolgenden vorgestellt wird. Im Rahmen der Vorstellung werden die Gründe für die Verwendung von FeatureIDE genannt und die Vorteile der Kombination von FeatureIDE und SPLDoctor beleuchtet. Weiterhin werden die Implementierungsdetails für SPLDoctor betrachtet und dessen Verwendung näher erklärt.

5.2.1 Integration von SPLDoctor in FeatureIDE

Für die Realisierung des Prototypen SPLDoctor ist es erforderlich Feature-orientieren Produktlinien verwalten und analysieren zu können, um dadurch den für SPLDoctor benötigen Input zu gewinnen. Das es enorm viel Aufwand darstellt, diese Unterstützung selbst zu implementieren, hilft es eine grundlegende Umgebung zu haben, in welche SPLDoctor integriert werden kann. Diese Umgebung muss mehrere Anforderungen erfüllen, damit sie für den Prototypen geeignet ist. Die wichtigste Anforderungen besteht darin, dass die Umgebung den nötigen Input für SPLDoctor liefern kann. Dafür muss die Umgebung das Einlesen und Zerlegen des Quelltextes und der JavaDoc-Kommentare unterstützen. Das bedeutet zum einen, dass die Umgebung die Sprache Java unterstützen muss und zum anderen, dass diese in der Lage ist Quelltext von SPLs, die mit FOP erstellt wurden zu verwalten. Das heißt sie muss den SPL-Quelltext einlesen und analysieren können, die Konfiguration einer SPL unterstützen und darauf aufbauend auch die entsprechenden Produkte generieren können. Außerdem muss sie auch in der Lage sein in gleicher Weise Kontext-Interfaces für SPLs zu generieren. Allerdings ist es nicht nur notwendig, dass die Umgebung die benötigten Daten generieren kann, sie muss auch in der Lage sein die Daten an SPLDoctor zu übergeben. Dafür ist es entweder nötigt, dass die Daten aus der Umgebung exportiert werden können oder das die Umgebung um die Funktionalität von SPLDoctor erweitert werden kann, sodass die Daten direkt übergeben werden können. Die zweite Methode ist wesentlich performanter und komfortabler, setzt aber voraus, dass die Umgebung erweiterbar ist. Eine weitere Funktionalität wäre zum Beispiel ein Syntax-Highlighting, welches beim Erstellen des Quelltextes und der Modul-Kommentare hilfreich ist. Eine Umgebung, die alle genannten Anforderungen erfüllt ist FeatureIDE. FeatureIDE ist eine in Java geschriebene Erweiterung (Plugin) der Entwicklungsumgebung Eclipse[2]. Mithilfe von FeatureIDE ist es möglich SPLs zu erstellen, zu modellieren, zu editieren und zu analysieren.
Zurzeit werden viele unterschiedliche Techniken zur Erstellung von SPLs unterstützt [TKB+14]. Darunter fällt die Aspekt-orientierte Programmierung (AOP) mit AspectJ [KHH+01], die Präprozessor-Techniken Antenna[3] oder Munge[4] und auch

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[2] https://www.eclipse.org/
[3] http://antenna.sourceforge.net/index.php
[4] https://sonatype.github.io/munge-maven-plugin/

die Feature-orientierte Programmierung (FOP) mit AHEAD [Bat04], FeatureHouse [AKL13] oder FeatureC++ [ALRS05]. FeatureIDE bietet zusätzlich noch viele weitere Unterstützungen für die Erstellung von SPLs an. So besteht die Möglichkeit des grafischen und textuellen Darstellens und Editieren von Feature-Modellen. FeatureIDE beinhaltet einen grafischen Konfigurationseditor und mehrere Navigationswerkzeuge für eine SPL, wie ein Kollaborationsdiagramm oder eine Outline. Weiterhin bietet FeatureIDE mehrere Analysen für SPLs zur Aufdeckung von Fehlern innerhalb des Quelltextes und des Modells und zur Erstellung von Statistiken an. Da FeatureIDE alle Anforderungen erfüllt und zudem noch ohne Probleme erweiterbar ist, wurde SPLDoctor mit Java implementiert und in FeatureIDE integriert.
Wie in Abschnitt 2.3 beschrieben ist bislang noch keine Möglichkeit bekannt Dokumentationen für FOP-Projekte zu erzeugen. So bietet auch FeatureIDE bisher noch keine Funktionalität dieser Art an. Lediglich die herkömmliche Generierung einer JavaDoc-Dokumentation ist mittels Eclipse möglich. Allerdings kann diese auch nur bei normalen Java Anwendungen verwendet werden.
Durch die Implementierung und Integration von SPLDoctor wurde FeatureIDE um die Funktionalität erweitert, Dokumentationen für FeatureHouse-Produktlinien zu generieren. Die Dokumentationen können für jeden in Abschnitt 3.2 vorgestellten Anwendungsfall generiert werden. Somit ist es nun möglich SPLDoctor und damit auch das Konzept mittels einiger Beispiel-Produktlinien zu testen. Momentan werden von FeatureIDE die benötigten Informationen zur Generierung der Dokumentation nur für FeatureHouse-Produktlinien generiert. Das bedeutet der implementierte Prototyp SPLDoctor ist bislang auf die mit FeatureHouse entwickelten Produktlinien beschränkt. Dennoch ist das entwickelte Konzept so allgemein gehalten, das es auch mit anderen FOP-Modellen funktionieren würde.

5.2.2 Details zur Implementierung von SPLDoctor

Die Implementierung von SPLDoctor liegt nahe an dem in Abschnitt 4.3 vorgestellten theoretischen Ablauf des Vereinigungsverfahrens. Von FeatureIDE wird der benötigte Input bereitgestellt.
FeatureIDE ist in der Lage den Quelltext einzulesen und zu analysieren. Es erstellt eine Liste mit allen in der SPL befindlichen Signaturen und filtert diese anschließend für den entsprechenden Anwendungsfall, sodass am Ende nur jene Signaturen übrig bleiben, die für den gewählten Anwendungsfall relevant sind. Dabei verwendet FeatureIDE zum Filtern der Signaturen den Sat-Solver Sat4J[5]. Ein Sat-Solver dient dazu das Sat-Problem eines aussagenlogischen Ausdrucks zu lösen, was dafür verwendet werden kann, um das Feature-Modell in Form eines aussagenlogischen Ausdrucks (siehe Abschnitt 2.2.2) zu analysieren. SPLDoctor verwendet die gefilterten Signaturen als Input und bearbeitet die Signaturen nach dem in Abschnitt 4.3 vorgestellten Ablauf. Dafür geht SPLDoctor die Signaturliste sequenziell durch und betrachtet jede Signatur mit ihren entsprechenden Modul-Kommentaren einzeln. Für jede Signatur werden die Modul-Kommentare in ihre einzelnen Tags zerlegt und jeweils eine Liste mit allgemeinen und mit Feature-spezifischen Tags erstellt. Beide Listen werden mit einer Art Sort-Merge-Join vereinigt und bilden so den fertigen Dokumentationskommentar für die jeweilige Signatur. Sobald die Signaturliste abgearbeitet ist, kann SPLDoctor aus den Signaturen den Pseudo-Quelltext erstellen

[5] http://www.sat4j.org/

und mit den Dokumentationskommentaren annotieren. Der Pseudo-Quelltext wird in einem eigenen Verzeichnis als Java-Dateien gespeichert. Abschließend wird das JavaDoc-Tool mit Verweis auf den Pseudo-Quelltext gestartet und generiert die finale Dokumentation.
Zurzeit muss die Generierung der Dokumentationen noch vom Nutzer manuell veranlasst werden, da noch keine automatisierte Generierung implementiert ist. SPLDoctor erweitert das Kontextmenü von FeatureIDE, sodass sich darüber die Dokumentationen für jeden Anwendungsfall erzeugen lassen. Manche Anwendungsfälle verlangen hier noch einen zusätzlichen Input. Für die Feature-Modul-Dokumentation muss das Features spezifiziert werden, für welches die Dokumentation generiert werden soll. Die Kontext-Dokumentation benötigt ebenso die Angabe eines Features, um das Kontext-Interface bilden zu können. Die Produkt-Dokumentation verwendet zur Erzeugung eines Produkts die aktuelle Konfiguration der Produktlinie. Ebenso lassen sich vor der Generierung der Dokumentation noch zusätzliche Parameter für das JavaDoc-Tool angeben. Ohne weitere Angabe eines bestimmten Doclets wird das Standard-Doclet von JavaDoc verwendet. Das Standard-Doclet erzeugt aus dem generierten Pseudo-Quelltext eine HTML Dokumentation.
Der Prozess der Generierung ist in Abbildung 5.4 noch einmal bildlich dargestellt. Bei dem abgebildeten Dokumentationsausschnitt handelt es sich um die SPL-Dokumentation der Chat-Produktlinie, mit den Methoden `incomingAction` und `sendMessage` aus der Klasse `Client`.

