Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Masterarbeit

Extraktion von Feature-Modellen aus Implementierungsartefakten

Autor:

Daniel Lüddecke

14. September 2012

VOLKSWAGEN

AKTIENGESELLSCHAFT

Lüddecke, Daniel:
Extraktion von Feature-Modellen aus Implementierungsartefakten
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2012.

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Masterarbeit

Extraktion von Feature-Modellen aus Implementierungsartefakten

Autor:
Daniel Lüddecke
Matrikelnummer: 180948

14. September 2012

Erstgutachter:
Prof. Dr. rer. nat. habil. Gunter Saake
Institut für Technische und Betriebliche Informationssysteme (ITI)
Otto-von-Guericke Universität Magdeburg

Zweitgutachter:
Dr. Moritz Neugebauer
Volkswagen Aktiengesellschaft

Betreuer:
Thomas Thüm
Dr.-Ing. Martin Kuhlemann
Andreas Lübcke
Institut für Technische und Betriebliche Informationssysteme (ITI)
Otto-von-Guericke Universität Magdeburg

Abstract

In this thesis, we present concepts, algorithms and tools which enable software developers to extract features and feature models from a given set of implementation artifacts in a semi-automated process. We will also show that feature models can be used to distribute features of software product lines to customers, using a central repository.

Inhaltsangabe

In dieser Arbeit werden Konzepte, Algorithmen und Werkzeuge vorgestellt, die es ermöglichen, aus einer gegebenen Menge an Implementierungsartefakten teilautomatisiert Features und Feature-Modelle zu extrahieren. Es wird auch gezeigt, dass Feature-Modelle dazu genutzt werden können, Features einer Software-Produktlinie über ein zentrales Repository an Kunden zu distribuieren.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich fortwährend bei der Erstellung dieser Arbeit unterstützt haben.

Dazu geht mein Dank zunächst an Thomas Thüm. Durch sein kontinuierliches, sehr konstruktives und vor allem hochqualitatives Feedback konnte diese Arbeit überhaupt erst entstehen. Als nächstes möchte ich Dr.-Ing. Martin Kuhlemann danken, der insbesondere zu Beginn dieser Arbeit maßgeblich an der Themenfindung beteiligt war. Mein Dank geht auch an Andreas Lübcke, der mich ebenfalls mit Anmerkungen unterstützte.

Ein weiteres großes Dankeschön geht an die Abteilung *Bedienkonzepte & Fahrer* der Konzernforschung der Volkswagen AG. Hier wurde ich ein weiteres Mal nach Kräften unterstützt. Mein Dank gilt hier insbesondere Dr. Moritz Neugebauer und Dr. Marius Spika für ihre exzellenten Anmerkungen und Kommentare.

Mein größter Dank geht auch dieses Mal an meine Familie und Freunde. Vor allem danken möchte ich dafür meiner Freundin Maria Müller und meinem Studienkollegen und Geschäftspartner Daniel Zehe. Eure Unterstützung ermöglichte es mir diese Arbeit anzufertigen – Vielen Dank!

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xi**

Quelltextverzeichnis **xiii**

Abkürzungsverzeichnis **xv**

1 Einführung **1**

2 Grundlagen **3**
2.1 Strukturierte Programmierung 3
2.2 Software-Produktlinien 4
2.3 Entwurf von Software-Produktlinien 8
2.4 Implementierung von Software-Produktlinien 13

3 Status quo & Anforderungen **21**
3.1 Status quo 21
3.2 Stakeholderanalyse 28
3.3 Stakeholderanforderungen 29
3.4 Soll-Zustand des Distributionsprozesses der HMIServer SPL 31
3.5 Zusammenfassung 34

4 Extrahieren von Feature-Mappings und Feature-Modellen **35**
4.1 Identifizieren von Features und Feature-Mappings 35
4.2 Extrahieren und Strukturieren eines Feature-Modells 45
4.3 Implementierung von FeatureMapGen und FeatureModGen 55
4.4 Zusammenfassung 56

5 Distribution von Software-Produktlinien **57**
5.1 Zentrale Speicherung von Basis- und Featurecode im SPL-Repository 57
5.2 Installationsprozess der HMIServer SPL 58
5.3 Implementierung der Distributionssoftware 60
5.4 Zusammenfassung 62

6 Evaluierung **65**
6.1 Aufbau & Durchführung der Fallstudie 65
6.2 Ergebnisse der Fallstudie 70

7 Verwandte Arbeiten **73**
7.1 Modulare Softwaresysteme 73
7.2 Abbildung zwischen Features und Implementierungsartefakten 75
7.3 Alternative Feature-Diagramme 76
7.4 Extraktion von Feature-Modellen 76
7.5 Distribution von Software-Produktlinien 77

8 Fazit & Ausblick **81**

A Aufwandsabschätzungen **83**
A.1 FeatureMapGen::All 83
A.2 FeatureMapGen::NoIn 83

B XML-Dateien **85**

C Fallstudie **87**

Literaturverzeichnis **89**

Abbildungsverzeichnis

2.1 Feature-Diagramm der Graphen-Produktlinie (entnommen aus [Kuh11]) 6

2.2 Entwicklungsprozess von Software-Produktlinien (entnommen aus [CE00]) 8

2.3 Proaktives Vorgehensmodell (entnommen aus [Kru02]) 11

2.4 Reaktives Vorgehensmodell (entnommen aus [Kru02]) 11

2.5 Extraktives Vorgehensmodell (entnommen aus [Kru02]) 12

2.6 Aufbau eines OSGi-Bundles 16

2.7 Lebenszyklus eines OSGi-Bundles (entnommen aus [WHKL08]) . . . 19

3.1 Bedienkonzepte, die dem Fahrer dabei helfen neue Fahrzeugfunktionen und deren Zustand zu verstehen 22

3.2 Abbildung zwischen OSGi-Bundles und durch ZIP-Dateien repräsentierte Features 27

3.3 Aufbau der Feature-ZIP-Dateien des HMIServers 27

3.4 Gewünschte Soll-Zustände aus Entwickler- und Kundensicht 32

4.1 Exemplarischer Abhängigkeitsgraph zur Visualisierung von Abhängigkeiten zwischen Implementierungsartefakten 37

4.2 Mögliche Varianten des FeatureModGen-Extraktionsschritts 49

4.3 Feature-Modell und Feature-Mapping nach dem ersten Teil des FeatureModGen-Initialisierungsschrittes 51

4.4 Feature-Modell und Feature-Mapping nach dem FeatureModGen-Initialisierungsschritt 51

4.5 Feature-Modell und Feature-Mapping nach der Extraktion des Implementierungsartefakts H aus den Features *Feature-H* und *Feature-X* . 52

4.6 Feature-Modell und Feature-Mapping nach der Extraktion des Implementierungsartefakts I aus den Features *Feature-H* und *Feature-X* . . 53

4.7 Finales FeatureModGen Feature-Modell und Feature-Mapping 54

4.8 Benutzeroberfläche von FeatureModGen 56

5.1 Benutzeroberfläche zum Installieren und Deinstallieren von Features des HMIServers 58

5.2 Feature-Diagramm zur Repräsentation des in Abbildung 5.1 zugrundeliegenden Feature-Modells 59

5.3 Benutzeroberfläche zum Installieren und Deinstallieren von Features des HMIServers während der Installation von Features 60

6.1 Abhängigkeitsgraph der OSGi-Bundles aus Tabelle C.1 67

6.2 In der Fallstudie durch FeatureModGen extrahiertes Feature-Modell . 68

7.1 Durchsatz am *Berlin Commercial Internet Exchange* während der Veröffentlichung von Apples iOS 5 80

Tabellenverzeichnis

3.1 Stakeholderanforderungen an die Distribution des HMIServers 29

4.1 Exemplarisches, durch FeatureMapGen extrahiertes Feature-Mapping 39

4.2 Feature-Mapping für FeatureMapGen::All 40

4.3 Feature-Mapping für FeatureMapGen::NoIn 41

4.4 Feature-Mapping für FeatureMapGen::MaxIn mit $k = 1$ 43

4.5 Feature-Mapping für FeatureMapGen::MaxInMinOut mit $k = 0$ und $m = 5$ 44

4.6 Vergleich der vier FeatureMapGen-Varianten 44

4.7 Feature-Mapping mit relativen Häufigkeiten der Implementierungsartefakte 50

6.1 Übersicht der in der Fallstudie extrahierten Features 69

7.1 Vergleich von Paketverwaltungssystemen und dem HMIServer 79

C.1 Übersicht der in der Fallstudie verwendeten OSGi-Bundles 87

Quelltextverzeichnis

2.1 Beispielhafte Interfacedefinition in OMG IDL 4

2.2 Generierte Interfacedefinition in Java 4

2.3 Graphen-Produktlinie beschrieben in der GUIDSL-Grammatik (erstellt aus Abbildung 2.1 mithilfe von *FeatureIDE* [KTS+09]) 7

2.4 Manifest-Datei des OSGi-Bundles *Foo* 17

4.1 FeatureMapGen Pseudocode 38

4.2 FeatureMapGen::All Pseudocode 40

4.3 FeatureMapGen::NoIn Pseudocode 41

4.4 FeatureMapGen::MaxIn Pseudocode 42

4.5 FeatureMapGen::MaxInMinOut Pseudocode 44

4.6 FeatureModGen Pseudocode 47

B.1 p2-Feature-Beschreibung für das Feature *Test Tuner* 85

B.2 Erweitertes FeatureIDE-Format zum Speichern von Feature-Modellen inkl. Feature-Mappings 86

Abkürzungsverzeichnis

AE	Application Engineering
AST	Abstract Syntax Tree
CIDE	Colored Integreated Development Environment
CVM	Compositional Variability Management
DE	Domain Engineering
FAMA	FeAture Model Analyser
FD	Feature-Diagramm
FDL	Feature Description Language
FM	Feature-Modell
FMP	Feature Modelling Plugin
FODA	Feature-Oriented Domain Analysis
GPL	Graphen-Produktlinie
HMI	Human Machine Interface
IDL	Interface Definition Language
IU	Installable Unit
JDK	Java Development Kit
OMG	Object Management Group
OOP	objektorientierte Programmierung
S.P.L.O.T.	Software Product Lines Online Tools
SPL	Software-Produktlinie
SXFM	Simple XML Feature Model format
VSL	Variability Specification Language

1. Einführung

Als Ada Lovelace in den Jahren 1842 und 1843 das erste Computerprogramm schuf [FF03], war ihr vielleicht bewusst, welches Potenzial sich hinter solchen Programmen verbirgt und welche Möglichkeiten eine Maschine, die solche Programme ausführen könnte, einer Gesellschaft eröffnen würde. Dass sich jedoch gut 170 Jahre später eine gesamte Industrie mit solchen Maschinen – in der Zwischenzeit als *Computer* bezeichnet – beschäftigen wird, deren Umsätze und Gewinne Rekord um Rekord jagen, war sehr wahrscheinlich auch Ada Lovelace nicht bewusst.
Was heutzutage als erstrebenswert gilt, konnte von Ada Lovelace als erste Programmiererin noch nicht betrieben werden – *Wiederverwendung* von Software. Das Wiederverwenden von Software und Softwareteilen spielt heutzutage in Form von Software-Produktlinien (SPLs), eine entscheidende Rolle in Wissenschaft und Industrie [vdLSR07]. Eine SPL ermöglicht es, durch die Wiederverwendung bestimmter Softwareteile, bestimmte Funktionalitäten in verschiedenen Produkten zu verwenden, ohne die Funktionalität für jedes Produkt erneut zu implementieren. SPLs ermöglichen es auch, bestimmte Funktionalitäten verschieden miteinander zu kombinieren. Die modular implementierten Funktionalitäten einer SPL werden als *Features* bezeichnet [KCH+90]. Die Kombination bestimmter Features resultiert in einer *Variante* der SPL [AK09].
Das Entwickeln einer SPL kann auf verschiedenen Wegen erfolgen. Ein in der Praxis sehr verbreitetes Vorgehen, ist das *extraktive Vorgehen* [Kru02]. Dabei werden aus einer Menge von Softwareprodukten gemeinsame Features extrahiert. Verschiedene Datenbankmanagementsysteme (DBMS) besitzen bspw. die Möglichkeit, atomare Operationen in einer neuen atomaren Operation zusammenzufassen. Das Ergebnis dieser Zusammenfassung wird *Transaktionen* genannt [SSH08]. Da nicht jedes DBMS Transaktionen unterstützt, lässt sich daraus der Schluss ziehen, dass ein DBMS das Feature *Transaktionen* aufweisen *kann*.
Die Implementation eines Features setzt sich im Allgemeinen aus einer bestimmten Menge an Implementierungsartefakten zusammen. Ein Implementierungsartefakt kann z.B. eine Anweisung im Quelltext, eine Methode oder eine gesamte Klasse sein [AK09, TKES11]. Es ist auch möglich, dass ein Implementierungsartefakt in mehreren Features benötigt wird. Daraus folgt, dass es eine Abbildung zwischen Features und

Implementierungsartefakten geben muss, die angibt, welches Feature aus welchen Implementierungsartefakten bestehen wird bzw. welches Implementierungsartefakt für welche Features benötigt wird [Käs07, TKES11].
Wurde eine Software nicht von Beginn ihrer Entwicklung an als SPL entwickelt, so kann das extraktive Vorgehen dazu genutzt werden, nachträglich Features aus der Software bzw. deren Implementierungsartefakten zu extrahieren. Um aus einer Menge von Softwareprodukten Features extrahieren zu können, mussten bisher zunächst von einem Experten dieser Softwareprodukte Features und deren gültige Kombinationen definiert werden. Im Anschluss musste der Experte festlegen, aus welchen Implementierungsartefakten die einzelnen Features bestehen. Dieser manuelle Prozess ist sehr zeitintensiv, fehleranfällig und erfordert eine hohe Expertise.

In dieser Arbeit werden Konzepte, Algorithmen und Werkzeuge vorgestellt, die es ermöglichen aus einer gegebenen Menge an Implementierungsartefakten und Abhängigkeiten zwischen diesen, automatisiert Features zu extrahieren und diese in einem Feature-Modell abzubilden. Es wird auch gezeigt, dass automatisiert extrahierte Feature-Modelle zur Distribution einer SPL genutzt werden können.

Diese Arbeit gliedert sich dazu in die folgenden Kapitel: in Kapitel 2 werden Grundlagen zu Software-Produktlinien eingeführt. Im Anschluss wird in Kapitel 3 erörtert, in welchem industriellen Umfeld diese Arbeit ihren praktischen Einsatz findet und welche Anforderungen an die Distribution einer SPL sich daraus ergeben. Kapitel 4 zeigt anschließend, wie Features, Feature-Mappings und Feature-Modelle aus einer gegebenen Menge an Implementierungsartefakten extrahiert werden können. In Kapitel 5 wird erläutert, wie Feature-Modelle zur Distribution von SPLs genutzt werden können. Kapitel 6 diskutiert die Ergebnisse einer durchgeführten Fallstudie zur Evaluierung der vorgestellten Algorithmen. Kapitel 7 ordnet diese Arbeit noch in den Kontext bereits bestehender Arbeiten ein, bevor in Kapitel 8 ein Fazit gezogen und ein Ausblick auf mögliche zukünftige Arbeiten gegeben wird.

2. Grundlagen

In diesem Kapitel werden konzeptionelle, sowie technische Grundlagen erläutert, die für den späteren Verlauf dieser Arbeit von Bedeutung sein werden. Dazu zählen insbesondere Grundlagen über Software-Produktlinien und deren Entwicklung, sowie Grundlagen über die OSGi Service Plattform.

2.1 Strukturierte Programmierung

Bereits in den 70er Jahren ist man zu der Erkenntnis gelangt, dass sich softwaretechnische Probleme besser lösen lassen, wenn sie in kleinere Teilprobleme aufgeteilt werden [Dij69, Par72, SMC74]. Seither wird argumentiert, dass Strukturierung und Modularisierung von Software *a*) dazu beiträgt, die Größe einer Software zu verringern, *b*) eine Überprüfung auf Korrektheit überhaupt erst ermöglicht und *c*) die Les- und Wartbarkeit einer Software verbessert [Dij69]. Softwareteile, die Teilprobleme lösen, werden als *Module* bezeichnet [Par72]. Module bringen den Vorteil mit sich, dass bei ihrer Implementierung nur sehr wenig Kenntnis über andere Module oder gar Kenntnis über das gesamte System beim Entwickler vorhanden sein muss [Par72]. Die Kommunikation zwischen verschiedenen Modulen eines Softwaresystems wird dabei über sogenannte *Schnittstellen* sichergestellt. Schnittstellen beschreiben dabei in erster Linie Funktionen, die andere Module benutzen können, um mit dem durch die Schnittstelle beschriebenen Modul zu kommunizieren. Die anderen Module benötigen keine weitere Kenntnis über die konkrete Implementierung, die sich hinter einer Schnittstellendefinition verbirgt [Sco05]. Außerdem wird durch eine abstrakte Beschreibung eines Moduls die Möglichkeit geschaffen, Module unabhängig und parallel voneinander zu entwickeln. Dazu ist es lediglich notwendig, vor der Entwicklung der Module, die Schnittstellen zu erstellen. Ein weiterer, wesentlicher Vorteil modularer Programmierung liegt darin, dass Module im späteren Verlauf der Entwicklung des Gesamtsystems einfach ausgetauscht und kombiniert werden können, da kein Teil des Systems eine konkrete Implementierung, sondern lediglich eine bestimmte Schnittstelle erwartet.

Schnittstellen können auf verschiedene Arten beschrieben werden. Steht noch nicht fest, in welcher Programmiersprache ein System entwickelt werden soll, oder soll ein System in verschiedenen Programmiersprachen entwickelt werden, können Schnittstellen durch programmiersprachenunabhängige Definitionssprachen, sogenannte Interface Definition Languages (IDLs), definiert werden. Ein Beispiel für solch eine IDL ist die von der Object Management Group (OMG) entwickelte *OMG IDL* [Vin93]. Mithilfe spezieller Werkzeuge kann die OMG IDL in die jeweils gewünschte Programmiersprache übersetzt werden. So könnte die exemplarische, in Quelltext 2.1 dargestellte, Schnittstellendefinition z.B. in die Programmiersprache Java übersetzt werden. Das dabei entstehende Ergebnis ist in Quelltext 2.2 dargestellt. Im gezeigten Beispiel würde so für das Attribut `active` ein *Getter* `isActive()` und ein *Setter* `setActive(boolean active)` im Java-Interface generiert werden, während die Methode `getName()` syntaktisch direkt übernommen wird.

```
1 interface Employee {
2
3     boolean getName();
4     attribute boolean active;
5 }
```

Quelltext 2.1: Beispielhafte Interfacedefinition in OMG IDL

```
1 public interface Employee {
2
3     boolean getName();
4     boolean isActive();
5     void setActive(boolean active);
6 }
```

Quelltext 2.2: Generierte Interfacedefinition in Java

Im weiteren Verlauf dieser Arbeit werden überwiegend Java-Interfaces zur Beschreibung von Schnittstellen verwendet, da als Grundlage für diese Arbeit bereits bestehende Softwareteile verwendet werden, die in der Programmiersprache Java implementiert worden sind. Welche Softwareteile das im Einzelnen sind wird in Abschnitt 3.1 näher betrachtet.

2.2 Software-Produktlinien

Eine SPL ist eine Menge von Softwaresystemen, die verschiedene Anforderungen in einem gemeinsamen Marktsegment erfüllen und auf einer gemeinsamen Basis entwickelt werden [CN01]. Motiviert werden SPLs zum einen dadurch, dass ein vom Kunden gewünschtes Produkt bestmöglich an seine Bedürfnisse angepasst werden kann und zum anderen dadurch, dass die Wiederverwendung von Quelltext durch die Verwendung von SPLs gesteigert werden und Software dadurch effizienter entwickelt werden kann. Produkte, die Teile des Quelltexts anderer Produkte wiederverwenden und Problemstellungen in der gleichen Domäne lösen, gehören einer gemeinsamen SPL an [CN01]. Um eine Wiederverwendung von Softwareteilen zu gewährleisten,

werden die Softwareteile, die eine bestimmte Anforderung erfüllen hervorgehoben. Diese Softwareteile werden als *Feature* bezeichnet [AK09, CN01, IEE98]. Neben der Erfüllung einer bestimmten Anforderung, stellen Features (z.B. durch ihren Quelltext) eine bestimmte Design-Entscheidung dar und bieten die Möglichkeit konfiguriert zu werden [AK09].
Features sind in einer hierarchischen Struktur organisiert, die es ermöglicht, Features semantisch zu gruppieren. Gruppierungen und andere Abhängigkeiten zwischen Features werden in einem *Feature-Modell (FM)* [KCH+90] erfasst. Durch eine hierarchische Gruppierung ergeben sich Abhängigkeiten zwischen Features, da ein, in der Hierarchie, untergeordnetes Feature abhängig von dem ihm übergeordneten Feature ist. Zum besseren Verständnis wird folgende Notation verwendet: Ein Kind-Feature ist abhängig von seinem Eltern-Feature.
Ein Feature kann ferner mehrere Kind-Features besitzen. Kind-Features eines Features können dabei als

a) *obligatorisch* — das entsprechende Kind-Feature muss ausgewählt werden, wenn sein Eltern-Feature ausgewählt wurde — oder

b) *optional* — das entsprechende Kind-Feature kann optional ausgewählt werden, wenn sein Eltern-Feature ausgewählt wurde —

gekennzeichnet werden. Da ein Feature mehrere Kind-Features haben kann, können auch die Kind-Features zueinander in Beziehung stehen. Ausgehend von der Stammbaum-Metapher, kann bei solchen Features von Geschwister-Features gesprochen werden. Die Beziehung zwischen Eltern- und all ihren Kind-Features kann dabei folgende Ausprägungen annehmen [Bat05]:

a) *Und-Verknüpfung* — alle (nicht-optionalen) Kind-Features müssen ausgewählt werden, wenn ihr Eltern-Feature ausgewählt wurde,

b) *Alternativ-Verknüpfung* — genau ein Kind-Feature muss ausgewählt werden, wenn sein Eltern-Feature ausgewählt wurde,

c) *Oder-Verknüpfung* — mindestens ein Kind-Feature muss ausgewählt werden, wenn ihr Eltern-Feature ausgewählt wurde.

Features können nur als *optional* oder *obligatorisch* gekennzeichnet werden, wenn sie in einer *Und-Verknüpfung* mit ihrem Eltern-Feature verknüpft sind.
Als abschließende Eigenschaft bleibt noch zu nennen, dass FMs in der Lage sind, beliebige Abhängigkeiten zwischen Features abzubilden [KCH+90]. Das bedeutet, dass ein Feature auch Abhängigkeiten aufweisen kann, die sich nicht durch die Hierarchie ergeben. So kann das Kind eines Eltern-Features bspw. eine Abhängigkeit zum Geschwister-Feature seines Eltern-Features aufweisen.

Feature-Modelle als Feature-Diagramme

Zum Zwecke der Veranschaulichung und Kommunikation zwischen Softwareentwicklern und potenziellen Kunden, kann ein FM bspw. durch ein Feature-Diagramm (FD) visualisiert werden [KCH+90]. Ein FD ist in der Lage, sämtliche zuvor beschriebene Eigenschaften eines FMs grafisch darzustellen. Am Beispiel einer *Graphen-Produktlinie (GPL)* [Kuh11, LHB01] lassen sich Vor- und Nachteile eines FDs diskutieren.

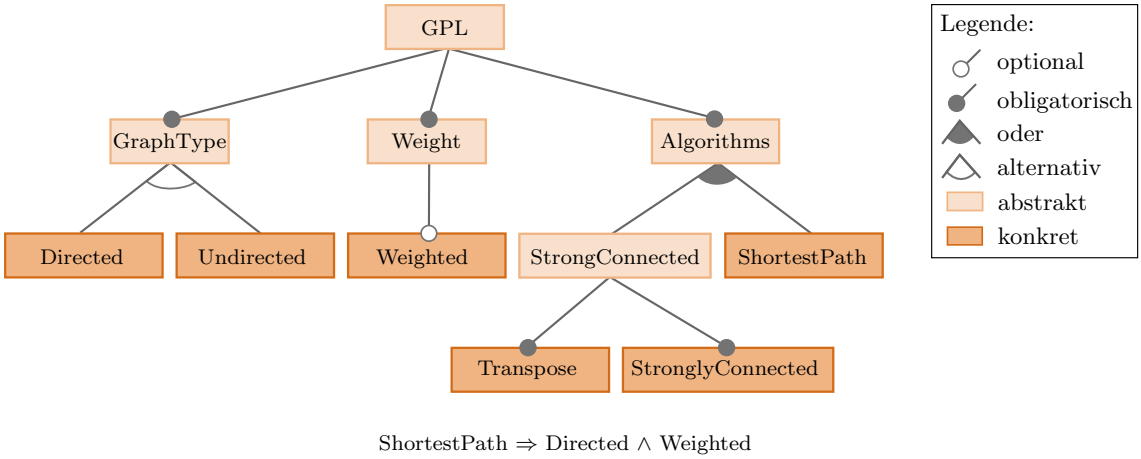


Abbildung 2.1: Feature-Diagramm der Graphen-Produktlinie (entnommen aus [Kuh11])

Abbildung 2.1 zeigt, wie ein mögliches FM der GPL als FD dargestellt werden kann. Die erwähnten Mittel zur Strukturierung (Kennzeichnung als *obligatorisch* oder *optional*, Verwenden von *Und-*, *Oder-* oder *Alternativ-Verknüpfungen*) sind an verschiedenen Beispielen zu erkennen: ein Graph kann bspw. nur gerichtet oder ungerichtet sein, nicht jedoch beides zur gleichen Zeit. Daher wurden die Features *Directed* und *Undirected* mithilfe einer Alternativ-Verknüpfung mit ihrem Eltern-Feature *GraphType* verknüpft. Eine andere Möglichkeit, eine Alternativ-Verknüpfung abzubilden, lässt sich am Feature *Weight* erkennen. Ein Graph kann gewichtet oder ungewichtet sein, aber nicht beides gleichzeitig. Repräsentiert ist dieser Umstand durch ein optionales Feature *Weighted* als Kind des Features *Weight*. Bei der Implementierung ist darauf zu achten, dass ein Graph a priori ein ungewichteter Graph ist. Ist das Feature *Weighted* ausgewählt, muss der Graph entsprechend gewichtet sein. Bei der Auswahl von Algorithmen, die ein Graph unterstützen soll, eröffnet sich die Wahl, das Feature *StrongConnected*, *ShortestPath* oder beide Features auszuwählen. Dem Feature *StrongConnected* sind ferner noch zwei weitere Features unterstellt, die beide ausgewählt werden müssen, da sie als obligatorisch gekennzeichnet wurden.
Die abschließende Formel $ShortestPath \implies Directed \wedge Weighted$ zeigt eine Schwachstelle von FDs auf. Ein FM muss nicht zwingend einer Baumstruktur folgen, wohingegen ein FD zunächst auf eine solche beschränkt ist [KCH+90]. Um die Konsistenz zwischen FM und FD wiederherzustellen, bedarf es zusätzlicher Bedingungen, die für eine gültige Featureauswahl erfüllt sein müssen. Solche Bedingungen werden in FDs üblicherweise mithilfe aussagenlogischer Ausdrücke realisiert und werden *cross-tree constraints* genannt. Im gezeigten Beispiel bedeutet dies, dass *ShortestPath*

nur ausgewählt werden kann, wenn auch *Directed* und *Weighted* ausgewählt wurden. Die gute Übersichtlichkeit einer Baumstruktur kann durch solch zusätzliche Bedingungen schnell verloren gehen – vor allem dann, wenn diese in Komplexität und Anzahl stark zunehmen.
Die in FDs verwendete Baumstruktur resultiert auch darin, dass die Auswahl eines Features immer dazu führt, dass sein Eltern-Feature und alle weiteren Feature auf dem *Weg* zur Wurzel des Baumes ausgewählt werden müssen. Im vorliegenden Beispiel bedeutet dies bspw., dass eine Auswahl des Features *Transpose* dazu führt, dass auch die Features *StrongConnected*, *Algorithms* und *GPL* ausgewählt werden müssen.

Maschinenlesbare Feature-Modelle

```
1  GPL : GraphType Weight Algorithms+ :: _GPL ;
2
3  GraphType : Directed
4      | Undirected ;
5
6  Weight : [Weighted] :: _Weight ;
7
8  Algorithms : Transpose StronglyConnected :: StrongConnect
9      | ShortestPath ;
10 %%
11 ShortestPath implies Directed and Weighted ;
```

Quelltext 2.3: Graphen-Produktlinie beschrieben in der GUIDSL-Grammatik (erstellt aus Abbildung 2.1 mithilfe von *FeatureIDE* [KTS+09])

Da eine grafische Repräsentation eines FMs zwar bei der Kommunikation von Features zwischen Menschen hilft, jedoch nicht schnell und einfach von Maschinen gelesen werden kann, wurden auch andere Beschreibungsformen von FMs entwickelt, die für Maschinen leichter zu lesen und zu verarbeiten sind. So können bspw. Grammatiken verwendet werden, um Hierarchien und andere Abhängigkeiten zu beschreiben. Ein populäres Beispiel hierfür ist die *GUIDSL*-Grammatik [Bat06]. Quelltext 2.3 zeigt die GPL, beschrieben durch die GUIDSL-Grammatik. Neben Grammatiken sind auch andere textuelle Beschreibungsformen für FMs entstanden. Dazu zählen unter anderem:

- die *Feature Description Language (FDL)* [vDK02],
- das *Feature Modelling Plugin (FMP)* [AC04],
- das Variantenmanagementsystem *pure::variants* [Beu03],
- der *FeAture Model Analyser (FAMA)* [BSTRC07],
- die web-basierten *Software Product Lines Online Tools (S.P.L.O.T.)* [MBC09] bzw. das darin verwendete *Simple XML Feature Model format (SXFM)*,
- das *Compositional Variability Management (CVM)* [Rei09] Framework, das zur Speicherung der FMs die *Variability Specification Language (VSL)* [Rei09] nutzt sowie,

- das XML-basierte *FeatureIDE-Format* [KTS+09].

Ein Vergleich [CBH11] der genannten textuellen Beschreibungsformen zeigte, dass sie sich im wesentlichen in menschlicher Lesbarkeit und ihrer Möglichkeit, Abhängigkeiten zwischen Features zu definieren (z.B. durch Kardinalitäten), unterscheiden. Es gibt daher nicht *die* beste Möglichkeit, Feature-Modelle zu beschreiben. Vielmehr hängt es vom Nutzungkontext der Modelle ab. Sollen bspw. einem möglichen Kunden mögliche Features und deren Zusammenhänge erklärt werden, würde man ihm kein FM in GUIDSL präsentieren, sondern auf grafische Repräsentationen wie FDs zurückgreifen. Eine parallele Nutzung verschiedener Beschreibungsformen kann daher nicht ausgeschlossen werden. Es muss jedoch sichergestellt werden, dass ein Feature-Modell verlustfrei von einer Beschreibungsform in eine andere überführt werden kann (unter Umständen auch zurück in die andere Richtung) [TKB+12].

2.3 Entwurf von Software-Produktlinien

Im Vergleich zu einer Softwareentwicklung nach dem Wasserfall- [Roy87] oder Spiralmodell [Boe88], lässt sich eine SPL nicht nach diesen klassischen Softwarelebenszyklen entwickeln. Im Allgemeinen wird der Entwicklungsprozess einer SPL in *Domain Engineering (DE)* und *Application Engineering (AE)* unterteilt [CE00]. Abbildung 2.2 zeigt sowohl die jeweils einzelnen Schritte im DE und AE, als auch deren Verzahnung.

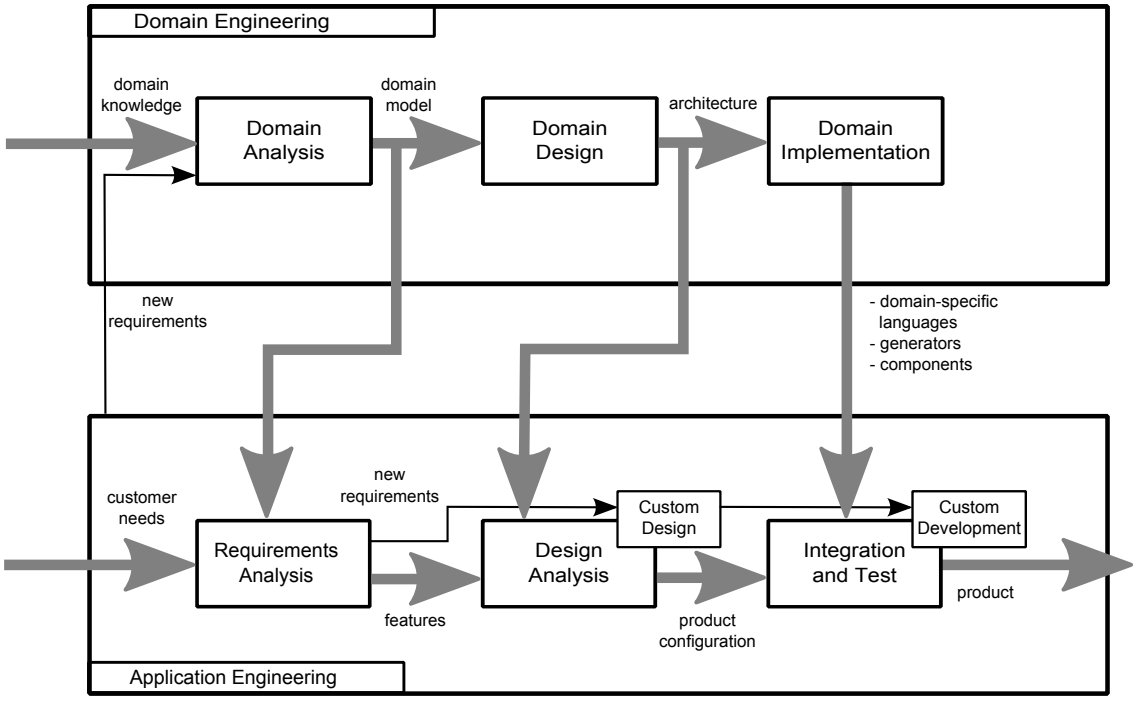


Abbildung 2.2: Entwicklungsprozess von Software-Produktlinien (entnommen aus [CE00])

Domain Engineering

Die Entwicklung einer SPL beginnt mit dem *Domain Engineering*, kurz DE, genauer gesagt mit der *Domain Analysis*. Mit dem über die Domäne vorhandenen Expertenwissen wird dabei analysiert, welche Features in der Domäne existieren und wie sie in Verbindung stehen. Das Ergebnis der Domain Analysis kann z.B. ein FM sein. Mithilfe der in Abschnitt 2.2 genannten Techniken können bereits zu diesem Zeitpunkt unerlaubte Feature-Kombinationen ausgeschlossen werden (wie z.B. die Tatsache, dass in der GPL ein Graph nicht gerichtet und gleichzeitig ungerichtet sein kann). Mit dem erstellten FM kann dann die Architektur der Domäne entworfen werden (*Domain Design*). Welche Möglichkeiten dabei zur Wahl stehen, wird in Abschnitt 2.4 erörtert. Nachdem sich für eine Architektur entschieden wurde, kann der Basiscode und die ermittelten Features mit der gewählten Technik implementiert werden. Diese Phase wird als *Domain Implementation* bezeichnet.

