Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Bachelorarbeit

Featurebaum-basierte Produktkonfiguration

Autor:

Daniel Püsche

03.04.2017

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Sebastian Krieter
M.Sc. Jacob Krüger
Institut für Technische und Betriebliche Informationssysteme

Püsche, Daniel:
Featurebaum-basierte Produktkonfiguration
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2017.

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

1 Einführung **1**

2 Grundlagen **5**
2.1 Softwareproduktlinien 5
2.2 Feature-Modelle 6
2.3 Konfigurationen 9
2.3.1 FeatureIDE 10
2.3.2 S.P.L.O.T. 11
2.4 Zusammenfassung 12

3 Konzept **15**
3.1 Nutzen 15
3.2 Anforderungen 15
3.3 Konzeptdetails 17
3.3.1 Auswahl des Konfigurationstools 17
3.3.2 Umsetzung der Anforderungen 17
3.4 Zusammenfassung 20

4 Implementierung **23**
4.1 Überblick über die FeatureIDE-Architektur 23
4.1.1 Configuration Editor 23
4.1.2 Feature Diagram Editor 24
4.2 Erstellen und Einbinden der Editorseite 24
4.3 Darstellung des Diagramms 24
4.4 Reagieren auf Eingaben 25
4.5 Zusammenfassung 26

5 Evaluierung **29**
5.1 Durchführung der Evaluierung 29
5.1.1 Fragestellungen 30
5.1.2 Produkte 30
5.2 Ergebnisse 32
5.3 Auswertung 35
5.3.1 Interpretation der Ergebnisse 35
5.3.2 Aussagekraft der Ergebnisse 36
5.4 Zusammenfassung 37

6 Verwandte Arbeiten **39**
6.1 S2T2 39
6.2 GEARS 40
6.3 KConfig 41

7 Zusammenfassung **43**

8 Zukünftige Arbeiten **47**

Literaturverzeichnis **51**

Abbildungsverzeichnis

2.1 Kostenvergleich Einzelprodukte und Produktlinie 6
2.2 Überblick über einen Softwareproduktlinienentwicklungsprozess . . . 7
2.3 Feature-Diagramm eines Online-Pizzabestellservices 8
2.4 Darstellung der vier Grundbeziehungen in Feature-Diagrammen . . . 9
2.5 Standard-Konfiguration und Erweiterte Konfiguration in FeatureIDE 11
2.6 Konfiguration in S.P.L.O.T. 12

3.1 Diagramm im Feature-Diagramm-Editor von FeatureIDE 18
3.2 Zustandsdiagramm für den Konfigurationsprozess 19
3.3 Darstellung der Zustände im Feature-Diagramm 19

4.1 Übersicht über die Editoren und Editorseiten 25
4.2 Übersicht der erzeugten EditParts 26

5.1 Erweitertes Feature-Diagramm eines Online-Pizzabestellservices . . . 31

1. Einführung

So wie es in anderen Industriezweigen Massenproduktion und Handarbeit gibt, so gibt es auch bei der Entwicklung von Software die Möglichtkeit, einheitliche Standardprodukte oder individualisierte Produkte herzustellen. Beide Varianten besitzen ihre Vor- und Nachteile. So sind individualisierte Produkte besonders zeit- und kostenintensiv, während den Standardprodukten die nötige Diversifikation fehlt [18], um verschiedene Kundenanforderungen zu erfüllen. Um sich die Vorteile beider Arten von Softwareprodukten zu Nutze machen zu können, gingen die Hersteller mit der Zeit dazu über, Softwareproduktlinien zu entwickeln [19].

Einzelne Produkte einer Produktlinie befinden sich in einer gemeinsamen Domäne und teilen sich daher bestimmte Eigenschaften. Der Begriff *Domäne* beschreibt hierbei ein Wissensgebiet mit eigenen Konzepten und Fachbegriffen, sowie Fachwissen über das Errichten von Softwaresystemen in diesem Gebiet [1]. Während des Softwareprodukliniententwicklungsprozesses werden wiederverwendbare Softwareartefakte geschaffen [4], mit deren Hilfe ein vollständiges Produkt erstellt werden kann. Durch eine solche Wiederverwendung von Code kann der Entwicklungsprozess eines Produkts stark beschleunigt werden und dementsprechend die Entwicklungskosten reduziert werden [1]. Zusätzlich kann ein Produkt im Laufe der Entwicklung beliebig erweitert werden, um die Anforderungen des Kunden zu erfüllen.

Das fertige Endprodukt entsteht aus einer so genannten *Produktkonfiguration*. Diese Konfiguration ist eine Menge von verschiedenen Komponenten beziehungsweise *Features*. Bei der Konfiguration eines Produkts müssen Abhängigkeiten zwischen den einzelnen Features beachtet werden. Eine Möglichkeit, diese Abhängigkeiten zu definieren, sind *Feature-Modelle* [5]. Diese können unter anderem als Baumdiagramme dargestellt werden, bei denen jeder Knoten ein Feature repräsentiert. Diese Repräsentation von Feature-Modellen wird üblicherweise Feature-Diagramm genannt.

Es existieren verschiedene Tools, mit denen Softwareproduktlinienentwicklung betrieben werden kann, wie zum Beispiel S.P.L.O.T. [16] oder FeatureIDE [22]. Mit diesen Tools lassen sich nicht nur Feature-Modelle erstellen, sondern sie bieten auch die Möglichkeit, eine Produktkonfiguration zusammen zu stellen. Dabei werden die

gewünschten Features mit Hilfe einer grafischen Oberfläche ausgewählt, um ein für den Kunden maßgeschneidertes Produkt zu kreieren. Die Konfigurationseditoren greifen hierbei auf ein Feature-Modell [10] zurück, um Konflikte bei der Auswahl von Features zu vermeiden. So werden zum Beispiel im Konfigurationseditor von S.P.L.O.T. alle Features abgewählt, die mit einem neu ausgewählten Feature kollidieren.

Viele Tools, wie S.P.L.O.T. oder FeatureIDE, nutzen in ihren Editoren Listen zur Darstellung der Konfiguration. Diese Form der Darstellung kann potentiell zu Problemen führen, da man in der Industrie oftmals mit mehreren tausend Features arbeitet [17], wodurch die Listenansicht schnell unübersichtlich werden kann. Dies wird besonders bei Editoren deutlich, die automatisch auf Veränderungen in der Konfiguration reagieren, wie S.P.L.O.T.. Hierbei kann es durchaus vorkommen, dass man ein Feature anwählt, aber aufgrund der Listenansicht nicht erkennen kann, welche Features dadurch abgewählt werden. Daraus lässt sich ein weiterer Nachteil ableiten: Durch die Listendarstellung lässt sich nicht nachvollziehen, welche Abhängigkeiten zwischen den einzelnen Features bestehen. Diese Probleme führen dazu, dass das Erstellen einer Produktkonfiguration sehr zeitaufwendig werden kann, was wiederum die Entwicklungskosten negativ beeinflusst. Des Weiteren kann es durch die Unübersichtlichkeit beim automatischen Abwählen von Features zum Erstellen ungewollter Konfigurationen kommen. Als Alternative könnte man die Features im Editor durch ein Baumdiagramm darstellen, ähnlich dem entsprechenden Feature-Diagramm. Dieses Konzept wird im Rahmen dieser Bachelorarbeit umgesetzt und anschließend in Hinblick auf Bedienbarkeit evaluiert.

Zielstellung

Das Ziel dieser Arbeit ist es, ein Konzept für einen neuen Konfigurationseditor zu erstellen und diesen anschließend zu implementieren. Der Editor soll hierbei als Darstellungsform ein Baumdiagramm nutzen, um die Probleme der Listendarstellung zu adressieren. Die Umsetzung soll in FeatureIDE erfolgen, welches bereits einen Konfigurationseditor mit Listendarstellung besitzt. Dadurch sollen im direkten Vergleich die Vor- und Nachteile der Listen- bzw. Baumdarstellung herausgearbeitet werden. Abschließend soll durch eine Nutzerstudie überprüft werden, ob der Editor mit Baumdarstellung tatsächlich eine erleichterte Bedienung ermöglicht. Dementsprechend sollen im Verlauf dieser Arbeit folgende wissenschaftliche Fragestellungen beantwortet werden:

- Welche Vor- und Nachteile weist die alternative Darstellungsform gegenüber der Listendarstellung auf?
- Ermöglicht die alternative Darstellungsform eine aus Nutzersicht einfachere Bedienung beim Erstellen von Produktkonfigurationen als die Listendarstellung?