5.3 Zusammenfassung

In diesem Kapitel wurde die Umsetzung des in Kapitel 4 vorgestellten Konzepts erläutert. Die Syntax der Modul-Kommentare wurde beschrieben und der Prototyp SPLDoctor (**S**oftware **P**roduct **L**ine **Doc**umentation Genera**tor**) vorgestellt.
Für die Umsetzung der neuen Schlüsselwörter in den Modul-Kommentare wurden die Vor- und Nachteile von HTML-Tags, JavaDoc-Tags und eigenständigen Kommentaren abgewogen. Dabei fiel die Wahl auf die eigenständigen Kommentare, welche die Informationen der Modul-Kommentare auf einen Kommentar pro Informationstyp und Priorität aufteilen. Dadurch erreichen die Modul-Kommentare eine gute Übersichtlichkeit und eine hohe Flexibilität.
Die Modul-Kommentare dienen als Input für den Prototypen SPLDoctor. Dieser setzt das in Kapitel 4 vorgestellte Vereinigungsverfahren um und generiert mithilfe des JavaDoc-Tools Dokumentationen für alle Anwendungsfälle. Integriert wurde SPLDoctor in das Eclipse-Plugin FeatureIDE. Dafür gab es mehrere Gründe. Der Hauptgrund bestand darin, dass FeatureIDE in der Lage ist den von SPLDoctor benötigten Input bereitzustellen. Insbesondere unterstützt FeatureIDE das Bilden von Kontext-Interfaces, welche die Grundlage der Kontext-Dokumentationen bildet. Des Weiteren bietet FeatureIDE den Entwicklern von Produktlinien ein breite Palette an Hilfen zur Erstellung, Modellierung und Analyse von Produktlinien an. Außerdem unterstützt FeatureIDE noch eine Vielzahl an weiteren Implementierungstechniken für SPLs, sodass SPLDoctor auch potentiell auf andere SPL-Paradigmen und Programmiersprachen erweitert werden könnte.

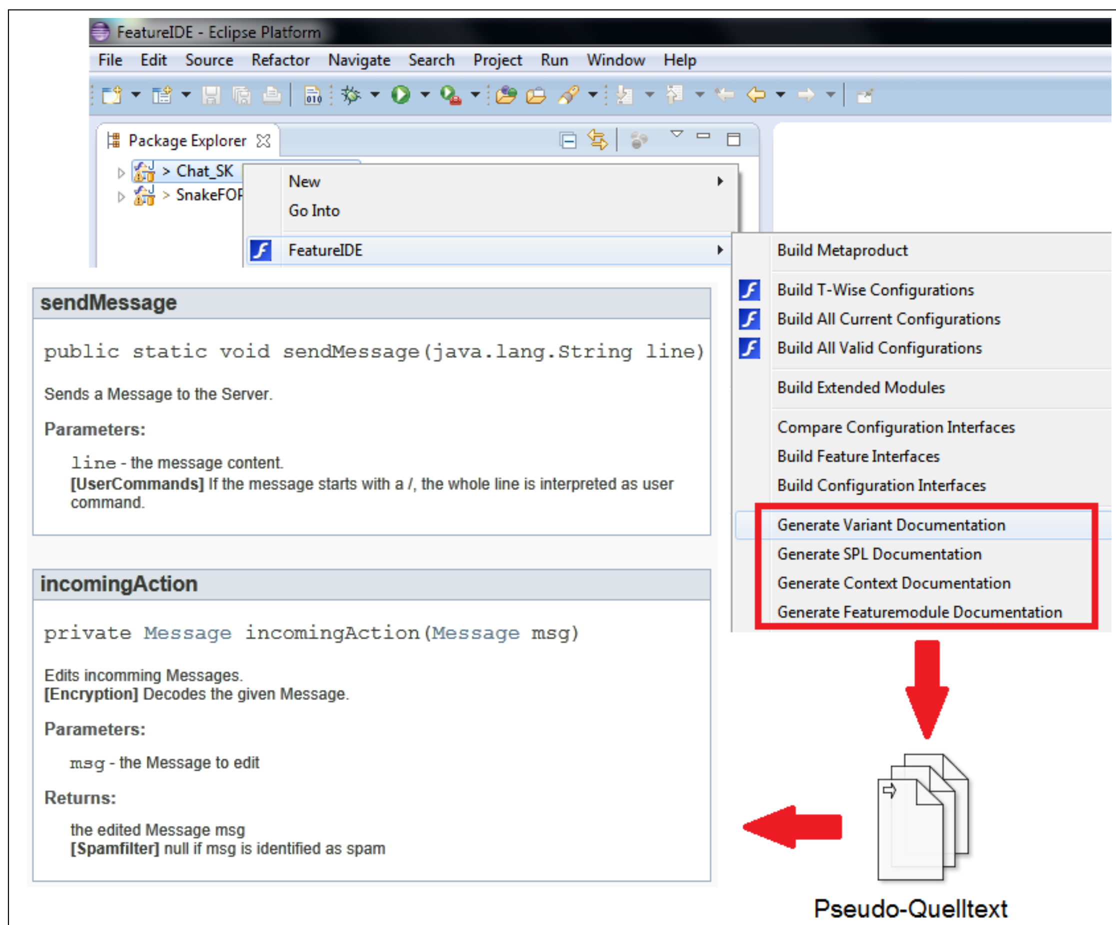

Abbildung 5.4: Erstellen einer Dokumentationen

Da es sich bei SPLDoctor zur Zeit noch um einen Prototypen handelt, existieren noch einige kleinere Einschränkungen. Bislang wird bedingt durch JavaDoc nur die Sprache Java unterstützt. Das könnte sich ändern, falls SPLDoctor um weitere Dokumentationsgeneratoren erweitert wird. Weiterhin kann FeatureIDE momentan nur für FeatureHouse-Produktlinien den benötigten Input für SPLDoctor liefern. Das bedeutet, dass bisher nur für FeatureHouse-Produktlinien die Dokumentationen generiert werden können.

6. Evaluierung

In diesem Kapitel wird das in Kapitel 4 vorgestellte Konzept und der darauf aufbauende Prototyp SPLDoctor evaluiert. SPLDoctor erstellt für vier verschiedene Anwendungsfälle mit Dokumentationskommentaren annotierten Pseudo-Quelltext. Anschließend generiert SPLDoctor unter Verwendung des JavaDoc-Tools aus dem Pseudo-Quelltext eine Dokumentation.
Es wird gezeigt, wie die Evaluierung durchgeführt wird und welche Art von Ergebnissen daraus resultiert. Die Evaluierung wird anhand von zwei Produktlinien durchgeführt, wobei der Fokus auf Bewertung der Effizienz und Benutzerfreundlichkeit von SPLDoctor liegt.

6.1 Ablauf der Evaluierung

Die Evaluierung wird in eine qualitative und eine quantitative Evaluierung aufgeteilt. Die qualitative Evaluierung bewertet die nicht messbaren Eigenschaften des Konzepts und den Modul-Kommentaren, wozu unter anderem die Benutzerfreundlichkeit zählt. Dagegen bewertet die quantitative Evaluierung die messbaren Ergebnisse der Modul-Kommentare, wie beispielsweise die Effizienz.

6.1.1 Qualitative Evaluierung

In dieser Arbeit wurden im Rahmen des Konzepts auch die dafür benötigten Modul-Kommentare eingeführt. Folglich gab es bisher auch keine Produktlinien, die mit Modul-Kommentaren annotiert waren. Die qualitative Evaluierung befasst sich daher mit der Frage, wie schwierig es war die Dokumentationen für die einzelnen Anwendungsfälle und die Modul-Kommentare zu erstellen und welche Probleme dabei auftraten. Dieses Vorgehen liefert eine Auskunft darüber, wie benutzerfreundlich SPLDoctor momentan ist.
Eine andere Frage für die qualitative Evaluierung wäre eine Bewertung über die Qualität der generierten Dokumentationen. Die Qualität einer Dokumentation kann allerdings nur äußerst schwierig bewertet werden. Das liegt daran, dass die Qualität

der Dokumentation stark von den Modul-Kommentaren abhängt, die die Entwicklern der Software-Produktlinie (SPL) erstellen. SPLDoctor schränkt JavaDoc in seiner Funktionalität in keiner Weise ein. Daher können alle Dokumentationen, die mit SPLDoctor erstellt werden potentiell genauso gut sein, wie jede andere Dokumentation, die mit JavaDoc erstellt werden kann.

6.1.2 Quantitative Evaluierung

Bei der quantitativen Evaluierung wird bewertet, wie effizient die von SPLDoctor genutzten Modul-Kommentare sind. Dazu werden die Modul-Kommentaren mit den Kommentaren der in Kapitel 3 vorgestellten, trivialen Ansätze verglichen.
Es werden zwei verschiedene Werte bei der quantitativen Evaluierung betrachtet. Die Anzahl der Tags in den Kommentare und die Anzahl der Zeichen aller Kommentare. Die Beschreibung eines Kommentars wird von SPLDoctor, wie in Abschnitt 4.3 beschrieben, als einzelner Tag gewertet. Daher zählt auch hier die Beschreibung eines Kommentars als einzelner Tag. Die Zeichenanzahl berücksichtigt nur den Inhalt der Kommentare, allerdings inklusive aller Schlüsselwörter. Die von JavaDoc vorangestellten * und Leerzeichen werden nicht berücksichtigt.