Application Engineering

Das *Application Engineering*, kurz AE, verfolgt das Ziel, aus einer SPL unter Berücksichtigung bestimmter Anforderungen ein konkretes Produkt zu generieren. In Abhängigkeit von der im Domain Engineering gewählten Architektur und Implementierung, kann dieser Prozess variieren. Allen Varianten ist jedoch gemein, dass zunächst die Anforderungen der Kunden gesammelt werden müssen. Aus diesen, vom Kunden möglicherweise vage formulierten, Anforderungen können dann in einem iterativen Prozess konkrete Anforderungen gemeinsam mit dem Kunden ermittelt werden. Dieser erste Schritt wird üblicherweise als Anforderungsanalyse bezeichnet (engl. *Requirements Analysis*). Als Ergebnis dieser Analyse entstehen möglicherweise Anforderungen, die nicht beim Entwurf der Domäne im Domain Engineering berücksichtigt wurden. Diese neuen Anforderungen machen es zum einen erforderlich, dass kundenspezifische Anpassungen am Produkt vorgenommen werden müssen und zum anderen, dass diese neuen Anforderungen zurück in das Domain Engineering geführt werden sollten, um auch künftige Kunden von den Anpassungen profitieren zu lassen [CE00].
Anhand der ermittelten Anforderungen und der zuvor für die Domäne entwickelten Features kann dem Kunden ein konkretes Produkt aus einer bestehenden SPL generiert werden. Das bedeutet, dass bestimmt wird, welche Features benötigt werden, um die Anforderungen des Kunden zu erfüllen, welcher Quelltext angepasst werden muss und welche neuen Features entwickelt werden müssen.
Abschließend kann dann ein auf den Kunden zugeschnittenes Produkt *generiert* werden. Diese Generierung darf dabei optimalerweise nicht mehr Aufwand erzeugen, als eine komplette Neuimplementierung erzeugt hätte. Um den Aufwand dieser Generierung möglichst gering zu halten, wird sie in der Regel von bestimmten Compilern oder Generatoren übernommen, die den entsprechenden Quelltext der einzelnen Features zu einem Software-Produkt zusammenfügen.

2.3.1 Featureorientierte Domänenanalyse

Das Domain Engineering stellt eine der Kernkomponenten der Entwicklung von SPLs dar. Die Domain Analysis wurde in Arbeiten wie bspw. *Feature-Oriented*

Domain Analysis (FODA) [KCH+90] ausführlich diskutiert. FODA's Ziel ist es, Gemeinsamkeiten und Unterschiede einer Menge von Systemen als solche zu identifizieren, um durch die Wiederverwendung von Gemeinsamkeiten später profitieren zu können. Die FODA ist dabei in drei Phasen unterteilt: in der ersten Phase, der *context analysis*, werden die Grenzen der Domäne definiert. Das Ergebnis dieser ersten Phase ist ein *context model*, das aufzeigt, in welcher Umgebung eine Domäne ihren Einsatz findet. Dazu zählen z.B. andere Applikationen mit denen die Domäne interagieren muss, bestimmte Dateiformate, die berücksichtigt werden müssen oder andere Eigenschaften der Umgebung, in der Produkte einer Domäne ausgeführt werden sollen [KCH+90].
In der zweiten Phase der FODA, dem *Domain Modelling*, wird zunächst durch die *Feature Analysis* ein Feature-Modell entwickelt, das Gemeinsamkeiten und Unterschiede der einzelnen Produkte einer Domäne zusammenfasst. In der darauf folgenden *Information Analysis*, werden Erfahrungen von Entwicklern und Wissenschaftlern zusammen mit Kenntnissen aus bereits existierenden Systemen festgehalten. Als letzten Schritt des *Domain Modellings* werden Gemeinsamkeiten und Unterschiede im Verhalten der Software durch Zustandsdiagramme eruiert. Das Ergebnis des *Domain Modellings* schließt also nicht nur ein FM ein, sondern gibt auch Aufschluss über bereits gemachte Erfahrung aus Industrie und Wissenschaft und definiert Unterschiede und Gemeinsamkeiten im Verhalten der möglichen, einzelnen Produkte einer Domäne [KCH+90].
Die dritte und letzte Phase der FODA wird von den Autoren *Architecture Modelling* genannt. Das Ergebnis dieser Phase ist ein Software-Design für die gesamte SPL, das es ermöglicht, die Gemeinsamkeiten der einzelnen Produkte so zu entwickeln, dass sie später auch gemeinsam genutzt werden können. Die speziellen Anforderungen der einzelnen Produkte können dann später, je nach Wunsch des Kunden, durch Hinzufügen von bestimmtem Quelltext, erfüllt werden. [KCH+90].

2.3.2 Vorgehensmodelle

Es existieren verschiedene Möglichkeiten des Vorgehens, eine Menge von Produkten als SPL zu entwickeln. Charles W. Krueger definierte die drei folgenden Vorgehensmodelle [Kru02]: proaktives Vorgehensmodell, reaktives Vorgehensmodell und extraktives Vorgehensmodell.

Proaktives Vorgehensmodell

Das proaktive Vorgehensmodell trifft die Grundannahme, dass noch keine Zeile Quelltext geschrieben wurde, um eine SPL zu entwickeln. Beim proaktiven Vorgehen wird zunächst Domain Engineering so vollständig wie möglich durchgeführt. Als Ergebnis entsteht ein Feature Modell mit allen möglichen Features der SPL. Produkte dieser SPL können dann entsprechend der Anforderungen eines Kunden generiert werden. Abbildung 2.3 zeigt das proaktive Vorgehensmodell. Dargestellt ist, dass die zu einer SPL zugehörigen Produkte erst dann aus der SPL abgeleitet werden, nachdem durch eine Domänenanlyse Architektur und Design der SPL erarbeitet und daraus Features und Beziehungen zwischen den Features entwickelt wurden.

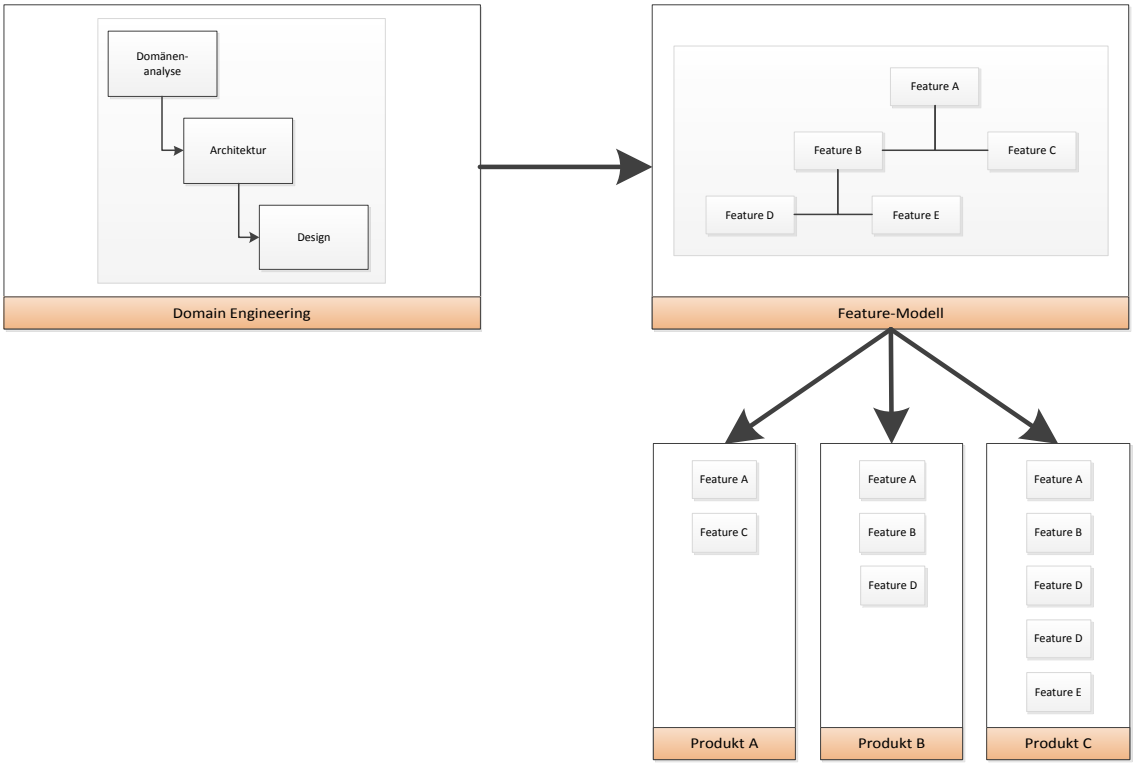


Abbildung 2.3: Proaktives Vorgehensmodell (entnommen aus [Kru02])

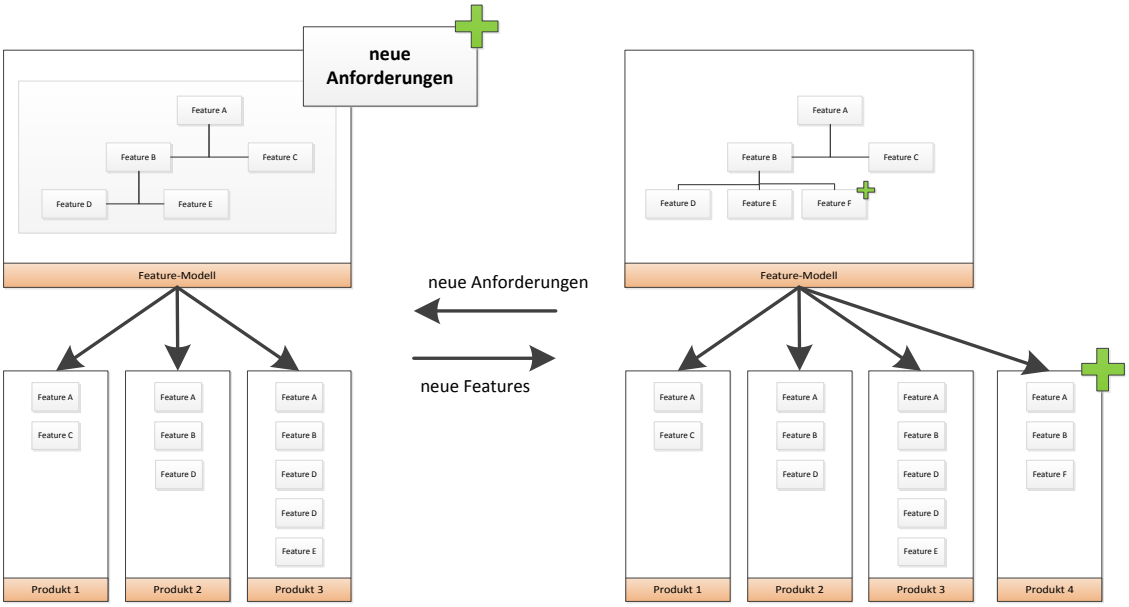


Abbildung 2.4: Reaktives Vorgehensmodell (entnommen aus [Kru02])

Reaktives Vorgehensmodell

Beim Verfahren nach dem reaktiven Vorgehensmodell werden Features, und damit Variabilität, im Laufe der Zeit zur SPL hinzugefügt. Im Vergleich zum proaktiven Vorgehen, werden beim reaktiven Vorgehen nicht von Anfang an alle Features berücksichtigt, weil z.B. noch nicht alle Features von Beginn an bekannt sind. Abbildung 2.4

schematisiert, dass eine bereits bestehende SPL durch das Hinzukommen neuer Anforderungen erweitert werden muss, um die neuen Kundenwünsche zu erfüllen. Die Erweiterung sieht dabei vor, dass deren Ergebnisse auch zurück in die bereits bestehende SPL übernommen werden können und nicht nur eine Sonderlösung für einen einzelnen Kunden darstellen. Die SPL entwickelt sich weiter, weil Anforderungen entstanden sind, die auf einen Mangel an Variabilität zurückzuführen sind. Durch das Hinzufügen neuer Variabilität können Anforderungen auch für zukünftige Kunden besser erfüllt werden.
Die Entwicklung einer SPL kann nach diesem Vorgehensmodell mit einer geringen Variabilität beginnen und immer dann Variabilität hinzufügen, wenn entsprechende Wünsche von Kunden entstehen. Der Vorteil: im Vergleich zum proaktiven Vorgehensmodell benötigt das reaktive Modell keine allzu langen Vorentwicklungszeiten für Architektur und Design der SPL. Der Nachteil liegt jedoch darin, dass das Hinzufügen neuer Variabilität mit der Zeit immer aufwendiger und komplexer werden kann.

Extraktives Vorgehensmodell

Das, im Vergleich zum proaktiven, umgekehrte Vorgehen wird durch das extraktive Vorgehensmodell beschrieben. Dabei werden aus bestehenden Produkten so lange Gemeinsamkeiten abgeleitet, bis diese Gemeinsamkeiten als Features verstanden werden und in einem FM sinnvoll abgebildet werden können. Analog zu den beiden anderen Vorgehensmodellen ist das extraktive Vorgehenmodell in Abbildung 2.5 abgebildet.

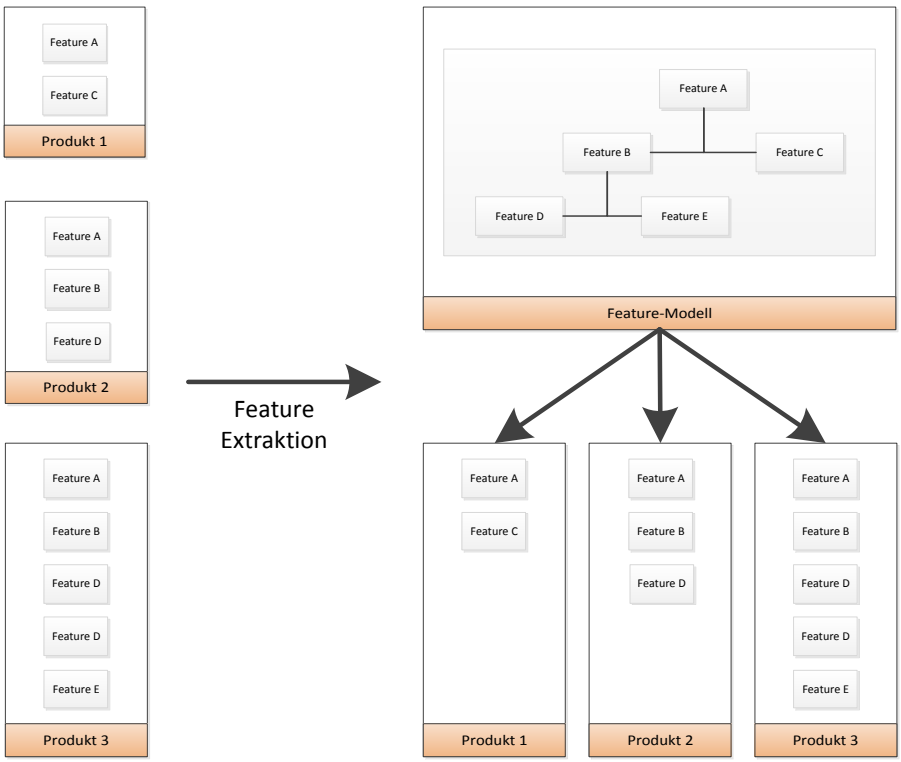


Abbildung 2.5: Extraktives Vorgehensmodell (entnommen aus [Kru02])

2.3.3 Abbildung zwischen Features und Implementierungsartefakten

Die Aufgabe eines Features besteht darin, ein Produkt einer SPL zu verändern, damit eine bestimmte Anforderung erfüllt wird. Um eine bestimmte Anforderung zu erfüllen, bedarf es in der Regel Quelltextänderungen an der Basis der SPL. Zu einem Feature existieren daher ganz bestimmte Implementierungsartefakte, die die tatsächlichen, durch Quelltext realisierte, Änderungen bewirken. Dabei muss die Anzahl der Implementierungsartefakte, die zu einem bestimmten Feature gehören, nicht auf eins beschränkt sein. Ein Feature kann auch aus mehreren Implementierungsartefakten bestehen. Auch kann ein Implementierungsartefakt nur dann zum Einsatz kommen, wenn mehrere Features gleichzeitig ausgewählt sind. Zwischen Features und Implementierungsartefakten besteht daher eine *n:m-Relation*.
Die Relation zwischen Features und deren Implementierungsartefakte wird auch als *Feature Mapping* [HKW08, TKES11] bezeichnet. Feature Mappings können entweder manuell, vom Entwickler erstellt und gepflegt werden, oder aber automatisiert aus den Implementierungsartefakten extrahiert werden, während der Entwickler vorrangig an der Entwicklung der Implementierungsartefakte eines Features arbeitet [HKW08]. Die automatisierte Extraktion von Feature-Mappings wird im späteren Verlauf dieser Arbeit von erhöhter Bedeutung sein, da Methoden und Werkzeuge zur Verfügung gestellt werden, die es ermöglichen, aus bereits vorhandenen Implementierungsartefakten Features, Featureabhängigkeiten und Feature-Modelle zu extrahieren.

2.3.4 Abstrakte Features

Das bereits in Abschnitt 2.2 gezeigte Feature-Diagramm in Abbildung 2.1 enthält sogenannte *abstrakte Features* [TKES11]. Abstrakte Features werden in Feature-Modellen verwendet, um darzustellen, dass diese Features keine Implementierungsartefakte beinhalten. Wird ein solches Feature ausgewählt, wird kein Quelltext zum finalen Produkt hinzugefügt oder verändert. Durch abstrakte Features ist es möglich, eine übersichtlichere Struktur in einem Feature-Modell zu schaffen. Abstrakte Feature führen aufgrund der höheren Anzahl an Features zu einer höheren Anzahl an möglichen Featurekombinationen, erhöhen dabei aber nicht die Anzahl der möglichen Programmvarianten, da durch sie kein Quelltext zum Produkt hinzugefügt und somit keine neue Variabilität der im Quelltext der SPL erzeugt wird [TKES11].

Abstrakte Features werden auch in dieser Arbeit eingesetzt, um bspw. die Struktur des extrahierten Feature-Modells für den Menschen verständlicher erscheinen zu lassen.

2.4 Implementierung von Software-Produktlinien

Wie bereits angesprochen, können SPLs auf verschiedene Arten entwickelt und implementiert werden. Zu den verbreitetsten Techniken gehören:

- *Annotative Techniken:*
  Eine SPL kann implementiert werden, indem der gesamte Quelltext mit Annotationen versehen wird, mithilfe derer gekennzeichnet wird, welche Teile des

Codes zu welchem Feature gehören [KA08]. Dabei wird zunächst das komplette System, mit allen Features implementiert, um bei der Erstellung eines konkreten Produkts, durch die Auswahl von Features den Quelltext zu entfernen, der nicht länger benötigt wird [Kuh11]. Im praktischen Einsatz weit verbreitet ist der `#ifdef`-Präprozessor [SC92] der Programmiersprache C.

- *Kompositionelle Techniken:*
  Eine SPL kann aber auch mit kompositionellen Techniken entwickelt werden. Dabei werden die einzelnen Features vom Entwickler in separaten Modulen entwickelt. Ein vom Kunden gewünschtes Produkt wird dann nach Auswahl der Features aus diesen Modulen zusammengestellt. Kompositionelle Ansätze zur SPL-Entwicklung beinhalten folgende Programmiertechniken:

  a) Eine SPL kann durch *objektorientierte Programmierung (OOP)* implementiert werden. Dabei können verschiedene, sogenannte *Design Patterns* [GHJV95] zum Einsatz kommen, um die einzelnen Features mittels OOP-Techniken zu implementieren. Design Patterns sind Entwurfsmuster von Lösungen für wiederkehrende Probleme. Design Patterns geben Techniken und Methoden an, die bei der Lösung von Problemen helfen. Je nach Art des Problems sind das verschiedene Design Patterns. Nach *Gamma et al.* lassen sich über 40 Design Patterns definieren [GHJV95]. Dazu zählt z.B. das *Framework*, das es ermöglicht eine Software an bestimmten Stellen durch bestimmte *Plug-ins* zu ergänzen.

  b) SPLs können auch mit *featureorientierter Programmierung* [BSR04] entwickelt werden. Dabei wird zunächst der Basiscode entwickelt, der dann über sogenannte *Refinements* [BSR04] zur Übersetzungszeit um die Funktionalität der entsprechenden Features *verfeinert* wird. Verfeinerungen können bestehende Methoden aus dem Basiscode bzw. dem bereits verfeinerten Code, erweitern, hinzufügen oder überschreiben [BSR04].

  c) *Aspektorientierte Techniken:*
  Eine weitere Möglichkeit SPLs zu entwickeln liegt darin, *aspektorientierte Programmierung* [Gri00] zu verwenden. Mithilfe von sogenannten *Aspekten* kann dabei an definierten Stellen im Basiscode (*sog. Joint Points*) der Programmfluss vom Entwickler beeinflusst werden. Zur Übersetzungszeit werden die Aspekte dann in den Basiscode *eingewebt*. Der Basiscode wird beim Einweben der entsprechenden Aspekte automatisch vom Übersetzer verändert. Ein Aspekt kann bspw. dazu genutzt werden, vor und nach der Ausführung einer bestimmten Methode eine *Logging*-Ausgabe zu erzeugen. Welche Features ausgewählt wurden entscheidet darüber, welche Aspekte in den bestehenden Quelltext integriert werden [AK09].

Da objektorientierte Techniken für den weiteren Verlauf dieser Arbeit von erhöhter Bedeutung sein werden, werden im folgenden Abschnitt objektorientierte Techniken zur Entwicklung von SPLs detaillierter betrachtet. Neben Komponenten und Frameworks wird dazu die *OSGi Service Platform* vorgestellt, die für die in dieser Arbeit entwickelten Konzepte und Algorithmen als Test- und Evaluierungsumgebung dienlich gewesen ist.

2.4.1 Objektorientierte Techniken zur Entwicklung von Software-Produktlinien

Komponenten

> *[...], software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system.*
>
> *Clemens Szyperski, 1997 [Szy97]*

Komponenten sind eine Möglichkeit, Software-Systeme modular zu gestalten. Die einzelnen Komponenten übernehmen dabei gezielt mehr oder weniger große Aufgabengebiete und bieten anderen Komponenten Methoden über eine Schnittstelle an, mithilfe derer sich Problemstellungen aus dem jeweiligen Aufgabengebiet lösen lassen [Szy97]. Sie kapseln also eine gewisse Funktionalität. Die Implementierungsdetails dieser Funktionalität bleiben dabei verborgen – eine Komponente wird daher auch als *Black Box* bezeichnet.
Das Zusammenfügen mehrerer Komponenten kann ein funktionsfähiges Gesamtsystem ergeben. Die Komponenten sind durch das Feature-Mapping, den im Domain Enginieering ermittelten Features zugeordnet. Aufgrund des Feature-Mappings können bei einer gegebenen Auswahl an Features, die entsprechenden Komponenten ermittelt werden, um die gewünschte Funktionalität des Gesamtsystems zu erzielen. Das Erstellen des Gesamtsystems geschieht durch einen Softwareentwickler, der die erforderlichen Komponenten durch sogenannten *Glue Code* miteinander verbindet. Glue Code ist Quelltext, der für jeden Kunden individuell angefertigt werden muss und daher je nach Anzahl der Komponenten einen erheblichen Aufwand bedeuten kann.

Frameworks

> *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes.*
>
> *Ralph E. Johnson und Brian Foote, 1988 [JF88]*

Frameworks kapseln keine direkte Funktionalität, sondern legen fest, wie die Vernetzung der Komponenten strukturiert ist. Dabei können abstrakte Klassen benutzt werden, um die Struktur zu beschreiben [JF88], die dazu genutzt werden kann, um ein konkretes Problem, für einen Kunden zu lösen. Bei der Entwicklung einer SPL werden Frameworks meist so eingesetzt, dass ein für alle Produkte gemeinsames Framework entwickelt wird, das bei der Erzeugung eines bestimmten Produkts durch *Plug-ins* angepasst und erweitert werden kann. Eine solche Anpassung oder Erweiterung kann an, während der Entwicklung des Frameworks, definierten Punkten geschehen – sogenannten *Hot Spots*. Wie stark sich ein Framework dadurch anpassen und erweitern lässt, hängt davon ab, ob es sich um ein *Black-Box*- oder ein *White-Box*-Framework

handelt. Wie Komponenten, gibt ein Black-Box-Framework keine Implementierungdetails bekannt. Das White-Box-Framework gestattet hingegen, die Ansicht und Modifizierung von Implementierungdetails. White-Box-Frameworks können daher meist angepasst werden, erfordern im Gegenzug jedoch ein Verständnis für das gesamte Framework.

In Kombination mit Komponenten ergibt sich ein mächtiges Werkzeug zur SPL-Entwicklung: betrachtet man das Framework als den Glue Code, der Komponenten miteinander verbindet, ergeben sich durch die Verwendung eines Frameworks auf der einen Seite die Vorteile der Wiederverwendbarkeit der Komponenten, ohne auf der anderen Seite zu viel Ressourcen und Zeit damit zu verlieren, die Komponenten zu verbinden, da diese Verbindung einmalig durch das Framework geschehen ist. Es muss jedoch vom Entwickler des Frameworks darauf geachtet werden, dass das Framework möglichst erweiterbar implementiert wird, um auch die Integration künftiger, unbekannter Komponenten zu ermöglichen.

2.4.2 Die OSGi Service Platform

Die OSGi Service Platform ist eine Java-basierte Softwareplattform, die es ermöglicht, einzelne Softwarekomponenten zur Laufzeit der gesamten Softwareplattform dynamisch zu starten und zu stoppen. Die Spezifizierung der Service Platform wird seit März 1999 von der OSGi Alliance (bis Anfang 2004 unter dem Namen *Open Service Gateway Initiative*) als Zusammenschluss von Unternehmen wie z.B. IBM und Oracle vorangetrieben [WHKL08].
Den Kern der OSGi Service Platform stellt das OSGi-Framework dar. Es ermöglicht es, einzelne Softwarekomponenten (sogenannte OSGi-Bundles), zu installieren, zu starten, zu stoppen und zu deinstallieren.

Aufbau eines OSGi-Bundles

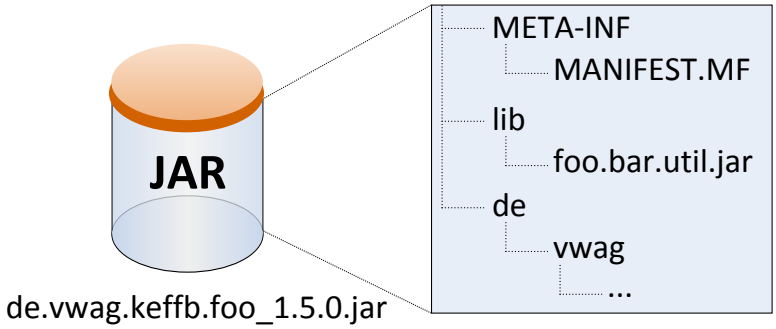


Abbildung 2.6: Aufbau eines OSGi-Bundles

OSGi-Bundles, die in einem OSGi-Framework verwendet werden sollen, müssen einem bestimmten Aufbau folgen, um vom OSGi-Framework genutzt werden zu können. Abbildung 2.6 visualisiert die wesentlichen Bestandteile dieses Aufbaus. Neben den kompilierten Klassen, die einer Ordnerstruktur äquivalent zu ihrem jeweiligen Javapackage folgen (im Beispiel *de/vwag/...*) und optionalen Bibliotheken (im

Beispiel *foo.bar.util.jar* im Ordner *lib*), muss jedes OSGi-Bundle eine Manifest-Datei beinhalten, in der Metainformationen zum jeweiligen OSGi-Bundle gespeichert sind. Exemplarischer Inhalt einer solchen Manifest-Datei ist in Quelltext 2.4 dargestellt.

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Foo
4 Bundle-SymbolicName: de.vwag.keffb.foo
5 Bundle-Version: 1.5
6 Bundle-Vendor: Volkswagen AG Konzernforschung – Daniel Lueddecke
7 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
8 Export-Package: de.vwag.keffb.foo.myPackage;version=1.1.0
```

Quelltext 2.4: Manifest-Datei des OSGi-Bundles *Foo*

Der Inhalt eines OSGi-Bundles ist schließlich in einer JAR-Datei archiviert. Der Dateiname dieser Jar-Datei ist dabei frei wählbar. Üblicherweise trägt er jedoch den sogenannte *symbolischen Namen* (vgl. *Bundle-SymbolicName* in Quelltext 2.4) des OSGi-Bundles, gefolgt von einem Trennzeichen und der Versionsnummer des OSGi-Bundles. Wird bei jeder Änderung am Quelltext die Versionsnummer inkrementiert, verhindert diese Namenskonvention Kollisionen im Dateinamen, da der symbolische Name eines OSGi-Bundles eindeutig sein muss.

Der OSGi-Service

Die in einem OSGi-Framework vorhandenen OSGi-Bundles können als *Services* miteinander kommunizieren. Ein Service ist die Inkarnation einer Java-Klasse, die über einen bestimmten Interface-Namen im OSGi-Framework bekannt gemacht wurde. Die *ServiceRegistry* verwaltet die im OSGi-Framework vorhandenen Services und ist beim Auffinden bestimmter Services dienlich [OSG09]. Durch diese serviceorientierte Infrastruktur, können OSGi-Bundles Daten und Informationen austauschen. Benötigt ein OSGi-Bundle einen bestimmten Service, fragt es bei der ServiceRegistry mit dem Interface-Namen, mit dem der Service sich registriert hat, alle OSGi-Bundles, die das entsprechende Interface implementieren, ab. Von nun an kann das OSGi-Bundle direkt mit einem anderen OSGi-Bundle kommunizieren. Dabei ist es völlig unerheblich, welche konkrete Implementierung hinter dem erhaltenen Java-Objekt steht. Wichtig ist nur, dass es das Interface implementiert und die durch das Interface versprochenen Funktionalität erfüllt. Da kein OSGi-Bundle eine konkrete Implementierung erwartet, können OSGi-Bundles, die einen Service zur Verfügung stellen, leicht ausgetauscht werden, ohne dass die Kommunikation mit dem ausgetauschtem OSGi-Bundle davon betroffen ist.

Abhängigkeiten zwischen OSGi-Bundles

Ein OSGi-Bundle kann die in ihm archivierten Klassen und Ressourcen anderen OSGi-Bundles zur Verfügung stellen. Dies geschieht jedoch nicht implizit, sondern muss explizit vom Entwickler, in der Manifest-Datei des OSGi-Bundles, angegeben werden. Dabei hat der Entwickler die Möglichkeiten anzugeben, welche *Java-Pakete* (Packages) das entsprechende OSGi-Bundle *exportieren* soll. Exportierte Packages

stehen anderen OSGi-Bundles dann zum Import zur Verfügung. Dadurch entsteht eine Abhängigkeit zwischen zwei OSGi-Bundles auf *Package-Ebene*.
In der Manifest-Datei des OSGi-Bundles *Foo* wurde vom Entwickler angegeben, dass *Foo* das Package *de.vwag.keffb.foo.myPackage* in Version *1.1.0* exportiert (siehe *Export-Package* in Quelltext 2.4). Das OSGi-Bundle *Bar*, dessen Manifest-Datei in Quelltext 2.5 dargestellt ist, importiert das von *Foo* exportierte Package in einer Version zwischen `1.1.0` (inklusive) und `2.0.0` (exklusive). Zur Ausführung des OSGi-Bundles *Bar*, muss also das OSGi-Bundle *Foo* im OSGi-Framework bekannt sein. Ist es dem OSGi-Framework nicht bekannt, kann *Bar* nicht verwendet werden.