Gliederung

Die Arbeit ist folgendermaßen aufgebaut. In Kapitel 2 werden die Grundlagen der Softwareproduktlinienentwicklung besprochen. Hierbei liegt der Schwerpunkt auf

Feature-Modellierung und Produktkonfigurationen, da diese Themengebiete wichtig für das Verständnis der Problemstellung sind. Außerdem gibt es einen ersten Überblick über Konfigurationssoftware. Anschließend wird in Kapitel 3 ein Konzept für einen neuen Konfigurationseditor vorgestellt. Zunächst wird beschrieben, welchen Nutzen der Editor haben soll und es werden verschiedene Anforderungen definiert. Aufbauend auf diesen Anforderungen wird das eigentliche Konzept entwickelt. Wie dieses Konzept implementierungstechnisch umgesetzt wurde, wird in Kapitel 4 erläutert. In Kapitel 5 wird der neu entwickelte Konfigurationseditor evaluiert. Dabei wird zunächst die Vorgehensweise bei der Evaluierung beschrieben. Anschließend werden die Ergebnisse vorgestellt und ausgewertet, um die wissenschaftlichen Fragen zu beantworten. In Kapitel 6 werden verwandte Themengebiete besprochen. Zum Abschluss bietet Kapitel 7 einen Überblick über die Resultate der Arbeit und Kapitel 8 geht auf Themengebiete für zukünftige Arbeiten ein.

2. Grundlagen

In diesem Kapitel sollen die Grundlagen von Softwareproduktlinienentwicklung vermittelt werden. Hierbei werden zunächst Softwareproduktlinien im Allgemeinen behandelt. Des Weiteren wird auf die Rolle von Feature-Modellen und Produktkonfigurationen eingegangen. Zusätzlich werden verschiedene Konfigurationstools vorgestellt.

2.1 Softwareproduktlinien

Ein möglicher Ansatz zur Entwicklung von Software sind Softwareproduktlinien. Der Kerngedanke dahinter ist eine kundenindividuelle Massenproduktion, die durch den Aufbau von individuellen Lösungen auf Grundlage von wiederverwendbaren Softwarekomponenten erreicht werden kann [1]. Softwareproduktlinien vereinen dementsprechend die Stärken von individualisierten Produkten und Standardprodukten, denn sie bieten schnelle und kosteneffiziente Entwicklung von Produkten, wie bei der Massenproduktion, aber sind dennoch so variabel wie maßgeschneiderte Softwareprodukte.

Es gibt verschiedene Gründe, warum sich die Hersteller von Softwareprodukten für Produktlinien entscheiden. Ein besonders wichtiger Punkt sind die verminderten Entwicklungskosten, da jedes Produkt auf bereits vorhandener Software aufbaut und demzufolge weniger Ressourcen für Design und Implementierung verwendet werden müssen [18, 24]. Diese Kosteneffizienz tritt allerdings erst bei größeren Produktmengen auf, wie Abbildung Abbildung 2.1 deutlich macht. Dafür ist die Ersparnis bei einer hohen Anzahl von Produkten umso größer. Ein weiterer Vorteil von Softwareproduktlinien ist die geringere Entwicklungsdauer. Einzelne Produkte können viel schneller geliefert werden, da die vom Kunden gewünschten Funktionen teilweise schon vorhanden sind und nur zusammengestellt werden müssen. Softwareproduktlinienentwicklung kann sich außerdem positiv auf die Qualität des Endprodukts auswirken [24]. Da Softwarekomponenten stetig wiederverwendet werden, werden sie immer wieder neuen Tests unterzogen. Komponenten, die sehr häufig verwendet werden, sind aufgrund der hohen Anzahl von Testläufen zuverlässiger [1].

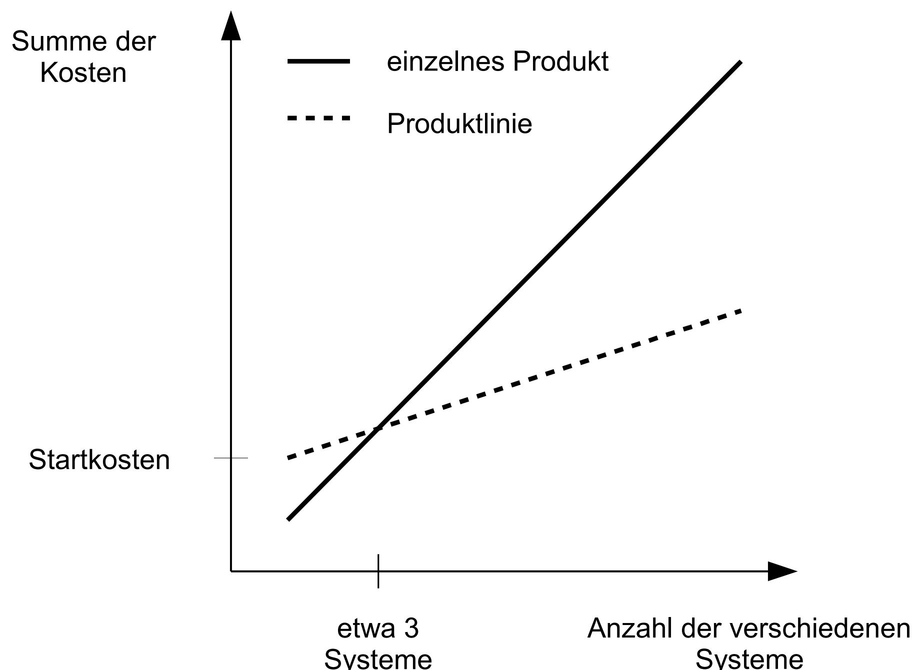

Abbildung 2.1: Kostenvergleich zwischen Einzelprodukten und Produkten einer Softwareproduktlinie [18]

Der Softwareproduktlinienentwicklungsprozess lässt sich außerdem in zwei Teilprozesse unterteilen, wie in Abbildung Abbildung 2.2 zu erkennen ist. Während des *domain engineering* wird die Domäne der Produktlinie analysiert [1]. Die daraus gewonnenen Erkenntnisse werden genutzt, um wiederverwendbare Artefakte zu entwickeln, welche später zur Entwicklung neuer Produkte genutzt werden können [4]. Zusätzlich werden Variabilitätsmodelle definiert, die die Abhängigkeiten zwischen diesen wiederverwendbaren Artefakten festlegen. Eine Form von Variabilitätsmodellen sind Feature-Modelle.

Beim *application engineering* wird auf den Ergebnissen des domain engineering aufgebaut. Das Resultat dieses Prozesses ist ein auf einen bestimmten Kunden maßgeschneidertes Softwareprodukt [1]. Zur Entwicklung dieses Produkts werden unter anderem die während des domain engineering entwickelten Artefakte verwendet. Dabei werden die durch das Variabilitätsmodell definierten Abhängigkeiten berücksichtigt. Der Auswahlprozess der Artefakte, die Teil eines bestimmten Produkts sein sollen, wird auch Produktkonfigurationsprozess genannt.

2.2 Feature-Modelle

Beim Erstellen von Produktkonfigurationen muss darauf geachtet werden, welche Abhängigkeiten zwischen einzelnen Features bestehen. Es ist daher wichtig, ein ent-

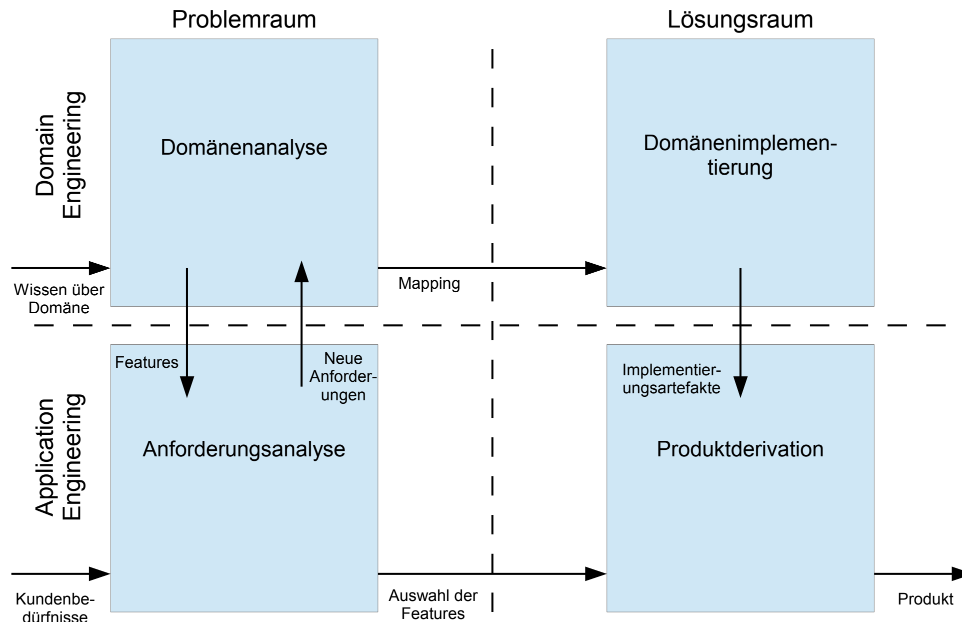

Abbildung 2.2: Überblick über einen Softwareproduktlinienentwicklungsprozess [1]

sprechendes Feature-Modell zu erstellen, da dieses die Features einer Produktlinie und deren Beziehungen definiert [1, 10].