Vergleichsmethoden

SPLDoctor soll in der quantitativen Evaluierung jeweils einem trivialen Ansatz gegenübergestellt werden, welcher zur Erstellung einer Dokumentation für einen der vier Anwendungsfälle genutzt wird. Das heißt jeweils einem Ansatz zur Erstellung einer SPL-, Produkt-, Kontext- und Feature-Modul-Dokumentation. Da es bisher noch keine andere Möglichkeit gab Dokumentationen für diese Anwendungsfälle zu erzeugen, ist es nicht möglich SPLDoctor mit einem anderen Ansatz direkt zu vergleichen. Es können allerdings Mutmaßungen über die Ansätze für die vorgestellten Anwendungsfälle angestellt werden. Dadurch kann SPLDoctor diesen theoretischen Ansätze für die vier Anwendungsfälle gegenübergestellt werden. Die theoretischen Ansätze sollen in der Lage sein die gleichen oder zumindest vergleichbare Dokumentationen zu erzeugen, welche auch SPLDoctor erzeugen würde. Das beutetet, dass für jeden Anwendungsfall eine alternative Methode zur Erzeugung der Dokumentation gefunden werden muss. So wie SPLDoctor aus den Modul-Kommentaren Dokumentationskommentare erzeugt, benötigt auch jede der alternativen Methoden als Input irgendeine Form von Kommentaren, um diese in Dokumentationskommentare umwandeln zu können. Daher ist es möglich die Modul-Kommentare mit dem Input der jeweiligen alternativen Methode zu vergleichen. Für Produkt-, SPL und Feature-Modul-Dokumentation wird jeweils der Ansatz aus Kapitel 3 verwendet werden. Für den Kontext wird ein ähnlicher Ansatz verwendet, wie für die Feature-Modul-Dokumentation. Nachfolgend werden die Details für die zu vergleichenden Methoden erläutert.

SPLDoctor

Der Input von SPLDoctor besteht aus den Modul-Kommentaren der SPL. Daher werden sämtliche in der SPL vorhandenen Modul-Kommentare zusammengezählt.

Summe aller Anwendungsfälle

In Kapitel 3 wurde schon gezeigt, dass die trivialen Ansätze für die einzelnen Anwendungsfälle nicht kompatibel zueinander sind. Werden demnach bei einer Produktlinie Dokumentationen für jeden Anwendungsfall benötigt, so müssten auch für jeden Anwendungsfall einzelne Kommentare für die jeweiligen trivialen Ansätze geschrieben werden. SPLDoctor ist jedoch in der Lage nur mithilfe der Modul-Kommentare für jeden Anwendungsfall die Dokumentation zu erzeugen. Somit wird den Modul-Kommentaren die Summe aller Kommentare für die trivialen Lösungsansätze der Anwendungsfälle gegenübergestellt.

Produkt

Wie in Kapitel 3 beschrieben, eignet sich für die Erzeugung einer Produkt-Dokumentation als trivialer Ansatz am besten das Uniformitätsprinzip. Auch der Aufbau der vom Prototypen genutzten Modul-Kommentare lehnt sich an das Uniformitätsprinzip an. Daher wird angenommen, dass kein signifikanter Unterschied zwischen der Anzahl und Länge der Kommentare besteht. Die Produkt-Dokumentation wird deshalb hier nicht weiter betrachtet.

SPL

Für die Erzeugung der SPL-Dokumentation werden die Kommentare des SPL-Ansatzes aus Kapitel 3 verwendet. Mithilfe von SPLDoctor lassen sich vollständigen Kommentare für den SPL-Ansatz erzeugen. Diese Kommentare werden anschließend direkt mit den Modul-Kommentaren von SPLDoctor verglichen.

Kontext

Wie in Abschnitt 2.2.3 beschrieben kann das Kontext-Interface für beliebige partielle Konfigurationen des Feature-Modells gebildet werden. Daher wurde in Abschnitt 3.1 auch die Nutzung einer statischen Methode als trivialer Ansatz für die Kontext-Dokumentation ausgeschlossen. Um trotzdem vergleichbare Werte für die Kontext-Dokumentation zu generieren, wird an dieser Stelle eine Einschränkung für die Bildung des Kontext-Interfaces getroffen. In den meisten Fällen ist die Restriktion für ein Kontext-Interface, dass ein bestimmtes Feature ausgewählt ist. So kann durch das Kontext-Interface gesehen werden, welche Signaturen in dem gewählten Feature-Module verfügbar sind. Daher greift hier die Einschränkung, dass nur Restriktionen dieser Art verwendet werden dürfen. Das schränkt die Anzahl der unterschiedlichen Kontext-Interfaces auf die Anzahl der Feature ein. Durch diese Einschränkung ist es möglich einen statischen Ansatz für die Kontext-Dokumentation zu finden. Das bedeutet, es werden wie auch bei der SPL- und Feature-Modul-Dokumentation vollständige Kommentare erzeugt, die bei der Generierung der Dokumentation nicht verändert werden. Alle Signaturen, die in jedem Kontext-Interface vorkommen erhalten genau einen vollständigen Kommentar. Für allen anderen Signaturen wird in jedem Feature-Modul jeweils ein passender und vollständiger Kommentar angelegt. Dabei enthält jedes Feature-Modul genau die Informationen, die für die Bildung der Kontext-Dokumentation nötig sind. Daraus resultieren einige redundante Informationen, aber nur so lassen sich auf einfache Weise Widersprüche und fehlende Informationen beseitigen. Es lässt sich mit diesem Ansatz für jedes Feature eine

Kontext-Dokumentation erzeugen. Daher werden für jedes Feature die Kommentare für eine Kontext-Dokumentation gebildet und die Summe aller Kommentare entspricht dem finalen Wert der Evaluierung. Dabei werden die Kommentare in den Feature-Modulen, die in jeder Konfiguration enthalten sind nur einmalig gezählt.

Feature-Modul

Für die Erzeugung der Feature-Modul-Dokumentation werden die Kommentare des Feature-Modul-Ansatzes benutzt (siehe Kapitel 3). Im Gegensatz zu den vollständigen Kommentaren, die bei der SPL-Dokumentation verwendet werden, werden hier Kommentare in allen Feature-Modulen der SPL benötigt. Das liegt daran, dass jede Feature-Modul-Dokumentation nur abhängig von den Kommentaren in dem entsprechenden Feature-Modul ist. Es werden keine Informationen aus anderen Modulen benutzt. Mithilfe von SPLDoctor lassen sich die vollständigen Kommentare für jedes Feature-Modul erzeugen. Der endgültige Wert der Evaluierung für den Feature-Modul-Anwendungsfall entspricht der Summe der Kommentare aus allen Feature-Modulen.

6.2 Verwendete Produktlinien

Für die Evaluierung wurden zwei verschiedene Produktlinien ausgewählt. Die Chat-Produktlinie besteht aus mehreren Varianten einer einfachen Chat-Anwendung zwischen einem Server und mehreren Clients. Sie wurde in dieser Arbeit bereits mehrfach als Beispiel verwendet. Die Snake-Produktlinie umfasst mehrere Varianten des bekannten Spiels Snake.
Die Produktlinien weisen einige Gemeinsamkeiten auf. So wurden beide in Java implementiert und durch die Feature-orientierte Programmierung (FOP) als Produktlinie umgesetzt. Da bisher keine Produktlinie über die in dieser Arbeit entwickelten Modul-Kommentare verfügt, mussten bei beiden Produktlinien die Modul-Kommentare erst hinzugefügt werden.
Dennoch gibt es auch deutliche Unterschiede zwischen den beiden Produktlinien, da beide auf eine unterschiedliche Art entstanden sind. Während die Chat-Produktlinie direkt als SPL implementiert wurde, wurde die Snake-Produktlinie durch die Zerlegung einer normalen Java-Anwendung erstellt. Dieser Umstand wurde bewusst gewählt, damit sich die Möglichkeit ergab die Modul-Kommentare für die Produktlinien auch auf unterschiedliche Weise zu erstellen. Im Nachfolgenden sind weitere Details zu den beiden Produktlinien beschrieben.

6.2.1 Chat-Produktlinie

Die Chat-Produktlinie wurde direkt als SPL entwickelt und war bislang noch nicht mit JavaDoc-Kommentaren annotiert. Sie eignet sich daher gut, um zu testen, wie schwer es ist eine Produktlinien ohne Kommentare mit den Modul-Kommentaren auszustatten. In der Praxis werden immer häufiger Produktlinien eingesetzt und infolgedessen auch von Grund auf neu erstellt. Ferner existieren auch bereits Produktlinien, welche über keine Kommentare verfügen und daher nicht in der Lage sind eine Dokumentation zu generieren. Insofern ist das Beispiel der Chat-Produktlinie durchaus relevant in der Praxis.