```
9  Manifest-Version: 1.0
10 Bundle-ManifestVersion: 2
11 Bundle-Name: Bar
12 Bundle-SymbolicName: de.vwag.keffb.bar
13 Bundle-Version: 1.0.0
14 Bundle-Vendor: Volkswagen AG Konzernforschung – Daniel Lueddecke
15 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
16 Import-Package: de.vwag.keffb.foo.myPackage;version=[1.1.0, 2.0.0)
```

Quelltext 2.5: Manifest-Datei des OSGi-Bundles *Bar*

Neben Abhängigkeiten auf Package-Ebene, können OSGi-Bundles auch auf einer *Service-Ebene* voneinander abhängig sein. Benötigt ein OSGi-Bundle einen bestimmten Service, um seine Funktionalität zu erfüllen, entsteht dabei eine Abhängigkeit zwischen dem OSGi-Bundle und einem weiteren OSGi-Bundle, dass den geforderten Service implementiert. Eine solche Abhängigkeit kann nicht explizit in der Manifest-Datei eines OSGi-Bundles definiert werden und lässt sich daher nur direkt im Quelltext eines OSGi-Bundles erkennen.

Lebenszyklus eines OSGi-Bundles

Dem OSGi-Framework sind OSGi-Bundles bekannt, wenn sie in ihm *installiert* sind. Das Installieren eines OSGi-Bundles (*install* genannt) lädt sämtliche Dateien, aus denen das OSGi-Bundle besteht in den Hauptspeicher des Rechners und stellt seine Klassen und Ressourcen damit im OSGi-Framework zur Verfügung. Welche anderen Zustände ein OSGi-Bundle annehmen kann, zeigt Abbildung 2.7.
Nach der Installation eines OSGi-Bundles wird das OSGi-Framework versuchen das OSGi-Bundle aufzulösen (*resolve*). Dabei werden zunächst eventuelle Abhängigkeiten zu anderen OSGi-Bundles detektiert und andere OSGi-Bundles gestartet, die für die Auflösung des OSGi-Bundles notwendig sind. Für das obige Beispiel von *Foo* und *Bar* würde dies bedeuten, dass, vorausgesetzt beide OSGi-Bundles sind installiert, eine Auflösung von *Bar* zu einer Auflösung von *Foo* führen würde, da *Bar* ein Package benötigt, das von *Foo* exportiert wird. *Bar* ist daher von *Foo* abhängig. Es könnte aber auch ein drittes OSGi-Bundle geben, dass das Package exportiert, das *Bar* benötigt. In einem solchem Fall wäre *Bar* von *Foo* oder dem dritten OSGi-Bundle abhängig. Welches OSGi-Bundle vom OSGi-Framework aufgelöst wird, um die Abhängigkeit zu erfüllen, ist nicht klar definiert. Als erstes würde das OSGi-Framework versuchen, dass OSGi-Bundle aufzulösen, dessen exportierte Package-Version *besser* zu der im Import spezifizierten Version passt. Abhängig von der konkreten Implementation

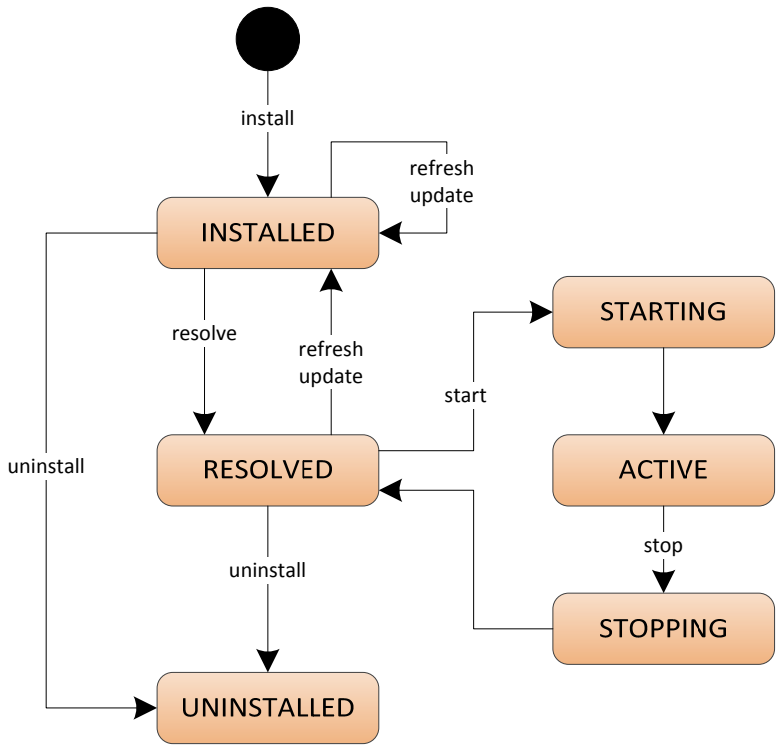


Abbildung 2.7: Lebenszyklus eines OSGi-Bundles (entnommen aus [WHKL08])

des OSGi-Frameworks gibt es jedoch keine Garantie, welches OSGi-Bundle in einem solchen Fall aufgelöst werden würde. Wurden alle Abhängigkeiten gefunden und aufgelöst, wird das OSGi-Bundle als *resolved* markiert.
Im Anschluss kann das OSGi-Bundle gestartet werden. Das Starten eines OSGi-Bundles führt dazu, dass vom OSGi-Framework in einer, in der Manifest-Datei spezifizierten, Activator-Klasse eine bestimmte Methode aufgerufen wird. Die Kontrolle des Programmflusses für dieses OSGi-Bundle liegt ab diesem Zeitpunkt beim OSGi-Bundle selbst. Das OSGi-Bundle wird daher als *active* gekennzeichnet.
Das OSGi-Framework hat jedoch jederzeit die Möglichkeit, das OSGi-Bundle zu stoppen (*stop*). Nach dem Aufruf einer bestimmten *stop*-Methode in der Activator-Klasse, werden sämtliche Threads beendet und Ressourcen zerstört, die vom OSGi-Bundle verwendet wurden.

Durch das Hinzufügen und Entfernen von OSGi-Bundles in bzw. von einem OSGi-Framework verändert das OSGi-Framework seinen Funktionsumfang. Betrachtet man das OSGi-Framework als SPL, wird die Variabilität dieser SPL über die aktivierten bzw. deaktivierten OSGi-Bundles definiert. Im weiteren Verlauf dieser Arbeit werden Algorithmen und Konzepte vorgestellt, die aus einer Menge von OSGi-Bundles, Features und ein dazugehöriges Feature-Modell extrahieren.

3. Status quo & Anforderungen

In diesem Kapitel wird zunächst das praktische Einsatzgebiet der in dieser Arbeit entwickelten Algorithmen und Konzepte, sowie die SPL, die später zur Evaluierung dienen wird, erläutert. Anschließend werden anhand dieser SPL Stakeholder und deren Anforderungen ermittelt, die es bei der Distribution der SPL zu erfüllen gilt.

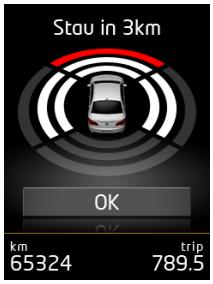
3.1 Status quo

In diesem Abschnitt wird vorgestellt, in welchem praktischen Einsatzgebiet die Distribution von SPLs analysiert und weiterentwickelt werden soll. Die Problemstellung dieser Arbeit ergab sich aus der Entwicklung neuer Bedienkonzepte in der Konzernforschung der Volkswagen AG.

3.1.1 Bedienkonzepte im Fahrzeug

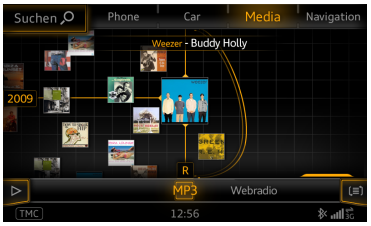
Ein Fahrzeug bietet heutzutage eine Vielzahl an Systemen, die verschiedenste Funktionen übernehmen. Viele Systeme arbeiten in der Regel ohne, dass der Fahrer einen Einfluss darauf hat, wie sie funktionieren. In vielen Fällen entspricht das auch dem Wunsch des Fahrers. Kein Fahrer möchte bspw. bestimmen müssen, wie viel Kraftstoff bei der nächsten Motorumdrehung in die Brennkammern des Motors eingespritzt wird. Eine solche Entscheidung überlässt der Autofahrer besser dem Motorsteuergerät seines Fahrzeugs. Es gibt aber auch Systeme im Fahrzeug, die der Fahrer überwachen oder sogar manipulieren möchte. Für alle diese Systeme bedarf es eines Konzepts, wie die Schnittstelle zwischen dem Fahrer und dem Fahrzeug gestaltet sein soll. Die Schnittstelle zwischen Mensch und Fahrzeug wird als *Human Machine Interface (HMI)* bezeichnet.[1]
Die für die Erforschung neuer Bedienkonzepte zuständige Abteilung in der Konzernforschung der Volkswagen AG (*Bedienkonzepte & Fahrer*) unterteilt die möglichen Systeme, mit denen ein Fahrer während der Benutzung des Fahrzeugs interagieren kann, in drei Kategorien:

[1] Die Bezeichnung *HMI* wird für die Schnittstelle zwischen Mensch und sämtlichen Maschinen verwendet. Maschinen können dabei neben Fahrzeugen auch Computer, Mobiltelefone, Fertigungsanlagen, Ticketautomaten, etc. sein.



(a) Ein HMI-Konzept zur Warnung des Fahrers vor einem Stauende (Quelle: intern)

(b) Ein HMI-Konzept, um den Fahrer über den aktuellen Zustand seiner Hybrid-Komponenten zu informieren (Quelle: intern)



(c) Ein kartenbasiertes Bedienkonzept, das Nutzern beim Durchstöbern ihrer Medienbibliothek hilft (Quelle: [MOW10])

Abbildung 3.1: Bedienkonzepte, die dem Fahrer dabei helfen neue Fahrzeugfunktionen und deren Zustand zu verstehen

1. *Assistenz- und Automatisierungssysteme:*
   In diese Kategorie sind Systeme einzuordnen, die den Fahrer bei der Quer- und Längsführung des Fahrzeugs unterstützen. In einem einfachen Fall kann ein HMI-Konzept eine blinkende LED im Kombinationsinstrument (Bezeichnung für die Instrumententafel hinter dem Lenkrad) sein, die den Fahrer darauf hinweist, dass das Fahrzeug aktuell durch ein eingreifendes ESP-System (Elektronisches Stabilitätsprogramm) in der Fahrspur gehalten wird. Ein komplexeres System könnte z.B. das in Abbildung 3.1(a) dargestellte Stauwarnsystem sein.

2. *Antriebssysteme:*
   Derzeit in aller Munde sind alternative Antriebskonzepte für Fahrzeuge [Bun11]. Neue Technologien machen es erforderlich, dass dem Nutzer zusätzliche Informationen zum aktuellen Fahrzeugzustand bereitgestellt werden. Exemplarisch seien hier *Plug-In-Hybrid-Konzepte* (Fahrzeuge, die mit haushaltsüblichem Wechselstrom geladen werden können) wie z.B. der Golf *twinDRIVE*[2] genannt. Hybrid-Konzepte machen es notwendig, dass dem Fahrer z.B. vermittelt wird, über welchen Motor sein Fahrzeug aktuell angetrieben wird oder wie der aktuelle Ladezustand der Batterie des Fahrzeugs zu bewerten ist. Dazu könnte bspw. ein Konzept, wie das in Abbildung 3.1(b) gezeigte, dienen. Es zeigt dem Fahrer im aktuellen Zustand, dass das Fahrzeug über den Elektromotor angetrieben wird und dass die Batterie zu 95% aufgeladen ist.

3. *Infotainmentsysteme:*
   Die dritte Kategorie, in die eine Fahrzeugfunktion im Sinne der HMI-Konzepte eingestuft werden kann, ist die Kategorie der Infotainmentdienste eines Fahrzeugs. Das in Abbildung 3.1(c) gezeigte, kartenbasierte Bedienkonzept [MOW10], hilft Nutzern bspw. besser dabei, große Medienbibliotheken zu durchstöbern als es ein klassisches, listenbasiertes System es könnte.

[2] https://www.bmu.de/pressemitteilungen/aktuelle_pressemitteilungen/pm/print/47536.php

3.1.2 Bedienkonzepte als Forschungsfeld

Die bereits angesprochene Konzernforschung der Volkswagen AG beschäftigt sich mit den erläuterten Bedienkonzepten eines Fahrzeuges aus Forschungsperspektive. Forschung bedeutet in diesem Zusammenhang, dass entwickelte Konzepte nicht zwingend in Serie produziert werden. Forschung bedeutet auch, dass entwickelte Konzepte nicht im nächsten, unter Umständen im übernächsten, vielleicht aber auch erst im über-übernächsten Produktzyklus zum Einsatz kommen. Daher ist es notwendig, dass Bedienkonzepte entwickelt werden, ohne zu stark von der technischen Machbarkeit beeinflusst zu werden. Denn wenn sich die technische Machbarkeit ändert (bspw. durch neue Technologien), ist es von Vorteil, das Konzept bereits ausgearbeitet zu haben.
Um zu entscheiden, welche entwickelten Konzepte weiter in Richtung Serienentwicklung gebracht werden sollen, muss ein Konzept erlebbar werden. Erlebbare Konzepte helfen besser dabei, die Idee dahinter zu verstehen und Konzepte bereits rechtzeitig durch Nutzerstudien zu evaluieren. Erlebbar wird ein Bedienkonzept aber nur dann, wenn die dafür notwendige Software implementiert wird. Da man sich bei dieser Entwicklung jedoch nicht von der technischen Machbarkeit einschränken lassen möchte, werden oft sog. Sitzkisten[3] oder sogar komplette Fahrzeuge mit prototypischer Hard- und Software ausgestattet. Das bedeutet in erster Linie, dass in solchen Umbauten vollständige Desktoprechner integriert werden, da diese über eine deutliche höhere Leistungsfähigkeit als eingebettete Systeme im Fahrzeug verfügen. Aufwendige Berechnungen lassen sich damit deutlich schneller durchführen. Aufgrund der rasanten, technologischen Entwicklung [Moo65] kann davon ausgegangen werden, dass die Leistungsfähigkeit eines heutigen Desktoprechners in einigen Jahren mit deutlich weniger Kostenaufwand, sowie deutlich weniger Platz- und Energieverbrauch in ein Fahrzeug integriert werden kann.
Nicht nur die Hardware, sondern auch die Software für neue Bedienkonzepte findet nicht zwingend mit Mitteln der Serienentwicklung statt. Um dennoch den Grad der Wiederverwendung von Quelltext zu erhöhen, wurde im Jahr 2009 der *HMIServer* entwickelt.

3.1.3 Der HMIServer

Der HMIServer stellt neuen Bedienkonzepten, deren Benutzeroberflächen aktuell überwiegend mithilfe von Flash oder HTML5 implementiert werden, grundlegende Fahrzeugfunktionen zur Verfügung. Unter solche grundlegenden Funktionen fallen bspw. der Zugriff auf die Daten von Abstandssensoren, auf Informationen wie Motordrehzahl und Geschwindigkeit des Fahrzeugs, oder die Möglichkeit den im Fahrzeug verbauten Radio-Tuner zu steuern, um die Frequenz des Radios zu ändern. Im Laufe der letzten Jahre kamen weitere Funktionen hinzu, die aufgrund der modularen Struktur des HMIServers ohne größeren Aufwand integriert werden konnten. Die Modularität des HMIServers beruht auf der Tatsache, dass es sich beim HMIServer um ein *OSGi-Framework* (siehe Abschnitt 2.4.2) handelt, das es ermöglicht die Funktionalität des HMIServers nachträglich durch OSGi-Bundles (in der Literatur auch als *Plug-ins* bezeichnet [WHKL08, MVA10]) zu erweitern.

[3] ein Fahrzeugcockpit ohne den Rest eines Fahrzeugs

Das folgende Beispiel zeigt, wie der HMIServer die Entwicklung und Präsentation eines neuen Bedienkonzeptes unterstützen kann: Es soll ein neues Bedienkonzept implementiert werden, das die Medienbibliothek des Nutzers auf eine neue Art und Weise darstellen soll (vgl. Abbildung 3.1(c)). Dazu benötigt die Implementierung dieses Konzepts Metadaten der Musikdateien, um bspw. Informationen wie Künstler oder Albumtitel anzuzeigen. Im HMIServer gibt es verschiedene OSGi-Bundles, die bei der ServiceRegistry einen *MediaRenderer*-Service registriert haben. Ein MediaRenderer stellt Metadaten aus einem Musikbestand zur Verfügung und bietet die Möglichkeit Songs aus diesem Bestand abzuspielen. Der Implementierung des neuen Bedienkonzepts ist es dann möglich von der ServiceRegistry Referenzen auf Objekte ermitteln zu lassen, die das MediaRenderer-Interface implementieren. Das Ergebnis ist eine Liste von Objektreferenzen, da sich mehrere OSGi-Bundles bei der ServiceRegistry als MediaRenderer registriert haben. Im Anschluss kann die Implementierung des Bedienkonzepts die Metadaten sämtlicher Musikstücke aus den verschiedenen MediaRenderern zusammentragen und entsprechend anzeigen.

3.1.4 Distribution des HMIServers

Wie Abbildung 3.1 bereits zeigte, werden neue Bedienkonzepte für verschiedene Projekte eingesetzt. All diese Projekte setzen auf den HMIServer, wenn es darum geht, auf Fahrzeug- oder Infotainmentdaten zuzugreifen. Dabei ist jedoch nur in sehr wenigen Fällen notwendig, dass eine Benutzeroberfläche auf alle Fahrzeug- und Infotainmentdaten zugreifen muss. Vielmehr ist es der Fall, dass gezielt ein neues Konzept vorgestellt und daher auch nur eine eingeschränkte Menge an Daten vom HMIServer abgefragt werden soll. Daher ist es notwendig, dass eine spezifische HMIServer-Installation mit genau den OSGi-Bundles ausgestattet wird, die für die erfolgreiche Absolvierung anfallender Aufgaben erforderlich sind. Einen HMIServer immer mit allen existierenden Bundles auszuliefern kann aus mehreren Gründen nicht empfohlen werden. Die drei wichtigsten dieser Gründen sind:

1. *Alternative OSGi-Bundles*
   Der wichtigste Grund, der gegen die Verwendung aller OSGi-Bundles zum gleichen Zeitpunkt in einem HMIServer spricht ist, dass für ein durch eine Schnittstelle definierten OSGi-Service mehrere OSGi-Bundles existieren, die die gleiche Schnittstelle implementieren und damit eine gleiche oder zumindest ähnliche Funktionalität bieten. So existieren bspw. OSGi-Bundles, die eine gewisse Testfunktionalität zur Verfügung stellen. Das Entwickeln einer Benutzeroberfläche ist oft komfortabler, wenn mit solchen *Testservices* gearbeitet wird. So wird bspw. für die, via GPS ermittelte Position des Fahrzeugs, während der Entwicklung der Benutzeroberfläche, ein Testservices verwendet, der ein fahrendes Fahrzeug simuliert.
   Wird der HMIServer im zu präsentierenden Fahrzeug installiert, sollen keine Testservices, sondern die Services verwendet werden, die realen Fahrzeugdaten (z.B. die reale GPS-Position) zur Verfügung stellen. Daher ist es notwendig bestimmte Gruppen von OSGi-Bundles an- und abschalten zu können.

2. *Performanz*
   Ein weiterer Grund nicht alle OSGi-Bundles gleichzeitig in einem HMIServer

zu verwenden, liefert die Tatsache, dass ein HMIServer mit allen OSGi-Bundles eine geringere Performanz aufweist, als ein Server mit einer eingeschränkten Anzahl an OSGi-Bundles. Die Gründe dafür liegen unter anderem darin, dass einige OSGi-Bundles wichtige Logging-Ausgaben erzeugen, oder wiederholt Festplattenzugriffe ausführen. Trotz sorgfältiger Optimierung, können solche Faktoren in der Summe dazu führen, dass das unter zu hoher Last betrieben wird, was sich dann wiederum im Erlebnis der Benutzeroberfläche negativ auswirken kann. Ferner benötigt ein HMIServer mit vielen Bundles beim Starten mehr Zeit um alle OSGi-Bundles zu laden und zu initialisieren. Untersuchungen im Rahmen dieser Arbeit haben ergeben, dass ein HMIServer mit sehr vielen OSGi-Bundles durchschnittlich doppelt so lang benötigt um zu starten, wie ein HMIServer mit nur einigen, wenigen OSGi-Bundles.

3. *Größe des HMIServers*
   Würde für jedes Projekt eines prototypischen Bedienkonzeptes ein vollständiger HMIServer – ein HMIServer mit allen verfügbaren OSGi-Bundles – installiert werden, so wäre dieser unverhältnismäßig groß. Ein solcher HMIServer würde weit über einem Gigabyte liegen. Beschränkt man die Funktionalität des HMI-Servers, durch Auswahl der entsprechenden OSGi-Bundles, ist ein HMIServer durchschnittlich 500MB groß.
   Da der HMIServer in der Regel auf einem gewöhnlichem Desktoprechner läuft, scheint es auf dem ersten Blick nicht von erhöhter Wichtigkeit auf die Größe des Servers zu achten. Ein großer HMIServer ist jedoch wesentlich aufwendiger während der Installation oder der Wartung. So ist es bspw. bei der Installation des HMIServers von Vorteil, wenn statt einem Gigabyte, nur 500MB von einem USB-Stick oder einem Netzlaufwerk kopiert werden müssen. Auch kommt es vor, dass vor wichtigen Präsentationen Kopien vom Server im jeweiligen Projekt erstellt werden, um bei möglichen Problemen unkompliziert zu einer, als funktionierend bekannten, Version zurück springen zu können. Es empfiehlt sich daher den HMIServer möglichst klein zu halten.

Aus den genannten Gründen ist es erstrebenswert, die Anzahl der OSGi-Bundles im HMIServer auf die notwendigen zu beschränken. Es muss daher einen Prozess geben, der einen konkreten HMIServer erstellt, der nur genau die OSGi-Bundles enthält, die in einem bestimmten Projekt benötigt werden.
Dieser Prozess existiert aktuell zwar bereits, wird jedoch von allen beteiligten Softwareentwicklern als unnötig kompliziert und sehr unkomfortabel empfunden. Der Prozess untergliedert sich in die zwei folgenden Teilschritte:

1. *Features erstellen*:
   OSGi-Bundles, die zu einem Feature gehören, werden von Softwareentwicklern in ZIP-Dateien gebündelt. Diese ZIP-Dateien können jeweils mehrere Bundles enthalten, die alle zur Erfüllung eines bestimmten Features notwendig sind. Wird ein OSGi-Bundle für mehrere Features benötigt, existieren Kopien dieses Bundles in mehreren ZIP-Dateien.
   Bundles, die zur Ausführung aller Features benötigt werden, werden in den so genannten *Basis-HMIServer* übernommen. Der Basis-HMIServer ist der Kern des HMIServers und beinhaltet neben den von alles Features benötigten

OSGi-Bundles auch OSGi-Bundles, die zur Ausführung des zugrundeliegenden OSGi-Frameworks dienen. Die für die Ausführung des Basis-HMIServers notwendigen Dateien werden ebenfalls in einer ZIP-Datei gebündelt. Das Ergebnis dieses ersten Schrittes ist in Abbildung 3.2 dargestellt.
Die Features (repräsentiert durch die jeweilige ZIP-Datei) unterliegen einer strengen Versionierung. Das bedeutet, dass Änderungen an Bundles, automatisch zu einer höheren Version des Features führen, dessen Teil das veränderte OSGi-Bundle ist.

2. *HMIServer distribuieren:*
Soll der HMIServer in einem Fahrzeug installiert werden, muss zunächst in intensiven Gesprächen zwischen Entwicklern (Softwareentwickler des HMIServers) und Projektmitarbeitern (Mitarbeiter aus dem Projekt, in dem der HMIServer eingesetzt werden soll) geklärt werden, welche Aufgaben der HMIServer im Fahrzeug erfüllen soll. Aus den zu erfüllenden Aufgaben müssen dann vom Entwickler die entsprechenden Features abgeleitet werden.
Im Anschluss kann der Entwickler dann zunächst den Basis-HMIServer aus der ZIP-Datei entpacken, um daraufhin die notwendigen Feature-ZIP-Dateien im gleichen Verzeichnis zu entpacken (die Ordnerstruktur der Feature-ZIP-Dateien ist so angelegt, dass die in der ZIP-Datei befindlichen Bundles automatisch am richtigen Pfad im Basis-HMIServer entpackt werden, wenn die Feature-ZIP-Datei im gleichen Ordner wie der Basis-HMIServer entpackt werden). Abbildung 3.3 zeigt die Ordnerstruktur der Feature-ZIP-Dateien und verdeutlicht, was beim Entpacken von Feature-1.zip geschehen würde: das OSGi-Bundle *Bundle-A* würde in zwei verschiedenen Versionen im HMIServer existieren. Nach dem Entpacken muss daher manuell vom Entwickler geprüft werden, ob die ältere Version bedenkenlos entfernt werden kann. Die ältere Version kann bspw. nicht ohne Weiteres entfernt werden, wenn in die neuere Version nicht abwärtskompatibel implementiert wurde. Dies ist z.B. dann der Fall, wenn Methoden (die als Schnittstelle für andere OSGi-Bundles dienen) entfernt werden oder ihre Signatur verändert wird, anstatt sie zu überladen.
Gleiche Bundles der gleichen Version haben immer den gleichen Dateinamen, was beim Entpacken dazu führt, dass sie einfach ersetzt werden können, wenn sie bereits im HMIServer vorhanden sind. Beim Entpacken der Feature-ZIP-Dateien ist jedoch darauf zu achten, dass gleiche OSGi-Bundles nicht in unterschiedlicher Version im finalen HMIServer vorhanden sind. Das kann z.B. dann passieren, wenn vergessen wurde eine neue Version eines OSGi-Bundles in einer Feature-ZIP-Datei zu aktualisieren. Beim Entpacken passiert es dann, dass beide Versionen des gleichen OSGi-Bundles nebeneinander im HMIServer liegen. Dadurch kann ein Systemverhalten verursacht werden, dass in vorherigen Tests vom jeweiligen Entwickler nicht berücksichtigt wurde.

Sowohl die Distribution des HMIServers (Entpacken der verschiedenen ZIP-Dateien), als auch die dafür notwendige Vorbereitungen (Erstellen der ZIP-Dateien, Gespräche mit Projektmitarbeitern), sind manuelle Aufgaben ohne jegliche Ansätze der Automatisierung. Dieser Mangel an Automatisierung führt zwangsläufig dazu, dass immer wieder Fehler bei der Distribution oder der Vorbereitung entstehen, die später durch zeitintensive Verbesserungsmaßnahmen wieder ausgeglichen werden müssen.

Mit dieser Arbeit wird ein Prozess vorgestellt, der sowohl die Distribution einer SPL als auch die dafür notwendigen Vorbereitungen beschreibt. Ferner wird für diesen Prozess ein geeignetes Werkzeug entworfen und entwickelt.

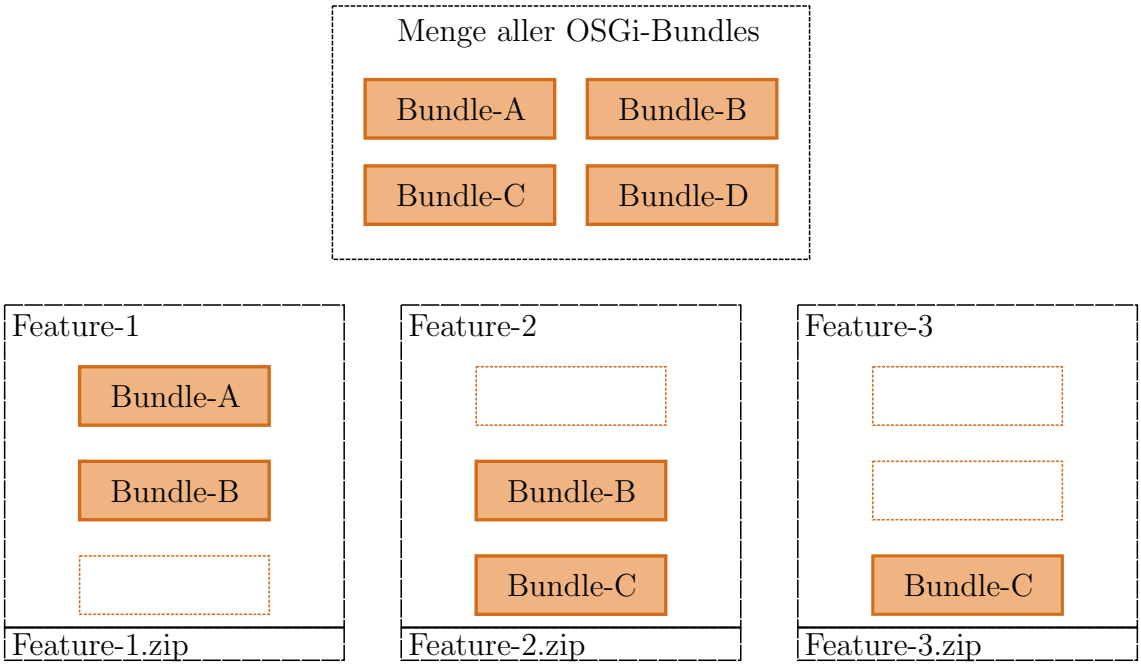


Abbildung 3.2: Abbildung zwischen OSGi-Bundles und durch ZIP-Dateien repräsentierte Features

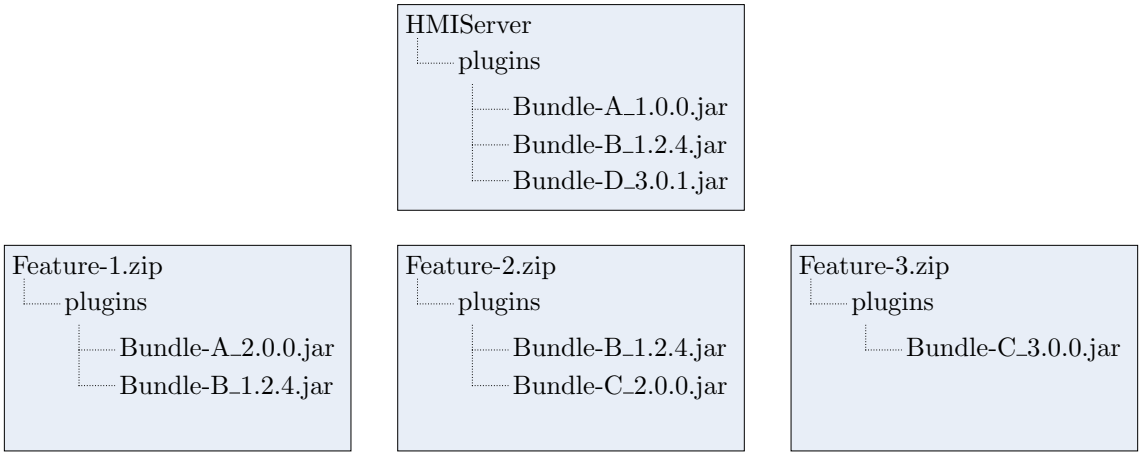


Abbildung 3.3: Aufbau der Feature-ZIP-Dateien des HMIServers

3.2 Stakeholderanalyse

Nach Diskussion der aktuellen Situation (vgl. Abschnitt 3.1), kann gemäß *Nuseibeh* und *Easterbrook* [NE00] mit der *Stakeholderanalyse* fortgefahren werden. Stakeholder sind, im betriebswirtschaftlichen Sinne, Personengruppen, die von den unternehmerischen Tätigkeiten gegenwärtig oder in Zukunft direkt oder indirekt betroffen sind [WD05]. Die im Sinne dieser Arbeit *unternehmerische Tätigkeit* kann dabei als Distributionsprozess des HMIServers verstanden werden. Zu beleuchten sind daher Personengruppen, die direkt oder indirekt, mit der Distribution des HMIServers in Verbindung stehen. Nach der Ermittlung der Stakeholder werden Anforderungen ermittelt, welche die jeweiligen Stakeholder an den Distributionsprozess stellen. Aus den ermittelten Anforderungen wird dann im Anschluss der gewünschte *Soll-Zustand* des Distributionsprozesses erstellt.

Eine SPL kommt im Laufe ihrer Entwicklung und ihrem Einsatz mit einer Vielzahl an Stakeholdern in Kontakt, die alle verschiedene, teilweise überlappende, Anforderungen an die SPL oder ein konkretes Produkt aus dieser SPL stellen. Bei Betrachtung der Distribution einer Software lässt sich leicht feststellen, dass *Softwareentwickler* und *Software-Architekten* der SPL den chronologisch ersten Kontakt zur SPL haben. Während die Entwickler den Quelltext der SPL in einer bestimmten Programmiersprache verfassen, muss im Anschluss sichergestellt werden, dass kompilierter oder interpretierbarer Quelltext in Form von ausführbaren Dateien zum Kunden gelangt. Ein Kunde setzt eine Variante einer SPL ein, um ein bestimmtes Ziel zu erreichen. Ein solches Ziel ist es, im Falle der Volkswagen AG, bspw. ein neuartiges Bedienkonzept für das Infotainmentsystem eines Fahrzeuges präsentier- und erlebbar zu machen. Dieses Ziel zu erreichen ist zunächst die Aufgabe eines *Projektleiters*. Der Projektleiter tritt hier als Kunde der HMIServer SPL auf, da er den HMIServer nutzen möchte, um ein bestimmtes Bedienkonzept zu präsentieren. Allerdings hat ein Projektleiter i.d.R. keinen, oder nur sehr wenig, direkten Kontakt mit dem HMIServer. Die Betreuung des HMIServers nimmt meist ein *Projektmitarbeiter* vor, der im Auftrag des Projektleiters bemüht ist, die HMIServer SPL so einzusetzen, dass das übergeordnete Ziel (neues Bedienkonzept präsentieren) erfüllt werden kann. Das Aufgabenfeld des Projektmitarbeiters umfasst eine ordnungsgemäße Konfiguration des HMIServers, das Aktualisieren einzelner Komponenten (OSGi-Bundles) des HMIServers, sowie das Testen des Zusammenspiels zwischen Bedienoberfläche und HMIServer. Auch der Projektmitarbeiter kann daher als Kunde des HMIServers verstanden werden. Bei abteilungsinternen Projekten ist auch nicht auszuschließen, dass die Stakeholder Softwareentwickler und Projektmitarbeiter in einer Person vereint auftreten. Dies bringt den glücklichen Umstand mit sich, dass eine solche Person sowohl Kenntnis und Verständnis über den Aufbau der SPL HMIServer als auch über das, durch den Projektleiter gewünschte, Zielverhalten des Systems hat. Da diese Tatsache jedoch nicht ohne Beschränkung der Allgemeinheit gilt, können daraus im weiteren Verlauf keine Vorteile gewonnen werden.
Ein vierter Stakeholder sind *Zulieferer*. Zulieferer treten in der HMIServer SPL als Dritthersteller auf. Sie bereichern den HMIServer durch zusätzliche Module (OSGi-Bundles) um neue oder veränderte Funktionalitäten. Zulieferer arbeiten typischerweise in den Räumlichkeiten ihres jeweiligen Unternehmens und haben daher keinen direkten Zugriff auf dem im Fahrzeug installierten HMIServer. Zum Testen ihrer ent-

wickelten OSGi-Bundles benötigen sie daher eine Kopie des im Fahrzeug installierten HMIServers.

Die durchgeführte Analyse der Stakeholder zeigt auf, dass es bei der Distribution einer SPL im Wesentlichen zwei große Sichten zu betrachten gilt. Zum einen die Sicht der *Kunden*, die eine Variante einer SPL für sich nutzen möchten, und zum anderen die Sicht der *Entwickler* einer SPL. Diese beiden Sichten müssen im Distributionsprozess berücksichtigt werden, da die Distribution die Schnittstelle zwischen Kunde und Entwickler darstellt.
Die zuvor beschriebenen Stakeholder lassen sich diesen beiden Sichten wie folgt zuordnen: trotz der direkte Kontakt mit dem HMIServer verschieden ausfällt, sind Projektleiter und Projektmitarbeiter bei der Distribution des HMIServers, als Kunde zu verstehen. Die Softwareentwickler lassen sich selbstverständlich der Entwickler-Sicht der Distribution zuordnen. Ein wenig unschärfer hingegen sind die Zulieferer zu betrachten. Sie gehören gewissermaßen beiden Sichten an. Kunden sind sie, weil sie einen HMIServer installieren und konfigurieren müssen. Das tun sie aber nur, um Plug-ins für den HMIServer zu entwickeln. Daher sind Zulieferer als Kunden und als Entwickler in gleichem Maße zu berücksichtigen.

3.3 Stakeholderanforderungen

Zu den vier isolierten Stakeholdern konnten im Anschluss durch Befragungen, Anforderungen ermittelt werden, die für die jeweiligen Stakeholder charakteristisch sind. Tabelle 3.1 fasst die ermittelten Anforderungen der Stakeholder vorab zusammen. Im Anschluss wird die Analyse der Anforderungen der einzelnen Stakeholder detailliert diskutiert.

Tabelle 3.1: Stakeholderanforderungen an die Distribution des HMIServers

Stakeholder	Zeitersparnis	verringerte Fehleranfälligkeit	erhöhte Transparenz	verringerte Komplexität	erhöhte Übertragbarkeit
Softwareentwickler	×	×	—	—	—
Projektleiter	×	—	—	—	—
Projektmitarbeiter	×	—	×	×	—
Zulieferer	×	—	—	×	×

Anforderungen der Softwareentwickler

Für den Entwickler einer SPL steht die Zeitersparnis, die er durch eine automatisierte Distribution gewinnen würde, an erster Stelle. Eine Zeitersparnis ergibt sich für ihn durch die Tatsache, dass er nicht für jeden neuen Kunden inhaltlich verstehen muss, welche Ergebnisse der Kunde gern durch ein Produkt der SPL erzielt haben möchte.

Ohne eine automatisierte Distribution müssen Kunden aufwendig beraten werden, welche Features einer SPL notwendig sind, um die Anforderungen des Kunden zu erfüllen. Weitere Zeit lässt sich für den Softwareentwickler sparen, wenn der zu leistende Support für die von ihm entwickelten Features auf ein Minimum reduziert werden kann.
Den zweiten, wichtigen Punkt für den Entwickler einer SPL stellt die verringerte Fehleranfälligkeit dar. Bei der Distribution des HMIServer ist es derzeit bspw. üblich, dass für ein neues (oder verändertes) Feature, der Quelltext, die Dokumentation und die ausführbaren Bestandteile eines Features *einzeln* erstellt und in einer komprimierten Form (bspw. einer ZIP-Datei) gespeichert werden. Dieser Prozess wird von allen Entwicklern als aufwendig, mühselig und sehr fehleranfällig eingeschätzt.
Wird dem Entwickler bei der Erstellung (oder Änderung) eines Features Unterstützung geleistet, ließe sich die Anzahl der Fehler reduzieren und die Belastung für den Entwickler senken.

Anforderungen der Projektleiter

Für den Projektleiter ist der Faktor Zeitersparnis ebenfalls sehr entscheidend. Nicht, weil seine eigene Arbeit schneller erledigt werden könnte, sondern vielmehr, weil das Installieren und Konfigurieren eines HMIServers ein Vorgang sein sollte, der im optimalen Fall keine oder nur sehr wenige Mitarbeiterressourcen benötigt. Ist ein HMIServer schnell installiert und eingerichtet, kann sich das gesamte Projektteam wieder auf die wesentliche Aufgabe (das neue Bedienkonzept) konzentrieren.
Dieser Vorteil kann auch direkt auf andere SPLs adaptiert werden. Da eine Variante einer SPL immer eine bestimmte Aufgabe bei einem Kunden erfüllen soll, möchte der, für die Einführung der Software verantwortliche, Mitarbeiter möglichst wenig Zeit für die Installation und Konfiguration dieser Software einplanen müssen. Ein hoher Grad an Automatisierung ist daher für den Projektleiter wünschenswert.

Anforderungen der Projektmitarbeiter

Neben dem Softwareentwickler bietet der Projektmitarbeiter das höchste Potenzial zur Verbesserung des Status Quo. Dies zeigte sich in den durchgeführten Befragungen durch vergleichsweise hohe und präzise formulierte Anforderungen. Ist der Projektmitarbeiter in der Lage einen HMIServer zu installieren, ohne dabei Unterstützung des Softwareentwicklers anzufragen, spart es für ihn in erster Linie Zeit. So muss er keine Zeit damit verbringen, den HMIServer umständlich über das Entpacken bereits genannter ZIP-Dateien zu installieren. Noch viel mehr Zeit spart er allerdings damit ein, dass er dem Softwareentwickler, der ihm beim Installieren des HMIServers zur Seite stehen müsste, nicht erklären muss, welche Funktionalität er vom Server benötigt, damit der Softwareentwickler die korrekten OSGi-Bundles auswählen könnte. Kann der Projektmitarbeiter Features anstelle von OSGi-Bundles auswählen, ist er bei der Installation des HMIServers nicht länger auf die Unterstützung eines Softwareentwicklers angewiesen. Für den Projektmitarbeiter ist der HMIServer dadurch auch nicht länger ein Softwareartefakt, das er nur schwer zu verstehen mag, sondern vielmehr eine Software, die ihn dabei unterstützt sein Ziel zu erreichen. Dies bietet somit die Chance, die Akzeptanz des HMIServers zu erhöhen und schafft damit einen Grundstein für einen höheren Verbreitungsgrad.

Neben der Zeitersparnis und der verringerten Komplexität ergeben sich aus Sicht des Projektmitarbeiters noch weitere, nicht minder wichtige, Anforderungen. So verlangt er bspw. nach mehr Transparenz darüber, welche Features im HMIServer aktuell installiert sind, welche er noch installieren kann und für welche Features ein Update zur Verfügung steht. Für die Features, die durch ein Update verbessert werden können, möchte der Projektmitarbeiter sehen können, was sich im Vergleich zu seiner installierten Version geändert hat. Auch möchte er darüber informiert werden, welche anderen Features beim Update eines bestimmtes Features aktualisiert werden müssen. Dies ist z.B. dann der Fall, wenn ein bestimmtes Feature ein anderes, bereits installiertes, Feature in einer neueren, als der bereits installierten, Version benötigt.
Diese Anforderungen lassen sich nicht selbstredend auf jede SPL übertragen. Sicher gibt es auch Nutzer einer SPL, die keinerlei Einzelheiten über evtl. Updates erfahren möchten. Umso wichtiger ist es daher, dass Installations- und Aktualisierungsprozesse fehlerfrei und ohne zu großen Nutzeraufwand ablaufen können.

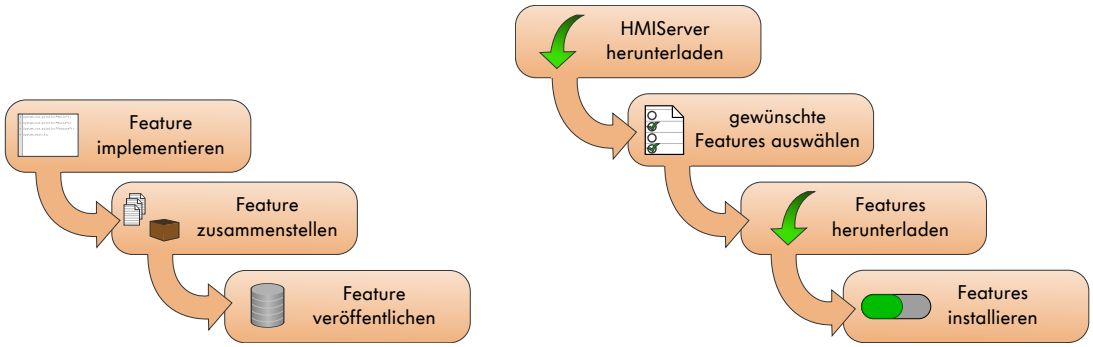
Anforderungen der Zulieferer

Da der Zulieferer, genau wie der Projektmitarbeiter, einen HMIServer installieren und konfigurieren muss, decken sich seine Anforderungen bzgl. Zeit und Komplexität mit denen des Projektmitarbeiters. Beim Zulieferer kommt, durch die Tatsache, dass er nicht mit dem HMISever im Fahrzeug arbeitet, sondern seine eigene, lokale Kopie verwendet, die Anforderung hinzu, dass er die Möglichkeit benötigt den HMIServer aus dem Fahrzeug zu kopieren. Dabei möchte er allerdings möglichst wenig binäre Daten kopieren. Vielmehr geht es ihm darum, eine bestimmte Auswahl von Features und eine bestimmte Konfiguration auf einen zweiten HMIServer zu übertragen.
Im allgemeinen Fall ist der Zulieferer ein Dritthersteller, der mit der SPL, für die er entwickelt, arbeiten muss, um seine entwickelten Komponenten zu testen.

3.4 Soll-Zustand des Distributionsprozesses der HMIServer SPL

Nachdem die Stakeholder und ihre jeweiligen Anforderungen ermittelt wurden, kann für den Distributionsprozess ein gewünschter Soll-Zustand erstellt werden. Die bereits erwähnten Sichten auf den Distributionsprozess werden auch hierbei wieder deutlich. So stellt sich die Frage, wie ein Entwickler (Softwareentwickler und Zulieferer) die von ihm entwickelten Features zur Verfügung stellen kann, während sich aus der anderen Sicht – der Kundensicht – die Frage stellt, wie ein Kunde (Projektmitarbeiter, Projektleiter und Zulieferer) die verfügbaren Features einsehen und installieren kann. Diese Fragen wurden in intensiven Einzelgesprächen mit den Stakeholdern diskutiert. In der Summe dieser Gespräche lies sich erkennen, welche Schwachstellen der aktuelle Distributionsprozess mit sich bringt und was von den verschiedenen Stakeholdern als besonders störend empfunden wird. Der entwickelte Soll-Zustand wurde nach dessen Erstellung in einem Gruppengespräch mit Vertretern aller Stakeholder präsentiert und verfeinert. Das Ergebnis dieser Verfeinerung wird in diesem Abschnitt vorgestellt.

Abbildung 3.4 veranschaulicht den gewünschten Soll-Zustand aus Entwickler- und Kundensicht. In den beiden folgenden Abschnitten werden die einzelnen Schritte der jeweiligen Sicht näher erläutert.



(a) Gewünschter Soll-Zustand für die Distribution eines Features aus Entwicklersicht

(b) Gewünschter Soll-Zustand für die Distribution eines Features aus Kundensicht

Abbildung 3.4: Gewünschte Soll-Zustände aus Entwickler- und Kundensicht

3.4.1 Soll-Zustand für den Entwickler

Der gemeinsam mit den Stakeholdern entwickelte Distributionsprozess ist in Abbildung 3.4(a) aus Entwicklersicht dargestellt.

1. *Feature implementieren* — im Vergleich zum Ist-Zustand (vgl. Abschnitt 3.1.4) soll sich für den Entwickler hier wenig ändern. Er sollte seine Implementierungsartefakte wie bisher auch implementieren können. Ergebnis dieser Implementierung muss sein, dass alle implementierten OSGi-Bundles (zumindest zu diesem Zeitpunkt) vom Entwickler durch Tests als fehlerfrei befunden wurden. Eine Menge von OSGi-Bundles wird dann vom Entwickler als Feature definiert. Damit sich der Kunde später selbst über die jeweiligen Features informieren kann, wird es nach wie vor erforderlich sein, dass der Entwickler sein Feature ausreichend dokumentiert. Dabei geht es nicht um Dokumentation auf Quelltextebene, sondern um eine Dokumentation, die es auch Nicht-Informatikern ermöglicht zu verstehen, welche Funktionalität ein Feature erfüllt.

2. *Feature zusammenstellen* — das Zusammenstellen eines Features bedeutet für den Entwickler bisher sehr viel Aufwand und eine hohe Komplexität, da er darauf achten muss, dass von ihm alle OSGi-Bundles zur Feature-ZIP-Datei hinzugefügt werden, die für die Ausführung der von ihm entwickelten OSGi-Bundles notwendig sind. Dieser Prozess ist Ursprung vieler Fehler. Das Zusammenstellen der Features soll daher automatisiert werden. Dem Entwickler soll die Möglichkeit gegeben werden, sich einzig und allein auf die von ihm entwickelten OSGi-Bundles zu konzentrieren. Abhängigkeiten zu anderen OSGi-Bundles sollen automatisch erkannt und aufgelöst werden.

3. *Feature veröffentlichen* — das Veröffentlichen eines Features gestaltet sich bisher als zeitintensiver Prozess, in dem große und damit unhandliche ZIP-Dateien auf Netzlaufwerke kopiert werden mussten. Es soll daher ein Konzept entwickelt werden, dass es dem Entwickler ermöglicht, die von ihm entwickelten Features an einer zentralen Stelle zur Verfügung zu stellen, die dabei sowohl für ihn, als auch für den Kunden schnell und einfach zu erreichen ist. Den Entwickler entlastet ein optimierter Veröffentlichungsprozess von Features, wenn dieser

Prozess einen hohen Grad an Automatisierung mit sich bringt. Im Idealfall löst der Entwickler durch eine einzige Interaktion eine Kette von Aktionen aus, die das Feature vollautomatisch veröffentlicht.

Die vom Softwareentwickler geforderte Zeitersparnis entsteht beim beschriebenen Prozess dadurch, dass er im Idealfall nach der Veröffentlichung lediglich mit der Behebung von Fehlern, die ihm durch Kunden mitgeteilt wurden, beschäftigt ist. Aktuell sind viele Softwareentwickler dazu gezwungen ihre Features individuell in mehreren Projekten zu betreuen. Dieser Aufwand würde sich durch einen standardisierten Distributionsprozess minimieren lassen.
Die ferner vom Softwareentwickler geforderte Anforderung der verringerten Fehleranfälligkeit, kann im dargestellten Prozess durch eine automatisierte Feature-Zusammenstellung gewährleistet werden.

3.4.2 Soll-Zustand für den Kunden

Wie eine Distribution aus Kundensicht gestaltet sein sollte, zeigt Abbildung 3.4(b). Für diesen Prozess ist es als Ausgangspunkt erforderlich, dass der Nutzer weiß, welche Funktionalität der HMIServer für ihn übernehmen soll. Da nicht angenommen werden kann, dass jeder potenzielle Nutzer weiß, welche Funktionalität er vom HMIServer benötigt, muss ein gewisses Maß an Beratungsmöglichkeiten dem Kunden gegenüber nach wie vor gewährleistet werden können. In vielen Fällen jedoch wissen Kunden (z.B. durch Erfahrung aus vorherigen Projekten) welche Funktionalität sie vom HMIServer benötigen.

1. *HMIServer herunterladen* — der Kunde soll die Möglichkeit bekommen, die Basis einer jeden HMIServer-Variante von einem Server herunterzuladen. Bisher ist dies nicht möglich, da der HMIServer immer von einem Softwareentwickler installiert und eingerichtet werden muss. Durch die gewünschte Entkopplung von Entwickler und Kunde, muss der Kunde daher die Möglichkeit bekommen den HMIServer selbstständig zu beziehen.

2. *Gewünschte Features auswählen* — der Kunde soll die Möglichkeit bekommen, Features selbstständig und seinen Bedürfnissen entsprechend, auswählen zu können. Dafür wird es notwendig werden, dem Kunden durch eine umfangreiche und reichhaltige Dokumentation die Funktionalität der einzelnen Features zu erläutern. Hat der Kunde sich für eine Menge von Features entschieden, müssen Abhängigkeiten zwischen Features automatisiert für den Nutzer aufgelöst werden. Nur so wird es möglich, dass er einen HMIServer ohne einen Softwareentwickler installieren kann.

3. *Features herunterladen* — nachdem der Kunde die gewünschten Features ausgewählt hat, kann von einem Distributionssystem berechnet werden, welche Features benötigt werden, um die Ausführung der ausgewählten Features zu gewährleisten. Die dadurch bestimmten Features können im Anschluss vom Webserver automatisch heruntergeladen werden.

4. *Features installieren* — abschließend werden die heruntergeladenen Features im HMIServer installiert und die neu hinzugefügten OSGi-Bundles gestartet, damit der Kunde die gewünschte und installierte Funktionalität nutzen kann.

Die Anforderungen sämtlicher Kunden (Projektleiter, Projektmitarbeiter und Zulieferer) können in diesem Prozess wie folgt erfüllt werden: die Zeitersparnis ergibt sich durch den Wegfall der Beratung durch einen Softwareentwickler. Der Kunde muss dem Softwareentwickler nicht länger vermitteln, welche Funktionalität er vom HMIServer benötigt. Stattdessen lädt sich der Kunde den HMIServer selbstständig von einem Webserver herunter, wählt die gewünschten Features aus und lässt die gewählten Features automatisch herunter laden und installieren.
Eine erhöhte Transparenz sowie eine verringerte Komplexität ergibt sich durch die selbstständige Auswahl der zur Verfügung stehenden Features. Der Kunde hat die volle Kontrolle darüber welche Features installiert werden. Durch eine, in den Prozess eingebettete, Dokumentation der Features, kann er sofort erkennen, welche Features ihm beim Erreichen seines Ziels helfen werden.

3.5 Zusammenfassung

Der beschriebene Status quo des Distributionsprozesses des HMIServers ist für alle Beteiligten sehr zeitintensiv, für Softwareentwickler an vielen Stellen fehleranfällig und vor allem für den Kunden sehr komplex und schwer nachvollziehbar. Ein Werkzeug, dass den Kunden beim Installieren des HMIServers unterstützt und ihn Features nachinstallieren lässt, würde sowohl dem Kunden helfen, als auch die Softwareentwickler entlasten. Noch weiter entlasten würde die Softwareentwickler ein Werkzeug, dass basierend auf OSGi-Bundles ein Feature-Modell extrahiert, das dem Kunden dabei helfen kann, Features und Abhängigkeiten zwischen Features des HMIServers besser zu verstehen.Im folgenden Kapitel werden daher Konzepte und Algorithmen vorgestellt, die es ermöglichen teilautomatisiert Feature-Modelle aus Implementierungsartefakten zu extrahieren.

4. Extrahieren von Feature-Mappings und Feature-Modellen

In diesem Kapitel werden Konzepte und Algorithmen vorgestellt, die es ermöglichen, aus gegebenen Implementierungsartefakten automatisiert Features und eine Abbildung zwischen diesen Features und den Implementierungsartefakten (Feature-Mapping) zu extrahieren. Dieser Prozess wird im weiteren Verlauf dieser Arbeit als *FeatureMapGen* (**Feature**-**Map**ping **Gen**erator) bezeichnet. Im Anschluss wird ein automatisierter Prozess vorgestellt, der die extrahierten Features in einem Feature-Modell zusammenfasst und in Relation zueinander setzt. Dieser Prozess wird *FeatureModGen* (**Feature**-**Mod**el **Gen**erator) genannt.

4.1 Identifizieren von Features und Feature-Mappings

In diesem Abschnitt wird erläutert, wie FeatureMapGen aus einer gegebenen Menge an Implementierungsartefakten Features identifiziert und Feature-Mappings aus den Abhängigkeiten zwischen den Implementierungsartefakten extrahiert.

Motivation

Aus der in Kapitel 3 beschriebenen Anforderungsanalyse geht hervor, dass sich die Softwareentwickler bei der Neuentwicklung eines bestimmten Features lediglich auf die von ihnen implementierten OSGi-Bundles konzentrieren können sollen. Um dies zu gewährleisten, wird mit FeatureMapGen ein Konzept zur Verfügung gestellt, dass es ermöglicht Features und ein dazugehöriges Feature-Mapping auf Basis einer bestimmten Menge von OSGi-Bundles zu extrahieren.

4.1.1 Voraussetzungen für FeatureMapGen

Ausgehend von einer Menge A_{all}, die sämtliche Implementierungsartefakte (OSGi-Bundles) umfasst, die zur Ausführung einer SPL relevant sind, d.h. die zur korrekten Funktionsweise der SPL und all ihrer Features benötigt werden, soll FeatureMapGen Features und ein dazugehöriges Feature-Mapping extrahieren. Dafür werden alle Implementierungsartefakte zunächst auf ihre Abhängigkeiten hin untersucht. Eine Abhängigkeit zwischen zwei Implementierungsartefakten aus A_{all} wird dabei wie folgt definiert:

Definition 1. *Implementierungsartefakt A ist abhängig von Implementierungsartefakt B, wenn B Funktionalität zur Verfügung stellt, die von A benutzt wird und $A, B \in A_{all}$ gilt.*

Alle Implementierungsartefakte aus A_{all} werden hinsichtlich ihrer bereitgestellten Funktionalität untersucht. Ergebnis dieser Untersuchung ist eine Datenstruktur, die Abhängigkeiten zwischen den Artefakten derart abbildet, dass ein von Artefakt A abhängiges Artefakt B direkt (d.h. mit einer Komplexität von $\mathcal{O}(1)$) ermittelt werden kann. Diese Datenstruktur wird in der Arbeit *Feature-Objekt* genannt. Feature-Objekte ermöglichen es später, für ein Artefakt X, ohne einen komplexen Rechenvorgang, alle Artefakte zu ermitteln, die zur Ausführung von X notwendig sind.
Das dabei entstehende Ergebnis lässt sich auch als gerichteter, ungewichteter Graph interpretieren, bei dem die Knoten Implementierungsartefakte und die gerichteten Kanten Abhängigkeiten zwischen diesen Artefakten darstellen. Die Richtung einer solchen Kante zeigt dabei *von* einem Implementierungsartefakt A, *zu* einem anderen Implementierungsartefakt B, genau dann wenn A von B abhängig ist, d.h. B benötigt wird, um A ausführen zu können. Ein solcher Graph wird in dieser Arbeit *Abhängigkeitsgraph* genannt.
In einem OSGi-Framework entsprechen die einzelnen OSGi-Bundles den zuvor beschriebenen Implementierungsartefakten. Die Abhängigkeit zwischen zwei OSGi-Bundles definiert sich dabei wie folgt:

Definition 2. *OSGi-Bundle A ist abhängig von OSGi-Bundle B, wenn B das Package P exportiert, P von A importiert wird, die vom Import geforderte Version von P mit der exportierten Version von P übereinstimmt und $A, B \in A_{all} \wedge P \in P_{all}$ gilt.*

Ein Abhängigkeitsgraph, der sich aus der Analyse von zehn OSGi-Bundles ableiten lies, die aktuell im HMIServer verwendet werden, ist in Abbildung 4.1 dargestellt. Um eine möglichst hohe Übersichtlichkeit zu gewährleisten, wurden die Namen der OSGi-Bundles vereinfacht.
Entscheidend für diesen Beispielgraphen sind die folgenden Punkte:

a) das OSGi-Bundle B stellt lediglich Schnittstelleninformationen (Java-Interfaces) zur Verfügung, die von anderen OSGi-Bundles verwendet werden, um standardisiert Objekte auszutauschen; B allein verändert die Funktionsweise eines OSGi-Frameworks nicht

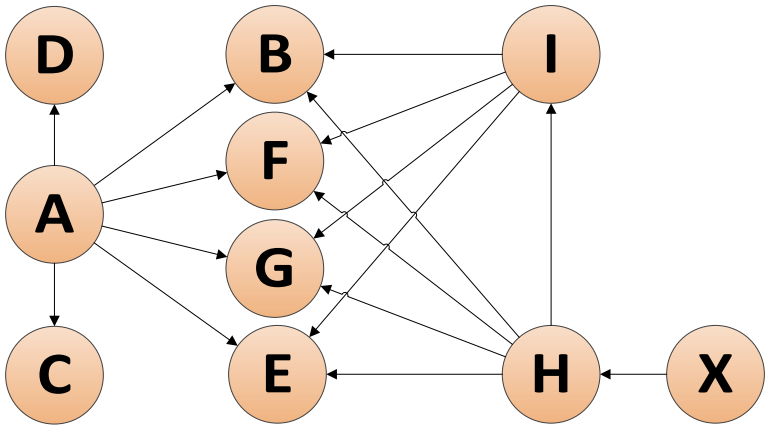


Abbildung 4.1: Exemplarischer Abhängigkeitsgraph zur Visualisierung von Abhängigkeiten zwischen Implementierungsartefakten

b) die OSGi-Bundles C, D und I bieten lediglich Klassen und Methoden an, die von anderen OSGi-Bundles benutzt werden können, um bspw. Mediendateien anzeigen oder abspielen zu können; C, D oder I allein verändern die Funktionsweise eines OSGi-Frameworks nicht

c) die OSGi-Bundles E, F und G stellen Methoden zum standardisierten Logging und andere OSGi-spezifische Kernfunktionalitäten zur Verfügung die von vielen OSGi-Bundles benutzt werden

d) bei den OSGi-Bundles A und H handelt es sich um OSGi-Bundles, die zum Abspielen von Mediendateien in einem OSGi-Framework dienen; ein Hinzufügen oder Entfernen von A oder H, verändert die Funktionsweise eines OSGi-Frameworks, da Mediendateien durch sie überhaupt oder aus verschiedenen Quellen abgespielt werden können

e) das OSGi-Bundle X erweitert das OSGi-Bundle H um die Möglichkeit Musik aus mehreren Speichermedien (USB, Smartphone, CD) parallel nutzen zu können.

Eine Diskussion unter Experten der Domäne hat ergeben, dass es als Feature verstanden wird, die OSGi-Bundles A, H und X einzeln oder in, den Abhängigkeitsgraphen respektierenden, Kombinationen zu einem OSGi-Framework hinzuzufügen oder zu entfernen. Das alleinige Hinzufügen oder Entfernen der anderen OSGi-Bundles (einzeln oder in beliebigen Kombinationen) wird dagegen nicht als sinnvolles Feature eingeschätzt. Ziel ist es daher, einen Algorithmus zu entwickeln, der auf Grund des Abhängigkeitsgraphen, automatisiert Features und ein dazugehöriges Feature-Mapping extrahiert, bei dem sich lediglich die OSGi-Bundles A, H und X als Feature an- und abschalten lassen.
Das ein solcher Algorithmus nur teil- und nicht vollautomatisiert arbeiten kann, wird bei der Betrachtung der OSGi-Bundles I, H und X deutlich: gelingt es, einen Algorithmus zu entwickeln, der H und X als an- und abschaltbar ermittelt, wird dieser Algorithmus sehr wahrscheinlich auch I als an- und abschaltbar ermitteln, da sich I und H bezüglich ihrer Knoteneigenschaften (Anzahl ein- und ausgehender Kanten) sehr ähnlich sind. Dem Abhängigkeitsgraphen unterliegt neben den offensichtlichen

Abhängigkeiten, auch eine gewisse *Semantik*, die sich algorithmisch nicht vollständig ermitteln lassen wird.
Ziel ist es daher, Konzepte und Algorithmen anzubieten, die einen Experten bei der Extraktion des Feature-Mappings unterstützen. Ferner erhebt der zu entwickelnde Algorithmus den Anspruch, auch auf SPLs angewendet werden zu können, die nicht als OSGi-Framework entwickelt wurden. OSGi-spezifische Informationen sollen daher bei der Entwicklung des Algorithmus nicht herangezogen werden. Statt von OSGi-Bundles wird im Folgenden daher nur noch von *Implementierungsartefakten* (kurz: Artefakte) gesprochen.

4.1.2 Funktionsweise von FeatureMapGen

Der in dieser Arbeit entwickelte Algorithmus *FeatureMapGen* erlaubt es, ausgehend von Implementierungsartefakten und Abhängigkeiten unter diesen Artefakten, Features und ein Feature-Mapping zu extrahieren. Grundgedanke ist dabei die Tatsache, dass ein Feature aus mehreren Implementierungsartefakten bestehen kann und das ein Implementierungsartefakt Bestandteil mehrerer Features sein kann.