Oftmals werden Feature-Modelle als Baumdiagramme dargestellt, auch Feature-Diagramme genannt. Ein Feature-Diagramm besteht aus einzelnen Knoten. Jeder Knoten steht hierbei stellvertretend für ein Feature [6]. Die Abhängigkeiten zwischen den Features werden durch die Kanten des Baums dargestellt. Daraus folgt, dass die Kind-Knoten immer von den Eltern-Knoten abhängig sind. Wird also für eine Konfiguration das Feature eines Kind-Knotens ausgewählt, muss auch das Feature des entsprechenden Eltern-Knotens ausgewählt werden, damit die Konfiguration gültig ist. Die Features der Kind-Knoten sind oftmals speziellere Varianten der Eltern-Features. Beispielsweise besitzt das Feature-Diagramm in Abbildung Abbildung 2.3 einen Knoten `Soße`, der durch die Kindknoten `Tomatensoße`, `BBQ` und `Käsesoße` genauer definiert wird.

Feature-Diagramme können vier grundlegende Arten von Kanten besitzen, welche jeweils eine bestimmte Form von Beziehung darstellen. Die ersten beiden Kantenarten werden durch einen gefüllten beziehungsweise einen leeren Kreis dargestellt, wie Abbildung Abbildung 2.3 zeigt. Der gefüllte Kreis steht hierbei für ein *zwingend notwendiges Feature*, der leere Kreis für ein *optionales Feature* [11]. Im Falle des Pizza-Beispiels wäre `Belag` also ein notwendiges Feature, da jede Pizza eine Form von Belag braucht. Der `Käserand` hingegen ist optional. Wählt man einen Eltern-Knoten aus, dessen Kind als zwingend notwendig markiert wurde, so muss auch dieser Kind-Knoten ausgewählt werden, um eine gültige Konfiguration zu erhalten

[20]. Im Gegensatz dazu müssen optionale Features nicht unbedingt ausgewählt werden, sobald das Feature des Eltern-Knotens gewählt wurde [20].

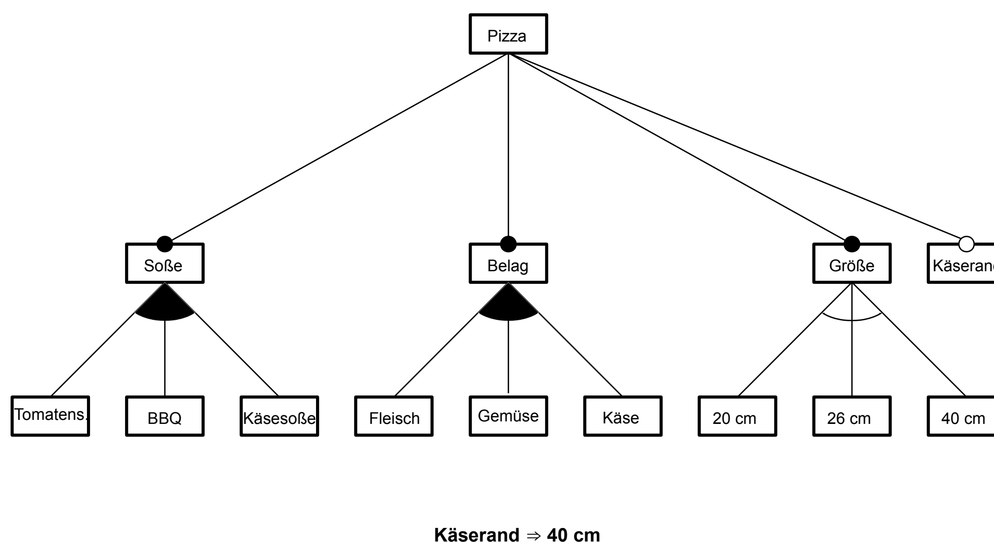

Abbildung 2.3: Feature-Diagramm eines Online-Pizzabestellservices

Zusätzlich zu diesen beiden Arten von Kanten gibt es noch *Alternativen* und *ODER-Kanten* [11]. Alternativen werden durch einen leeren Kegel, ODER-Kanten durch einen gefüllten Kegel dargestellt, wie in Abbildung Abbildung 2.4 zu erkennen ist. Besitzt ein Feature ausgehende ODER-Kanten, bedeutet das, dass mindestens eins der untergeordneten Features ausgewählt werden muss [20].Es können allerdings noch zusätzliche Features ausgewählt werden. Die Funktionsweise ähnelt hierbei dem logischen ODER, daher auch der Name. Im Beispiel muss mindestens ein Belag gewählt werden, es ist aber auch möglich, seine Pizza mit mehreren Sachen zu belegen. Wenn ein Feature Alternativen besitzt, muss genau ein untergeordnetes Feature ausgewählt werden, damit die Produktkonfiguration gültig ist [20]. Die Alternativen-Beziehung wurde im Beispiel für das Feature `Größe` verwendet, da eine Pizza immer nur genau eine Größe haben kann.

Durch Hinzufügen so genannter *cross-tree constraints* ist es außerdem möglich Feature-Diagramme zu verfeinern. Hierbei handelt es sich oftmals um aussagenlogische Statements [12], mit denen auch Abhängigkeiten zwischen verschiedenen Teilbäumen eines Feature-Diagramms definiert werden können. Im Pizza-Beispiel impliziert die Auswahl des Käserands die Auswahl einer 40cm-Pizza, da die Pizzeria den Käserand nur für große Pizzen anbietet.

Neben einzelnen cross-tree constraints ist es außerdem möglich, komplette Feature-Diagramme als aussagenlogische Formeln darzustellen [12, 20]. Dafür wird für jedes Feature eine entsprechende Variable eingeführt. Wird ein Feature ausgewählt, wird der entsprechenden Variable der Wert 1 oder TRUE zugewiesen. Die Variablen abgewählter Features erhalten dementsprechend den Wert 0 oder FALSE. Eine Konfiguration ist eine Belegung der verschiedenen Variablen. Sie ist erst dann gül-

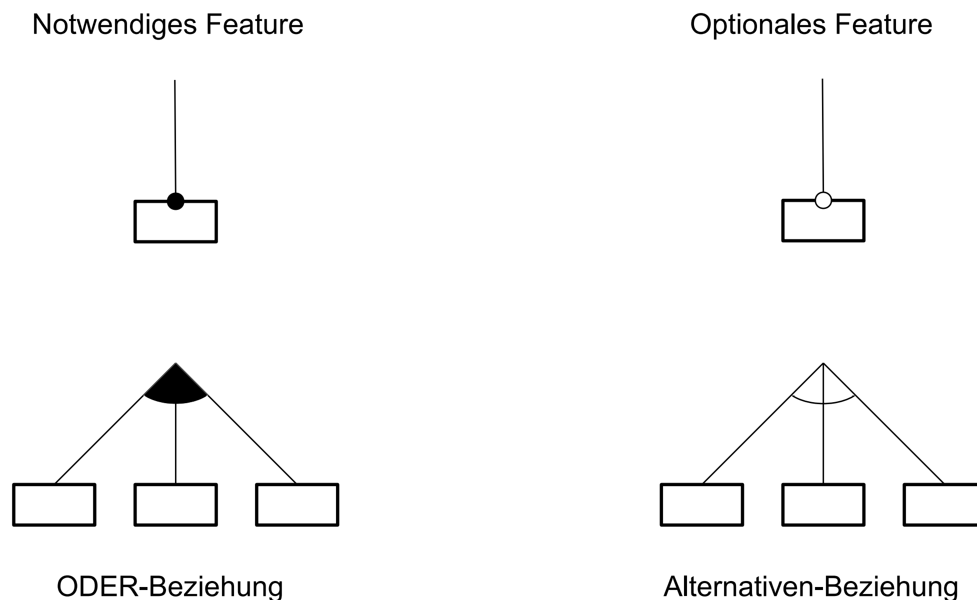

Abbildung 2.4: Darstellung der vier Grundbeziehungen in Feature-Diagrammen

tig, wenn die aussagenlogische Formel durch diese Belegung erfüllt werden kann. Dementsprechend würde eine aussagenlogische Formel für das Beispiel aus Abbildung Abbildung 2.3 wie folgt aussehen:

$$
\begin{aligned}
&Pizza \wedge Soße \wedge Belag \wedge Größe \wedge \\
&(Soße \iff (Tomatensoße \vee BBQ \vee Käsesoße)) \wedge \\
&(Belag \iff (Fleisch \vee Käse \vee Gemüse)) \wedge \\
&(Größe \iff (20\ cm \veebar 26\ cm \veebar 40\ cm)) \wedge \\
&(Käserand \rightarrow 40\ cm)
\end{aligned}
$$