In Kapitel 2 wurde das Feature-Modell der Chat-Produktlinie bereits gezeigt (siehe Abbildung 2.4). Aus dem Modell geht hervor, dass die Produktlinie aus 13 Featuren besteht und 120 valide Konfigurationen besitzt. Die Chat-Produktlinie verfügt über 12 Klassen, 70 Methoden und 47 Felder und weist insgesamt 18 Klassen- und 22 Methodenverfeinerungen auf. Die Werte belegen, dass die Chat-Produktlinie eine eher kleine SPL ist.

6.2.2 Snake-Produktlinie

Die Snake-Produktlinie entstand aus der Zerlegung einer normalen Java-Anwendung. Der Quelltext war vor der Zerlegung bereits mit normalen JavaDoc-Kommentaren annotiert. Im Zuge der Zerlegung von Snake wurden daher auch die Kommentare zerlegt und angepasst. Somit konnte getestet werden, wie schwer es ist bereits vorhandene Kommentare in Modul-Kommentare umzuwandeln. Die Zerlegung eines oder mehrerer Produkte zur Umwandlung in eine Produktlinie findet sich häufig in der Praxis, wenn die Entwickler sich aufgrund der Vorteile entschließen ab einem gewissen Punkt eine SPL für ihre Produkte zu nutzen. Daher kann auch die Zerlegung von Snake als praxisnahes Beispiel angesehen werden.
Einen guten Überblick über die Variabilität der Snake-Produktlinie gibt das in Abbildung 6.1 abgebildete Feature-Modell. Dieses zeigt, dass die Produktlinie aus 21 Featuren besteht und 5580 valide Konfigurationen besitzt. Weiterhin besitzt die Snake-Produktlinie 28 Klassen, 197 Methoden und 133 Felder und insgesamt 28 Klassen- und 34 Methodenverfeinerungen. Im Vergleich zur Chat-Produktlinie ist die Snake-Produktlinie deutlich größer.

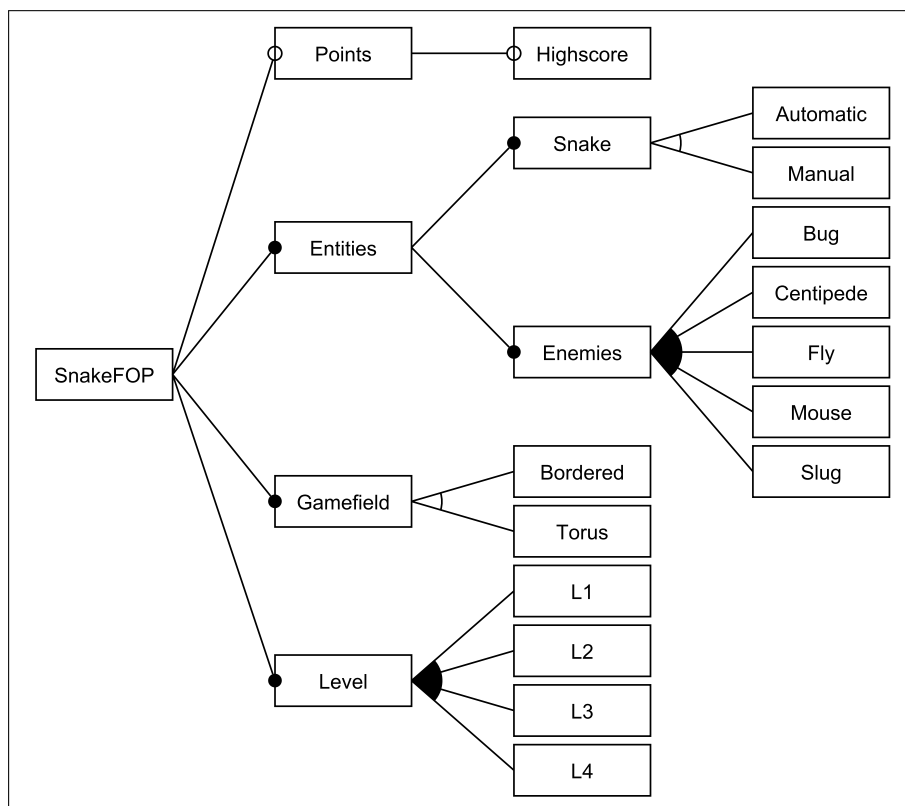

Abbildung 6.1: Feature-Modell der Snake-Produktlinie

6.3 Erhaltene Ergebnisse

Die Evaluierung fand anhand zweier mit FOP erstellten Produktlinien statt. Beide wurden mit den in Kapitel 4 beschriebenen Modul-Kommentaren versehen.

6.3.1 Chat-Produktlinie

Nachfolgend werden die Ergebnisse der qualitativen und der quantitativen Evaluierung für die Chat-Produktlinie beschrieben.

Qualitative Evaluierung

Die Kommentare in der Chat-Produktlinie wurden von Grund auf als Modul-Kommentare verfasst. Durch die Struktur der SPL war es leicht die Kommentare so modular wie nötig aufzubauen. Der schwierige Teil beim Erstellen der Modul-Kommentare bestand darin die allgemeinen Informationen zu identifizieren. Danach ließen sich relativ einfach alle Feature-spezifischen Informationen finden, indem jeweils die neu hinzukommende Funktionalität bei Verfeinerungen betrachtet wurde. Priorität wurden immer gleich 0 gewählt, außer bei Informationen, die vorhergehenden widersprachen. Daher musste bei der Priorisierung der Feature-spezifischen Informationen auch stark auf die Feature-Reihenfolge geachtet werden.
Ein geringfügiges Problem stellte die nicht vorhandene Unterstützung von Modul-Kommentaren durch die Entwicklungsumgebung (Integrated Development Environment) (IDE) dar. Die Schlüsselwörter mussten alle per Hand erstellt werden, da keine Templates für diese in der IDE existieren.
Insgesamt wurden 80 verschiedene Modul-Kommentare erstellt. Die Tabelle 6.1 gibt Auskunft über die Anzahl der verwendeten Schlüsselwörter und deren Prioritäten. Die Zeile Summe der Tabelle gibt die Anzahl aller Modul-Kommentare wieder.

Schlüsselwort	Priorität = 0	Priorität > 0	Priorität ≥ 0
General	43	0	43
Feature	28	7	35
New	2	0	2
Summe	73	7	80

Tabelle 6.1: Übersicht über die Modul-Kommentare der Chat-Produktlinie

Nach der Erstellung der Modul-Kommentare wurde SPLDoctor getestet, indem für jeden Anwendungsfall eine Dokumentation erstellt wurde. SPLDoctor war in der Lage für alle vier Anwendungsfälle die Dokumentationen zu erstellen, wobei alle generierten Dokumentationen die gewünschten Informationen mit dem entsprechenden Detaillierungsgrad enthielten.

Quantitative Evaluierung

Die Ergebnisse der quantitativen Evaluierung sind in Tabelle 6.2 aufgelistet und nochmals grafisch in Abbildung 6.2 dargestellt. Es ist zu erkennen, dass der Vergleich mit den Anwendungsfällen recht unterschiedlich ausfällt. Im Vergleich mit dem SPL-Ansatz benötigt SPLDoctor mehr Informationen und daher auch mehr

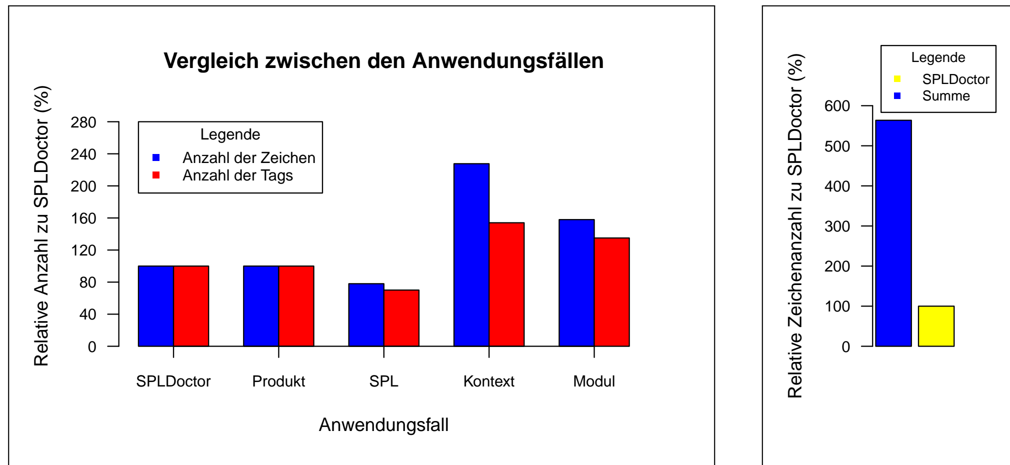

Abbildung 6.2: Grafische Repräsentation der quantitativen Evaluierung für die Chat-Produktlinie

beziehungsweise größere Kommentare im Quelltext. Hingegen benötigen die Ansätze für die Kontext- und die Feature-Modul-Dokumentation wesentlich mehr Input. Wie abzusehen, liegt die Größe der Modul-Kommentare deutlich unter der Summe aller Kommentare für die vier Anwendungsfälle. Werden für die Chat-Produktlinie alle vier trivialen Ansätze für die Anwendungsfälle verwendet, nehmen die dafür nötigen Kommentare rund 5,6 mal mehr Platz ein, als die Modul-Kommentare für die Chat-Produktlinie.