```
1  parameters: A_all – set of all artifacts
2  featureMapGen(A_all) {
3      //init return value
4      features ← ∅
5
6      //detect feature candiate artifacts
7      A_init ← ∅
8      for each artifact ∈ A_all {
9          if(isFeatureCandidate(artifact))
10             A_init ← A_init ⋃ artifact
11     }
12
13     //create and fill feature for each feature candidate
14     for each c ∈ featureCandidates {
15         //do a deep search for feature candidate artifact
16         requiredArtifacts ← deepSearch(c)
17
18         //create new feature and
19         //add feature candidate and its transitive dependencies
20         feature ← c
21         feature ← feature ⋃ requiredArtifacts
22
23         //add the newly generated feature to the set of features
24         features ← features ⋃ feature
25     }
26
27     return features
28 }
```

Quelltext 4.1: FeatureMapGen Pseudocode

Der in Quelltext 4.1 dargestellte Pseudocode zeigt die prinzipielle Vorgehensweise von FeatureMapGen. In einem ersten Schritt werden die Implementierungsartefakte bestimmt, die eine Grundlage für die später extrahierten Features sind. Ob ein

Artefakt $x \in A_{all}$ ein Kandidat für ein Feature ist, wird aufgrund des Rückgabewertes der Methode `isFeatureCandidate(`x`)` entschieden. Im weiteren Verlauf dieser Arbeit werden verschiedene Implementierungsvarianten für `isFeatureCandidate(x)` vorgestellt. Für den Moment sei angenommen, dass eine Menge A_{init} generiert wird, die alle Artefakte enthält, die potenzielle Kandidaten für ein Feature sind. Aufgrund dieser Menge A_{init} kann für jeden Kandidaten ein neues Feature-Objekt erstellt werden, welches das Feature-Mapping für das entsprechende Feature repräsentiert. Ein Feature-Mapping für ein Feature F muss alle Implementierungsartefakte enthalten, die für den fehlerfreien Ablauf von F notwendig sind. Für ein gegebenes Artefakt X (den Kandidaten), sind dies alle anderen Artefakte, von denen X — auch transitiv — abhängig ist. Ausgehend vom Abhängigkeitsgraphen kann daher eine Tiefensuche vom Knoten X ausgehend durchgeführt werden (vgl. Zeile 16 in Quelltext 4.1). Alle bei dieser Tiefensuche besuchten Knoten sind notwendig um Artefakt X in der SPL ausführen zu können. Um ein fehlerfrei arbeitendes Feature-Mapping zu extrahieren werden daher X und alle besuchten Knoten (bzw. die dazugehörigen Artefakte) dem aus X entstandenem Feature-Objekt hinzugefügt. Das generierte Feature-Objekt enthält daher sowohl das Artefakt X, als auch alle transitiven Abhängigkeiten dieses Kandidaten.
Das folgende Beispiel verdeutlicht die Funktionsweise von FeatureMapGen: angenommen, bei der Suche nach Feature-Kandidaten (Zeile 7–11 in Quelltext 4.1, wurde A_{init} mit den OSGi-Bundles A, H und H befüllt. Die für diese drei OSGi-Bundles durchgeführte Tiefensuche ermittelt dann die in Tabelle 4.1 dargestellten Features.

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-H	—	•	—	—	•	•	•	•*	•	
Feature-X	—	•	—	—	•	•	•	•	•	•*

Tabelle 4.1: Exemplarisches, durch FeatureMapGen extrahiertes Feature-Mapping

Entscheidend für das Ergebnis von FeatureMapGen bleibt jedoch die Funktionsweise der Kandidatenbestimmung durch die Methode `isFeatureCandidate(x)`. Im Folgenden werden vier Varianten mit ihren jeweiligen Vor- und Nachteilen vorgestellt.

FeatureMapGen::All

Ein trivialer Ansatz zur Bestimmung der Feature-Kandidaten ist es, FeatureMapGen mit allen Implementierungsartefakten aus A_{all} zu initialisieren. Formal gilt bei dieser Variante $A_{init} = \{x \mid x \in A_{all}\}$ bzw. $A_{init} = A_{all}$. Die Methode `isFeatureCandidate(x)` wird in diesem Fall keinerlei Kriterien des Implementierungsartefakts überprüfen, sondern einfach jedes Artefakt als Kandidaten einstufen (siehe Quelltext 4.2).
Wird für jedes Artefakt untersucht, von welchen anderen Artefakten es abhängig ist und wird dies als Feature verstanden, wird als Ergebnis eine Menge an Features entstehen, deren Mächtigkeit gleich der Mächtigkeit der Menge aller Artefakte A_{all}

```
parameters: x – artifact which may be an artifact
isFeatureCandidate(x) {
    return true
}
```

Quelltext 4.2: FeatureMapGen::All Pseudocode

ist. Die dabei entstehenden Features sowie das dazugehörige Feature-Mapping ist in Tabelle 4.2 dargestellt.

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-B	—	•*	—	—	—	—	—	—	—	—
Feature-C	—	—	•*	—	—	—	—	—	—	—
Feature-D	—	—	—	•*	—	—	—	—	—	—
Feature-E	—	—	—	—	•*	—	—	—	—	—
Feature-F	—	—	—	—	—	•*	—	—	—	—
Feature-G	—	—	—	—	—	—	•*	—	—	—
Feature-H	—	•	—	—	•	•	•	•*	•	—
Feature-I	—	•	—	—	•	•	•	—	•*	—
Feature-X	—	•	—	—	•	•	•	•	•	•*

— dieses Implementierungsartefakt ist nicht Teil des Features
• dieses Implementierungsartefakt ist Teil des Features
* dieses Implementierungsartefakt ist als Feature-Kandidat für die Bildung des Features verantwortlich

Tabelle 4.2: Feature-Mapping für FeatureMapGen::All

Der Vorteil von FeatureMapGen::All liegt darin, dass eine komplexe Bestimmung initialer Implementierungsartefakte nicht stattfinden muss, da zur Initialisierung *alle* vorhanden Artefakte benutzt werden.
Der Nachteil von FeatureMapGen::All ist jedoch, dass Features extrahiert werden, die zwar lauffähig sind (d.h. deren Artefakte aufgelöst werden können), die den Funktionsumfang der SPL jedoch nicht erweitern würden. *Feature-B* beispielsweise, ist zwar lauffähig, da das Artefakt B keine anderen Artefakte benötigt, der Funktionsumfang der SPL wird durch dieses Feature jedoch nicht erweitert, da das Artefakt B in diesem Beispiel lediglich Schnittstelleninformationen zur Verfügung stellt, die von den Artefakten A, I und H benutzt werden. Das alleinige Hinzufügen dieser Schnittstellen wäre damit zwar nicht falsch, würde jedoch einen falschen Eindruck

von Variabilität vermitteln. Die erzeugte Variabilität wäre deutlich höher als die vom Experten gewünschte. Einen weiteren Nachteil liefert eine Betrachtung der Skalierbarkeit. Abschnitt A.1 zeigt, dass FeatureMapGen::All mit einer Komplexität von $\mathcal{O}(n^3)$ überaus schlecht für große Graphen skaliert. Es empfiehlt sich daher nicht, alle Implementierungsartefakte als Feature-Kandidaten einzustufen.

FeatureMapGen::NoIn

Die Nachteile von FeatureMapGen::All, der zu viel erzeugten Variabilität und der schlechten Skalierbarkeit, ließen sich reduzieren, wenn FeatureMapGen nur mit Artefakten initialisiert wird, die keine eingehenden Kanten aufweisen. Formal gilt dabei $A_{init} = \{x \mid x \in A_{all} \ \wedge \ \#_{in}(x) = 0\}$, wobei die Funktion $\#_{in}(x)$ die Anzahl eingehender Kanten für ein Artefakt X ermittelt. Die Methode `isFeatureCandidate(x)` wird in diesem Fall nur die Artefakte als Kandidat einstufen, die nicht von anderen Artefakten benötigt werden (siehe Quelltext 4.3). Knoten, die keine eingehenden Kanten aufweisen, werden von keinem anderen Knoten benötigt und legen daher die Vermutung nah, dass die Artefakte hinter diesen Knoten den *Beginn* einer Funktionalität darstellen. Die aus diesen Artefakten extrahierten Features, könnte daher Features sein, die auch ein Experte als Feature eingestuft hätte.

```
1 parameters: x – artifact which may be an artifact
2 isFeatureCandidate(x) {
3     return x.incomingDependencies.count == 0
4 }
```

Quelltext 4.3: FeatureMapGen::NoIn Pseudocode

Initialisiert mit Artefakten ohne eingehende Kanten, wird FeatureMapGen die in Tabelle 4.3 dargestellten Features und das dazugehörige Feature-Mapping extrahieren. Die Artefakte A und X sind die einzigen Artefakte, die keine eingehenden Kanten aufweisen. Diesen beiden Artefakte allein, werden daher als Feature-Kandidaten bestimmt. Da das Artefakt H eine eingehenden Kante vom Artefakt X aufweist, wird es nicht als Feature-Kandidat bestimmt und somit auch kein Feature ausgehend von Artefakt H extrahiert.
Die so extrahierten Features erfüllen daher auch unter Verwendung von FeatureMapGen::NoIn nicht das Maß an Variabilität, das zuvor durch Expertenbefragungen für diese ausgewählten Artefakte ermittelt wurden war.

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-X	—	•	—	—	•	•	•	•	•	•*

Tabelle 4.3: Feature-Mapping für FeatureMapGen::NoIn

Die Vorteile von FeatureMapGen::NoIn liegen darin, dass zum einen Variabilität vermieden wird, die von Experten als *unnötig* eingestuft wurde und zum anderen,

dass die Menge der Artefakte, für die eine Tiefensuche durchgeführt werden wird, stark eingeschränkt wird. Abschnitt A.2 zeigt, dass die Komplexität zwar weiterhin bei $\mathcal{O}(n^3)$ liegt, jedoch aufgrund der geringeren Anzahl an Tiefensuchen von einem geringen Aufwand ausgegangen werden kann.
Der Nachteil von FeatureMapGen::NoIn liegt darin, dass unter Umständen zu wenig Variabilität erzeugt wird, um alle *sinnvollen* Produkte aus der SPL ableiten zu können. Im vorliegenden Beispiel wäre es daher wünschenswert, wenn FeatureMapGen *Feature-A*, *Feature-H* und *Feature-X* extrahieren würde.

FeatureMapGen::MaxIn

Der zu wenig erzeugten Variabilität von FeatureMapGen::NoIn kann entgegengewirkt werden, indem als Feature-Kandidaten auch Implementierungsartefakte bestimmt werden, deren eingehende Kanten eine bestimmte Anzahl nicht überschreiten. Die ermittelten Feature-Kandidaten werden dabei durch folgende Menge beschrieben: $A^k_{init} = \{x \mid x \in A_{all} \ \wedge \ \#_{in}(x) \leq k\}, k \in \mathbb{N}$.
Da diese Variante von FeatureMapGen einen Parameter benötigt, der das Ergebnis (die extrahierten Features) verändert, muss die Aussteuerung dieses Parameters von einer Person überwacht werden, die mit der gesamten SPL vertraut ist und Kenntnis darüber hat, welche Features benötigt werden und welche Features unter keinen Umständen erzeugt werden sollten. Der Parameter k dient demjenigen, der die Extraktion der Features überwacht, dazu, mehr oder weniger Features zu extrahieren. Wählt er $k = 0$ ist das Ergebnis von FeatureMapGen::MaxIn äquivalent zum Ergebnis von FeatureMapGen::NoIn, wählt er – bei n Artefakten – $k = n$, ist das Ergebnis hingegen äquivalent zum Ergebnis von FeatureMapGen::All.
Die Signatur der Methode `isFeatureCandidate(x)` wird für FeatureMapGen::MaxIn um den Parameter `k` auf `isFeatureCandidate(x, k)` erweitert. Der Rumpf der Methode wird, wie in Quelltext 4.4 gezeigt, angepasst.