2.3 Konfigurationen

Um ein fertiges Endprodukt bereitstellen zu können, werden verschiedene Features kombiniert. Diese Menge von Features wird (Produkt-)Konfiguration genannt. Es kann jedoch vorkommen, dass bestimmte Features sich gegenseitig ausschließen. Befinden sich diese Features gemeinsam in einer Konfiguration, so wird diese als *nicht gültig* bezeichnet. Aus einer nicht gültigen Konfiguration kann somit kein Produkt erstellt werden. Eine Konfiguration ist also erst *gültig*, wenn die an- und abgewählten Features vom Feature-Modell zugelassen werden [23], da es definiert, welche Abhängigkeiten und Beziehungen zwischen den Features bestehen. Es beschreibt die Menge aller gültigen Produkte und bildet daher die Basis des Produktkonfigurationsprozesses [17]. Unter Berücksichtigung des Feature-Modells ist es möglich, die Konfiguration manuell zu erstellen, indem man die gewünschten Features auswählt. Diese Art der Konfiguration ist allerdings sehr fehleranfällig, da der Nutzer ohne regelmäßige Überprüfung des Modells nicht sicher sein kann, ob das von ihm er-

stellte Produkt valide ist. Daher kann dieser Prozess bei großen Feature-Modellen zeitaufwendig werden.

Um diesen Problemen entgegenzuwirken, können spezielle Tools genutzt werden, so genannte *Konfiguratoren.* Ein Konfigurator unterstützt den Nutzer dabei auf verschiedenste Art und Weise. So ermöglicht er beispielsweise eine schrittweise Konfiguration. Bei dieser Form der Konfiguration kann jedes Feature einen von drei möglichen Zuständen besitzen: undefiniert, positiv oder negativ. Ein Feature, das sich im positiven Zustand befindet, wurde angewählt und ist somit ein Teil des Endprodukts. Hingegen sind Features im negativen Zustand kein Teil des Endprodukts. Features, für die noch keine Auswahl getroffen wurde, befinden sich im undefinierten Zustand. Zu Beginn einer schrittweisen Konfiguration befinden sich alle Features in diesem Zustand. Im Verlauf des Konfigurationsprozesses werden nacheinander die Zustände der einzelnen Features auf positiv oder negativ gesetzt. Sobald für alle Features eine Auswahl getroffen wurde, es also keine Features im undefinierten Zustand gibt, ist der Konfigurationsprozess abgeschlossen. Das Resultat des Prozesses ist eine *vollständige Konfiguration.* Wurde noch nicht für alle Features eine Auswahl getroffen, spricht man von einer *partiellen Konfiguration* [7]. In jedem Zwischenschritt des Konfigurationsprozesses wird eine partielle Konfiguration erzeugt.

Mit Konfiguratoren lassen sich aber auch bestimmte Teile des Konfigurationsprozesses automatisieren, da sie auf die Eingabe des Nutzers reagieren und die Konfiguration dementsprechend anpassen. Dieser Prozess wird auch *interaktive Konfiguration* genannt [9]. Dabei überprüft der Konfigurator nach jedem Konfigurationsschritt, also beim An- oder Abwählen eines Features, ob die derzeitige Konfiguration gültig ist. Ist dies nicht der Fall, werden automatisch, unter Berücksichtigung bisheriger Entscheidungen, weitere Features an- oder abgewählt, um eine gültige Konfiguration zu kreieren. Dieser Vorgang wird auch *decision propagation* genannt [17]. Um eine solche Form der Automatisierung ermöglichen zu können, greifen die Programme auf das Feature-Modell der entsprechenden Softwareproduktlinie zurück. Die Auswahl der Features erfolgt in der Regel über eine grafische Oberfläche. Eine häufige Form der Darstellung ist es, Feature-Modelle in Listen zu überführen.

2.3.1 FeatureIDE

Bei FeatureIDE handelt es sich um ein Eclipse-Plugin, welches Feature-orientierte Softwareentwicklung ermöglicht. Dementsprechend lassen sich mit Hilfe von FeatureIDE auch Produktkonfigurationen erstellen. Auf der ersten Seite des Configuration View befindet sich eine Listenansicht des aktuell gewählten Feature-Diagramms. Von hier aus können durch An- und Abwählen der Features entsprechende Konfigurationen zusammengestellt werden. Da der Konfigurator mit dem Feature-Diagramm zusammenhängt, werden automatisch die Abhängigkeiten zwischen den Features überprüft, um sicherzustellen, dass die Konfiguration gültig ist. Zudem unterstützt der Editor den Nutzer beim Erstellen einer Konfiguration, indem er Features, die für eine gültige Konfiguration notwendig sind, grün markiert. Features, die für eine gültige Konfiguration abgewählt werden müssen, werden blau markiert.

Die Seite *Advanced Configuration* bietet einige zusätzliche Funktionen. Beispielsweise werden in dieser Ansicht verschiedene Abhängigkeiten zwischen den Features,

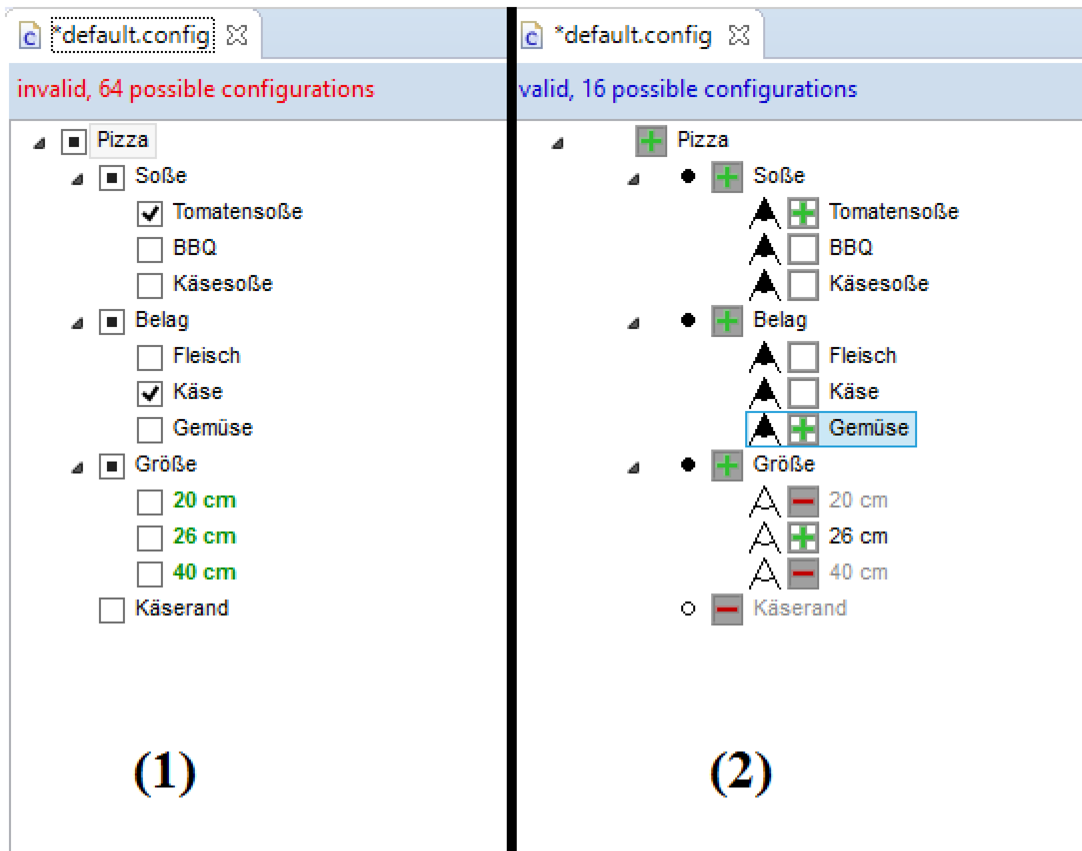

Abbildung 2.5: Standard-Konfiguration (1) und Erweiterte Konfiguration (2) in FeatureIDE

wie zum Beispiel eine OR-Beziehung, angezeigt. Außerdem lassen sich Features per Rechtsklick markieren, um sie automatisch abzuwählen.

FeatureIDE bietet zusätzlich eine Texteditoransicht der Konfiguration, die es erleichtert Konfigurationen zu kopieren oder einzufügen. Hierbei steht in jeder Zeile ein ausgewähltes Feature.