Typ	Zeichen	Zeichendifferenz	Tags	Tagdifferenz
SPLDoctor	5088	100%	137	100%
Produkt	5088	100%	137	100%
SPL	3969	78%	96	70%
Kontext	11582	227%	211	154%
Feature-Modul	8036	157%	185	135%
Summe	28675	563%	629	459%

Tabelle 6.2: Übersicht über die Ergebnisse der quantitativen Evaluierung für die Chat-Produktlinie

6.3.2 Snake-Produktlinie

In den nächsten Abschnitten werden die Ergebnisse der qualitativen und der quantitativen Evaluierung für die Snake-Produktlinie beschrieben.

Qualitative Evaluierung

Im Zuge der Zerlegung der Snake-Anwendung in eine SPL wurden auch die normalen JavaDoc-Kommentare zerlegt und angepasst. Die Kommentare wurde auf genau den selben Weg zerlegt, wie auch die Klassen und Methoden des Projekts. Auf diese Weise ergab sich sehr leicht die Einteilung in allgemeine und Feature-spezifische Kommentare. Eine umfassende Priorisierung der Modul-Kommentare war nicht nötig, da bei der Zerlegung einer Anwendung wenig bis keine Alternativen auftreten

und so das Konfliktpotential stark reduziert wird. Daher wurden bei Snake auch nachträglich noch einige wenige Alternativen wurde hinzugefügt.
In Tabelle 6.3 lässt sich die Anzahl der verwendeten Schlüsselwörter erkennen.

Schlüsselwort	**Priorität = 0**	**Priorität > 0**	**Priorität ≥ 0**
General	190	1	191
Feature	22	22	44
New	0	0	0
Summe	212	23	235

Tabelle 6.3: Übersicht über die Modul-Kommentare der Snake-Produktlinie

Nach der Umwandlung der JavaDoc- in Modul-Kommentare, ließen sich mithilfe von SPLDoctor die Dokumentation für alle vier Anwendungsfälle erstellen. Wie auch bei der Chat-Produktlinie enthielten diese jeweils die gewünschten Informationen.

Quantitative Evaluierung

In Tabelle 6.4 sind die Ergebnisse der quantitativen Evaluierung zu sehen. Eine grafische Darstellung der Werte ist in Abbildung 6.3 abgebildet.
Die Ergebnisse der Quantitativen Evaluierung der Snake-Produktlinie gleichen denen der Chat-Produktlinie. Die Größe der Kommentare für die Dokumentation einer SPL liegen auch hier unter der Größe der Modul-Kommentare. Bei der Feature-Modul- und der Kontext-Dokumentation schneiden die Modul-Kommentare besser ab. Die Modul-Kommentare liegen wiederum deutlich unter der Summe aller Kommentare der vier Anwendungsfälle.

Typ	**Zeichen**	**Zeichendifferenz**	**Tags**	**Tagdifferenz**
SPLDoctor	20210	100%	530	100%
Produkt	20210	100%	530	100%
SPL	19696	97%	475	89%
Kontext	36963	182%	820	154%
Feature-Modul	26294	130%	594	112%
Summe	103163	510%	2419	456%

Tabelle 6.4: Übersicht über die Ergebnisse der quantitativen Evaluierung für die Snake-Produktlinie

6.3.3 Interpretation der Ergebnisse

Beide Produktlinien lieferten sehr ähnliche Resultate in der Evaluierung. Das vereinfacht es im Nachfolgenden einige Schlüsse aus den erhaltenen Ergebnissen zu ziehen.

Erstellung der Modul-Kommentare

In beiden Fällen bedurfte es keines großen Mehraufwandes zur Erstellung der Modul-Kommentare, gegenüber einer normalen Annotation mit JavaDoc-Kommentaren.

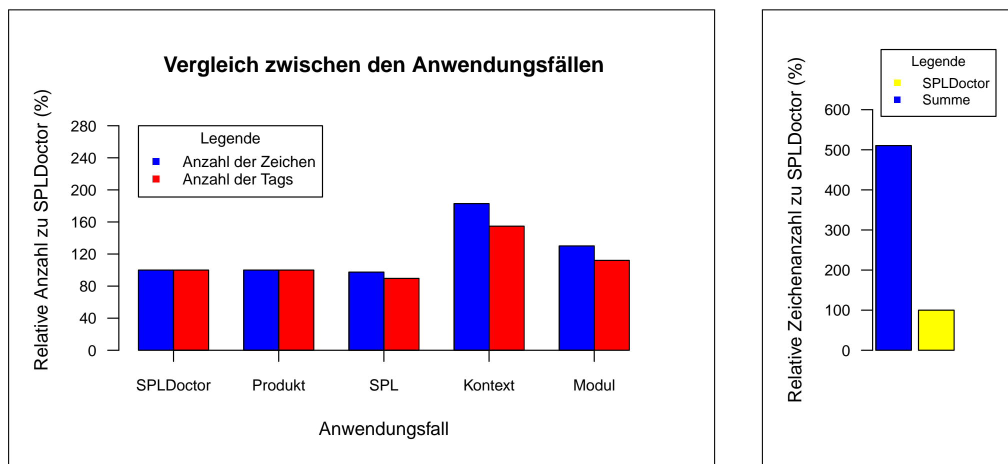

Abbildung 6.3: Grafische Repräsentation der quantitativen Evaluierung für die Snake-Produktlinie

Das Erstellen der Modul-Kommentare ist intuitiv, da es sich nach dem Uniformiäts-prinzip richtet. Ein normaler JavaDoc-Kommentar beschreibt komplett die Funktionalität der Signatur, welche er annotiert. Analog dazu verhalten sich die Modul-Kommentare. Bei Verfeinerungen der Signatur enthält der Kommentar ausschließlich die neuen Informationen. Somit ist es intuitiv und leicht nachvollziehbar, wie der Modul-Kommentar für eine Signatur erstellt wird. Lediglich die Identifizierung der allgemeinen Informationen ist in einigen Fällen nicht sofort ersichtlich, da dafür jede Verfeinerung der Signatur betrachtet werden muss.
Die Generierung der Dokumentationen für die einzelnen Anwendungsfälle funktionierte bei beiden Produktlinien. In jedem Fall enthielten die Dokumentationen alle Informationen, die durch die Modul-Kommentare für den jeweiligen Anwendungsfall vorgegeben waren.
Aus diesen empirischen Ergebnissen lässt sich schlussfolgern, dass sowohl das Konzept der Modul-Kommentare, als auch der implementierte Prototyp SPLDoctor in der Praxis funktionieren. Weiterhin ist ersichtlich, dass die Erstellung der Modul-Kommentare einfach zu handhaben ist. Wobei es keine Rolle spielt, ob die Kommentare neu erstellt oder aus normalen JavaDoc-Kommentaren umgewandelt werden. Daraus folgt, dass SPLDoctor im Rahmen der prototypischen Entwicklung so benutzerfreundlich wie möglich ist.

Quantitative Bewertung der Modul-Kommentare

Zur Erstellung einer SPL-Dokumentation, wurden die vollständigen Kommentare des SPL-Ansatzes mit den Modul-Kommentaren von SPLDoctor verglichen. Bei beiden evaluierten Produktlinien ist die Summe der vollständigen Kommentare kleiner als die Summe der Modul-Kommentare. Das liegt an den zusätzlichen Feature-spezifischen Informationen in den Modul-Kommentaren. Diese Informationen werden beispielsweise für die Produkt-Dokumentation benötigt. Bei der SPL-Dokumentation werden sie hingegen nicht verwendet. Das bedeutet für den Fall, falls nur eine reine SPL-Dokumentation von den Entwicklern gewünscht ist, dass das Verfahren nicht so effizient arbeitet, wie der triviale SPL-Ansatz. Daher ist SPLDoctor laut den Resultaten der quantitativen Evaluierung für diesen Fall nicht geeignet. Dabei sind jedoch noch zwei Dinge zu berücksichtigen. Zum einen existiert bislang kein