```
1 parameters: x – artifact which may be an artifact
2             k – maximum amount of incoming edges
3 isFeatureCandidate(x, k) {
4     return x.incomingDependencies.count <= k
5 }
```

Quelltext 4.4: FeatureMapGen::MaxIn Pseudocode

Um die drei gewünschten Features *Feature-A*, *Feature-H* und *Feature-X* zu extrahieren, müsste bei der Durchführung von FeatureMapGen::MaxIn $k = 1$ gesetzt werden. Insgesamt betrachtet, entstehen dadurch jedoch die in Tabelle 4.4 dargestellten Features mit dem dazugehörigen Feature-Mapping.
Die dabei extrahierten Features *Feature-C*, *Feature-D* und *Feature-I* lassen die Variabilität der SPL jedoch über den gewünschten Grad hinaus steigen. Eine Durchführung von FeatureMapGen::MaxIn mit $k = 0$ erzeugt jedoch eine Variabilität äquivalent zu der von FeatureMapGen::NoIn (siehe Tabelle 4.3).
Der Vorteil von FeatureMapGen::MaxIn liegt darin, dass durch die Feature-Extraktion keine unveränderliche Menge an Features entsteht, sondern eine Menge, die durch Veränderung des Parameters k erweitert oder eingeschränkt werden kann. Zwar benötigt FeatureMapGen::MaxIn dazu die Expertise eines Mitarbeiters, der mit

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-C	—	—	•*	—	—	—	—	—	—	—
Feature-D	—	—	—	•*	—	—	—	—	—	—
Feature-H	—	•	—	—	•	•	•	•*	•	
Feature-I	—	•	—	—	•	•	•	—	•*	—
Feature-X	—	•	—	—	•	•	•	•	•	•*

Tabelle 4.4: Feature-Mapping für FeatureMapGen::MaxIn mit $k = 1$

der gesamten SPL vertraut ist, im Sinne der Skalierbarkeit ist aber dennoch eine deutliche Unterstützung durch FeatureMapGen gewährleistet: unabhängig davon, wie viele Implementierungsartefakte eine SPL enthält, der Experte muss immer nur einen Parameter verändern. Die Berechnungskomplexität bleibt jedoch auch für FeatureMapGen::MaxIn $\mathcal{O}(n^3)$, da für $k = n$ der Aufwand dem in Abschnitt A.1 erläuterten Aufwand entspricht.
Nachteilig bleibt, dass durch FeatureMapGen::MaxIn im gezeigten Beispiel nicht die gewünschte Menge an Features extrahiert werden konnte.

FeatureMapGen::MaxInMinOut

Um mehr Feature-Kandidaten als FeatureMapGen::MaxIn mit $k = 0$ zu extrahieren, ohne so viel Variabilität wie FeatureMapGen::MaxIn mit $k = 1$ zu extrahieren, berücksichtigt FeatureMapGen::MaxInMinOut neben den eingehenden Kanten der durch Knoten repräsentierten Artefakte, auch die von den Knoten ausgehenden Kanten. Dabei wird FeatureMapGen::MaxIn um eine minimale Anzahl ausgehender Kanten erweitert. Da die Menge der Feature-Kandidaten im Vergleich zu FeatureMapGen::MaxIn (mit $k = 0$) vergrößert werden soll, werden die beiden Bedingungen an die ein- ausgehende Kantenanzahl mit einem logischen *oder* verbunden. Die Menge der Feature-Kandidaten ergibt sich damit formal aus $A_{init}^{k,m} = \{x \mid x \in A_{all} \ \wedge \ (\#_{in}(x) \leq k \vee \#_{out}(x) \geq m)\}, k, m \in \mathbb{N}$. Ein Implementierungsartefakt wird Feature-Kandidat, wenn er höchstens k eingehende Kanten hat, oder wenn er mindestens m ausgehende Kanten hat.
Die Signatur und der Rumpf der Methode `isFeatureCandidate(x)` würde, wie in Quelltext 4.5 dargestellt, erneut angepasst werden.

Für den Experten, der die Extraktion der Features überwacht, kommt im Vergleich zu FeatureMapGen::MaxIn, ein zweiter Parameter m hinzu, der die Menge der Feature-Kandidaten und damit die Menge der Features beeinflussen kann. Bei geeigneter Parameterwahl lassen sich somit die gewünschten Features extrahieren. Um die Features *Feature-A*, *Feature-H* und *Feature-X* (und keine weiteren Features) zu extrahieren, müsste $k = 0$ und $m = 5$ gewählt werden. Die dabei entstehenden Features sowie das dazugehörige Feature-Mapping ist in Tabelle 4.5 dargestellt.
Die erzeugte Variabilität unter Verwendung von FeatureMapGen::MaxInMinOut,

```
parameters: x – artifact which may be an artifact
            k – maximum amount of incoming edges
            m – minimum amount of outgoing edges
isFeatureCandidate(x, k, m) {
    return  x.incomingDependencies.count <= k OR
            x.outgoingDependencies.count >= m;
}
```

Quelltext 4.5: FeatureMapGen::MaxInMinOut Pseudocode

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-H	—	•	—	—	•	•	•	•*	•	
Feature-X	—	•	—	—	•	•	•	•	•	•*

Tabelle 4.5: Feature-Mapping für FeatureMapGen::MaxInMinOut mit $k = 0$ und $m = 5$

entspricht genau der durch Experten gewünschten Variabilität. Durch das Berücksichtigen der Anzahl von eingehenden und ausgehenden Kanten kann die Menge an Feature-Kandidaten, und damit auch die Menge an Features, im gewählten Beispiel sehr gut durch die Parameter k und m angepasst werden.
Ein Nachteil ist jedoch, dass es dem Experten unter Umständen schwerfallen wird, eine Funktion mit zwei Unbekannten, die sich gegenseitig beeinflussen, zu parametrisieren. Bei entsprechender Parametrisierung erzeugt FeatureMapGen::MaxInMinOut den gleichen, in Abschnitt A.1 beschriebenen, Aufwand wie FeatureMapGen::All.

Fazit

In Tabelle 4.6 sind Vor- und Nachteile der einzelnen Varianten zusammengefasst. Eine optimale Variante von FeatureMapGen würde eine geringe Berechnungskomplexität aufweisen, das gewünschte Maß an Variabilität erfüllen und eine geringe Bedienkomplexität erfordern.

FeatureMapGen-Variante	Berechnungs-komplexität	Maß an Variabilität	Bedien-komplexität
All	hoch	hoch	gering
NoIn	gering	gering	gering
MaxIn	gering – hoch	gering – hoch	hoch
MaxInMinOut	gering – hoch	gering – hoch	sehr hoch

Tabelle 4.6: Vergleich der vier FeatureMapGen-Varianten

FeatureMapGen::All sollte aufgrund der hohen Berechnungskomplexität (Tiefensuche

für alle Artefakte), aber vor allem auch wegen der dabei entstehenden, im gezeigten Beispiel zu hohen, Variabilität nicht im produktiven Alltag eingesetzt werden.
FeatureMapGen::NoIn ist deutlich weniger rechenintensiv, benötigt keine Eingaben eines Experten, erzeugt jedoch unter Umständen eine zu geringe Variabilität.
FeatureMapGen::MaxIn kann im Vergleich zu FeatureMapGen::NoIn eine erhöhte Berechnungskomplexität aufweisen, da unter Umständen für eine nicht unerhebliche Anzahl an Artefakten eine Tiefensuche durchgeführt wird. Die Berechnungskomplexität variiert durch die Aussteuerung des Parameters k. Auch erzeugt FeatureMapGen::MaxIn im gezeigten Beispiel eine zu hohe Variabilität. Die Bedienkomplexität ist erhöht, da FeatureMapGen::MaxIn parametrisiert werden muss.
FeatureMapGen::MaxInMinOut weist ebenfalls eine variable Berechnungskomplexität auf, da die Anzahl der Feature-Kandidaten je nach Aussteuerung von k und m variiert. Zwar erfüllt FeatureMapGen::MaxInMinOut das Maß an Variabilität exakt, jedoch ist dafür eine sehr hohe Bedienkomplexität erforderlich, da zwei sich beeinflussende Parameter ausgesteuert werden müssen.

Keine der gezeigten Varianten erfüllt alle Kriterien optimal. FeatureMapGen::NoIn erfüllt zwei der drei Anforderungen (geringe Berechnungs- und Bedienkomplexität) optimal. Der Mangel an erzeugter Variabilität könnte ausgeglichen werden, wenn ein Experte Implementierungsartefakte festlegen würde, die unter allen Umständen später als Feature an- und abschaltbar sein sollen. Entscheidend für die Be- bzw. Entlastung des Experten ist dabei der Zeitpunkt zu dem er diese Artefakte definiert. Dabei erscheint es nicht entlastend für den Experten, wenn er vor der Bestimmung der Feature-Kandidaten die Artefakte festlegt, die seiner Meinung nach Feature-Kandidaten sein sollten, da er so, sehr wahrscheinlich Artefakte als Feature-Kandidat bestimmt, die FeatureMapGen::NoIn ohnehin für ihn bestimmt hätte. Ein möglicherweise geeignetere Ablauf ist es, durch FeatureMapGen::NoIn Feature-Kandidaten zu bestimmen und einen Experten im Anschluss Artefakte, die nicht als Feature-Kandidat bestimmt wurden, als solche nachnominieren zu lassen. Diese Anpassung an FeatureMapGen::NoIn würde zwar eine erhöhte Bedienkomplexität nach sich ziehen, jedoch ist das nachträgliche Bestimmen von Feature-Kandidaten für nur wenige Artefakte zu erwarten. Hinzu kommt, dass dieser zusätzliche Bedienschritt weniger abstrakt ist, als einen Parameter (k für FeatureMapGen::MaxIn) oder gar zwei Parameter (k und m für FeatureMapGen::MaxInMinOut) zu verändern. Ob diese Hypothesen tatsächlich der Realität entsprechen, gilt es in einer entsprechenden Evaluation zu untersuchen.
Mit FeatureMapGen steht ein Algorithmus zur Verfügung, der es ermöglicht aus Implementierungsartefakten teilautomatisiert Features zu extrahieren. Um einen Nutzer der SPL später die extrahierten Features installieren lassen zu können, empfiehlt es sich die Features in einem Feature-Modell abzubilden.

4.2 Extrahieren und Strukturieren ein Feature-Modells

FeatureModGen ermöglicht es aus einem gegebenen Feature-Mapping ein Feature-Modell zu extrahieren. Zur graphischen Repräsentation dieser Feature-Modelle werden in diesem Abschnitt Feature-Diagramme (siehe Abschnitt 2.2) verwendet.

Bei einem durch FeatureMapGen extrahierten Feature-Mapping, entstehen Features, die unabhängig voneinander funktionsfähig sind. Das bedeutet, dass kein extrahiertes Feature ein anderes Feature benötigt, um korrekt zu funktionieren. Bei der Installation bzw. dem Entfernen von Features aus einem System, muss sichergestellt werden, dass die überlappenden Implementierungsartefakte nur einmal installiert werden, bzw. nur dann entfernt werden, wenn kein Feature installiert ist, dass das fragliche Implementierungsartefakt benötigt. Für eine korrekte Funktionsweise von FeatureModGen ist jedoch entscheidend, dass keine Abhängigkeiten zwischen Features existieren.
Zunächst wird die generelle Funktionsweise von FeatureModGen erklärt. Im Anschluss wird anhand eines beispielhaften Feature-Mappings gezeigt, wie aus Features ein Feature-Modell extrahiert werden kann.

4.2.1 Funktionsweise von FeatureModGen

Der Ausgangspunkt zur Extraktion eines Feature-Modells sind voneinander unabhängige Features und ein dazugehöriges Feature-Mapping. FeatureModGen extrahiert in mehreren Iterationen gemeinsame Implementierungsartefakte aus Features und fasst die extrahierten Artefakte in neuen Features zusammen. Das zuvor erstellte Feature-Mapping wird daher durch FeatureModGen verändert. Der in Quelltext 4.6 gezeigte Pseudocode veranschaulicht die Funktionsweise von FeatureModGen.
Nach der Erstellung eines initialen Feature-Modells im *Initialisierungsschritt*, werden gemeinsame Implementierungsartefakte im *Extraktionsschritt* so weit wie möglich aus Features heraus extrahiert. Dieser Schritt kann Feature-Modelle extrahieren, die sich durch einen *Vereinfachungsschritt* vereinfachen lassen. Im Folgenden werden die drei Schritte detailliert erläutert.

Initialisierungsschritt

FeatureModGen's Initialisierung sieht vor, ein triviales Feature-Modell zu extrahieren, bei dem alle extrahierten Features unter einem abstrakten *Kern-Feature*, als optional gekennzeichnet, angeordnet sind (Zeile 3 in Quelltext 4.6). Features unter dem Kern-Feature werden als optional gekennzeichnet, damit es später möglich, den Kern der SPL an den Kunden zu distribuieren und nachträglich Feeatures hinzu installieren zu können. Der Kern der SPL muss daher eine eigenständige, gültige Variante der SPL sein. Das von FeatureMapGen extrahierte Feature-Mapping bleibt während FeatureModGen's erstem Schritt unverändert. Im Anschluss (Zeile 6 in Quelltext 4.6) wird eine Liste aller verfügbaren Implementierungsartefakte (A_{sorted}) erzeugt und (absteigend) nach den relativen Häufigkeiten (h_n) sortiert. Ein Artefakt hat eine relative Häufigkeit von 1, wenn es Bestandteil eines jeden Features ist und eine relative Häufigkeit von $h_n = \frac{1}{n}$, wenn es Bestandteil von nur einem Feature ist. Alle Artefakte mit $h_n = 1$ können, da sie von jedem Feature benötigt werden, als Bestandteil des Kern-Features abgebildet werden (Zeile 9 – 18 in Quelltext 4.6). Das zuvor erstellte, abstrakte Kern-Feature wird in diesem Fall in ein konkretes Feature umgewandelt. Das Feature-Mapping verändert sich, da alle Implementierungsartefakte mit $h_n = 1$ nur noch Bestandteil des Kern-Features sind. Die in das Kern-Feature verschobenen Artefakte werden aus A_{sorted} entfernt. Es werden auch die Artefakte entfernt, die eine relative Häufigkeit von $h_n = \frac{1}{n}$ aufweisen, da diese Artefakte

lediglich Bestandteil eines einzigen Features sind und daher in einem späteren Schritt nicht aus mehreren Features extrahiert werden können. A_{sorted} umfasst nach dem FeatureModGen-Initialisierungsschritt, noch nicht als Bestandteil des Kern-Features definierte Artefakte und Artefakte, die Teil von mehr als einem Feature sind. Da sie Teil mehrerer Features sind, besteht für diese Artefakte die Möglichkeit, sie als Bestandteil neuer Features zu definieren, und die Features, deren Bestandteil die Artefakte zuvor waren, unterhalb dieses neuen Features anzuordnen. Bestimmte Implementierungsartefakte werden also aus ihren Features heraus extrahiert. Der nächste FeatureModGen-Schritt wird daher als *Extraktionsschritt* bezeichnet.