2.3.2 S.P.L.O.T.

Im Gegensatz zu FeatureIDE handelt es sich bei S.P.L.O.T. um ein webbasiertes Tool. Der Konfigurationseditor verwendet ebenfalls eine Listenansicht, bietet aber zusätzlich eine Tabelle, in der automatisch die einzelnen Konfigurationsschritte erfasst werden. Ähnlich wie bei der Advanced Configuration in FeatureIDE, lassen sich Abhängigkeiten anhand von Symbolen neben den Featurebezeichnungen ablesen.

Da S.P.L.O.T. mit decision propagation arbeitet, wird die Konfiguration automatisch angepasst, sobald ein Feature an- oder abgewählt wurde. Außerdem gibt es zwei weitere Symbole neben dem Featurenamen, welche signalisieren, ob die Aus- oder Abwahl des Features automatisch oder manuell erfolgt ist. Hierbei steht das Kopf-Symbol für einen manuellen Konfigurationsschritt und das Zahnrad-Symbol für einen automatischen. Zusätzlich sind im Tooltip dieser Icons vermerkt, in welchem Konfigurationsschritt diese Entscheidung getroffen wurde. Ein Dialogfenster

Step	Decision	#Decisions (cummulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	✔ Pizza	4 (28.6%)	3	7	1 ms
2	✔ Fleisch	5 (35.7%)	0	4	1 ms
3	✘ Kaesesosse	6 (42.9%)	0	4	0 ms
4	✔ Kaeserand	10 (71.4%)	3	5	0 ms
5	✘ Kaese	11 (78.6%)	0	2	0 ms
6	✘ Gemuese	12 (85.7%)	0	2	0 ms
7	✔ BBQ	13 (92.9%)	0	2	0 ms
8	✘ Tomatensosse	14 (100.0%)	0	0	0 ms

Abbildung 2.6: Konfiguration in S.P.L.O.T.

gibt außerdem Auskunft darüber, welche Konflikte durch die getroffene Entscheidung entstehen würden.

2.4 Zusammenfassung

In diesem Kapitel wurde die Grundidee von Softwareproduktlinienentwicklung erläutert, sowie auf deren Nutzen eingegangen. Softwareproduktlinien vereinen die Vorteile von Massenproduktion und individualisierten Produkten. Sie zeichnen sich deshalb durch hohe Variabilität aus. Dadurch sollen vor allem die Kosteneffizienz erhöht und die Entwicklungsdauer gesenkt werden. Um dieses Ziel zu erreichen, entscheidet man sich häufig für die Wiederverwendung von Programmcode. Dies kann zusätzlich eine Verbesserung der Qualität der entwickelten Produkte zur Folge haben, da Code, der häufig wiederverwendet wird, regelmäßig getestet wird. Ein Paradigma der Softwareproduktlinienentwicklung ist Feature-orientierte Programmierung. Dabei wird die Produktlinie in verschiedene Features aufgeteilt.

Des Weiteren wurden die Grundlagen zu Feature-Modellen und Produktkonfigurationen erklärt, welche wichtige Teile der Feature-orientierten Softwareentwicklung sind. Feature-Modelle bilden die Grundlage des Konfigurationsprozesses, da sie die Abhängigkeiten zwischen den Features definieren. Eine häufig verwendete Darstellungsform sind Baumdiagramme, auch Feature-Diagramme genannt. In ihnen wird jedes Feature durch einen Knoten repräsentiert. Die Beziehungen zwischen den Features werden durch vier verschiedene Arten von Kanten dargestellt. Mit Hilfe von cross-tree constraints können Feature-Diagramme noch weiter spezifiziert werden. Alternativ lassen sich Feature-Diagramme durch aussagenlogische Formeln darstellen.

Der Begriff Konfiguration beschreibt eine Menge von Features, die zusammen ein fertiges Produkt ergeben. Nicht jede Konfiguration ist gültig, da es vorkommen kann,

dass sich zwei Features gegenseitig ausschließen. Der Konfigurationsprozess kann daher bei großen Produktlinien mit vielen Features zeitaufwendig und fehleranfällig werden. Aufgrunddessen gibt es mittlerweile verschiedene Konfigurationstools, die Teile des Prozesses automatisieren und die verschiedenen Abhängigkeiten zwischen den Features berücksichtigen. Zu solchen Tools zählen unter anderem FeatureIDE und S.P.L.O.T.. Beide Tools besitzen Konfiguratoren, die die einzelnen Features als Liste darstellen.

3. Konzept

Im nachfolgenden Kapitel wird das Konzept eines Konfigurationseditors beschrieben, der die Features, ähnlich einem Feature-Diagramm, in einer Baumstruktur darstellt. Dabei wird zunächst auf den Nutzen des Editors eingegangen. Des Weiteren werden verschiedene Anforderungen definiert. Anschließend wird unter Berücksichtigung dieser Anforderungen das eigentliche Konzept des Konfigurationseditors erläutert.

3.1 Nutzen

Die Listenansicht in Konfiguratoren wie FeatureIDE und S.P.L.O.T. kann unter Umständen den Konfigurationsprozess erschweren. Besonders bei großen Produktlinien mit einer hohen Anzahl an Features kann der Nutzer schnell die Übersicht verlieren und unbeabsichtigte Produkte erstellen. Um diese Probleme zu adressieren soll der neue Konfigurationseditor, im folgenden *Tree Configurator* genannt, die Features als Baumdiagramm darstellen.

Ziel ist es, den Konfigurationseditor so zu gestalten, dass er eine nutzerfreundlichere Bedienung als der alte Konfigurationseditor ermöglicht. Nutzerfreundlich bedeutet hierbei, dass der Konfigurationsprozess insgesamt beschleunigt wird und weniger fehleranfällig ist. Dies soll in erster Linie durch eine bessere Übersichtlichkeit innerhalb des Editors erreicht werden. Der Tree Configurator soll vor allem dabei helfen, bei einer großen Anzahl an Features den Überblick zu behalten und dementsprechend eine reibungsloseren Konfigurationsprozess ermöglichen.

3.2 Anforderungen

Es gibt verschiedene Anforderungen, die der Tree Configurator erfüllen soll. Bei der Definition der Anforderungen wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden. Dabei beschreiben funktionale Anforderungen die einzelnen Funktionalitäten des Tree Configurators. Nicht-funktionale Anforderungen beschreiben hingegen die Qualität der Funktionalitäten. In diesem Abschnitt wird nur auf die funktionalen Anforderungen des Tree Configurators eingegangen.

Folgende funktionale Anforderungen ergeben sich für den Tree Configurator:

R1 Darstellen einer Konfiguration

Das System soll die Features als Baumdiagramm darstellen. Im Baumdiagramm sollen die Abhängigkeiten, die zwischen den einzelnen Features bestehen, erkennbar sein. Jede Abhängigkeit soll durch ein eigenes Symbol repräsentiert werden. Hierbei soll das Baumdiagramm des Tree Configurators dem entsprechenden Feature-Diagramm ähneln. Somit werden vier Arten von Abhängigkeiten dargestellt: OR-Beziehung und Alternativenbeziehung, sowie optionale und notwendige Features. Die Farbe des Features soll außerdem darüber Aufschluss geben, ob es sich um ein abstraktes oder ein konkretes Feature handelt.

R2 Darstellen einer Legende

Der Konfigurationseditor soll eine Legende besitzen. In dieser Legende sollen alle Icons des Baumdiagramms aufgeführt und erklärt werden.

R3 Erstellen einer Konfiguration

Der Nutzer soll in der Lage sein mit Hilfe des Tree Configurators eine Konfiguration zu erstellen. Das bedeutet, dass der Nutzer einzelne Features an- oder abwählen kann. Außerdem soll signalisiert werden, ob ein Feature Teil der aktuellen Konfiguration ist oder nicht. Dabei wird zwischen den Zuständen undefiniert, positiv und negativ unterschieden. Jeder dieser Zustände besitzt eine andere Kennzeichnung im Baumdiagramm.

R4 Aktualisieren und Speichern der Konfiguration

Auf An- oder Abwählen von Features reagiert das System , indem es intern die Konfiguration aktualisiert. Der Nutzer soll außerdem in der Lage sein, die aktuelle Konfiguration speichern zu können. Gespeicherte Konfigurationsdateien können geöffnet und wieder bearbeitet werden.

R5 Aktualisieren des Diagramms

Das Baumdiagramm soll nicht im Konfigurationseditor bearbeitet werden können. Es können keine neuen Features hinzugefügt werden und vorhandene Features können nicht verschoben oder umbenannt werden. Veränderungen des Feature-Diagramms im Feature-Diagramm-Editor wirken sich jedoch direkt auf das Diagramm im Konfigurationseditor aus, sobald die Änderungen gespeichert wurden.