Verfahren, das den trivialen SPL-Ansatz umsetzt, sodass dieser auch nicht praktisch eingesetzt werden kann und zum anderen ist für die Entwickler zumeist mehr als nur die SPL-Dokumentation von Interesse.
Für die Erstellung von Feature-Modul-Dokumentationen wurden die vollständigen Kommentare des Feature-Modul-Ansatzes mit den Modul-Kommentaren verglichen. Die Summe der vollständigen Kommentare war in beiden evaluierten Produktlinien deutlich größer als die Summe der Modul-Kommentare. Dieses Ergebnis war zu erwarten, da sich unter den vollständigen Kommentaren des Feature-Modul-Ansatzes viele redundante Informationen befinden. Diese redundanten Informationen bestehen aus den allgemeinen Informationen. Bei den Modul-Kommentaren werden die allgemeinen Informationen hingegen nur einmal pro Signatur verwendet. Der Aufwand beim Erstellen und Warten der Modul-Kommentare ist somit geringer als bei vollständigen Kommentaren. Daraus folgt, dass SPLDoctor, dem trivialen Ansatz überlegen ist und bei der Erstellung von Feature-Modul-Dokumentationen immer verwendet werden sollte.
Im Hinblick auf die Kontext-Dokumentationen wurden die Modul-Kommentare mit den vollständigen Kommentaren des Kontext-Ansatzes verglichen. Die Größe der Kommentare für den Kontext-Ansatz liegt bei beiden Produktlinien deutlich oberhalb der Größe der Modul-Kommentare. Dies liegt vor allem an der Redundanz der Kommentare in dem gewählten statischen Ansatz für die Kontext-Dokumentation. Das bedeutet, dass SPLDoctor wesentlich effizienter ist als der triviale Kontext-Ansatz. Zusätzlich ist SPLDoctor im Gegensatz zum trivialen Ansatz in der Lage für jedes Kontext-Interface die Dokumentation zu erzeugen. Daher lässt sich ableiten, dass SPLDoctor sich hervorragend zur Generierung von Kontext-Dokumentationen eignet.
Im Regelfall ist für die Entwickler einer SPL nicht nur einer der Anwendungsfälle interessant, sondern es werden mehrere genutzt. Da die Kommentare der trivialen Ansätze für einzelnen Anwendungsfälle nicht kompatibel sind, müssten ohne SPLDoctor für jeden Ansatz eigene Kommentare erstellt werden, um alle vier Anwendungsfälle bedienen zu können. Werden die Kommentare der einzelnen Anwendungsfälle allerdings aufsummiert, so zeigt sich, dass die Summe wesentlich größer ist als die Summe aller Modul-Kommentare. Das bedeutet, dass sobald mehr als ein Anwendungsfall von Bedeutung ist, sich der Einsatz von SPLDoctor beziehungsweise der Modul-Kommentare lohnt.

6.3.4 Aussagekraft der Ergebnisse

In der Evaluierung wurden nur zwei, eher kleine Produktlinien verwendet. Da für die Beispiel-Produktlinien noch die Modul-Kommentaren erstellt werden mussten, konnten aufgrund der zeitlichen Beschränkung dieser Arbeit nicht mehr oder wesentlich größere Produktlinien gewählt werden. Die Aussagekraft der Evaluierung scheint daher eingeschränkt zu sein. Allerdings wurden bewusst zwei verschiedenartige Produktlinien ausgewählt, um unterschiedliche Ergebnisse bei der Evaluierung zu provozieren. Dennoch zeigten beide Produktlinien sehr ähnliche Resultate, welche auch durchaus den theoretischen Überlegungen entsprachen. Demnach sind die Ergebnisse der Evaluierung durchaus glaubwürdig und belegen die Funktionalität von SPLDoctor und die Eignung der Modul-Kommentare.

6.4 Zusammenfassung

Die Evaluierung bewertet mithilfe der Chat- und der Snake-Produktlinie die Effizienz und die Benutzerfreundlichkeit von SPLDoctor. SPLDoctor ist in der Lage aus den gegebenen Kommentaren die Dokumentationen für alle vier Anwendungsfälle zu erzeugen. Vergleicht man die Größe der Modul-Kommentare, die der Prototyp benötigt mit den Kommentaren für die einzelnen Anwendungsfälle, so liegt der Prototyp etwa in der Mitte. Betrachtet man jedoch die Summe aller Anwendungsfälle, so benötigt der Prototyp wesentlich weniger Kommentare. Somit ist SPLDoctor in der Summe den bisherigen Methoden überlegen. Insgesamt zeigt die Evaluierung, dass SPLDoctor funktioniert und wesentlich effizienter und benutzerfreundlicher ist, als alle anderen vorgestellten Ansätze.

7. Verwandte Arbeiten

In dieser Arbeit wurde die Kombination von Quelltext-Dokumentationen und der Feature-orientierten Programmierung (FOP) untersucht. Mit dem Ziel effizient maßgeschneiderte Dokumentationen für verschiedene Anwendungsfälle von Software-Produktlinien (SPLs) zu erstellen. Daher werden in diesem Kapitel zwei weitere Themengebiete vorgestellt, welche sich ebenso auf die Verbindung von SPLs und Dokumentationen beziehen. Das erste Themengebiet befasst sich mit der Generierung von begleitenden Dokumentationen beziehungsweise allgemeinen Dokumenten für SPLs. Das zweite Themengebiet widmet sich formalen Spezifikationen für SPLs und untersucht Ansätze, die es ermöglichen den Quelltext von SPLs mit formalen Spezifikationen zu annotieren.

Dokumentationsproduktlinien

Wie in Kapitel 2 beschrieben, gibt es neben der Dokumentation des Quelltextes noch viele weitere Arten von Software-Dokumentationen. Für die Entwickler von Software-Produkten sind vor allem noch die System- und Benutzerdokumentationen interessant [PB96, Som92]. Die Entwickler von SPLs stehen hier allerdings vor dem Problem, dass durch eine Produktlinie viele verschiedene Produkte generiert werden können und sie so für jedes mögliche Produkt eine eigene maßgeschneiderte Dokumentation angelegen müssten. Da dies jedoch einen viel zu hohen Aufwand darstellt, sind potentielle Ansätze von Interesse, welche die Erstellung von maßgeschneiderten System- oder Benutzerdokumentationen für die Produkte einer SPL ermöglichen. Ein solcher Ansatz beruht dabei darauf, die Dokumentation ebenfalls als Produktlinie zu behandeln, was dazu genutzt werden kann, um Variabilität in die Dokumentationen zu bringen und diese zusammen mit einer SPL verwenden zu können. Für die Entwickler ergibt sich daraus der Vorteil, dass die Dokumentationen zu den Produkten genauso schnell erzeugt werden können, wie die Produkte selbst und nicht die Notwendigkeit besteht eine eigene Dokumentation für jedes Produkt anzulegen.
DocLine ist ein Tool, das die Umsetzung von Benutzerdokumentationen als Dokumentationsproduktlinie ermöglicht [KR08]. Das Tool basiert auf der Nutzung der

XML basierten Sprache DRL (Documentation Reuse Language), mit deren Hilfe eine angepasste Wiederverwendung von Textbausteinen möglich ist. Die Hauptidee ist die mehrfache Verwendung von Textbausteinen, aus denen die Dokumentation zusammengesetzt wird. Durch den Einsatz von Platzhaltern, optionalen oder alternativen Texten in den Textbausteinen, können diese im Zuge der Wiederverwendung an die jeweilige Situation optimal angepasst werden.
Rabiser et al. schlagen einen etwas allgemeinerer Ansatz zur Modellierung von Variabilität in Dokumenten vor [RHE+10]. Der Ansatz der Dokumentationsproduktlinien wird hierbei auf beliebige Dokumente erweitert. Die Erstellung von Dokumenten erfolgt entscheidungsbasiert mithilfe der Tool Suite DOPLER (Decision-oriented product line engineering).
Im Gegensatz zu den vorgestellten Arbeiten, welche sich mit begleitenden Dokumentationen beschäftigten, konzentrierte sich diese Arbeit auf die Quelltext-Dokumentationen. Diese sind wesentlich stärker an den Quelltext gebunden als die begleitenden Dokumentationen und lassen sich mithilfe anderer Werkzeuge erzeugen.