```
1  parameters: Features – all extracted features containing the artifacts
2                         they are mapped to
3              Artifacts – all artifacts
4  featureModGen(Features, Artifacts) {
5      //init a new feature model with all given features
6      featureModel ← newFeatureModel(Features)
7
8      //sort artifacts according to their relative frequency
9      artifacts_sorted ← artifacts.sort()
10
11     //move artifacts to root that occur in each and every feature
12     for each a ∈ artifacts_sorted {
13         if (a.getFrequency() = 1) {
14             featureModel.moveArtifactToRoot(a);
15             artifacts_sorted ← artifacts_sorted \ a
16         } else {
17             //we can break this loop if we found an artifact with
18             //a relative frequency less than 1
19             //because artifacts are sorted
20             break
21         }
22     }
23
24     //extract artifacts
25     for each a ∈ artifacts_sorted {
26
27         didExtraction ← true
28         //try to extract the same artifact
29         //until it could not be extracted anymore
30         while(didExtraction) {
31             didExtraction ← featureModel.extractArtfact(a)
32
33             //if extraction was successful
34             //clean up the feature model
35             if(didExtraction)
36                 featureModel.cleanUp()
37         }
38     }
39     return featureModel
40 }
```

Quelltext 4.6: FeatureModGen Pseudocode

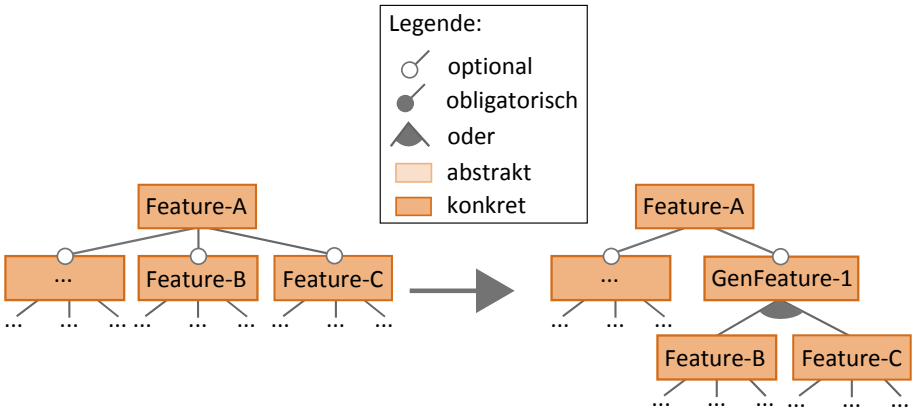
Extraktionsschritt

Während des Extraktionsschritts (Zeile 21 – 34 in Quelltext 4.6) wird für jedes Implementierungsartefakt $X \in A_{sorted}$, versucht, es aus Features, die es enthalten, zu extrahieren. Artefakte, die häufiger als andere vorkommen, werden dabei (aufgrund der Sortierung von A_{sorted}) zuerst untersucht. Dies gewährleistet, dass häufige Artefakte möglichst *nah* am Kern-Feature angeordnet werden können. Für jedes Artefakt wird untersucht, welche Features, die im Feature-Diagramm auf der selben Ebene liegen und das gleiche Eltern-Feature haben, das zu untersuchende Artefakt enthalten. Das Ergebnis ist eine Menge $F_{sameArtifact}$, die die Features enthält, die alle das aktuell untersuchte Artefakt enthalten. Gilt für diese Menge $|F_{sameArtifact}| > 1$, ist das Artefakt Bestandteil mehrerer Features und kann daher aus den Features extrahiert werden, die dieses Artefakt enthalten. Dazu wird ein neues Feature erzeugt, welches das aktuell untersuchte Artefakt enthält. Alle Features aus $F_{sameArtifact}$ werden unterhalb des neu erzeugten Features in einer Oder-Verknüpfung angeordnet und enthalten das aktuell untersuchte Artefakt nicht länger.
Abbildung 4.2 stellt dar, wie das neu erzeugte Feature in das Feature-Modell integriert wird: grundsätzlich wird es unter dem Eltern-Feature der Features aus $F_{sameArtifact}$ angeordnet. Waren die Features aus $F_{sameArtifact}$ als optional gekennzeichnet, wird das neu erzeugte Feature ebenfalls als optional gekennzeichnet (siehe Abbildungen 4.2(a) und 4.2(c)). Traten die Features aus $F_{sameArtifact}$ in einer Oder-Verknüpfung auf und existierten neben ihnen noch andere, benachbarte Features, wird das neu erzeugte Feature in die bestehende Oder-Verknüpfung integriert (siehe Abbildung 4.2(b)). Existierten neben den Features aus $F_{sameArtifact}$ keine benachbarten Features, wird das neu erzeugte Feature, als obligatorisch gekennzeichnet, unterhalb des Eltern-Features der Features aus $F_{sameArtifact}$, angeordnet (siehe Abbildung 4.2(d)). Führt ein Extraktionsschritt zu keiner Veränderung im Feature-Modell, wird der aktuell untersuchte Implementierungsartefakt aus A_{sorted} gestrichen. Nach jeder erfolgreichen Extraktion eines Artefakts, wird das extrahierte Feature-Modell im *Vereinfachungsschritt* vereinfacht.

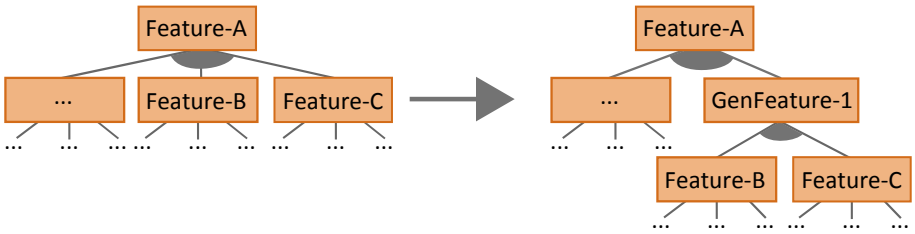
Vereinfachungsschritt

Im Vereinfachungsschritt (Zeile 28 in Quelltext 4.6) werden die zwei folgenden Vereinfachungen am Feature-Modell vorgenommen:

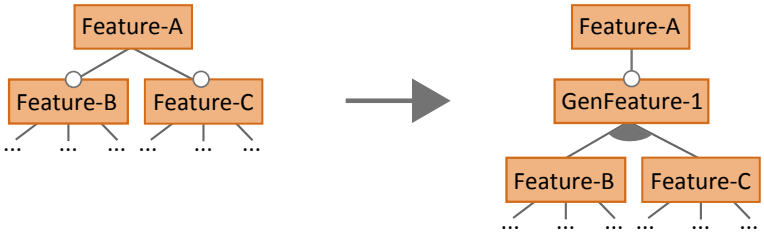
1. Durch die Extraktion von Implementierungsartefakten aus Features kann es dazu kommen, dass ein Feature keine Artefakte mehr enthält (genau dann, wenn das extrahierte Artefakt der einzig verbliebene Implementierungsartefakt dieses Features war). Solche Features werden daher zu abstrakten Features. Da abstrakte Features keine Implementierungsartefakte enthalten und die Funktionsweise der SPL somit nicht verändern, können sie ohne weitere Umstände aus dem Feature-Modell entfernt werden. War das entfernte Feature durch eine Oder-Verknüpfung mit nur einem anderen Feature verknüpft, muss das verbleibende Feature als optional gekennzeichnet, damit für diesen Teilbaum die gleichen Varianten generiert werden können[AGM+06, TBK09, STKS12]. Kind-Features des entfernten Features sind nicht zu berücksichtigen, da nur abstrakte Features entfernt werden und abstrakte Features während der Durchführung



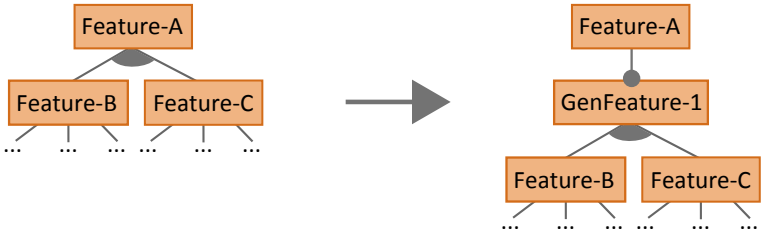
(a) Extraktion eines von *Feature-B* und *Feature-C* gemeinsamen Implementierungsartefakts, für den Fall, dass *Feature-B* und *Feature-C* als optional gekennzeichnet sind und noch benachbarte Features haben



(b) Extraktion eines von *Feature-B* und *Feature-C* gemeinsamen Implementierungsartefakts, für den Fall, dass *Feature-B* und *Feature-C* in einer Oder-Verknüpfung zusammen mit weiteren, benachbarten Features auftreten



(c) Extraktion eines von *Feature-B* und *Feature-C* gemeinsamen Implementierungsartefakts, für den Fall, dass *Feature-B* und *Feature-C* als optional gekennzeichnet sind und keine benachbarten Features haben



(d) Extraktion eines von *Feature-B* und *Feature-C* gemeinsamen Implementierungsartefakts, für den Fall, dass *Feature-B* und *Feature-C* in einer Oder-Verknüpfung ohne weitere, benachbarte Features auftreten

Abbildung 4.2: Mögliche Varianten des FeatureModGen-Extraktionsschritts

von FeatureModGen niemals Kind-Features haben werden (mit Ausnahme des Kern-Features).

2. Features, die als obligatorisch gekennzeichnet als einziges Kind-Feature unterhalb ihres Eltern-Features angeordnet sind, können mit ihrem Eltern-Feature *verschmolzen* werden, da ohnehin nur beide gleichzeitig ausgewählt werden können. Verschmelzen heißt, dass die Artefakte und eventuelle Kind-Features des obligatorischen Kind-Features auf das entsprechenden Eltern-Feature übertragen werden. Im Anschluss kann das obligatorische Kind-Feature aus dem Feature-Modell entfernt werden, da es nun keine Kind-Features mehr hat und ein abstraktes Feature geworden ist.

Das gesamte Feature-Modell wird so lang nach möglichen Vereinfachungen durchsucht, bis bei einer kompletten Iteration keine mögliche Vereinfachung mehr gefunden werden kann. Anschließend wird im Extraktionsschritt mit dem nächsten Artefakt aus A_{sorted} fortgefahren.

Wurden alle möglichen Implementierungsartefakte extrahiert und das Feature-Modell abschließend vereinfacht, ist FeatureModGen beendet. Anhand eines Beispieles soll im nächsten Abschnitt die Funktionsweise von FeatureModGen verdeutlicht werden.

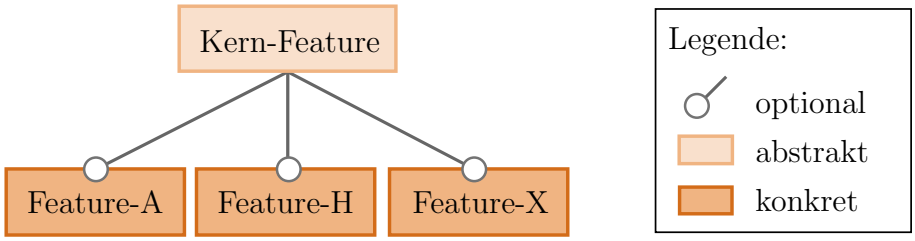
4.2.2 FeatureModGen am Beispiel des HMIServers

In diesem Abschnitt wird ein durch FeatureMapGen extrahiertes Feature-Mapping verwendet, um mithilfe von FeatureModGen ein Feature-Modell zu extrahieren. Dabei dient das in Tabelle 4.7 dargestellte Feature-Mapping als exemplarischer Ausgangspunkt für FeatureModGen.

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•*	•	•	•	•	•	•	—	—	—
Feature-H	—	•	—	—	•	•	•	•*	•	—
Feature-X	—	•	—	—	•	•	•	•	•	•*
relative Häufigkeit (h_n)	$\frac{1}{3}$	1	$\frac{1}{3}$	$\frac{1}{3}$	1	1	1	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$

Tabelle 4.7: Feature-Mapping mit relativen Häufigkeiten der Implementierungsartefakte

Das im ersten Teil des Initialisierungsschrittes extrahierte Feature-Modell, bei dem sämtliche Features direkt unterhalb eines abstrakten Kern-Feature angeordnet werden, ist in Abbildung 4.3(a) dargestellt. Abbildung 4.3(b) zeigt das (noch) unveränderte Feature-Mapping. Der zweite Teilschritt des Initialisierungsschrittes sieht vor, die Liste der Implementierungsartefakte ihrer relativen Häufigkeit nach zu sortieren. Auf eine Visualisierung dieses Schrittes wird an dieser Stelle verzichtet. Nicht verzichtet wird jedoch auf das *Verschieben* der Implementierungsartefakte, die Bestandteil eines

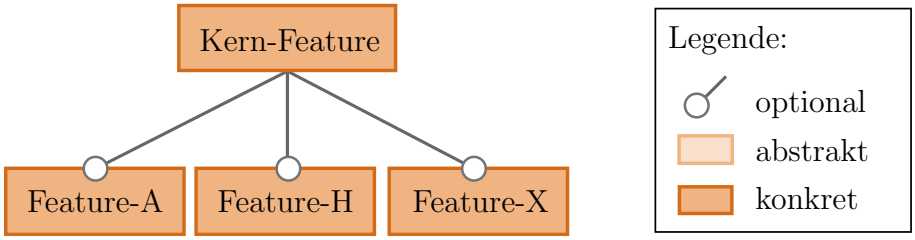


(a) *Feature-Modell als Feature-Diagramm*

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•	•	•	•	•	•	•	—	—	—
Feature-H	—	•	—	—	•	•	•	•	•	—
Feature-X	—	•	—	—	•	•	•	•	•	•
Kern-Feature	—	—	—	—	—	—	—	—	—	—

(b) *Feature-Mapping*

Abbildung 4.3: Feature-Modell und Feature-Mapping nach dem ersten Teil des FeatureModGen-Initialisierungsschrittes



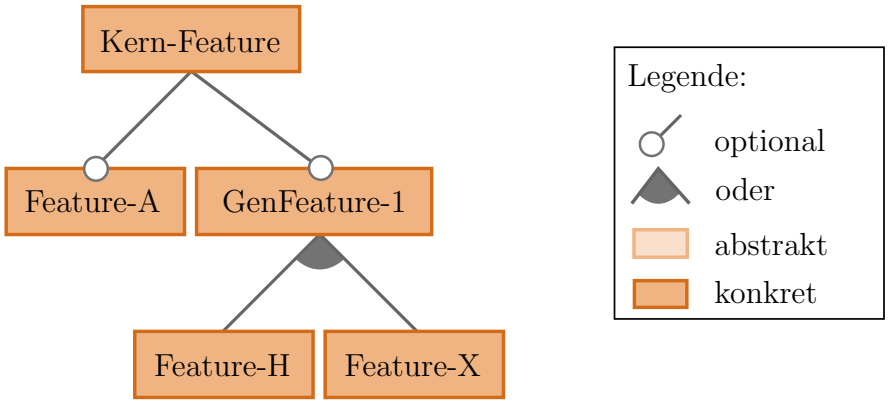
(a) *Feature-Modell als Feature-Diagramm*

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•	—	•	•	—	—	—	—	—	—
Feature-H	—	—	—	—	—	—	—	•	•	—
Feature-X	—	—	—	—	—	—	—	•	•	•
Kern-Feature	—	•	—	—	•	•	•	—	—	—

(b) *Feature-Mapping*

Abbildung 4.4: Feature-Modell und Feature-Mapping nach dem FeatureModGen-Initialisierungsschritt

jeden Features sind. Abbildung 4.4 zeigt das dabei entstehende Feature-Modell als Feature-Diagramm, sowie das dazugehörige Feature-Mapping. Tabelle 4.4(b) verdeutlicht, dass die Artefakte B, E, F und G nur noch Bestandteil des Kern-Features sind



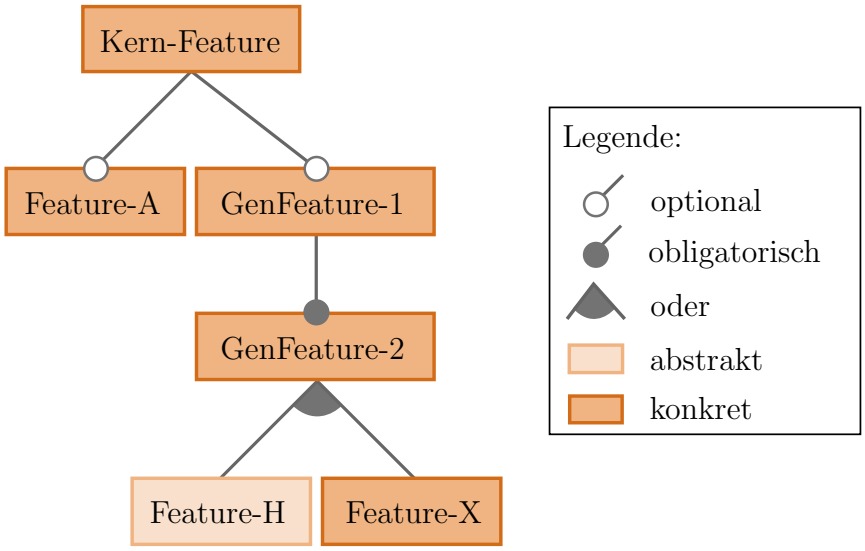
(a) *Feature-Modell als Feature-Diagramm*

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•	—	•	•	—	—	—	—	—	—
Feature-H	—	—	—	—	—	—	—	—	•	—
Feature-X	—	—	—	—	—	—	—	—	•	•
Kern-Feature	—	•	—	—	•	•	•	—	—	—
GenFeature-1	—	—	—	—	—	—	—	•	—	—

(b) *Feature-Mapping*

Abbildung 4.5: Feature-Modell und Feature-Mapping nach der Extraktion des Implementierungsartefakts H aus den Features *Feature-H* und *Feature-X*

und, dass sie aus den Features *Feature-A*, *Feature-H* und *Feature-X* entfernt wurden. Im Anschluss an den Initialisierungschritt folgt der Extraktionsschritt. Gemäß der Sortierung von A_{sorted}, wird in diesem Schritt zunächst versucht, das Artefakt H aus Features heraus zu extrahieren, die H enthalten und ein gemeinsames Eltern-Feature haben. Beim Vergleich der Features *Feature-A*, *Feature-H* und *Feature-X* wird festgestellt, dass sowohl *Feature-H*, als auch *Feature-X* das Artefakt H enthalten. Wie bereits in Abbildung 4.2(c) gezeigt, wird daher ein neues Feature *GenFeature-1* erzeugt und unterhalb des Eltern-Features von *Feature-H* und *Feature-X* angeordnet. Da *Feature-H* und *Feature-X* als optional gekennzeichnet waren, wird auch *GenFeature-1* als optional gekennzeichnet. *Feature-H* und *Feature-X* selbst, werden in einer Oder-Verknüpfung unterhalb des neu erzeugten *GenFeature-1* angeordnet. Das Ergebnis ist in Abbildung 4.5 dargestellt. Tabelle 4.5(b) macht deutlich, dass es bei diesem Schritt dazu kommen kann, dass die Namen der Features nicht mehr deren enthaltenen Artefakten entsprechen (siehe *Feature-H*). In *einfach* Fällen, z.B. wenn das Feature nur noch ein Implementierungsartefakt enthält, könnte der Name des Features angepasst werden. FeatureModGen verzichtet jedoch zunächst auf eine Umbenennung der Features.
Nach der erfolgreichen Extraktion des Implementierungsartefakts H aus den Features *Feature-H* und *Feature-X*, wird im Vereinfachungsschritt überprüft, ob das bisher



(a) *Feature-Modell als Feature-Diagramm*

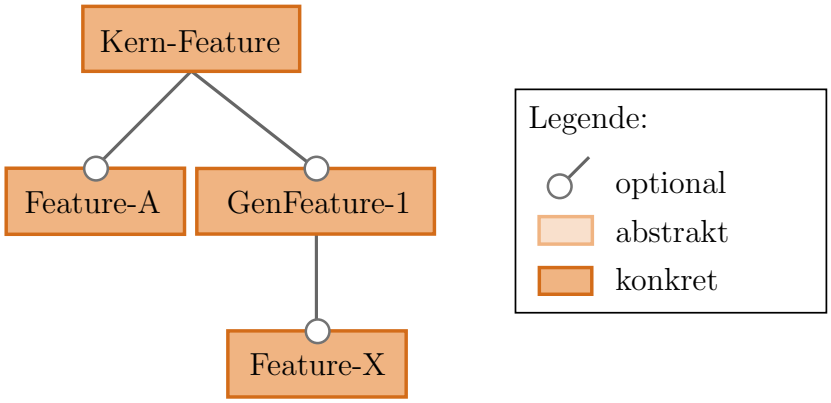
Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•	—	•	•	—	—	—	—	—	—
Feature-H	—	—	—	—	—	—	—	—	—	—
Feature-X	—	—	—	—	—	—	—	—	—	•
Kern-Feature	—	•	—	—	•	•	•	—	—	—
GenFeature-1	—	—	—	—	—	—	—	•	—	—
GenFeature-2	—	—	—	—	—	—	—	—	•	—

(b) *Feature-Mapping*

Abbildung 4.6: Feature-Modell und Feature-Mapping nach der Extraktion des Implementierungsartefakts I aus den Features *Feature-H* und *Feature-X*

extrahierte Feature-Modell, abstrakte Features, oder Features, die als einziges und als obligatorisch gekennzeichnet unter ihrem Eltern-Feature angeordnet sind, enthält. Abbildung 4.5(a) zeigt, dass das Feature-Modell keine solcher Features enthält. Eine Vereinfachung des Feature-Modells ist daher zu diesem Zeitpunkt nicht erforderlich. Nach der Überprüfung möglicher Vereinfachungen, wird der Extraktionsschritt mit dem nächsten Implementierungsartefakt fortgesetzt. In der Liste der sortierten Artefakte ist das Artefakt *I* das nächste Element. Ein Vergleich der Features *Feature-A* und *GenFeature-1* ergibt, dass nicht beide Features das Artefakt *I* beinhalten. Ein Vergleich der Features *Feature-H* und *Feature-X* ergibt, dass sowohl *Feature-H* als auch *Feature-X* das Artefakt *I* enthalten. Artefakt *I* kann daher aus diesen beiden Features extrahiert werden. Gemäß Abbildung 4.2(d), entsteht dabei ein neues Feature *GenFeature-2*, dass, da *Feature-H* und *Feature-X* in einer Oder-Verknüpfung als einzige Kind-Features unterhalb von *GenFeature-1* angeordnet waren, als obligatorisch gekennzeichnet unterhalb von *GenFeature-1* angeordnet wird. *Feature-H* und

Feature-X werden in einer Oder-Verknüpfung unterhalb des neu erzeugten Features *GenFeature-2* angeordnet. Der Implementierungsartefakt *I*, wird aus *Feature-H* und *Feature-X* entfernt und zu *GenFeature-2* hinzugefügt. Das dabei entstehende Feature-Modell sowie das dazugehörige Feature-Mapping, sind in Abbildung 4.6 dargestellt. Da *Feature-H* keine Artefakte beinhaltet, wurde es im Feature-Modell als abstrakt gekennzeichnet. Im, auf diesen Teil des Extraktionsschritts folgenden, Vereinfachungs-



(a) *Feature-Modell als Feature-Diagramm*

Feature	Implementierungsartefakt									
	A	B	C	D	E	F	G	H	I	X
Feature-A	•	—	•	•	—	—	—	—	—	—
Feature-X	—	—	—	—	—	—	—	—	—	•
Kern-Feature	—	•	—	—	•	•	•	—	—	—
GenFeature-1	—	—	—	—	—	—	—	•	•	—

(b) *Feature-Mapping*

Abbildung 4.7: Finales FeatureModGen Feature-Modell und Feature-Mapping

schritt, wird zunächst das abstrakte Feature *Feature-H* aus dem Feature-Modell und dem Feature-Mapping entfernt. Da *Feature-H* in einer Oder-Verknüpfung mit *Feature-X* auftrat, wird *Feature-X* in diesem Schritt als optional gekennzeichnet. Anschließend wird das Feature-Modell nach Features durchsucht, die als einziges Kind-Feature und als obligatorisch gekennzeichnet unterhalb ihres Eltern-Features angeordnet sind. Dabei wird ermittelt, dass dieser Umstand auf *GenFeature-2* zutrifft. Die Implementierungsartefakte von *GenFeature-2* können daher ohne Weiteres auf sein Eltern-Feature *GenFeature-1* übertragen werden. *GenFeature-2* kann im gleichen Zug aus Feature-Modell und -Mapping entfernt werden. Das verbleibende Kind-Feature *Feature-X* wird unterhalb von *GenFeature-1* angeordnet. Das dabei entstehende Feature-Modell sowie das dazugehörige Feature-Mapping sind in Abbildung 4.7 dargestellt. Da die Liste der Implementierungsartefakte keine weiteren Artefakte enthält, die Potenzial für eine mögliche Extraktion bieten würden, ist das in Abbildung 4.7 gezeigte Feature-Modell und Feature-Mapping das finale Ergebnis von FeatureModGen.

Durch die bereits erwähnte Expertenumfrage wurde zuvor ermittelt, dass die OSGi-Bundles A, H und X an- und abschaltbar sein sollten. Das erstellte Feature-Modell ermöglicht genau diesen Umstand. Das Feature *Feature-A* ermöglicht das An- und Abschalten des OSGi-Bundles A. Das Feature *GenFeature-1* ermöglicht es, das OSGi-Bundle H an- und abzuschalten. Durch *Feature-X* wird es möglich, das OSGi-Bundle X an- und abzuschalten. Da das OSGi-Bundle X immer das OSGi-Bundle H benötigt, ist *Feature-X* während FeatureModGen ein Kind-Feature von *GenFeature-1* geworden.

4.3 Implementierung von FeatureMapGen und FeatureModGen

Um die Machbarkeit von FeatureMapGen und FeatureModGen zu zeigen, wurde ein Werkzeug entwickelt, das einen Domänenexperten dabei unterstützt, aus einer Menge OSGi-Bundles, ein Feature-Modell zu extrahieren. Dazu wählt der Experte zunächst das Verzeichnis aus, in dem sich alle relevanten OSGi-Bundles befinden. Daraufhin bekommt er in der linken Spalte der Benutzeroberfläche (siehe Abbildung 4.8) die symbolischen Namen aller OSGi-Bundles angezeigt. Die Auswahlboxen vor den Namen der OSGi-Bundles fordern den Experten dazu auf, Feature-Kandidaten auszuwählen. Dies kann der Experte entweder manuell erledigen, oder Feature-Kandidaten durch FeatureMapGen-Algorithmen[1] bestimmen lassen. Auch bei einer Bestimmung der Feature-Kandidaten durch FeatureMapGen, hat der Experte die Möglichkeit die Menge der Feature-Kandidaten durch An- bzw. Abhaken der entsprechenden Auswahlboxen zu verändern.
Ist der Experte mit der Auswahl der Feature-Kandidaten zufrieden, kann er durch FeatureModGen ein Feature-Modell erstellen lassen. Dieses Feature-Modell wird ihm dann in der rechten Spalte der Benutzeroberfläche angezeigt. Im Anschluss hat der Experte die Möglichkeit, das Feature-Modell zu bearbeiten. Dazu stehen ihm auf der einen Seite einfache Bearbeitungsmöglichkeiten direkt in der gezeigten Benutzeroberfläche zu Verfügung. Auf der anderen Seite ermöglicht die Benutzeroberfläche dem Experten den Export des Feature-Modells in andere Feature-Modell-Beschreibungsformen. Zur einfachen Bearbeitung besteht in der Benutzeroberfläche derzeit die Möglichkeit Features umzubenennen. Für erweiterte Bearbeitungsmöglichkeiten steht der Export des Feature-Modells in das FeatureIDE-Format [KTS+09] zu Verfügung. Beim Export des Feature-Modells in andere Beschreibungsformen ist wichtig, dass neben den Features und deren Abhängigkeiten auch das Feature-Mapping teil des Export ist, da die Arbeit von FeatureMapGen und die des Domänenexperten ansonsten verloren gehen würde. Quelltext B.2 im Anhang zeigt, wie das FeatureIDE-Format erweitert wurde. Durch diese Erweiterung kann das Feature-Modell nach dem Bearbeiten mit FeatureIDE zurück in die FeatureModGen-Benutzeroberfläche geladen werden, ohne das Informationen verloren gehen.

[1] Aktuell unterstützt die Benutzeroberfläche nur FeatureMapGen::NoIn

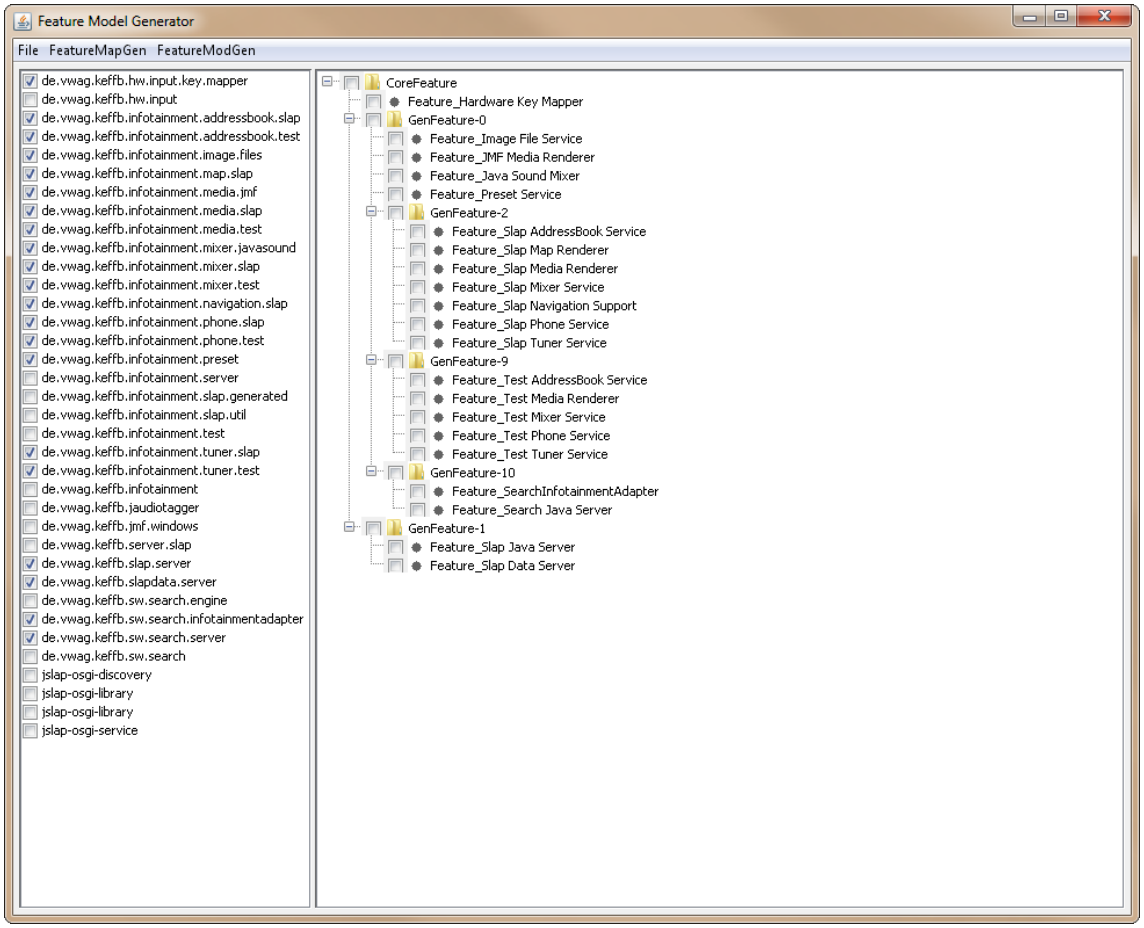


Abbildung 4.8: Benutzeroberfläche von FeatureModGen

4.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Feature-Modelle aus Implementierungsartefakten und deren Abhängigkeiten extrahiert werden können. Entscheidend für eine solche Extraktion ist, dass sämtliche Abhängigkeiten zwischen allen Implementierungsartefakten bekannt sind. Andernfalls entsteht unter Umständen ein nicht valides Feature-Modell. Ergebnisse einer durchgeführten Fallstudie zur Extraktion eines Feature-Modells, werden in Kapitel 6 vorgestellt und diskutiert.

5. Distribution von Software-Produktlinien

Nachdem im vorherigen Kapitel 4 das Erstellen von Features, Feature-Mapping und Feature-Modellen beschrieben wurde, wird in diesem Kapitel am Beispiel des HMIServers ein möglicher Distributionsprozess einer SPL vorgestellt. Das hier vorgestellte Konzept lässt sich auf die Distribution aller SPLs anwenden, bei denen ein bestimmter Basisquelltext nach seiner Kompilierung durch anderen kompilierten Quelltext erweitert werden kann.

5.1 Zentrale Speicherung von Basis- und Featurecode im SPL-Repository

Um eine Software, auch wenn sie keine SPL ist, bei einem Kunden zum Einsatz zu bringen, muss die Software auf irgendeine Art und Weise den Weg vom Hersteller zum Kunden nehmen. Soll die Software zudem, wie in Kapitel 3 gefordert, möglichst ohne individuelle Beratung des Kunden durch den Hersteller den Weg zum Kunden nehmen, muss dieser Weg automatisiert werden. Dazu ist es notwendig, dass der Kunde sich die Software von einem zentralen, für alle möglichen Kunden zugänglichen, Speichermedium herunterladen kann. Solch ein zentrales Speichermedium kann z.B. ein Server im Internet sein.[1]
Auf diesem Webserver werden sowohl der kompilierte Basisquelltext der zu distribuierenden SPL, als auch der kompilierte Quelltext aller Features abgelegt. Dies ermöglicht es dem Kunden die Basis der SPL und die für diese SPL verfügbaren Features individuell nach eigenen Bedürfnissen und eigenem Ermessen herunterzuladen. Der Server, auf dem Basis- und Featurecode der SPL abgelegt sind, wird fortan an als *SPL-Repository* bezeichnet. Neben Basis- und Featurecode befindet sich im SPL-Repository auch das durch FeatureModGen extrahierte Feature-Modell, sowie das durch FeatureMapGen erstellte Feature-Mapping.

[1] Über welches Protokoll mit diesem Server auf Applikationsebene kommuniziert wird ist dabei zunächst unerheblich. Denkbar sind hierbei das *Hypertext Transfer Protocol (HTTP)* oder das *File Transfer Protocol (FTP)* bzw. deren *sichereren* Varianten *HTTPS* und *SFTP*.

Im Falle der Volkswagen AG handelt es sich bei der zu distribuierenden SPL um den bereits mehrfach erwähnten HMIServer. Eine Distribution über das Internet wäre in diesem Fall weder notwendig, noch wäre sie mit den Sicherheitspolitiken des Konzerns vereinbar. Die Bereitstellung des HMIServers im konzernweiten Netzwerk ist hingegen vollkommen ausreichend und sicherheitspolitisch unkritisch.

5.2 Installationsprozess der HMIServer SPL

Der Installationsprozess beschreibt, wie ein potenzieller HMIServer-Kunde einen HMIServer mit den Features installiert, die ihm dabei helfen, sein neues Bedienkonzept zu präsentieren. Der in Kapitel 3 erstellte Soll-Zustand für die Distribution des HMIServers sieht vor, dass der HMIServer zunächst ohne vorinstallierte Features aus dem SPL-Repository heruntergeladen wird. Führt der Kunde den HMIServer aus, macht ihn eine spezielle Benutzeroberfläche darauf aufmerksam, dass noch keine Features im HMIServer installiert sind und der HMIServer somit keine, für den Kunden interessanten, Funktionen zur Verfügung stellt. Um dies zu ändern, wird der Kunde aufgefordert Features auszuwählen, die nach quittieren dieser Auswahl aus dem Repository *nachgeladen* und im Anschluss installiert werden. Abbildung 5.1 zeigt die im Rahmen dieser Arbeit entwickelte Benutzeroberfläche zur Installation und Deinstallation von Features des HMIServers.

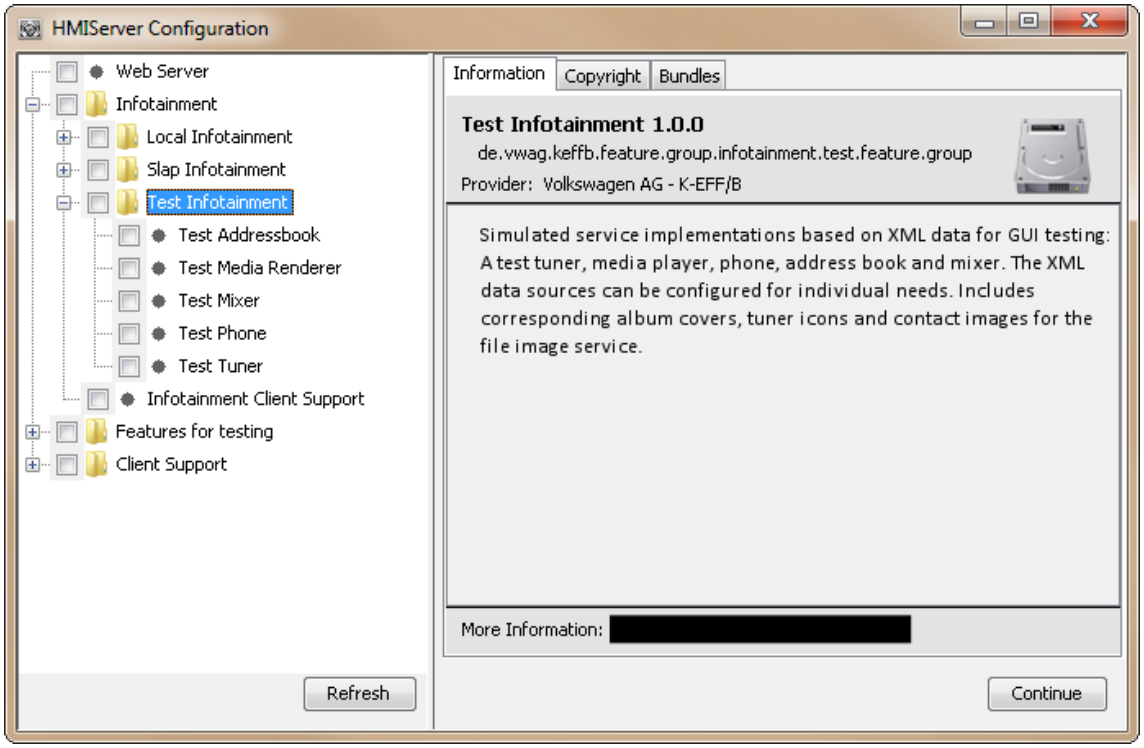


Abbildung 5.1: Benutzeroberfläche zum Installieren und Deinstallieren von Features des HMIServers

Der Auswahlbaum am linken Rand der Benutzeroberfläche wird dabei aus dem, im SPL-Repository befindlichen, Feature-Modell erstellt. Abbildung 5.2 zeigt das für diesen Auswahlbaum zugrundeliegende Feature-Modell als Feature-Diagramm.

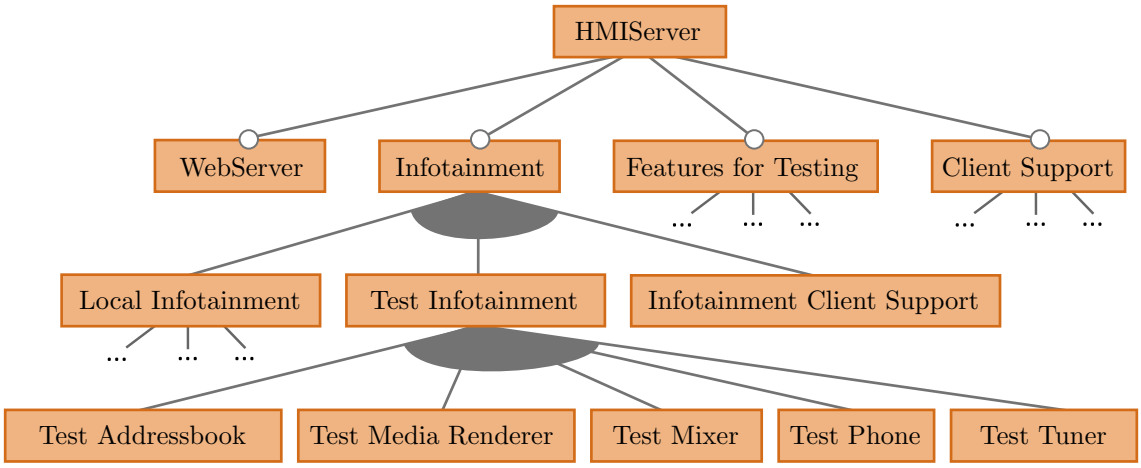


Abbildung 5.2: Feature-Diagramm zur Repräsentation des in Abbildung 5.1 zugrundeliegenden Feature-Modells

Da FeatureModGen eine einfache Baumstruktur erzeugt und auf cross-tree constraints (vgl. Abschnitt 2.2) verzichtet, kann das Feature-Modell dem Kunden als Baumstruktur präsentiert werden. Durch diesen Auswahlbaum ist der Kunde in der Lage Features auszuwählen, die er gern im HMISever verwenden und daher gern installieren lassen würde.
Bei der Auswahl der Features findet die aus Feature-Diagrammen bekannte und in Abschnitt 2.2 eingeführte Semantik von *Und*-, *Oder*- und *Alternativ-Verknüpfungen* ihre Anwendung. Im gezeigten Beispiel wurde das Feature *Test Media Renderer* ausgewählt, was zur Folge hat, dass sowohl sein Eltern-Feature *Test Infotainment* als auch dessen Eltern-Feature *Infotainment* ausgewählt wurde. Die Auswahl einer bestimmten Menge an Features ergibt nach deren Installation eine bestimmte Variante des HMIServers. Die Benutzeroberfläche ist so implementiert, dass jede aktuell ausgewählte Menge an Features eine gültige Variante des HMIServers ergibt. Würde bspw. *Test Media Renderer* abgewählt werden, würde auch *Test Infotainment* (und dadurch auch *Infotainment*) abgewählt werden, da alle Kind-Features von *Test Infotainment* in einer Alternativ-Verknüpfung miteinander verbunden sind und daher keine Variante des HMIServer existiert, in der *Test Infotainment* ausgewählt, jedoch keines der Kind-Features von *Test Infotainment* ausgewählt sind. Das Auswählen des Features würde dazu führen, dass alle Kind-Features ausgewählt werden. Nicht benötigte Features können vom Benutzer dann abgewählt werden. Das automatische Auswählen von Features funktioniert nur, das FeatureModGen nur optionale Features und Oder-Verknüpfungen erzeugt. Würde FeatureModGen bspw. Alternativ-Verknüpfungen erzeugen, so ließe sich dieses Verfahren so nicht aufrecht erhalten.

Hat der Kunde alle Features ausgewählt, die er benötigt, werden die erforderlichen Implementierungsartefakte nach Quittierung des Kunden durch einen Klick auf *Continue*, anhand des im SPL-Repository befindlichen Feature-Mappings ermittelt, heruntergeladen und im HMIServer installiert. In Abhängigkeit der Größe der Implementierungsartefakte kann dieser Prozess einige Zeit in Anspruch nehmen. Dem Kunden wird daher, wie in Abbildung 5.3 gezeigt, der Fortschritt des Prozesses

visualisiert.

Deinstallationen von Features bzw. deren Implementierungsartefakten erfolgt über die gleiche Benutzeroberfläche. Beim Starten der Oberfläche sind alle Features selektiert, die aktuell im HMIServer installiert sind. Durch Abwahl zu Beginn selektierter Features, werden diese nach dem Quittieren deinstalliert.

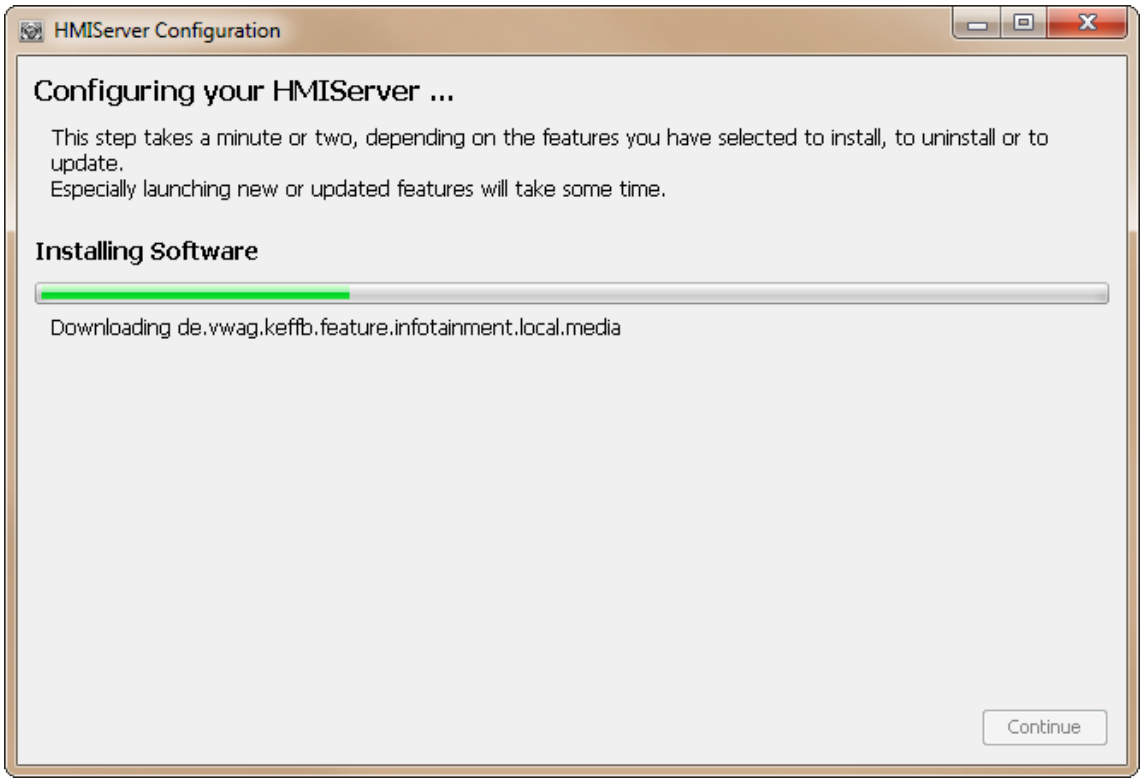


Abbildung 5.3: Benutzeroberfläche zum Installieren und Deinstallieren von Features des HMIServers während der Installation von Features

5.3 Implementierung der Distributionssoftware

Die im vorherigen Abschnitt beschriebenen Prozesse der Installation und Deinstallation wurde für den HMIServer mithilfe von *Equinox p2* realisiert. Equinox p2 ein von der *Eclipse Foundation* im Rahmen des Equniox Projektes[2] entwickeltes Distributionssystem für OSGi-basierte Systeme (wie z.B. den HMIServer). P2 ermöglicht es, eine OSGi-basierte Anwendung durch sogenannte *Installable Units (IUs)* zu erweitern. IUs können durch p2 auch entfernt werden oder ein Update erfahren. Um eine OSGi-basierte Anwendung durch IUs zu erweitern, müssen diese installiert werden. Eine solche Installation wird von p2 überwacht, durchgeführt und kontrolliert. IUs werden auch als *Features* bezeichnet und entsprechen auch dem dieser Arbeit zugrundeliegenden und bereits erläuterten Verständnis eines Features. P2-Features bestehen aus Plug-ins, also OSGi-Bundles, und können andere p2-Features inkludieren. Die Informationen über mögliche p2-Features werden in einem Repository erwartet, das

[2] Equinox Project Website — http://eclipse.org/equinox/

sowohl auf einem entfernten Server als auch im lokalen Dateisystem positioniert sein kann. Eine wichtige Eigenschaft dieser Repositories ist, dass in diesen, Metadaten getrennt von den Artefakten abgespeichert werden. Das bringt den Vorteil mit sich, dass Informationen über die p2-Features aus dem Repository geladen werden können, ohne die OSGi-Bundles zu laden.
P2 ermöglicht es, OSGi-Bundles in Features zusammenzufassen und diese Features in Relation zueinander zu setzen. Auch für die eigentliche Distribution der Features (also das Kopieren der OSGi-Bundles) bieten p2 dem Entwickler einer OSGi-basierten Anwendung Schnittstellen an. Warum die Distribution des HMIServers und dessen Features mit p2 realisiert wurden, erklären die folgenden vier Punkte:

1. *OSGi-Kompatibilität*:
   Zunächst ist p2 für die Installation von Features in einem OSGi-Framework konzipiert und kann durch Verwendung verschiedener OSGi-Bundles in einem solchem schnell und einfach integriert werden. Durch diverse Schnittstellen kann auf die Funktionalität von p2 durch andere OSGi-Bundles zugegriffen werden.

2. *Feature-Mapping*:
   Da der OSGi-Standard keine Features spezifiziert, sondern als Kapselungselement lediglich OSGi-Bundles vorsieht, ist es von entscheidender Bedeutung, dass eine Distributionssoftware Feature-Mapping unterstützt. P2 eignet sich in diesem Punkt sehr gut als Distributionssoftware, da sich mit p2 Features durch OSGi-Bundles definieren lassen und somit eine Abbildung zwischen Features und Implementierungsartefakten erstellt werden kann.

3. *Auflösung von Abhängigkeiten*:
   Neben der Abbildung von Features zu Implementierungsartefakten bietet p2 auch die Möglichkeit, Features in Beziehung zu einander zusetzen. So können mit p2 bspw. Features abhängig von anderen Features sein. Ist z.B. *Feature-A* von *Feature-B* abhängig, so würde bei einer Installation von *Feature-A* von p2 automatisch *Feature-B* mitinstalliert werden. Durch diese Möglichkeit der Abhängigkeitsmodellierung lassen sich durch FeatureModGen hierarchischen Feature-Modelle realisieren. Auch wenn FeatureModGen aktuell auf cross-tree constraints verzichtet, könnten *einfache* cross-tree constraints durch p2 später durch das Hinzufügen von Abhängigkeiten zwischen Features berücksichtigt werden. Ein einfaches cross-tree constraint ist beispielsweise: $FeatureA \rightarrow FeatureB \wedge FeatureC$.

4. *Entwicklungsstand*:
   Abschließend bleibt zu nennen, dass p2 als Teil des Equinox und damit als Teil des Eclipse Projekts einen hohen Verbreitungsgrad aufweist. So wird p2 bspw. genutzt, um die Eclipse IDE durch das Installieren neuer Plug-Ins mit zusätzlicher Funktionalität auszustatten oder durch das Aktualisieren von Plug-Ins auf einem aktuellen Stand zu halten. Ein hoher Verbreitungsgrad ist positiv zu erwähnen, da p2 auch außerhalb des HMIServers eingesetzt wird und Fehler dadurch schneller durch die Open-Source-Gemeinschaft gefunden und behoben werden.

Neben diesen Vorteilen, mussten bei der Verwendung von p2 zur Distribution des HMIServers und dessen Features auch einige Kompromisse eingegangen werden:

1. *Abhängigkeiten zwischen Features*:
   Die Modellierung der Abhängigkeiten zwischen Features erlaubt p2 lediglich in eine Richtung. In Bezug auf das als Feature-Diagramm gezeigte Feature-Modell in Abbildung 5.2 bedeutet dies, dass z.B. für das Feature *Test Tuner* modelliert wird, das *Test Tuner* das Feature *Test Infotainment* benötigt. P2 erlaubt es nicht, bei der Modellierung von *Test Infotainment* anzugeben, dass *Test Tuner* ein Kind-Feature von *Test Infotainment* ist. Da Abhängigkeiten in p2 also immer *vom* Kind- *zum* Eltern-Feature definiert werden, ist während der Definition eines Kind-Features nie bekannt, ob es noch Geschwister-Features hat. Auch während der Definition des Eltern-Features ist die Menge seiner Kind-Features nicht bekannt. Diese Tatsache impliziert, dass p2 es nicht ermöglicht, die aus Feature-Diagrammen bekannten *Und*-, *Oder*- und *Alternativ-Verknüpfungen* zu modellieren.

2. *Proprietäres SPL-Repository*:
   Features werden für ein OSGi-Framework unter Verwendung von p2 aus einem sogenannten p2-Repository installiert. Im wesentlichen handelt es sich dabei um ein SPL-Repository, in dem jedoch lediglich sämtliche Implementierungsartefakte (und nicht der Basiscode der SPL selbst) abgespeichert sind. Das dabei verwendete Format entspricht zwar dem XML-Standard, entspricht oder erweitert jedoch keinen der in Abschnitt 2.2 vorgestellten Standard zu Speicherung von Feature-Modellen. Bei der Verwendung von p2 werden die einzelnen Features mit all ihren Abhängigkeiten zu anderen Features und dem zum Feature gehörigen Feature-Mapping in jeweils einer XML-Datei gespeichert. Für das Feature *Test Tuner* ist eine solche XML-Datei im Anhang in Quelltext B.1 dargestellt.
   Um aus den OSGi-Bundles Features automatisiert zu extrahieren und diese mit Hilfe von p2 in ein OSGi-Framework zu distribuieren, muss FeatureModGen um die Funktionalität ergänzt werden, die erzeugten Features und das erzeugte Feature-Modell im *p2-Feature-XML-Format* abspeichern zu können.

Wegen der genannten Vorteile und trotz der dadurch entstehenden Nachteile, wurde sich im Rahmen dieser Arbeit dafür entschieden, für die exemplarisch Implementierung der Distribution einer SPL, auf p2 zurückzugreifen. Zur vollständigen Automatisierung muss FeatureModGen jedoch noch dahingehend erweitert werden, dass dessen Ergebnis in einem p2-konformen Format abgespeichert werden kann.

5.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Feature-Modelle zur Distribution einer SPL genutzt werden können. Das distribuieren einer SPL bietet sich insbesondere dann an, wenn kompilierte Softwareteile zur Laufzeit nachgeladen werden können, da dann Artefakte einzelner Features nachträglich installiert werden können.
Entscheidend für das Nachladen einzelner Artefakte ist das Feature-Mapping. Erst

durch das Feature-Mapping kann aus der Feature-Auswahl eines Kunden auf die nachzuladenen Implementierungsartefakte geschlossen werden. Die Implementation einer solchen Distributionssoftware muss sicherstellen, dass stets alle benötigten Artefakte im System vorhanden sind. Bei der Deinstallation eines Features muss überprüft werden, welche Artefakte entfernt werden können und welche Artefakte noch von anderen Features benötigt werden und daher nicht entfernt werden dürfen.

6. Evaluierung

In diesem Kapitel wird anhand einer Fallstudie untersucht, inwieweit FeatureMapGen und FeatureModGen eingesetzt werden können, um aus Implementierungsartefakten Feature-Modelle zu extrahieren. Es wird ferner eine Einschätzung gegeben, wie viel Aufwand eines Experten notwendig ist, um die extrahierten Feature-Modelle zur Distribution einer SPL und deren Features an einen Kunden zu ermöglichen.

6.1 Aufbau & Durchführung der Fallstudie

Den Grundstein der durchgeführten Fallstudie bildet der in Abschnitt 3.1 vorgestellte HMIServer der Volkswagen AG. Durch die Modularität des zugrundeliegenden OSGi-Frameworks, kann der HMIServer als Software-Produktlinie betrachtet werden. Durch das Hinzufügen und Entfernen verschiedener OSGi-Bundles entsteht eine Vielzahl an HMIServer-Varianten. Nicht alle Kombinationen an OSGi-Bundles sind lauffähig oder sinnvoll. Abhängigkeiten zwischen den OSGi-Bundles aufzulösen und sie in einem Feature zusammenzufassen, ist Aufgabe von FeatureMapGen. Die dabei extrahierten Features in einem Feature-Modell aufbereitet zu präsentieren, ist Aufgabe von FeatureModGen.
Damit die Abhängigkeiten zwischen den OSGi-Bundles aufgelöst werden können, ist eine gründliche Pflege der Metadaten, schon während der Entwicklung eines OSGi-Bundles zwingend erforderlich. So muss durch den Softwareentwickler, der ein OSGi-Bundle entwickelt, klar definiert werden, welche anderen OSGi-Bundles das von ihm entwickelte OSGi-Bundle benötigt. Wie bereits in Abschnitt 2.4.2 eingeführt, können OSGi-Bundles auf zwei Ebenen voneinander abhängig sein: auf Package- und auf Service-Ebene. Abhängigkeiten auf Package-Ebene ergeben sich aus der Manifest-Datei des jeweiligen OSGi-Bundles. Durch die in der Manifest-Datei geforderten Packages, können andere OSGi-Bundles als erforderlich ermittelt werden. Abhängigkeiten auf Service-Ebene lassen sich jedoch nicht anhand der Manifest-Datei erkennen, sondern ergeben sich lediglich aus dem Quelltext des jeweiligen OSGi-Bundles. Um diese Unzulänglichkeit zu umgehen, wurde in Vorbereitung der Durchführung der Fallstudie Abhängigkeiten auf Service-Ebene *nachgepflegt*. So konnte sichergestellt werden, dass sämtliche Abhängigkeiten zwischen den OSGi-Bundles automatisch

erfasst wurden. Das detektieren einer Abhängigkeit geschah über das Auslesen von Import und Export von Java-Packages aus der Manifest-Datei eines jeden OSGi-Bundles (vgl. Abschnitt 2.4.2). Nicht-optionale[1] Abhängigkeiten auf Service-Ebene wurden als Export bzw. Import eines neuen, nicht existierenden Java-Packages realisiert. Benötigt ein OSGi-Bundle A bspw. ein OSGi-Bundle B, weil B einen bestimmten Service zur Verfügung stellt, den A benötigt, so wurde der Manifest-Datei von B ein Export eines neuen Java-Packages (bspw. *service.a*) hinzugefügt. Gleichzeitig wurde die Manifest-Datei von A um einen Import des gleichen Java-Packages erweitert.
Zur Vorbereitung auf FeatureMapGen und FeatureModGen, wurden sämtliche, für den HMIServer relevanten OSGi-Bundles, manuell in einem Ordner aggregiert. Die Menge dieser OSGi-Bundles ist im Anhang in Tabelle C.1 dargestellt. Im Anschluss konnte gemäß Kapitel 4, aus der Menge an OSGi-Bundles zunächst ein Abhängigkeitsgraph erzeugt werden (siehe Abbildung 6.1). Dieser beinhaltet sowohl Abhängigkeiten zwischen OSGi-Bundles auf Service- als auch auf Package-Ebene. Es wurde weder zwischen Abhängigkeiten verschiedener Ebenen unterschieden, noch wurden Abhängigkeiten unterschiedlich gewichtet. Anschließend wurden mithilfe von FeatureMapGen::NoIn Feature-Kandidaten bestimmt. FeatureMapGen::NoIn wurde gewählt, da dieser für die HMIServer SPL die besten Ergebnis versprach. Aus dem Abhängigkeitsgraphen geht hervor, dass 21 OSGi-Bundles existieren, die keine eingehenden Kanten haben, da sie von keinem anderen OSGi-Bundle benötigt werden. Diese OSGi-Bundles wurden aufgrund der Verwendung von FeatureMapGen::NoIn als Feature-Kandidaten bestimmt. Daraufhin konnte für jeden dieser Feature-Kandidaten ein Feature extrahiert werden. Die dabei entstandenen Features konnten im Anschluss durch FeatureModGen in einem Feature-Modell zueinander in Beziehung gebracht und strukturiert werden.
Die Distribution der dabei extrahierten Features konnte im Anschluss über die in Kapitel 5 erläuterte Technik *p2* prototypisch getestet werden. Dazu war es zuvor jedoch notwendig, die extrahierten Features manuell in das p2-Feature-Format zu überführen. Abschließend konnte das entstandene Feature-Modell Domänenexperten zur Bewertung vorgelegt werden.

[1] Eine nicht-optionale Serivce-Abhängigkeit liegt vor, wenn die Funktionalität eines OSGi-Bundles *nicht* erfüllt werden kann, wenn ein bestimmter Service im OSGi-Framework nicht vorhanden ist.

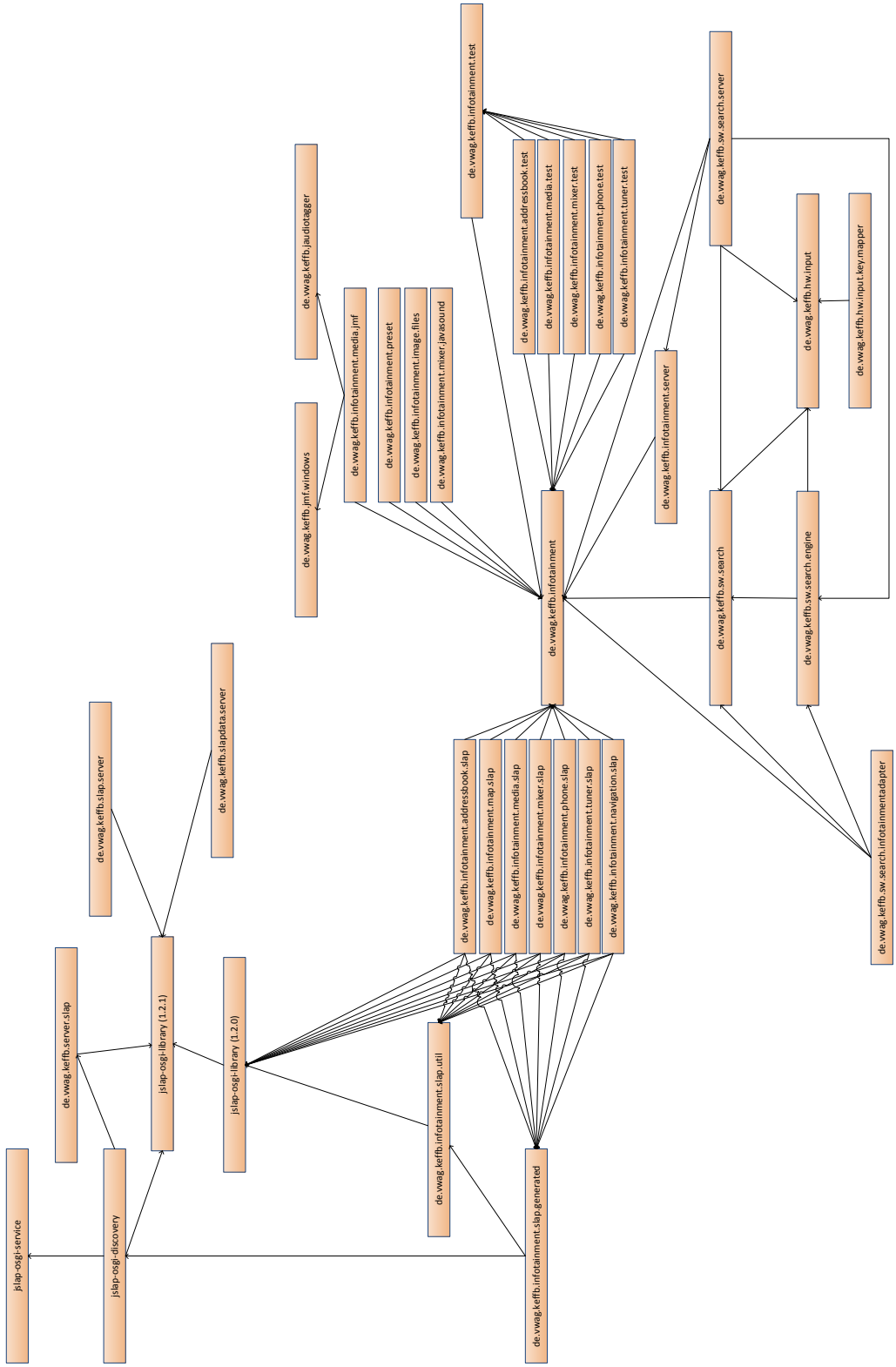


Abbildung 6.1: Abhängigkeitsgraph der OSGi-Bundles aus Tabelle C.1

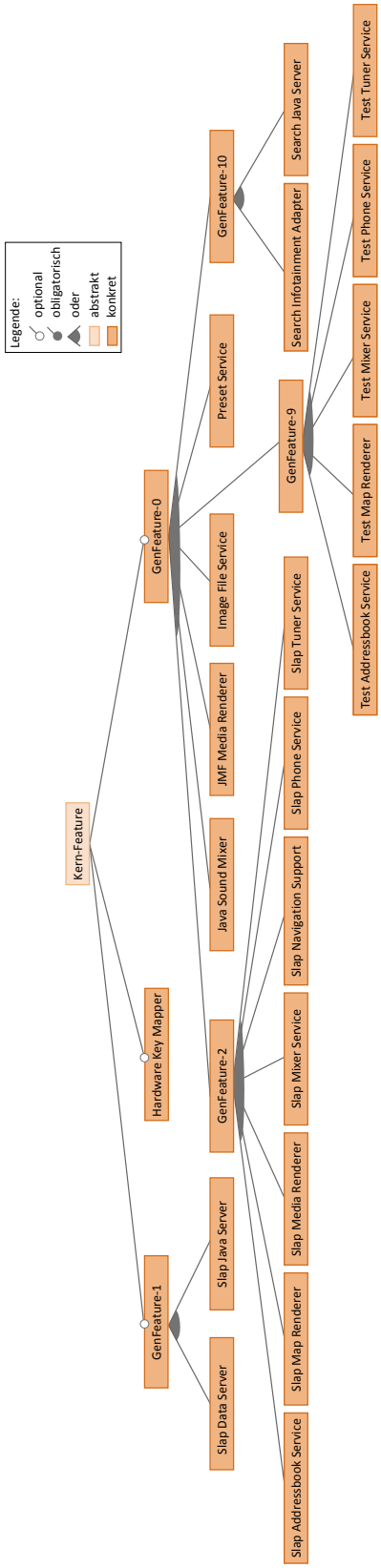


Abbildung 6.2: In der Fallstudie durch FeatureModGen extrahiertes Feature-Modell

Tabelle 6.1: Übersicht der in der Fallstudie extrahierten Features

Feature	Implementierungsartefakt																																		
	⋆.hw.input.key.mapper	⋆.hw.input	⋆.infotainment.addressbook.slap	⋆.infotainment.addressbook.test	⋆.infotainment.image.files	⋆.infotainment.map.slap	⋆.infotainment.media.jmf	⋆.infotainment.media.slap	⋆.infotainment.media.test	⋆.infotainment.mixer.javasound	⋆.infotainment.mixer.slap	⋆.infotainment.mixer.test	⋆.infotainment.navigation.slap	⋆.infotainment.phone.slap	⋆.infotainment.phone.test	⋆.infotainment.preset	⋆.infotainment.server	⋆.infotainment.slap.generated	⋆.infotainment.slap.util	⋆.infotainment.test	⋆.infotainment.tuner.slap	⋆.infotainment.tuner.test	⋆.infotainment	⋆.jaudiotagger	⋆.jmf.windows	⋆.server.slap	⋆.slap.server	⋆.slapdata.server	⋆.sw.search.engine	⋆.sw.search.infotainmentadapter	⋆.sw.search.server	⋆.sw.search	jslap-osgi-discovery	jslap-osgi-library (1.2.0)	jslap-osgi-library (1.2.1)	jslap-osgi-service
Kern-Feature	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Hardware Key Mapper	●	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Addressbook Service	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Test Addressbook Service	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Image File Service	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Map Renderer	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
JMF Media Renderer	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	●	-	-	-	-	-	-	-	-	-	-	-
Slap Media Renderer	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Test Media Renderer	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Java Sound Mixer	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Mixer Service	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Test Mixer Service	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Navigation Support	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Phone Service	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Test Phone Service	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Preset Service	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Tuner Service	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Test Tuner Service	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-
Slap Java Server	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-
Slap Data Server	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-
Search Infotainment Adapter	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-
Search Java Server	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-
GenFeature-0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-
GenFeature-1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-
GenFeature-2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	●	-	-	-	-	-	●	-	-	-	-	-	-	●	●	●	●
GenFeature-9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GenFeature-10	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	●	-	-	-	-

⋆ de.vwag.keffb ● dieses Implementierungsartefakt ist Teil des Features - dieses Implementierungsartefakt ist nicht Teil des Features

6.2 Ergebnisse der Fallstudie

Als primäres Ergebnis der durchgeführte Fallstudie entstand das als Feature-Diagramm in Abbildung 6.2 dargestellte Feature-Modell. Das dazugehörige Feature-Mapping ist in Tabelle 6.1 abgebildet. Durch FeatureMapGen und FeatureModGen entstanden insgesamt 27 Features. Fünf dieser 27 Features, sind durch FeatureModGen neuentstanden und tragen einen *generierten* Namen (*Genfeature-0*, *Genfeature-1*, *Genfeature-2*, *Genfeature-9*, *Genfeature-10*). Die extrahierten Features tragen jeweils die Namen der OSGi-Bundles, die als Feature-Kandidat für die Extraktion des Features verantwortlich waren. Anhand des abstrakten Kern-Features lässt sich erkennen, dass *keine* OSGi-Bundles existieren, die von allen anderen OSGi-Bundles benötigt werden.
Die Ergebnisse der durchgeführten Fallstudie zeigen, dass es möglich ist, aus komplexen Abhängigkeitsstrukturen (vgl. Abbildung 6.1) in einem extraktiven Vorgehen Features und ein Feature-Modell zu extrahieren.

Diskussion der Ergebnisse

Nach der Überführung der extrahierten Features in das p2-Feature-Format, konnte für in der Praxis häufig vorkommende Kombinationen von Features belegt werden, dass die extrahierten Features funktionsfähig sind. Sie konnten mittels p2 installiert werden und im Anschluss reibungslos im OSGi-Framework gestartet werden. Die Bewertung des extrahierten Feature-Modells durch Domänenexperten ergab, dass trotz der erfolgreichen Extraktion eines Feature-Modells, aufgrund der folgenden Punkte nicht davon ausgegangen werden kann, dass das entstandene Feature-Modell einem potenziellen Kunden des HMIServers präsentiert werden kann, ohne dass ein Domänenexperte das Modell zuvor bewertet und bearbeitet hat:

- *Semantische Abhängigkeiten:*
  Selbst unter der Annahme, dass Softwareentwickler alle Abhängigkeiten der von ihnen entwickelten Implementierungsartefakte vollständig (insbesondere auch Abhängigkeiten auf Service-Ebene) pflegen, kann und wird es semantische Abhängigkeiten zwischen Artefakten geben, die ein Softwareentwickler nicht kennen kann, ohne nicht alle anderen Implementierungsartefakte zu kennen.
  Eine solche semantische Abhängigkeit zeigt sich in der durchgeführten Fallstudie bspw. in dem Feature *Java Sound Mixer*. Eine Auswahl dieses Features bzw. des in dem Feature enthaltenen Artefakts *de.vwag.keffb.infotainment.mixer.-javasound* erweitert die SPL HMIServer um die Fähigkeit Soundausgaben auf dem Rechner, auf dem er HMIServer läuft, zu kontrollieren (Lautstärke ändern, Sound aus verschiedenen Quellen zusammenmischen). Dieses Feature auszuwählen und zu installieren erweitert den HMIServer daher tatsächlich nur dann, wenn ein beliebiges anderes Artefakt eine Soundausgabe erzeugt. Dadurch entsteht eine Abhängigkeit, die sich weder aus Metainformationen, noch aus dem Quellcode eines Implementierungsartefakts ableiten lässt. Eine solche Abhängigkeit kann daher nur von einem Experten der Domäne erkannt und berücksichtigt werden.

- *Namen der extrahierten Features:*
  FeatureMapGen extrahiert unter Anderem Features, die aus lediglich einem

Artefakt bestehen. Die so extrahierten Features können daher den Namen des Artefakts erhalten, aus dem sie entstanden sind. FeatureModGen erzeugt im Anschluss an FeatureMapGen Features, die aus mehreren Artefakten bestehen. Der Name der dabei entstehenden Features wird von FeatureModGen auf *GenFeature-X* (wobei X eine fortlaufende, natürliche Zahl ist) festgelegt. In Abhängigkeit der in dem Feature enthaltenden Artefakte, muss ein Experte der Domäne nach der Extraktion des Feature-Modells durch FeatureModGen die generierten Features umbenennen, damit sie Kunden dabei helfen, die Struktur des Feature-Modells zu verstehen. In der durchgeführten Fallstudie könnte *GenFeature-0* so bspw. in *Infotainment* und *GenFeature-9* in *Test Infotainment* umbenannt werden. Enthält das Feature am Ende von FeatureModGen nur ein Artefakt, könnte der Name des Features automatisch auf den Namen dieses einen Artefakts gesetzt werden. Das Feature-Mapping aus Tabelle 6.1 zeigt jedoch, dass generierte Features existieren (z.B. *GenFeature-2*), die aus mehreren Implementierungsartefakten bestehen. Für die Benennung solcher Features wird weiterhin das Wissen eines Domänenexperten benötigt.

- *Freies Gruppieren von Features:*
  FeatureModGen extrahiert basierend auf den Abhängigkeiten zwischen Implementierungsartefakten ein Feature-Modell. Würde ein Domäneneexperte Feature-Modell und Feature-Mapping manuell erstellen, würde er Features so anordnen, wie es seiner Meinung nach am sinnvollsten scheint. Dazu zählt auch, Features nach ihrem Einsatzzweck zu ordnen. In der durchgeführten Fallstudie zeigt sich dies z.B. an dem Feature *JMF Media Renderer*. Dieser wird überwiegend zu Testzwecken eingesetzt und würde daher, aus Sicht des Experten, als Kind-Feature des Features *GenFeature-9* (bzw. *Test Infotainment* nach einer Umbenennung) angeordnet werden.
  Wird dem Experten die Möglichkeit gegeben, das Feature *JMF Media Renderer* zu verschieben, muss der Algorithmus, der ein Feature verschiebt, darauf achten, dass alle Artefakt-Abhängigkeiten weiterhin erfüllt werden können und, dass keine neuen *Pseudo-Abhängigkeiten* erzeugt werden. Eine Pseudo-Abhängigkeit entsteht z.B. durch ein einfaches Verschieben von *JMF Media Renderer* unter *GenFeature-9*. Solch einfache Verschiebung würde implizieren, dass beim Installieren von *JMF Media Renderer* auch *Genfeature-9* mit installiert werden muss. Das wiederum würde bedeuten, dass das Implementierungsartefakt *de.vwag.keffb.infotainment.test* dem System hinzugefügt wird. Dieses Artefakt ist jedoch für die Ausführung des Features *JMF Media Renderer* nicht notwendig. Das Verschieben erzeugt daher eine Abhängigkeit zwischen dem Feature *JMF Media Renderer* und dem Artefakt *de.vwag.keffb.infotainment.test*, die es eigentlich nicht geben müsste.
  Eine Möglichkeit dies zu Umgehen wäre es, vor dem Verschieben des Features *JMF Media Renderer*, alle Artefakte aus *GenFeature-9* zu entfernen und zu all seinen Kind-Features hinzuzufügen. Wird das Feature *JMF Media Renderer* nun verschoben, ist *GenFeature-9* abstrakt, wodurch bei dessen Installation dem System keine Artefakte hinzugefügt werden.

Aufgrund der genannten Punkte kann, ausgehend von einem durch FeatureModGen extrahierten Feature-Modell, in einigen wenigen Schritten durch einen Domänenex-

perten ein Feature-Modell erstellt werden, das einem potenziellen Endkunden der SPL zur Installation von Features angeboten werden kann. In der durchgeführten Fallstudie müssten ca. 19% der Features durch einen Experten umbenannt werden, damit ein Kunde die Funktionalität des Features an dessen Namen erkennen kann. Das Verschieben von einzelnen Features wäre nach Aussagen der Experten nicht zwingend erforderlich. Zwar hätten sie die Features anders angeordnet als FeatureModGen, das extrahierte Feature-Modell kann aus Sicht der Experten dennoch genutzt werden, um Kunden Features installieren zu lassen.
Die Anzahl an Entscheidungen, die ein Kunde beim Installieren von Features treffen muss, konnte in der gezeigten Fallstudie um 25% reduziert werden. Anstatt aus 36 OSGi-Bundles muss der Kunde nun nur noch aus 27 Features wählen. Hinzu kommt, dass die Features, im Vergleich zur Menge der OSGi-Bundles, einer Struktur folgen, die das Auswählen von Features zusätzlich erleichtert.

Im Vergleich zum manuellen Erstellen von Features und Feature-Modell, sind FeatureMapGen und FeatureModGen nach Aussagen der Experten eine große Unterstützung. Würden die Experten Features manuell extrahieren müssen, könnte nicht sichergestellt werden, dass keine Abhängigkeiten zwischen Implementierungsartefakten übersehen werden und somit ein falsches Feature-Modell extrahiert werden würde. FeatureMapGen und FeatureModGen helfen daher auch dabei Fehler zu vermeiden.

7. Verwandte Arbeiten

In diesem Kapitel werden sämtliche Aspekte dieser Arbeit in Hinblick auf bereits bestehende Arbeiten oder Werkzeuge untersucht. Das Kapitel gliedert sich in die folgenden Abschnitte: Abschnitt 7.1 erläutert zunächst andere modulare Systeme, die die Entwicklung einer SPL als Framework ermöglichen. In Abschnitt 7.2 werden andere Arbeiten vorgestellt, die sich bereits mit der Abbildung zwischen Implementierungsartefakten und Features auseinandergesetzt haben. Abschnitt 7.3 zeigt alternative Feature-Diagramme auf, die benutzt werden können, um Features weniger restriktiv in Relation zu einander zu setzen. Abschnitt 7.4 erläuert andere Arbeiten, in denen sich mit der Extraktion von Feature-Modellen auseinandergesetzt wurde und setzt die vorliegende Arbeit in den entsprechenden Bezug dazu. Abschließend werden in Abschnitt 7.5 Systeme vorgestellt, die es ermöglichen SPLs zu distribuieren.

7.1 Modulare Softwaresysteme

Das OSGi-Framework stellt ein verbreitetes Modulsystem für Java dar, das in den letzten Jahren immer mehr Beliebtheit erlangte.[1] Die unter Softwareentwicklern bekannte Entwicklungsumgebung *Eclipse*, basiert auf einem OSGi-Framework namens *Equinox*, das, wie auch Eclipse selbst, von der *Eclipse Foundation* entwickelt wird [MVA10]. Nach Angaben der Eclipse Foundation wird Equinox derzeit auf „Millionen von Rechnern“ eingesetzt [Ecl08]. Zu den bekanntesten Firmen, die Equinox einsetzen, zählen die IBM Corporation, die Oracle Corporation, die NASA sowie die Deutsche Post [Ecl08]. Trotz des großen Erfolgs des OSGi-Frameworks soll an dieser Stelle geklärt werden, welche Alternativen zu OSGi existieren.

Individuelle Frameworks

Eine mögliche Alternative ist, das die Module umschließende Framework in eigener Regie zu entwickeln. Findet eine solche Entwicklung in der Programmiersprache Java statt, besteht z.B. die Möglichkeit ein bestimmtes Verzeichnis nach JAR-Dateien

[1] Über die Verbreitung von OSGi — http://heise.de/-842445

zu durchsuchen und die in den JAR-Dateien befindlichen Klassen und Interfaces zu importieren. Das Nachladen solcher Klassen und Interfaces übernimmt ein so genannter *Class-Loader*. Die Entwicklung eines eigenen *Class-Loaders* kann aufwendig werden, da es nicht ausreicht die Klassen zu laden. Sie müssen auch ihrer Funktion entsprechend koordiniert und für andere Klassen ansprechbar gemacht werden. Soll kein allgemeingültiges Framework entwickelt werden liegt der Vorteil darin, dass die Komplexität des *Class-Loaders* reduziert werden kann, da er auf eine bestimmte Domäne zugeschnitten ist und somit Mehraufwand vermieden werden kann, der in einem allgemeingültigen Framework sowohl zur Entwicklungs- als auch zur Laufzeit entstehen würde. Mehraufwand während der Entwicklung entsteht z.B. dadurch, dass vom Entwickler zunächst ein Grundgerüst konzipiert und implementiert werden muss, das es dem *Benutzer* des Frameworks (ein Softwareentwickler, der das zuvor entwickelte Framework nutzt) ermöglicht, die konkreten Probleme in einer Domäne zu lösen. Erhöhte Komplexität des Programmcodes führt schließlich auch zu erhöhten Laufzeiten des Programms. Maßgeschneiderte Frameworks sind daher allgemeinen Frameworks gegenüber im Vorteil, wenn es um Laufzeitumgebungen geht, in denen es auf effiziente Speicher- und Zeitnutzung ankommt (z.B. in Echtzeit- oder eingebetteten Systemen).

Project Jigsaw

Ein ebenfalls modulares Softwaresystem wird derzeit unter dem Namen *Project Jigsaw* entwickelt.[2] Anders als Modulsysteme die auf eine vorhandene Java Platform aufsetzen, verfolgt Project Jigsaw das Ziel, ein Modulsystem direkt in die Java Platform zu integrieren. Das ursprüngliche Ziel, durch dieses Modulsystem lediglich das Java Development Kit (JDK) zu modularisieren, wurde inzwischen dahingehend erweitert, das Modulsystem als Standard in die Java Platform zu integrieren. Das bedeutet, dass dieses Modulsystem allen Java-Entwicklern in ihren Applikationen zur Verfügung stehen wird.
Ein entscheidender Konzeptunterschied zwischen OSGi und Project Jigsaw betrifft die Auflösung von Modulabhängigkeiten. Während die Abhängigkeiten zwischen OSGi-Bundles im OSGi-Framework zur Laufzeit aufgelöst werden, werden Abhängigkeiten zwischen Modulen in Jigsaw zur Installationszeit aufgelöst. Module, die Abhängigkeiten zu anderen, nicht im System installierten Modulen besitzen, können nicht im System installiert werden. Ein OSGi-Bundle könnte im Vergleich dazu installiert, aber zur Laufzeit nicht gestartet werden.
Ein weiterer konzeptioneller Unterschied zwischen OSGi und Project Jigsaw ist die Tatsache, wie Module definiert werden. Im OSGi-Standard ist vorgesehen, dass OSGi-Bundles immer über eine separate Manifest-Datei verfügen, die sämtliche Metainformationen zum entsprechenden OSGi-Bundle in einer eigenen Syntax bereitstellt. In Jigsaw werden Module in einer Datei namens `module-info.java` definiert. Moduldefinitionen finden auch in separaten Dateien statt, orientieren sich allerdings an der Java-Syntax. Ein Minimalbeispiel zur Jigsaw-Syntax ist in Quelltext 7.1 dargestellt.
Das bereits vor Beginn dieser Arbeit in der Konzernforschung der Volkswagen AG verwendete OSGi-Framework deckt sämtliche Anforderungen an die Modularität des HMIServers ab. Die Entwicklung eines eigenen Frameworks erscheint daher überflüssig und wurde nicht angestrebt. Der Einsatz von Project Jigsaw ließe sich derzeit

[2] Project Jigsaw Website — http://openjdk.java.net/projects/jigsaw/