R6 Verbergen von Teilbäumen

Das System soll es ermöglichen Teilbäume ein- und wieder ausklappen zu können. Dabei sollen alle Kinder des ausgewählten Features eingeklappt werden. Wird ein Feature automatisch angewählt, wird der entsprechende Teilbaum automatisch ausgeklappt.

R7 Validieren der Konfiguration

Das System soll automatisch überprüfen, ob eine Konfiguration gültig oder ungültig ist. Die Gültigkeit soll entsprechend signalisiert werden.

R8 Automatische Konfiguration

Das System kann unter bestimmten Bedingungen automatische Konfigurationsschritte durchführen. Beispielsweise werden Eltern-Knoten automatisch angewählt, sobald ein dazu gehöriger Kind-Knoten ausgewählt wurde.

3.3 Konzeptdetails

3.3.1 Auswahl des Konfigurationstools

Für die Umsetzung eines Konfigurationseditors ist es zunächst wichtig zu entscheiden, ob dieser Teil eines bereits vorhandenen Tools oder eine eigenständige Anwendung werden soll. Im Rahmen dieser Arbeit soll vor allem erörtert werden, welche Vor- und Nachteile der Tree Configurator gegenüber Konfiguratoren mit Listendarstellung hat. Daher ist es besser, auf einem bereits vorhandenen Tool aufzubauen, da dies den direkten Vergleich zweier Konfiguratoren vereinfacht. Dazu ist es notwendig, dass der vorhandene Editor ein die Features in Form einer Liste darstellt. Hierfür bieten sich verschiedene Tools an, zum Beispiel S.P.L.O.T., FeatureIDE oder fmp [3]. Außerdem lässt sich das Konzept auf diese Weise einfacher implementieren, da viele Funktionen eines bereits bestehenden Konfigurators wiederverwendet werden können, um den Tree Configurator zu entwickeln.

Als Grundlage für den Tree Configurator wurde das Eclipse-Plugin FeatureIDE ausgewählt. Ein Grund für diese Entscheidung ist der mehrseitige Konfigurationseditor. Dieser ist leicht erweiterbar und bietet deshalb eine gute Grundlage für die Entwicklung des Tree Configurators. Des Weiteren ist der Konfigurationseditor eng mit dem Feature-Diagramm verknüpft. Dies sollte es erleichtern, die Baumdarstellung auf den Konfigurationseditor zu übertragen.

3.3.2 Umsetzung der Anforderungen

Entsprechend der Anforderung R1 erfolgt der Konfigurationsprozess durch die Nutzung eines Baumdiagramms. Jeder Knoten repräsentiert hierbei ein Feature der Produktlinie, ähnlich dem Feature-Diagramm. Durch die Baumdarstellung ist bereits

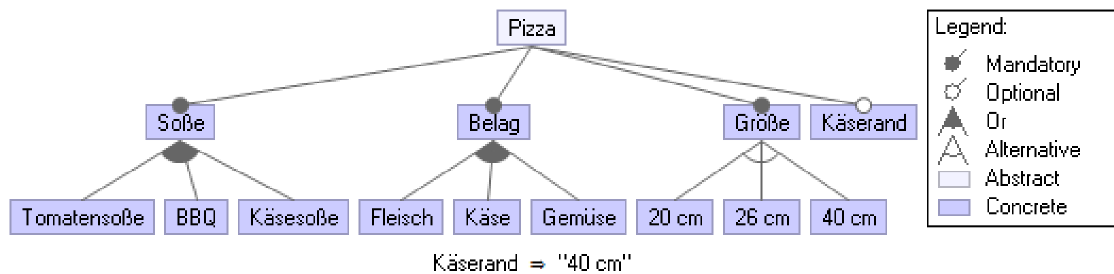

Abbildung 3.1: Diagramm im Feature-Diagramm-Editor von FeatureIDE

auf den ersten Blick ersichtlich, welche Features in einer Eltern-Kind-Beziehung stehen. Die restlichen Beziehungsarten, wie beispielsweise notwendige Features, sind im Diagramm ebenfalls erkennbar. Diese Abhängigkeiten werden durch die aus dem Feature-Diagramm bekannten Symbole dargestellt. Dadurch finden sich Nutzer, die bereits mit dem Feature-Diagramm vertraut sind, besser im Editor zurecht. Durch die klare Darstellung der Abhängigkeiten im Tree Configurator soll der Nutzer sofort erkennen können, welche Features sich gegenseitig beeinflussen. Damit soll eines der Probleme der alten Konfiguratoren umgangen werden. Als Beispiel für das Baumdiagramm dient Abbildung Abbildung 3.1, die das Diagramm der Pizza-Produktlinie innerhalb des Feature-Diagramm-Editors zeigt.

Zum besseren Verständnis des Diagramms, enthält der Tree Configurator eine Legende (R2). Sie beschreibt die verschiedenen Arten von Beziehungen im Baumdiagramm. Dadurch können auch Nutzer ohne Vorkenntnisse über das Feature-Diagramm erkennen, welche Abhängigkeiten zwischen den Features bestehen.

Das Erstellen einer Konfiguration (R3) erfolgt durch einen Doppelklick auf einzelne Features. Features können die Zustände undefiniert, positiv oder negativ annehmen. Zu Beginn des Konfigurationsprozesses sind alle Features im undefinierten Zustand. Durch das Doppelklicken eines undefinierten Features wechselt dieses zunächst in den positiven Zustand und ist dementsprechend Teil der Konfiguration. Durch weiteres Doppelklicken kann zwischen dem positiven und negativen Zustand gewechselt werden. Features, die sich im negativen Zustand befinden, sind kein Teil der Konfiguration. Die möglichen Zustandsübergänge werden in Abbildung Abbildung 3.2 verdeutlicht.

Zur Darstellung der Zustände werden für die Knoten des Baums verschiedenfarbige Rahmen verwendet. Die ursprüngliche Farbe der Knoten bleibt dabei unverändert, damit weiterhin zwischen abstrakten und konkreten Features unterschieden werden kann. Features, die sich noch im undefinierten Zustand befinden, besitzen keinen speziellen Rahmen. Sobald ein Feature angewählt wurde, also sich im positiven Zustand befindet, erhält es einen grünen Rahmen. Abgewählte und somit im negativen Zustand befindliche Features erhalten einen roten Rahmen. Abbildung 3.3 verdeutlicht die farbliche Kennzeichnung der verschiedenen Zustände.

Durch Auswählen der Features wird die interne Konfiguration entsprechend angepasst (R4). Die Veränderungen wirken sich automatisch auch auf die anderen Konfigurationseditoren aus, da sie alle auf die selben Konfigurationsdaten zugreifen. Die

undefiniert
positiv
negativ
Zustandswechsel des Features erfolgt durch Doppelklick

Abbildung 3.2: Zustandsdiagramm für den Konfigurationsprozess

positiver Zustand
negativer Zustand
undefinierter Zustand

Abbildung 3.3: Darstellung der Zustände im Feature-Diagramm

Konfiguration kann außerdem gespeichert werden und zu einem späteren Zeitpunkt wieder bearbeitet werden.

Das Diagramm selbst kann jedoch nicht verändert werden (R5). Umbenennungen sowie das Verschieben, Hinzufügen oder Löschen von Features kann nur im Feature-Diagramm-Editor erfolgen. Dadurch soll gewährleistet werden, dass der Nutzer während es Konfigurationsprozesses keinen Zugriff auf Funktionen hat, die für ebendiesen Prozess nicht relevant sind. Hierdurch soll die Fehleranfälligkeit verringert werden. Allerdings können weiterhin Anpassungen im Feature-Diagramm-Editor vorgenommen werden. Jede Anpassung wird in das Diagramm im Tree Configurator übernommen.

Wie in Anforderung R6 beschrieben, soll der Nutzer außerdem in der Lage sein einzelne Teilbäume ein- und ausklappen zu können. Hierdurch kann er sich während des Konfigurationsprozesses besser auf einzelne Teile des Feature-Diagramms konzentrieren und Teile, die momentan nicht relevant sind, ausblenden. Das Ein- und Ausklappen erfolgt über das Kontextmenü oder mit Hilfe einer Tastenkombination. Davon betroffen sind alle Kind-Knoten des ausgewählten Features. Die Anzahl der

eingeklappten Knoten kann anhand einer Markierung am Eltern-Knoten abgelesen werden. Beeinflusst die Zustandsänderung eines Features ein anderes Feature, welches gerade eingeklappt ist, so wird dieses automatisch wieder ausgeklappt. Dadurch soll der Nutzer stets mitverfolgen können, welche Auswirkung eine Veränderung der aktuellen Konfiguration hat.