Formale Spezifikationen

Die in dieser Arbeit verwendete Form der Dokumentation wird auch als informelle Spezifikation bezeichnet. Ein weitere Form ist die formale Spezifikation. Die informelle Spezifikation beschreibt die Funktionalität einer Signatur mithilfe von natürlicher Sprache und ist deshalb für den Menschen einfach zu lesen. Formale Spezifikationen beschreiben die Funktionalität einer Signatur hingegen mittels einer eindeutig definierten (formalen) Sprache. Der Vorteil von formalen Spezifikationen ist, dass Computer in der Lage sind Aussagen in einer formalen Sprache zu lesen und zu analysieren. Mithilfe von formalen Spezifikationen können beispielsweise die Grenzen von Parameter und Rückgabewerten von Funktionen beschrieben und deren korrekte Verwendung im Programmfluss untersucht werden. Somit lassen sich die Interaktionen von Methoden in einer SPL durch Analysewerkzeuge überprüfen und Programmierfehler aufdecken. So kann eine automatisierte Überprüfung einer SPL auf ihre Richtigkeit stattfinden.
Eine Art von formalen Spezifikationen sind Kontrakte, welche ursprünglich für die Objekt-orientierte Programmierung (OOP) eingeführt wurden [Mey92]. Die Kontrakte werden beispielsweise dazu benutzt Methoden mit Vor- und Nachbedingungen zu versehen, um die Interaktionen von Methoden untereinander zu analysieren. Thüm et al. adaptieren das Konzept der Kontrakte für Produktlinien, mit dem Ziel ein korrektes Verhalten bei allen Produkten einer Produktlinie sicherstellen zu können [TSK+12]. Mithilfe von Kontrakten lässt sich das Verhalten der einzelnen Produkte einer SPL zu untersuchen. So sind die Entwickler in der Lage ihre SPL so zu entwickeln, dass in keinem Produkt ein ungewolltes Verhalten auftritt, weil die Methoden auf eine unvorhergesehene Weise miteinander interagieren.
Sowohl bei der eben vorgestellten, als auch in dieser Arbeit traten ähnliche Probleme mit der Verfeinerungen von Quelltext-Annotationen auf. Diese Probleme ließen sich in beiden Arbeiten nicht durch das bloße Anwenden des Uniformitätsprinzips lösen und mussten daher auf anderem Wege behoben werden. Dennoch unterscheiden sich diese Arbeiten grundlegend von einander, da sie völlig unterschiedliche Probleme adressieren.

8. Zusammenfassung und Fazit

Um Zeit und Entwicklungskosten bei neuen Projekten zu sparen, versuchen viele Entwickler einmal geschriebenen Programme wiederzuverwenden. Damit die Wiederverwendung von Quelltexten überhaupt erst möglich ist, muss der Quelltext zwei Anforderungen erfüllen. Der Quelltext muss zum einen so verständlich sein, dass die Entwickler dessen Funktion auch noch einige Zeit nach der Implementierung nachvollziehen können. Zum anderen muss der Quelltext so strukturiert sein, dass er leicht an andere Situationen angepasst werden kann. Um die erste Anforderung zu erfüllen werden zwangsläufig Quelltextdokumentationen benötigt, um die Funktion und das Verhalten des Quelltextes zu beschreiben. Für die zweite Anforderung spielt die Wahl der Implementierungstechnik eine entscheidende Rolle. In diesem Zusammenhang bietet die Feature-orientierte Programmierung (FOP) durch ihre Struktur, in der Feature in einzelnen lokal separierten Modulen entwickelt werden, die Möglichkeit Quelltext effizient wiederzuverwenden. Allerdings fehlte FOP bisher die Unterstützung für die Dokumentation des Quelltextes. Insbesondere konnten bisher keine spezifischen Dokumentationen für verschiedenen Anwendungsfälle erzeugt werden. Im Rahmen der Arbeit wurde daher ein Konzept entwickelt, um maßgeschneiderte Quelltext-Dokumentationen von FOP-Produktlinien für verschiedenartige Anwendungsfälle zu ermöglichen.
In Kapitel 3 wurden vier verschiedene Anwendungsfälle für Dokumentation von Software-Produktlinien (SPLs) vorgestellt. Die Anwendungsfälle sind die Produkt-, die SPL-, die Kontext- und die Feature-Modul-Dokumentation. Zwar existieren für die meisten dieser Anwendungsfälle triviale Lösungsansätze zur Erstellung einer Dokumentation, doch funktionieren diese Ansätze nur für die Dokumentation des jeweiligen spezifischen Anwendungsfalles. Infolgedessen wurde ein Konzept zur Erstellung von Dokumentation einer SPL entworfen, welches alle Anwendungsfälle berücksichtigt. Das Konzept besteht aus drei aufeinander aufbauenden Phasen. Der Definition von Modul-Kommentaren durch den Entwickler, der Erzeugung von Pseudo-Quelltext mit entsprechenden Dokumentationskommentaren und der anschließenden Generierung der Dokumentation durch JavaDoc. Ein Vorteil dieses modularen Aufbaus ist, dass das JavaDoc-Tool ohne weitere Änderungen verwendet werden kann und genauso konfigurierbar ist, wie bei der Generierung einer normalen JavaDoc-

Dokumentation für die Objekt-orientierte Programmierung (OOP).
Durch die Entwicklung des Konzepts zur Generierung von maßgeschneiderten JavaDoc-Dokumentationen in FOP wurde zugleich die erste Forschungsfrage der Arbeit positiv beantwortet. Dementsprechend wurde gezeigt, dass das neu entwickelte Konzept funktioniert und in der Lage ist für jeden der vorgestellten Anwendungsfälle eine Dokumentation zu erzeugen. Zur Beantwortung der Frage, wurde das Konzept entwickelt und mit Hilfe einer qualitativen Evaluierung die Funktionalität dieses Konzepts bestätigt. Hierfür wurde das Konzept mit zwei unterschiedlich erstellten Produktlinien auf Effizienz und Benutzerfreundlichkeit untersucht. Dabei handelte es sich bei der ersten Produktlinie um eine Chat-Anwendung, welche direkt als Produktlinie implementiert wurden und keine JavaDoc-Kommentare enthielt. Diese musste daher von Grund auf mit Modul-Kommentaren annotiert werden. Die zweite Produktlinie umfasst mehrere Varianten des Spiels Snake und wurde durch Zerlegung einer normalen Java-Anwendung erstellt, welche bereits mit JavaDoc-Kommentare versehen war. Die vorhandenen JavaDoc-Kommentare wurden während der Zerlegung in Modul-Kommentare umgewandelt. Bei beiden Produktlinien gelang die Erstellung der Modul-Kommentare ohne Probleme. Das bedeutet, dass die Entwickler in der Lage sein werden, Modul-Kommentare so intuitiv zu erstellen, wie sie auch den Quelltext einer FOP-Produktlinie erstellen können. Nach der Erstellung der Modul-Kommentare war es bei beide Produktlinien möglich Dokumentationen für jeden Anwendungsfall zu erzeugen.
Eine weitere offene Forschungsfrage, die im Laufe der Arbeit beantwortet wurde, bezieht sich auf die effiziente Umsetzung der Modul-Kommentare, welche für die Dokumentation aller Anwendungsfälle genutzt werden. Dabei ist mit der Effizienz der anfallende Aufwand für die Entwickler zur Erstellen der Kommentare gemeint. Um diese Frage zu beantworten, wurde eine quantitative Evaluierung des Konzepts durchgeführt. Dazu wurde das Konzept mit intuitiven Ansätzen zur Erzeugung von Dokumentationen für jeden Anwendungsfall verglichen. Diese Ansätze eignen sich zwar jeweils optimal dazu einen speziellen Anwendungsfall, wie zum Beispiel die Dokumentation eines Produkts zu unterstützen, erlauben jedoch keine Nutzung für andere Anwendungfälle. Weiterhin zeigte sich, dass die intuitiven Ansätze teilweise sogar weniger effizient sind als das in dieser Arbeit entwickelte Konzept. Insgesamt zeigte sich, dass im Vergleich zur normalen Dokumentation für OOP-Projekte keinen signifikanten Mehraufwand gibt, um die Modul-Kommentare zu erzeugen beziehungsweise umzuwandeln. In der Summe ist das entwickelte Konzept sogar deutlich effizienter, da für die Entwickler einer SPL meistens mehrere Anwendungsfälle interessant sind. Daraus lässt sich ableiten, dass es den Entwicklern möglich ist, durch Nutzung des Konzept effizient Dokumentationen erzeugen zu können.
Zusammenfassend lassen sich die folgenden Schlüsse ziehen. Mithilfe des entwickelten Konzepts ist es möglich den Quelltext einer mit FOP erstellten Produktlinie zu dokumentieren. Dabei werden vier verschiedene Sichtweisen auf den Quelltext unterstützt, welche sich jeweils an verschiedene Gruppen von Entwicklern richtet. Für die Dokumentation aller vier Anwendungsfälle werden lediglich die einmalig erstellten Modul-Kommentare benötigt. Dabei unterscheidet sich die Erstellung der Modul-Kommentare nicht im großen Maße von der Erstellung gewöhnlicher JavaDoc-Kommentare. Die Erstellung ist intuitiv und verursacht keinen nennenswerten Mehraufwand im Vergleich zu einer JavaDoc-Dokumentation für OOP, da

die Modul-Kommentare nach dem Uniformitätsprinzip aufgebaut sind. Die Entwickler einer FOP-Produktlinie sind somit einfach in der Lage Dokumentationen ihres Quelltextes für sich und auch für andere Entwickler zu erstellen.

9. Zukünftige Arbeiten

In diesem Kapitel werden einige offene Fragen präsentiert, die sich bei der Erstellung des Konzepts ergeben haben. Des Weiteren werden verschiedene Möglichkeiten vorgeschlagen, wie der Prototyp SPLDoctor in zukünftigen Arbeiten erweitert werden könnte, um eine effizientere Entwicklung mithilfe der Feature-orientierten Programmierung (FOP) zu gewährleisten.