```
17 module foo {
18     exports foo;
19     exports foo.spi;
20     exports foo.util; }
```

Quelltext 7.1: Beispiel einer Moduldefinition in Jigsaw-Syntax

nur schwer umsetzen, da es sich aktuell in einer prototypischen Phase befindet, in der Ideen und Konzepte entwickelt werden. Jigsaw soll nach Angaben von Oracle in das JDK 8 integriert werden.[3] Die offizielle Referenzimplementierung des JDK – das OpenJDK[4] – ist zur Veröffentlichung Ende 2013 geplant.[5] Aktuelle Vorabversionen sind zwar verfügbar,[6] eignen sich jedoch nicht für den produktiven Einsatz. Ob und inwieweit die in dieser Arbeit gestellten Anforderungen an ein Modulsystem durch Jigsaw erfüllt werden, soll an dieser Stelle nicht weiter untersucht werden, da eine Überführung des bestehenden, auf OSGi basierenden Systems, in Jigsaw, mit erheblichem Mehraufwand verbunden wäre. Für die Neuentwicklung modularer Java-Systeme lohnt sich – nach Veröffentlichung des JDK 8 – sicher ein Blick auf Jigsaw.

7.2 Abbildung zwischen Features und Implementierungsartefakten

Im Grundlagenkapitel wurde bereits erläutert, dass ein Feature aus mehreren Implementierungsartefakten zusammensetzt und dass ein Implementierungsartefakt wiederum Bestandteil mehrerer Features sein kann. Mit dieser Tatsache beschäftigten sich in der Vergangenheit auch andere Arbeiten. So wurde das *Colored Integreated Development Environment (CIDE)* [KTA08] bspw. entwickelt, um die Variabilität einer SPL beim Bearbeiten des Quelltextes zu visualisieren. Durch das Markieren bestimmter Elemente des Quelltextes mit einer bestimmten Farbe, werden diese Elemente einem mit der jeweiligen Farbe assoziierten Feature zugeordnet. Diese Elemente können als Implementierungsartefakte verstanden werden. Ein Implementierungsartefakt, der durch CIDE markiert werden kann, ist ein Element aus dem *Abstract Syntax Tree (AST)* des Quelltextes der SPL. Ein Implementierungsartefakt kann daher eine Klasse, eine Methode, ein bestimmte Anweisung oder ein Parameter sein [KTA08]. Durch das Markieren mehrerer Implementierungsartefakte mit der selben Farbe, kann ein Feature aus mehreren Implementierungsartefakten zusammengesetzt sein. Ein Implementierungsartefakt kann auch mit CIDE mehreren Features zugeordnet werden, indem das entsprechende Artefakt mit mehreren Farben gekennzeichnet wird. Dabei werden die Farben, mit denen der Implementierungsartefakt markiert wurde, miteinander gemischt [Käs07].
Auch bei der Analyse zweier großer Betriebssysteme (Linux und FreeBSD) wurde

[3] Oracle JDK Roadmap — http://www.oracle.com/us/corporate/press/173782

[4] OpenJDK Website — http://openjdk.java.net/projects/jdk8/

[5] OpenJDK Mailinglisten-Eintrag
— http://mail.openjdk.java.net/pipermail/jdk8-dev/2012-April/000942.html

[6] JDK 8 Download-Website — http://jdk8.java.net/download.html

eine Abbildung zwischen Features und Implementierungsartefakten verwendet, um Elemente des AST bestimmten Features zuzuordnen [BSL+10]. Dabei wurde für jedes Implementierungsartefakt eine Existenzbedingung in Form eines aussagenlogischen Ausdruckes verwendet, der genau dann wahr wurde wenn das entsprechende Artefakt aufgrund einer bestimmten Wahl an Features Teil einer Variante sein sollte [BSL+10]. Das Implementierungsartefakt I_1 wird bei folgendem Ausdruck bspw. genau dann ausgewählt, wenn die Feature F_1 und F_2, nicht aber F_3 ausgewählt wurden: $I_1 \Leftrightarrow F_1 \wedge F_2 \wedge \neg F_3$.

7.3 Alternative Feature-Diagramme

In Kapitel 2 wurden Feature-Modelle und eine graphische Repräsentation dieser durch Feature-Diagramme eingeführt. Für Feature-Diagramme gilt, dass Features immer nur genau ein Eltern-Feature haben dürfen. Es existieren jedoch auch Ansätze, in denen ein Feature mehreren Eltern-Features zugewiesen sein kann [SHTB07, CW07]. Die Baumstruktur des Feature-Diagramms kann dadurch nicht länger aufrecht erhalten werden und das Feature-Diagramm wird zu einem gerichteten, azyklischen Graphen [Thü08]. Diese Möglichkeit ist vor allem dann interessant, wenn es aufgrund einer logischen Sortierung von Features erforderlich wird, ein bestimmtes Features unterhalb verschiedener anderer Features anzuordnen.
Eine noch flexiblere Möglichkeit Feature-Diagramme zu erstellen besteht darin, die Beziehungen zwischen Eltern- und Kind-Features zu lockern. So existieren bspw. Ansätze, bei der die Auswahl eines Features nicht die Auswahl des jeweiligen Eltern-Features bedingt [SLB+10]. Dadurch können Features nicht nur nach ihren logischen Zusammenhängen gruppiert werden, sondern können auch einer Sortierung folgen, die sich nicht direkt aus den Features und deren Abhängigkeiten untereinander ergeben.

7.4 Extraktion von Feature-Modellen

In [CW07] wurden bereits theoretische Probleme und Herausforderungen bei der Extraktion von Feature-Modellen analysiert. So den wesentlichen Erkenntnissen dabei zählen *a*) dass es niemals nur ein korrektes Feature-Modell geben kann, *b*) dass nicht alle Abhängigkeiten zwischen automatisch extrahierten Features für den Nutzer des Feature-Modells wertvoll sind und daher nicht alle logischen Abhängigkeiten im Feature-Modell abgebildet werden sollten und *c*) dass es Abhängigkeiten zwischen extrahierten Features geben wird, die keiner logischen Abhängigkeit entspringen, sondern sich aus einer Semantik ergeben, die nur ein Experte der Domäne kennen kann. Diese theoretischen Probleme konnten in der durchgeführten Fallstudie dieser Arbeit bestätigt werden.
Ebenfalls theoretisch wurde gezeigt, dass die Extraktion von Feature-Modellen, ausgehend von einer Menge an Varianten möglich ist. So ist es bspw. gelungen, aus einer Menge verwandter Produkte, eine SPL zu extrahieren [RC12], indem für jede Variante, aus der Menge an Varianten die zur Extraktion der SPL genutzt wurden, ein Feature erzeugt wurde. Als Ergebnis entstand ein Feature-Modell, in dem die einzelnen Features in einer Alternativ-Verknüpfung miteinander verknüpft sind. Daraus folgt, dass die entstandene SPL nur die Varianten erzeugen kann, die zuvor bereits ein Produkt gewesen sind. Es können anhand des extrahierten Feature-Modells keine

neuen Varianten erzeugt werden.
In jüngster Vergangenheit gab es Bestrebungen, aus den Beschreibungen verschiedener Produkte einer Produktfamilie, Feature-Modelle zu extrahieren [ACP+12, HLHE11]. Dabei wurde aus einer Menge an Produkten und einer tabellarischen Beschreibung dieser Systeme ein einfaches Feature-Modell extrahiert. Dieses Feature-Modell hilft jedoch lediglich dabei, die Features einer bestimmten Produktfamilie zu verstehen. Was im Vergleich zu den vorgestellten Algorithmen und Konzepten dieser Arbeit fehlt, ist eine Abbildung zwischen diesen Features und den Implementierungsartefakten der einzelnen Produkte.
Es wurde auch gezeigt, dass ordnungtheoretische Konzepte wie bspw. die *Formale Begriffsanalyse* [GW96], dazu genutzt werden können, um aus einer Menge an Varianten, Beziehungen zwischen Features aufzudecken [LP07]. Dadurch können bspw. Mengen an Features detektiert werden, die (in einer bestimmten Menge an Varianten) immer gemeinsam auftreten. Es kann auch erkannt werden, wenn ein Feature in keinen der bekannten Varianten vorkommt. Solche Feststellungen können dann zurück in das (bereits bestehende) Feature-Modell der analysierten SPL integriert werden und dieses dadurch vereinfachen.
Anhand einer Fallstudie [ACC+11] wurde aufgezeigt, dass das Extrahieren eines Feature-Modells aus einer Implementation des *SCA Standards*[7] und Plug-ins dieser Implementation zwar möglich ist, das Ergebnis jedoch nicht mit einem manuell erstellten Feature-Modell eines Domänenexperten äquivalent sein wird. Die vorliegende Arbeit bestätigt diese Ergebnisse, wählt jedoch ein anderen Ansatz um automatisch und manuell extrahierte Feature-Modelle zu vereinen: während in [ACC+11] das manuell erstellte Feature-Modell mit dem automatisch extrahierten Feature-Modell verglichen wird, um ein endgültiges Feature-Modell zu generieren, basieren Algorithmen und Konzepte dieser Arbeit darauf, das automatisch extrahierte Feature-Modell durch einen Domänenexperten editieren zu lassen.

7.5 Distribution von Software-Produktlinien

Soll ein konkretes Produkt einer SPL zum Einsatz beim Kunden kommen, stellt sich die Frage, wie man aus den Features und deren, im Feature Modell festgehaltenen, Abhängigkeiten ein finales, lauffähiges Produkt erstellt oder generiert und wie dieses Produkt schließlich beim Kunden installiert wird.
Dieses Kapitel zeigt verschiedene Möglichkeiten, eine SPL zu distribuieren. Die Möglichkeiten der Distribution hängen stark von den Techniken ab, die verwendet wurden, um die SPL zu entwickeln.
Nutzt man bspw. die bereits in Kapitel 2 erwähnten annotativen Techniken, wie z.B. Präprozessoren, müssen zur Generierung eines bestimmten Produkts lediglich die entsprechend gewünschten Parameter aktiviert bzw. deaktiviert werden und das Programm kompiliert werden. Das kompilierte Programm unterscheidet sich nach der Erstellung nicht von *klassischen* Software-Produkten, die nicht aus einer SPL heraus erstellt oder generiert wurden.
Wurde zur Entwicklung der SPL auf die objektorientierte Technik des Frameworks gesetzt, wurden die einzelnen Bestandteile bereits kompiliert, bevor ein Kunde ausgewählt hat, welche Features er gern in seinem Produkt nutzen möchte. Die,

[7] Software Component Architecture (SCA) Standard – http://www.oasis-opencsa.org/

sich daraus ergebende, Fragestellung lautet daher, wie die einzelnen Bestandteile eines Frameworks – die Plug-Ins – den Wünschen des Kunden entsprechend, an den Kunden verteilt werden können.

7.5.1 Software-Produktlinien manuell distribuieren

Dass das manuelle verteilen einer SPL entweder viel Expertenwissen auf Seiten des Kunden voraussetzt oder sehr zeitintensiv für den Distributor ist, zeigte bereits Abschnitt 3.1.4. Für SPLs mit einer geringen Anzahl an Features und einer geringen Komplexität der Abhängigkeiten zwischen den Features, kann eine manuelle Distribution einer SPL dennoch sinnvoll sein. Der Nachteil des intensiven Kundenkontakts wird hierbei unmittelbar zum Vorteil: direkter Kundenkontakt ermöglicht es dem Distributor zu erleben, wie das vom Kunden gewünschte Produkt der SPL eingesetzt wird und welche funktionalen Erweiterungen dem Kunden helfen würden, ein bestimmtes Problem noch besser zu lösen. Die aus solch einem Erlebnis gewonnene Erfahrung kann dann dazu genutzt werden, bestehende Features weiter zu entwickeln oder neue Features zu entwerfen.
Für SPLs mit der Größe (Größe hier gemessen an der Anzahl der Features) des HMI-Servers, ist eine manuelle Distribution jedoch aus den in Abschnitt 3.1.4 genannten Gründen nicht zu empfehlen.

7.5.2 Paketverwaltungssysteme

Ein Betriebssystem kann aufgrund der Tatsache, dass einzelne Applikationen nachinstalliert werden können, auch als SPL betrachtet werden. Applikationen in Linuxdistribution werden z.B. durch sogenannte Paketverwaltungssysteme verwaltet und bestehen aus mehreren Komponenten. Die Komponenten wiederum besitzen Abhängigkeiten zu anderen Komponenten. An dieser Stelle lassen sich daher starke Parallelen zwischen der Distribution einer Linuxdistributon und der Distribution des HMIServers ziehen. Tabelle 7.1 zeigt die für diese Arbeit wesentlichen Unterschiede und Gemeinsamkeiten zwischen Paketverwaltungssystemen (hier die Linuxdistribution Debian) und dem HMIServer:
Gemäß Tabelle 7.1 unterscheiden sich Paketverwaltungssysteme, neben der Begrifflichkeiten, lediglich darin, dass ein Paketverwaltungssystem in der Lage sein muss, mit deutlich mehr Komponenten umgehen zu können, als der HMIServer es muss. Neben den in der Tabelle aufgezeigten Gemeinsamkeiten und Unterschieden, gibt es noch weitere Aspekte, die ein Paketverwaltungssystem zur Distribution eines Betriebssystems berücksichtigen muss, die bei der Distribution des HMIServers eine untergeordnete Rolle spielen, für die Distribution anderer SPLs jedoch zu beachten sind:

Verfügbarkeit & Last

Um ein Betriebssystem und zusätzliche Applikationen an eine große Menge von Endkunden zuverlässig zu verteilen, muss sichergestellt werden, dass das Repository, aus dem Applikationen und Komponenten geladen werden, hochverfügbar ist. Häufigere oder lang andauernde Ausfälle des Repositories würden einen nicht unerheblichen Imageschaden nach sich ziehen. Paketverwaltungssysteme versuchen diese hohe Verfügbarkeit durch redundante Repositories zu lösen, sodass nicht nur ein Repository

Kriterium	Debian Paket-verwaltung	HMIServer	Bemerkung
Bezeichnungen	Applikation	Feature	—
	Komponente	Bundle	—
Anzahl der Komponenten	35.309[8]	ca. 150	Paketverwaltungs-systeme müssen deutlich besser skalieren als der HMIServer
Inhalt der Komponenten	ausführbarer Binärcode[9]	interpretierbarer Java-Bytecode	in beiden Systemen kann der Code direkt auf dem System aus-geführt werden
Definition von Abhängigkeiten	durch *control information files* in der jeweiligen Komponente	durch *MANIFEST-Dateien* im jeweiligen Bundle	in beiden Systemen sind Abhängigkeiten dezentral definiert

Tabelle 7.1: Vergleich von Paketverwaltungssystemen und dem HMIServer

existiert, aus dem Applikationen und Komponenten geladen werden können, sondern mehrere, identische Repositories existieren. Beim Ausfall eines Repositories kann dann auf eines der anderen zurückgegriffen werden.
Neben der erhöhten Verfügbarkeit kann durch redundante Repositories auch die Last auf die einzelnen Repsoitories verringert werden. Sind die Repositories weltweit verteilt, ergibt sich eine bessere Verteilung der Datenlast an den Internetknoten. Was geschieht, wenn viele Menschen eine bestimmte Datei von einem bestimmten Server laden wollen, zeigt Abbildung 7.1. Dargestellt ist der Datendurchsatz des Berliners Internetknoten *Berlin Commercial Internet Exchange* vom 12. Oktober 2011, als Apple Inc. um 19 Uhr deutscher Zeit *iOS 5* veröffentlichte. Grafik entsprechend einem Online-Artikel aus *The Guardian* vom 13. Oktober 2011[10].
Sowohl eine hohe Verfügbarkeit, als auch eine hohe Last, werden für den HMIServer keine Probleme sein, für die es einer Lösung bedürfen wird, da weder eine hohe Verfügbarkeit notwendig noch ein hohes Lastaufkommen realistisch ist. Für andere, vor allem öffentlich verfügbare SPLs sind die Verfügbarkeit und die Reduzierung des Lastaufkommens jedoch ein wichtiges Thema der Distribution.

[8] Auflistung aller *stabilen* Debian Pakete — http://packages.debian.org/stable/allpackages

[9] Debian Policy Manual — http://www.debian.org/doc/debian-policy/index.html

[10] iOS 5 update causes massive internet traffic spike - to users' frustration — http://www.guardian.co.uk/technology/2011/oct/13/ios-5-update-internet-traffic-spike

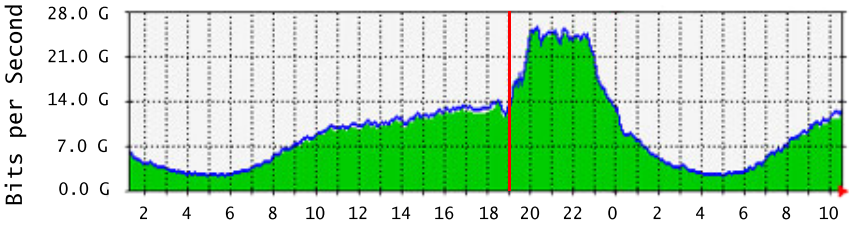


Abbildung 7.1: Durchsatz am *Berlin Commercial Internet Exchange* während der Veröffentlichung von Apples iOS 5

Sicherheit

Ein weiterer wichtiger Aspekt bei der Verteilung von Betriebssystemen ist die Sicherheit. Es muss zum einen stets sichergestellt sein, dass Komponenten des Betriebssystems oder Applikationen auf dem Weg vom Repository zum Zielrechner nicht durch Dritte verändert werden. Die Möglichkeit Komponenten während der Übertragung zu verändern könnte dann ausgenutzt werden, um Schadcode in die zu verteilenden Komponenten einzufügen. Verhindert werden kann solch ein Angriff z.B. durch *HTTPS*- anstelle von *HTTP*-Kommunikation. Zum anderen muss verhindert werden, dass die Repositories durch Angriffe von außen geschützt werden. Angreifer könnten so versuchen Komponenten im Repository zu verändern und mit Schadcode zu versehen. Verhindert werden könnten solche Angriffe z.B. durch restriktiv konfigurierte Firewalls, die die Datenverbindung von und zum Repository überwacht und Angreifer entsprechend abweist. Die zu verteilenden Komponenten können auch mit einer verifizierten Signatur versehen werden. Damit wäre sichergestellt, dass der Inhalt einer Komponente nicht verändert werden könnte (ohne die Signatur zu verändern). Die Signaturen könnten abschließend noch von einer zentralen Stelle als vertrauenswürdig gekennzeichnet werden. Somit könnten nur Komponenten installiert werden, die aus vertrauenswürdiger Quelle stammen und weder im Repository, noch auf dem Weg zum Zielrechner verändert wurden.

8. Fazit & Ausblick

Features und Feature-Modelle können genutzt werden, um eine Software-Produktlinie in, für den Menschen verständliche, Bestandteile zu zerlegen. Die dabei entstehenden Bestandteile können dann in verschiedenen Kombinationen dazu genutzt werden, eine Vielzahl an Softwarevarianten zu erzeugen.

Mit dieser Arbeit wurde gezeigt, dass das Extrahieren von Features, Feature-Mappings und Feature-Modellen aus den Implementierungsartefakten einer bestehenden SPL möglich ist und Experten der Domäne dabei helfen kann, eine bessere Übersicht über die Abhängigkeiten innerhalb ihrer SPL zu bekommen. Das extrahierte Feature-Modell kann mit geringem Aufwand von einem Experten so angepasst werden, dass es zur Kommunikation mit einem Kunden der SPL genutzt werden kann.
Es wurde auch gezeigt, dass die Distribution einer SPL anhand der angepassten Feature-Modelle vorgenommen werden kann. Dazu ist es erforderlich, dass sowohl die Struktur, die Namen als auch die Beschreibungen der Features so gewählt werden, dass Endkunden in der Lage sind, eigenständig zu entscheiden, welche Features sie installieren müssen, um eine gewünschte Funktionalität zu erzielen.

Die Ergebnisse der durchgeführten Fallstudie bestätigten auch die zu Beginn aufgestellte These, dass die Extraktion eines Feature-Modells kein vollautomatischer Prozess sein kann, sondern, dass das extrahierte Feature-Modell anschließend von einem Experten bewertet und editiert werden sollte. Die Änderungen, die der Experte dabei am Feature-Modell vornehmen muss, beschränken sich – zumindest in dieser Fallstudie – auf das Ändern von Namen der Features und das Hinzufügen einiger, weniger, semantischer Abhängigkeiten. Insofern unterstützen die vorgestellten Algorithmen den Experten beim extraktiven Erstellen von Features, Feature-Mappings und Feature-Modellen.

Es wurde auch gezeigt, dass bestehende Techniken genutzt werden können, um eine SPL zu distribuieren. Wie eine SPL distribuiert werden kann, hängt stark von der bei der Implementation der SPL verwendeten Technik ab. Zeichnet sich eine SPL dadurch aus, dass nach der Kompilierung Softwareteile nachgeladen werden

können (z.B. durch das Nachladen von OSGi-Bundles), kann die SPL durch Techniken wie *p2* distribuiert werden. Wurde die SPL hingegen mit Techniken implementiert, die Features während der Kompilierung zu einer Variante zusammenfügen (wie z.B. *feature-* oder *aspektorientierte* Techniken), muss die SPL nach der Auswahl der Features für den Kunden kompiliert werden.

Ausblick

Um die Ergebnisse der durchgeführten Fallstudie zu stützen, müssen FeatureMapGen und FeatureModGen auf Abhängigkeitsstrukturen größerer SPLs angewendet werden. Zur Evaluierung der dabei entstehenden Feature-Modelle, müssen die jeweiligen Experten der SPL befragt werden, wie gut das extrahierte Feature-Modell ihrer Meinung nach die Variabilität ihrer SPL widerspiegelt. Im Anschluss muss während einer Beobachtung dokumentiert werden, wie viele Änderungen Experten an dem extrahierten Feature-Modell vornehmen müssen, um zu einem Ergebnis zu kommen, das es auch Nicht-Experten erlaubt, die Features einer SPL und deren Struktur zu verstehen.

Es muss auch evaluiert werden, ob FeatureMapGen::NoIn die geeignetsten Feature-Kandidaten extrahiert, oder ob eine der anderen FeatureMapGen-Varianten, trotz dem erforderlichen Aufwand der Parametrisierung, Feature-Kandidaten ermitteln kann, die zu Features führen, die die Variabilität der SPL noch besser repräsentieren.

Das in Abschnitt 4.3 vorgestellte Werkzeug zur Durchführung von FeatureMapGen und FeatureModGen kann an einigen Punkten erweitert werden. So könnten neben FeatureMapGen::NoIn auch die anderen FeatureMapGen-Varianten implementiert werden. Außerdem könnte das Exportieren des Feature-Modells auf andere maschinenlesbare Beschreibungsformen (vgl. Abschnitt 2.2) erweitert werden. In weiteren Arbeiten könnten auch anderen Algorithmen zur Bestimmung der Feature-Kandidaten gefunden werden, wodurch noch weniger Nacharbeit eines Experten am entstehenden Feature-Modell nötig wäre.

A. Aufwandsabschätzungen

A.1 FeatureMapGen::All

Das Durchführen einer Tiefensuche erzeugt bei n Knoten und k Kanten zwischen diesen Knoten, eine Komplexität von $\mathcal{O}(n+k)$. Da in einem gerichteten Graphen mit n Knoten und ohne doppelte Kanten maximal $n \cdot (n-1)$ Kanten existieren, lässt sich die Komplexität auf $\mathcal{O}(n+(n\cdot(n-1)))$ bzw. gemäß Formel A.1 auf $\mathcal{O}(n^2)$ vereinfachen.

$$\begin{aligned} n + (n \cdot (n-1)) &= n^2 && \mid -n \\ n \cdot (n-1) &= n \cdot n - n && \mid : n \\ n - 1 &= n - 1 && \mid +1 \\ n &= n && q.e.d. \end{aligned} \tag{A.1}$$

Wird für jeden der n Knoten des Graphen eine Tiefensuche durchgeführt, entsteht dadurch eine Komplexität von $\mathcal{O}(n^3)$.

A.2 FeatureMapGen::NoIn

Es sei angenommen, ein Graph enthält n Knoten. Davon haben $0 \leq m \leq n$ Knoten *keine* eingehenden Kanten. Wird eine Tiefensuche bei einem Knoten ohne eingehende Kanten gestartet, können bei dieser Tiefensuche alle Kanten zwischen den Knoten mit mindestens einer eingehenden Kante $((n-m) \cdot (n-m-1))$ und die Kanten vom Startknoten zu allen Knoten mit mindestens einer eingehenden Kante $((n-m))$ besucht werden. Formel A.2 zeigt, dass bei einer von einem Knoten ohne eingehende Kanten gestarteten Tiefensuche, in einem Graphen mit n Knoten und m Knoten ohne eingehende Kanten, maximal $(n-m)^2$ besucht werden können.

$$\begin{aligned} & (n-m) \cdot (n-m-1) + (n-m) \\ = & (n-m) \cdot (n-m-1+1) \\ = & (n-m)^2 \end{aligned} \tag{A.2}$$

Bleibt noch zu klären, wie viele Knoten bei solch einer Tiefensuche maximal besucht

werden können: es kann zum einen jeder Knoten besucht werden, der mindestens eine eingehende Kante hat $(n-m)$ und zum anderen der Knoten, bei dem die Tiefensuche gestartet wurde. Alle anderen Knoten $(m-1)$, können nicht besucht werden, da sie keine eingehenden Kanten haben.
Der Gesamtaufwand einer einzelnen Tiefensuche ergibt sich aus der Summe der Kanten und Knoten, die maximal besucht werden können: $(n-m)^2 + (n-m+1)$. Soll nun für jeden der m Knoten eine Tiefensuche durchgeführt werden, ergibt sich ein Gesamtaufwand von $m \cdot ((n-m)^2 + (n-m+1))$.

Da der Aufwand offensichtlich nicht nur von der Menge der Knoten insgesamt (n), sondern auch von der Anzahl von Knoten ohne eingehende Kanten (m) abhängt, kann der Aufwand nur in Abhängigkeit von m abgeschätzt werden:

a) $m = 0$
Soll keine einzige Tiefensuche durchgeführt werden, entsteht für n Knoten eine konstante Komplexität: $\mathcal{O}(1)$.

b) $m = n$
Existieren in einem Graphen nur Knoten ohne eingehende Kanten, ist die Anzahl der Kanten im Graphen 0. Es muss daher nur für jeden der $m = n$ Knoten eine Tiefensuche gestartet werden, die keine Kanten zu verfolgen hat. Somit entsteht ein linearer Aufwand: $\mathcal{O}(n)$.

c) $m = \frac{n}{2}$
Unter der Annahme, dass die Hälfte aller Knoten im Graphen keine eingehenden Kanten haben, ergibt sich gemäß Formel A.3 ein Gesamtaufwand von $\frac{n^3+2n^2+4n}{8}$.

$$\begin{aligned} & \frac{n}{2} \cdot ((n - \frac{n}{2})^2 + (n - \frac{n}{2} + 1)) \\ = & \frac{n}{2} \cdot (\frac{n^2}{4} + \frac{n}{2} + 1) \\ = & \frac{n^3}{8} + \frac{n^2}{4} + \frac{n}{2} \\ = & \frac{n^3 + 2n^2 + 4n}{8} \end{aligned} \tag{A.3}$$

Im Sinne der $\mathcal{O}$-Notation bleibt der Aufwand damit bei $\mathcal{O}(n^3)$. Es ist jedoch auch ersichtlich, dass für große n, $\frac{n^3+2n^2+4n}{8} < n^3$ gilt und der Gesamtaufwand von FeatureMapGen::NoIn somit i.d.R. geringer als der Gesamtaufwand von FeatureMapGen::All ist.

B. XML-Dateien

Dieser Abschnitt ergänzt Kapitel 4 und 5.
Quelltext B.1 zeigt eine p2-Feature-Beschreibungsdatei des Features *Test Tuner*. Zu erkennen sind neben Beschreibung, Copyrights und Lizenzen, dass das Feature *Test Tuner* das Feature *Test Infotainment* benötigt und dass es das Artefakt *de.vwag.keffb.infotainment.tuner.test* beinhaltet.