Des Weiteren soll sich der Tree Configurator nicht nur grafisch am Feature-Diagramm orientieren, sondern auch intern damit verknüpft sein. Dadurch wirken sich Veränderungen am Feature-Diagramm automatisch auch auf den Tree Configurator aus. Damit kann der Nutzer sicher sein, dass er immer auf der aktuellen Menge von Features arbeitet. Wie in Anfoderung R7 und R8 beschrieben wurde, werden Teile des Konfigurationsprozesses automatisiert. Zum Einen wird zu jedem Zeitpunkt automatisch geprüft, ob die Konfiguration gültig ist oder nicht. Eine Konfiguration ist nur dann gültig, wenn sich kein Feature im undefinierten Zustand befindet und die Belegung der einzelnen Features nicht mit den im Feature-Modell definierten Abhängigkeiten kollidiert. Zum Anderen kann das System automatische Konfigurationsschritte durchführen. Dies kann zum Beispiel der Fall sein, wenn ein Feature ausgewählt wird, welches einen Eltern-Knoten besitzt. Dadurch wird automatisch der Eltern-Knoten ausgewählt, da dies durch die Definition des Feature-Modells vorgeschrieben wird. Features können aber auch automatisch abgewählt werden, wenn beispielsweise ein Feature gewählt wird, welches Teil einer Alternativen-Beziehung ist. Dabei werden die restlichen Knoten der Beziehung abgewählt.

3.4 Zusammenfassung

In diesem Kapitel wurde das Konzept des Tree Configurator vorgestellt. Der Tree Configurator soll eine Alternative zu herkömmlichen Konfigurationseditoren darstellen und dem Nutzer eine schnelle und einfache Bedienung ermöglichen. Der neue Editor wird kein eigenständiges Tool, sondern baut auf FeatureIDE auf. Dadurch wird es einfacher den Tree Configurator mit Konfiguratoren mit Listendarstellung zu vergleichen. Des Weiteren können so Teile des Programmcodes der vorhandenen FeatureIDE-Konfiguratoren genutzt werden, um den Tree Configurator zu implementieren.

Im Gegensatz zu den meisten anderen Konfiguratoren wird der Tree Configurator die Features in Form eines Baumdiagramms darstellen. Dadurch soll dem Nutzer ein besserer Überblick über die Produktlinie verschafft werden. Das Anzeigen der verschiedenen Abhängigkeiten im Baumdiagramm trägt ebenfalls zur Verständlichkeit der Produktlinie bei. Durch diese Maßnahmen soll vor allem vermieden werden, dass der Nutzer ungewollte Konfigurationen erstellt, ohne es zu bemerken. Zum bessere Verständnis des Diagramms gibt es, wie auch im Feature-Diagramm-Editor, eine Legende.

Das eigentliche Erstellen der Konfiguration erfolgt direkt über das Baumdiagramm. Durch Anklicken der einzelnen Features können diese an- oder abgewählt werden. Der Zustand der Features wird durch eine farbige Umrahmung der Knoten gekennzeichnet. Das Bearbeiten des Diagramms ist im Konfigurationseditor nicht möglich. Dadurch soll vermieden werden, dass der Nutzer während des Konfigurationsprozesses unwissentlich das Feature-Diagramm verändert, was wiederum in einer fehlerhaften beziehungsweise ungewollten Konfiguration resultieren kann.

Der Tree Configurator ist außerdem mit dem Feature-Diagramm verknüpft. Dies ermöglicht eine automatische Validierung der aktuellen Konfiguration unter Berücksichtigung der Abhängigkeiten des Feature-Modells.

Aus diesem Konzept wurde ein Prototyp entwickelt. Wie genau dieser umgesetzt wurde, wird im nächsten Kapitel erläutert.

4. Implementierung

In diesem Kapitel wird erläutert, wie das im vorherigen Kapitel entwickelte Konzept umgesetzt wurde. Zunächst werden die Komponenten des FeatureIDE-Plugins, auf denen aufgebaut wird, näher erläutert. Der Fokus liegt dabei auf dem Configuration Editor und dem Feature Diagram Editor, da der Tree Configurator auf diesen beiden Komponenten aufbaut. Anschließend wird darauf eingegangen, wie die Kernfunktionen des Tree Configurators umgesetzt wurden.

4.1 Überblick über die FeatureIDE-Architektur

Das FeatureIDE-Plugin wurde in Java entwickelt. Dementsprechend wird auch der Tree Configurator in Java implementiert. Das Plugin setzt sich aus verschiedenen Komponenten zusammen. Zu den Wichtigsten gehören hierbei der Configuration Editor, der Feature Diagram Editor, das Collaboration Diagram und die FeatureIDE-Outline. Für die Umsetzung des vorgestellten Konzepts sind jedoch nur der Configuration Editor und der Feature Diagram Editor interessant.

4.1.1 Configuration Editor

Im Configuration Editor können unter Berücksichtigung des Feature-Diagramms Produktkonfigurationen erstellt werden. Der Editor verfügt über drei verschiedene Menüseiten: Configuration, Advanced Configuration und Source. Über jede dieser Seiten können Konfigurationen erstellt werden. Die verschiedenen Konfiguratoren greifen dabei immer auf die selben Konfigurationsdaten zu, wodurch die aktuelle Konfiguration auf allen Seiten stets gleich ist. Jede Seite wurde durch eine eigene Klasse umgesetzt. Die Verwaltung der einzelnen Seiten erfolgt über die `ConfigurationEditor`-Klasse. Hier werden die Seiten dem Editor hinzugefügt und initialisiert. Dementsprechend wird auch der Tree Configurator dort eingefügt.

Über Objekte der ConfigurationEditor-Klasse kann auch auf die Konfiguration zugegriffen werden. Diese wird durch die Klasse `Configuration` repräsentiert. Die verschiedenen Features der Produktlinie sind in einer Array-List gespeichert. Mit

Hilfe einer Hashtabelle können die gewünschten Features durch Angabe des Feature-Namens aus der Liste extrahiert werden. Jedes Feature speichert außerdem die Information darüber, ob es sich im positiven, negativen oder undefinierten Zustand befindet. Durch Methoden wie `setManual()` oder `setAutomatic()` kann der Zustand des Features verändert werden.

4.1.2 Feature Diagram Editor

Im Feature Diagram Editor wird das Feature-Diagramm erstellt. Über ein Kontextmenü können verschiedene Diagramm-Elemente hinzugefügt oder bereits vorhandene Elemente bearbeitet werden. Der Editor selbst wird durch die Klasse `FeatureDiagramEditor` realisiert. Die grafischen Elemente des Editors werden über die `GraphicalEditPartFactory` erzeugt. Diese Klasse erzeugt, je nach Art der einzelnen Elemente, EditParts, welche als Bausteine des Editors fungieren. Die Klassen `FeatureEditPart`, `ConstraintEditPart` und `ConnectionEditPart` repräsentieren dementsprechend Features, cross-tree constraints und die Verbindungselemente zwischen den Features.

Die grafische Darstellung der FeatureEditParts erfolgt über die Klasse `FeatureFigure`. In ihr kann unter anderem die Farbe der Features verändert werden. Die globalen Farbeinstellungen können in einem separaten Menü angepasst werden.

Die Interaktion mit dem Diagramm wird durch verschiedene Action-Klassen ermöglicht. Beispielsweise sorgt die Klasse `SelectionAction` dafür, dass die einzelnen Teile des Diagramms ausgewählt werden können.

4.2 Erstellen und Einbinden der Editorseite

Da das Diagramm des Tree Configurators große Ähnlichkeit mit dem Feature-Diagramm hat, wurden zur Umsetzung verschiedene Teile der FeatureDiagramEditor-Klasse übernommen, um die neue TreeConfigurationPage-Klasse zu kreieren. Die TreeConfigurationPage erbt ebenso wie der FeatureDiagramEditor von ScrollingGraphicalViewer. Somit enthält diese Klasse wichtige Kernfunktionen, zum Beispiel eine Funktion zur Initialisierung des GraphicalViewers.

Zum Einbinden des TreeConfigurators in den vorhandenen Konfigurationseditor, musste zunächst eine neue Seite im Konfigurationseditor von FeatureIDE angelegt werden. Das Erstellen neuer Seiten geschieht in der createPages-Methode der ConfigurationEditor-Klasse. Da die verschiedenen Seiten des Konfigurationseditors von der Klasse ConfigurationTreeEditorPage erben, war es problematisch die neue Seite einzufügen, da diese von ScrollingGraphicalViewer erbt. Dementsprechend mussten verschiedene Anpassungen vorgenommen werden.