Offene Fragen

Das in dieser Arbeit vorgestellte Konzept ist für die Feature-orientierte Programmierung (FOP) entwickelt wurden. Daher ergeben sich mehrere interessante Fragestellungen, wie sich das Konzept in verschiedenen Richtungen ausbauen lässt. Dies gilt sowohl bezogen auf FOP selbst, als auch auf andere Paradigmen zur Erstellung einer Software-Produktlinie (SPL).
Ein interessanter Punkt bei der Entwicklung von SPLs ist die veränderliche Vererbungshierarchie. Die Objekt-orientierte Programmierung (OOP) gibt immer eine feste Vererbungshierarchie vor. In FOP hingegen lässt sich diese Vererbungshierarchie auch verändern, sodass manche Produkte eine andere Vererbungshierarchie haben als andere Produkte. Daraus ergeben sich Probleme bei der Generierung der SPL- und der Kontext-Dokumentation. Es wäre daher interessant zu wissen, wie sich eine veränderliche Vererbungshierarchie bei der Generierung einer Dokumentation berücksichtigen lässt.
Abseits von der konkreten Implementierungstechnik ergeben sich noch weitere Fragen über das Erstellen und Editieren der Modul-Kommentare. In der Anforderungsanalyse in Kapitel 3 wurde herausgestellt, dass doppelte und widersprüchliche Informationen innerhalb der Kommentare ein Problem darstellen können. Diese Probleme lassen sich zwar mithilfe der Modul-Kommentare umgehen, jedoch müssen die Entwickler selbst darauf achten, dass ihre Modul-Kommentare richtig aufgebaut sind. Dafür ist es momentan nötigt, dass die Entwickler einen Überblick über alle miteinander interagierenden Modul-Kommentare behalten. Daher wäre es von großem Interesse, ob sich Ähnlichkeiten oder Dopplung in den Kommentaren automatisch

erkennen lassen. Ebenso wäre das automatisch Erkennen von Widersprüche innerhalb der Kommentare vorteilhaft.
Modul-Kommentare beeinflussen sich nicht nur untereinander, sie sind auch stark abhängig vom Feature-Modell der zugrunde liegenden SPL. Änderungen am Feature-Modell können gravierenden Auswirkungen auf die SPL haben. Nicht nur die Anzahl und Art der Produkte können sich ändern, es können auch viele Probleme bei der Zusammensetzung der Feature-Module auftreten. Da die Modul-Kommentare auf die Verfeinerungen abgestimmt sind, werden sie mit beeinflusst. Daher stellt sich die Frage, wie stark Änderungen am Feature-Modell die Modul-Kommentare beeinflussen. Ferner stellt sich die Frage, wie die Entwickler bei derartigen Änderungen unterstützt werden können.
Wie in Kapitel 2 erwähnt gibt es noch weitere Paradigmen um eine SPL zu implementieren. FeatureIDE unterstützt unter anderem die Implementierung einer SPL mittels Aspekt-orientierter Programmierung (AOP) [KLM+97] (AspectJ [KHH+01]) und Präprozessoren [KAK08] (Antenna[1], Munge[2]). Diese Paradigmen unterscheiden sich in ihrer Funktionsweise stark von FOP. Während Aspekte eine andere Art der modularen Entwicklung von Features ermöglichen, nutzen Präprozessoren Annotationen, um Features von einander abzugrenzen. Daher stellt sich die Frage, ob das Konzept auch auf andere Paradigmen, wie das AOP- oder das Präprozessoren-Paradigma übertragbar ist.

Verbesserungen für SPLDoctor

Durch die Integration in FeatureIDE eröffnen sich Möglichkeiten eine Vielzahl an weiteren Hilfen für die Entwickler von SPLs einzubauen. So bietet Eclipse zum Beispiel standardmäßig eine Unterstützung für JavaDoc an. Durch Tooltipps können die Kommentare einer Signatur direkt in der formatierten Form gelesen werden. Es wäre sinnvoll diese Funktion der Entwicklungsumgebung (Integrated Development Environment) (IDE) zu erweitern, sodass vorab das Vereinigungsverfahren auf die Modul-Kommentare angewandt wird, um eine SPL- oder Kontext-Dokumentation zu generieren. Damit könnten Informationen aus diesen Dokumentationen direkt im Quelltext eingesehen werden. Weiterhin ist es auch möglich dem Entwickler beim Erstellen der Modul-Kommentare Hilfe anzubieten. Das momentane Prinzip bei der Aufteilung der Modul-Kommentare geht davon aus, dass der Entwickler einen Überblick über alle Kommentare zu einer Signatur behält. Dies kann bei großen Produktlinien und mehreren Entwickler recht schwierig sein. Somit wäre die Unterstützung einer Problemsuche in den Modul-Kommentaren besonders hilfreich für die Entwickler. So sollten zum Beispiel ähnliche Kommentare und Einschränkungen bei Parameter- oder Rückgabewerten erkannt werden. Im Idealfall hilft die IDE nicht nur die Probleme zu erkennen, sondern diese auch zu beheben. Kommentare mit redundanten Informationen können zerlegt und Kommentare mit widersprüchlichen Informationen priorisiert werden.
Ein weiterer Aspekt ist die für die SPL verwendete Programmiersprache. Das in der Arbeit vorgestellte Konzept ist auf die Sprache Java und den Dokumentationsgenerator JavaDoc ausgelegt. Dokumentationsgeneratoren gibt es jedoch auch für viele

[1] http://antenna.sourceforge.net/index.php
[2] https://sonatype.github.io/munge-maven-plugin/

andere Sprachen. FeatureIDE unterstützt beispielsweise auch den FOP-Ansatz FeatureC++ [ALRS05], welcher auf der Sprache C++ aufbaut. Daher wäre es sinnvoll auch andere populäre Dokumentationsgeneratoren zu unterstützen.

Literaturverzeichnis

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. (zitiert auf Seite 9 und 13)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013. (zitiert auf Seite 15 und 51)

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Generative Programming and Component Engineering*, pages 125–140. Springer, 2005. (zitiert auf Seite 15, 51 und 75)

[AR92] Egil P Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152. Springer Berlin Heidelberg, 1992. (zitiert auf Seite 13)

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the International Conference on Software Engineering*, pages 702–703. IEEE Computer Society, 2004. (zitiert auf Seite 15 und 51)

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. (zitiert auf Seite 2 und 15)

[CE05] Krzysztof Czarnecki and Ulrich W Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2005. (zitiert auf Seite 10 und 11)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. (zitiert auf Seite 15)

[CN06] Paul Clements and Linda Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2006. (zitiert auf Seite 9)

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Computer Science Department, Technical University of Ilmenau, 1998. (zitiert auf Seite 1)

[FL02] Andrew Forward and Timothy C Lethbridge. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the ACM symposium on Document engineering*, pages 26–33. ACM, 2002. (zitiert auf Seite 2 und 6)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. (zitiert auf Seite 1)

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std*, 610121990:1–84, 1990. (zitiert auf Seite 5)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 311–320. ACM, 2008. (zitiert auf Seite 13 und 74)

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990. (zitiert auf Seite 9)

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001. (zitiert auf Seite 50 und 74)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. (zitiert auf Seite 13 und 74)

[KR08] Dmitry V Koznov and Konstantin Yu Romanovsky. DocLine: A Method for Software Product Lines Documentation Development. *Programming and Computer Software*, 34(4):216–224, 2008. (zitiert auf Seite 67)

[Kru92] Charles W Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992. (zitiert auf Seite 1)

[Mey92] Bertrand Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992. (zitiert auf Seite 68)

[Ora14] Oracle. JavaDoc Tool. Website, JAN 2014. http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html. (zitiert auf Seite 6, 9 und 32)

[PB96] Gustav Pomberger and Günther Blaschek. *Object-Orientation and Prototyping in Software Engineering*. Prentice-Hall, 1996. (zitiert auf Seite 5, 6 und 67)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. (zitiert auf Seite 2, 9, 10, 11 und 13)

[Pre01] Christian Prehofer. Feature-Oriented programming: A New Way of Object Composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001. (zitiert auf Seite 2 und 13)

[RHE+10] Rick Rabiser, Wolfgang Heider, Christoph Elsner, Martin Lehofer, Paul Grünbacher, and Christa Schwanninger. A Flexible Approach for Generating Product-Specific Documents in Product Lines. In *Software Product Lines: Going Beyond*, pages 47–61. Springer, 2010. (zitiert auf Seite 68)

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992. (zitiert auf Seite 5, 6 und 67)

[SSTS14] Reimar Schröter, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. Unpublished Manuscript, 2014. (zitiert auf Seite 15)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014. (zitiert auf Seite 50)

[TSK+12] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 255–269. Springer, 2012. (zitiert auf Seite 68)

[vH14] Dimitri van Heesch. Doxygen. Website, JAN 2014. http://www.stack.nl/~dimitri/doxygen/. (zitiert auf Seite 6)

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

Magdeburg, den 23.04.2014

Sebastian Krieter