```
1  <feature
2        id="de.vwag.keffb.feature.infotainment.test.tuner"
3        label="Test Tuner"
4        version="1.0.0"
5        provider-name="Volkswagen AG - K-EFF/B">
6
7     <description url="http://www.example.com/description">
8        [Enter Feature Description here.]
9     </description>
10    <copyright url="http://www.example.com/copyright">
11       [Enter Copyright Description here.]
12    </copyright>
13    <license url="http://www.example.com/license">
14       [Enter License Description here.]
15    </license>
16
17    <includes
18          id="de.vwag.keffb.feature.infotainment.test"
19          version="1.0.0"
20          name="Test Infotainment"/>
21    <plugin
22          id="de.vwag.keffb.infotainment.tuner.test"
23          download-size="16"
24          install-size="16"
25          version="1.1.4"
26          unpack="false"/>
27 </feature>
```

Quelltext B.1: p2-Feature-Beschreibung für das Feature *Test Tuner*

Quelltext B.2 zeigt, wie das FeatureIDE-Format [KTS+09] zum Speichern von Feature-Modellen um die Fähigkeit erweitert wurde, neben Features und deren Abhängigkeiten auch Feature-Mappings abspeichern zu können.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <featureModel chosenLayoutAlgorithm="1">
3    <struct>
4      <feature mandatory="true" name="Feature_JMF_Media_Renderer">
5          <includes
6              artifact-name="de.vwag.keffb.infotainment.media.jmf"
7              artifact-version="1.1.6"/>
8          <includes
9              artifact-name="de.vwag.keffb.jaudiotagger"
10             artifact-version="1.1.0"/>
11         <includes
12             artifact-name="de.vwag.keffb.jmf.windows"
13             artifact-version="2.1.1"/>
14     </feature>
15   </struct>
16   <constraints/>
17   <comments/>
18   <featureOrder userDefined="false"/>
19 </featureModel>
```

Quelltext B.2: Erweitertes FeatureIDE-Format zum Speichern von Feature-Modellen inkl. Feature-Mappings

Oben gezeigte XML-Datei kann aktuell nicht von FeatureIDE gelesen werden, da ein Element *includes* innerhalb eines Elements *feature* nicht dem erwarteten XML-Schema entspricht.

C. Fallstudie

Dieser Abschnitt zeigt mit Tabelle C.1 ergänzend zu Kapitel 6, welche OSGi-Bundles bei der Durchführung der Fallstudie für FeatureMapGen und FeatureModGen verwendet wurden.

Tabelle C.1: Übersicht der in der Fallstudie verwendeten OSGi-Bundles

Lfd.-Nr.	Bundlename	Version
1	de.vwag.keffb.hw.input	1.2.00
2	de.vwag.keffb.hw.input.key.mapper	1.0.04
3	de.vwag.keffb.infotainment	1.3.02
4	de.vwag.keffb.infotainment.addressbook.slap	1.1.09
5	de.vwag.keffb.infotainment.addressbook.test	1.1.04
6	de.vwag.keffb.infotainment.image.files	1.3.03
7	de.vwag.keffb.infotainment.map.slap	1.1.09
8	de.vwag.keffb.infotainment.media.jmf	1.1.06
9	de.vwag.keffb.infotainment.media.slap	1.1.09
10	de.vwag.keffb.infotainment.media.test	1.1.07
11	de.vwag.keffb.infotainment.mixer.javasound	1.0.06
12	de.vwag.keffb.infotainment.mixer.slap	1.1.09
13	de.vwag.keffb.infotainment.mixer.test	1.1.03

Fortsetzung auf der nächsten Seite

Fortsetzung von Tabelle C.1

Lfd.-Nr.	Bundlename	Version
14	de.vwag.keffb.infotainment.navigation.slap	1.1.09
15	de.vwag.keffb.infotainment.phone.slap	1.1.09
16	de.vwag.keffb.infotainment.phone.test	1.1.03
17	de.vwag.keffb.infotainment.preset	1.0.04
18	de.vwag.keffb.infotainment.server	1.6.02
19	de.vwag.keffb.infotainment.slap.generated	1.2.02
20	de.vwag.keffb.infotainment.slap.util	1.2.03
21	de.vwag.keffb.infotainment.test	1.1.03
22	de.vwag.keffb.infotainment.tuner.slap	1.1.09
23	de.vwag.keffb.infotainment.tuner.test	1.1.04
24	de.vwag.keffb.jaudiotagger	1.1.00
25	de.vwag.keffb.jmf.windows	2.1.01
26	de.vwag.keffb.server.slap	1.0.15
27	de.vwag.keffb.slap.server	1.3.01
28	de.vwag.keffb.slapdata.server	1.2.03
29	de.vwag.keffb.sw.search	1.6.00
30	de.vwag.keffb.sw.search.engine	1.6.01
31	de.vwag.keffb.sw.search.infotainmentadapter	1.2.07
32	de.vwag.keffb.sw.search.server	1.3.14
33	jslap-osgi-discovery	1.2.01
34	jslap-osgi-library	1.2.00
35	jslap-osgi-library	1.2.01
36	jslap-osgi-service	1.2.01

Literaturverzeichnis

[AC04] Antkiewicz, M. und K. Czarnecki: *FeaturePlugin: Feature Modeling Plug-In for Eclipse*. In: *Proceedings of the Workshop Eclipse Technology Exchange*, Seiten 67–72, New York, NY, USA, 2004. ACM.

[ACC+11] Acher, M., A. Cleve, P. Collet, P. Merle, L. Duchien und P. Lahire: *Reverse engineering architectural feature models*. In: *Proceedings of the European Conference on Software Architecture (ECSA)*, Seiten 220–235, Berlin, Heidelberg, New York, London, 2011. Springer Verlag.

[ACP+12] Acher, M., A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet und P. Lahire: *On extracting feature models from product descriptions*. In: *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Seiten 45–54, New York, NY, USA, 2012. ACM.

[AGM+06] Alves, V., R. Gheyi, T. Massoni, U. Kulesza, P. Borba und C. Lucena: *Refactoring Product Lines*. In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, Seiten 201–210. ACM, 2006.

[AK09] Apel, S. und C. Kästner: *An Overview of Feature-Oriented Software Development*. Journal of Object Technology (JOT), 8(5):49–84, Juli 2009.

[Bat05] Batory, D.: *Feature Models, Grammars, and Propositional Formulas*. In: *Proceedings of the International Software Product Line Conference (SPLC)*, Band 3714 der Reihe *Lecture Notes in Computer Science*, Seiten 7–20, Berlin, Heidelberg, New York, London, 2005. Springer Verlag.

[Bat06] Batory, D.: *A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*. In: *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, Seiten 3–35, Berlin, Heidelberg, New York, London, 2006. Springer Verlag.

[Beu03] Beuche, D.: *Variant Management with pure::variants*. Technischer Bericht, pure-systems GmbH, 2003.

[Boe88] Boehm, B. W.: *A spiral model of software development and enhancement.* In: *Computer*, Band 21, Seiten 61–72. IEEE, Washington, DC, USA, Mai 1988.

[BSL+10] Berger, T., S. She, R. Lotufo, K. Czarnecki und A. Wąsowski: *Feature-to-code mapping in two large product lines.* In: *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 498–499, Berlin, Heidelberg, New York, London, 2010. Springer Verlag.

[BSR04] Batory, D., J. N. Sarvela und A. Rauschmayer: *Scaling Step-Wise Refinement.* IEEE Transactions on Software Engineering (TSE), 30(6):355–371, 2004.

[BSTRC07] Benavides, D., S. Segura, P. Trinidad und A. Ruiz-Cortés: *FAMA: Tooling a Framework for the Automated Analysis of Feature Models.* In: *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Seiten 129–134, Ireland, 2007. Lero-University of Limerick.

[Bun11] Bundesministerium für Verkehr, Bau und Stadtentwicklung: *Elektromobilität – Deutschland als Leitmarkt und Leitanbieter.* Broschüre, Juli 2011.

[CBH11] Classen, A., Q. Boucher und P. Heymans: *A text-based approach to feature modelling: Syntax and semantics of TVL.* Science of Computer Programming (SCP), 76(12):1130–1143, Dezember 2011.

[CE00] Czarnecki, K. und U. W. Eisenecker: *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, New York, NY, USA, 2000.

[CN01] Clements, P. und L. Northrop: *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, USA, 2001.

[CW07] Czarnecki, K. und A. Wasowski: *Feature Diagrams and Logics: There and Back Again.* In: *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 23–34, Washington, DC, USA, 2007. IEEE.

[Dij69] Dijkstra, E. W.: *Structured programming (EWD268).* online unter: http://userweb.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF, abgerufen am 10. September 2012, 1969.

[Ecl08] Eclipse Foundation: *Component Oriented Development and Assembly with Equinox.* White Paper, online unter: http://eclipse.org/eclipsert/whitepaper/20080310_equinox.pdf, abgerufen am 10. September 2012, März 2008.

[FF03] Fuegi, J. und J. Francis: *Lovelace & Babbage and the creation of the 1843 'notes'.* Annals of the History of Computing, IEEE, 25(4):16–26, Oktober–Dezember 2003.

[GHJV95] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gri00] Griss, M. L.: *Implementing product-line features by composing aspects.* In: *Proceedings of the first conference on Software product lines : experience and research directions*, Seiten 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[GW96] Ganter, B. und R. Wille: *Formal Concept Analysis – Mathematical Foundations.* Springer Verlag, Berlin, Heidelberg, New York, London, 1996.

[HKW08] Heidenreich, F., J. Kopcsek und C. Wende: *FeatureMapper: Mapping Features to Models.* In: *Companion International Conference on Software Engineering (ICSEC)*, Seiten 943–944, New York, NY, USA, Mai 2008. ACM. Informal demonstration paper.

[HLHE11] Haslinger, E.N., R.E. Lopez-Herrejon und A. Egyed: *Reverse Engineering Feature Models from Programs' Feature Sets.* In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Seiten 308–312, Oktober 2011.

[IEE98] IEEE: *Standard for Software Test Documentation.* IEEE Std. 829–1998, Seite i, 1998.

[JF88] Johnson, R. E. und B. Foote: *Designing Reusable Classes.* Journal of Object-Oriented Programming (JOOP), 1(2):22–35, Juni 1988.

[KA08] Kästner, C. und S. Apel: *Integrating Compositional and Annotative Approaches for Product Line Engineering.* In: *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, Seiten 35–40, Passau, Deutschland, Oktober 2008.

[Käs07] Kästner, C.: *CIDE: Decomposing Legacy Applications into Features.* In: *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 149–150. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[KCH+90] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technischer Bericht CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[Kru02] Krueger, C. W.: *Easing the Transition to Software Mass Customization.* In: *Proceedings of the International Workshop on Software Product-Family Engineering (PFE)*, Band 2290 der Reihe *Lecture Notes in Computer Science*, Seiten 282–293, Berlin, Heidelberg, New York, London, 2002. Springer Verlag.

[KTA08] Kästner, C., S. Trujillo und S. Apel: *Visualizing Software Product Line Variabilities in Source Code.* In: *Proceedings of the International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, Seiten 303–313, September 2008.

[KTS+09] Kästner, C., T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz und S. Apel: *FeatureIDE: A Tool Framework for Feature-Oriented Software Development.* In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 611–614, Washington, DC, USA, Mai 2009. IEEE. Formal demonstration paper.

[Kuh11] Kuhlemann, M.: *Refactoring Feature Modules: Disciplined Generation of Reusable Modules.* Doktorarbeit, Universität Magdeburg, Deutschland, November 2011.

[LHB01] Lopez-Herrejon, R. E. und D. Batory: *A Standard Problem for Evaluating Product-Line Methodologies.* In: *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, Seiten 10–24, Berlin, Heidelberg, New York, London, 2001. Springer Verlag.

[LP07] Loesch, F. und E. Ploedereder: *Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations.* In: *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, Seiten 159–170, März 2007.

[MBC09] Mendonca, M., M. Branco und D. Cowan: *S.P.L.O.T.: Software Product Lines Online Tools.* In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Seiten 761–762, New York, NY, USA, 2009. ACM. demonstration paper.

[Moo65] Moore, G. E.: *Cramming More Components onto Integrated Circuits.* Electronics, 38(8):114–117, April 1965.

[MOW10] Mischke, M., P. Oel und M. Wittkämper: *Kartenbasierte Bedienung automobiler Infotainmentsysteme – Ein Ausweg aus der Liste.* In: *Useware*, Band 5, Seiten 215–226, Oktober 2010.

[MVA10] McAffer, J., P. Vanderlei und S. Archer: *OSGi and Equinox: Creating Highly Modular Java Systems.* Addison-Wesley, Upper Saddle River, NJ, 2010.

[NE00] Nuseibeh, B. und S. Easterbrook: *Requirements engineering: a roadmap.* Context, 1(258):35–46, 2000.

[OSG09] OSGi Alliance: *OSGi Service Platform, Service Compendium, Release 4, Version 4.2.* Createspace, August 2009.

[Par72] Parnas, D. L.: *On the criteria to be used in decomposing systems into modules.* Communications of the ACM, 15(12):1053–1058, Dezember 1972.

[RC12] Rubin, J. und M. Chechik: *Combining related products into product lines.* In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, Seiten 285–300, Berlin, Heidelberg, New York, London, 2012. Springer Verlag.

[Rei09] Reiser, M.-O.: *Core concepts of the compositional variability management framework (CVM).* Technischer Bericht, Technische Universität Berlin, Deutschland, 2009.

[Roy87] Royce, W. W.: *Managing the development of large software systems: concepts and techniques.* In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[SC92] Spencer, H. und G. Collyer: *#ifdef Considered Harmful, or Portability Experience With C News.* In: *Proceedings of the USENIX Conference*, Seiten 185–198, USENIX Association Berkeley, USA, 1992.

[Sco05] Scott, M. L.: *Programming Language Pragmatics, Second Edition.* Morgan Kaufmann, November 2005.

[SHTB07] Schobbens, P., P. Heymans, J. Trigaux und Y. Bontemps: *Generic Semantics of Feature Diagrams.* Computer Networks, 51(2):456–479, 2007.

[SLB+10] She, S., R. Lotufo, T. Berger, A. Wasowski und K. Czarnecki: *The Variability Model of the Linux Kernel.* In: *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, Januar 2010.

[SMC74] Stevens, W. P., G. J. Myers und L. L. Constantine: *Structured design.* IBM Systems Journal, 13(2):115–139, 1974.

[SSH08] Saake, G., K. Sattler und A. Heuer: *Datenbanken: Konzepte und Sprachen.* mitp-Verlag/Bonn, 2008.

[STKS12] Schulze, S., T. Thüm, M. Kuhlemann und G. Saake: *Variant-Preserving Refactoring in Feature-Oriented Software Product Lines.* In: *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Seiten 73–81, New York, NY, USA, Januar 2012. ACM.

[Szy97] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Boston, MA, USA, Dezember 1997.

[TBK09] Thüm, T., D. Batory und C. Kästner: *Reasoning about Edits to Feature Models.* In: *Proceedings of the International Conference on Software Engineering (ICSE)*, Seiten 254–264, Washington, DC, USA, Mai 2009. IEEE.

[Thü08] Thüm, T.: *Reasoning about Feature Model Edits.* Bachelorarbeit, Universität Magdeburg, 2008.

[TKB+12] Thüm, T., C. Kästner, F. Benduhn, J. Meinicke, G. Saake und T. Leich: *FeatureIDE: An Extensible Framework for Feature-Oriented Software Development*. Science of Computer Programming (SCP), 2012. noch nicht erschienen; angenommen am 6. Juni 2012.

[TKES11] Thüm, T., C. Kästner, S. Erdweg und N. Siegmund: *Abstract Features in Feature Modeling*. In: *Proceedings of the International Software Product Line Conference (SPLC)*, Seiten 191–200, Washington, DC, USA, August 2011. IEEE.

[vDK02] Deursen, A. van und P. Klint: *Domain-Specific Language Design Requires Feature Descriptions*. Computing and Information Technology, 10(1):1–17, 2002.

[vdLSR07] Linden, F. J. van der, K. Schmid und E. Rommes: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Verlag, Berlin, Heidelberg, New York, London, 2007.

[Vin93] Vinoski, S.: *Distributed Object Computing with CORBA*. C++ Report, 5(6):32–38, August 1993.

[WD05] Wöhe, G. und U. Döring: *Einführung in die Allgemeine Betriebswirtschaftslehre*, Band 22. Verlag Franz Vahlen, Berlin / Frankfurt/Main, 2005.

[WHKL08] Wütherich, G., N. Hartmann, B. Kolb und M. Lübken: *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt, Heidelberg, 2008.

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

Magdeburg, den 14. September 2012