4.3 Darstellung des Diagramms

Wie auch beim Feature Diagram Editor, basiert die Darstellung des Diagramms im Tree Configurator auf der Nutzung von EditParts. Da sich Anpassungen der EditPart-Klassen auch auf den Feature Diagram Editor auswirken würden, mussten zunächst eigene EditParts erstellt werden. Dabei wurden große Teile der alten

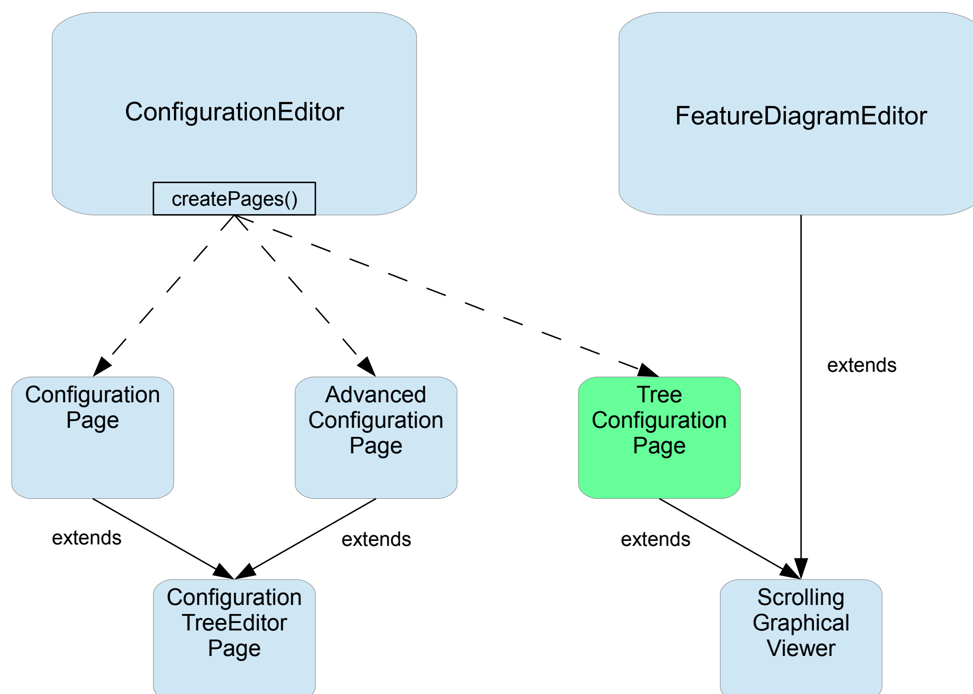

Abbildung 4.1: Übersicht über die Editoren und Editorseiten

EditParts übernommen, da die neuen EditParts von ihnen erben. Der wichtigste der neuen EditParts ist der `ConfigTreeFeatureEditPart`, da im Tree Configurator nur mit den Features des Diagramms interagiert werden soll. Zusätzlich musste eine eigene EditPartFactory entwickelt werden, um die neuen EditParts zu initialisieren.

Für das Implementieren der farbigen Rahmen der Features musste die Klasse FeatureFigure angepasst werden, da sie die Grafiken der Features liefert. In der Methode `setProperties()` können verschiedene Eigenschaften der Grafik festgelegt werden, wie zum Beispiel Farbe oder Größe. Damit der Grafik des Features ein farbiger Rahmen zugewiesen werden kann, muss zunächst abgefragt werden, in welchem Zustand sich das Feature befindet. Danach wird der Rahmen mit Hilfe der Methode `setBorder()` hinzugefügt. Befindet sich das Feature im positiven Zustand, erhält es einen grünen Rahmen, befindet es sich im negativen Zustand, erhält es einen roten Rahmen. Wurde für das Feature noch keine Auswahl getroffen, befindet es sich also undefinierten Zustand, erhält es einen schwarzen Rahmen. Damit die Grafik immer aktuell bleibt, muss nach jeder Zustandsänderung setProperties() aufgerufen werden.

4.4 Reagieren auf Eingaben

Der ConfigTreeFeatureEditPart besitzt die `performRequest()`-Methode, mit deren Hilfe auf Eingaben reagiert werden können. Bei Auswahl eines Features sendet die SelectionAction, welche in der TreeConfigurationPage initialisiert wurde, eine so

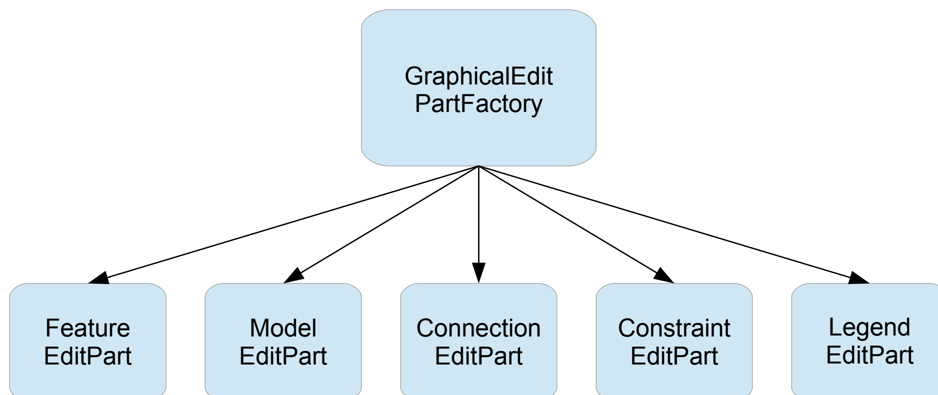

Abbildung 4.2: Übersicht der erzeugten EditParts

genannte Selection-Request. Der ConfigTreeFeatureEditPart verarbeitet diese Anfrage, indem er den Zustand des selektierten Features verändert und die aktuelle Konfiguration anpasst.

Um die Konfiguration anzupassen, muss zunächst der Zustand des ausgewählten Features ausgelesen werden. Durch Anwählen eines Features im undefinierten Zustand wird es zunächst in den positiven Zustand versetzt. Befindet ein Feature sich nicht mehr im undefinierten Zustand, wird zwischen dem positiven und negativen Zustand hin- und hergeschalten, sobald dieses Feature angewählt wurde.

4.5 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie ein erster Prototyp des Tree Configurators entwickelt wurde. Dieser Prototyp basiert auf dem zuvor entwickelten Konzept und erfüllt somit verschiedene Nutzeranforderungen. Er wurde als Teil des Eclipse-Plugins FeatureIDE implementiert und wurde daher in Java programmiert.

Über eine neue Seite im Konfigurationseditor von FeatureIDE kann auf den Tree Configurator zugegriffen werden, die TreeConfigurationPage. Anders als die ConfigurationPage und die AdvancedConfigurationPage erbt sie jedoch nicht von der Klasse ConfigurationTreeEditorPage. Da der Tree Configurator sich bei der Darstellung des Diagramms am Feature Diagram Editor orientiert, erbt die TreeConfigurationPage stattdessen von der Klasse ScrollingGraphicalViewer. Um die TreeConfigurationPage dennoch in den Konfigurationseditor einbinden zu können, mussten daher verschiedene Anpassungen vorgenommen werden.

Das Darstellen der einzelnen Diagrammelemente erfolgte über das Implementieren einer zusätzlichen GraphicalEditPartFactory, welche verschiedene EditParts, wie beispielsweise FeatureEditParts oder ConnectionEditParts, erzeugt. Da ein Verändern dieser EditParts auch das Modell im Feature Diagram Editor beeinflusst hätte, wurden für den Tree Configurator neue EditPart-Klassen angelegt. Innerhalb einer

dieser Klassen, der ConfigTreeFeatureEditPart-Klasse, wurde der eigentliche Konfigurationsprozess umgesetzt. Durch Anwählen der Features im Editor lässt sich so der Zustand des Features verändern. Sobald sich der Zustand geändert hat, wird zugleich die Grafik des Features angepasst. Dafür wurden in der Klasse FeatureFigure verschieden farbige Rahmen erstellt, die dem Feature je nach Zustand zugewiesen werden.

Mit Hilfe dieses Prototyps kann nun das zuvor entwickelte Konzept bewertet werden.

5. Evaluierung

In diesem Kapitel wird das Konzept des Tree Configurators mit Hilfe des entwickelten Prototyps evaluiert. Hierfür wird eine Gruppe von Testpersonen zwei Produkte einer Produktlinie erstellen. Das erste Produkt wird mit dem klassischen Konfigurator von FeatureIDE erstellt, das zweite mit dem Tree Configurator. Die Ergebnisse der darauf folgenden Befragung werden gesammelt und anschließend ausgewertet.

5.1 Durchführung der Evaluierung

Im Zuge der Evaluierung sollen die zuvor gestellten wissenschaftlichen Fragen beantwortet werden:

- Welche Vor- und Nachteile weist die alternative Darstellungsform gegenüber der Listendarstellung auf?
- Ermöglicht die alternative Darstellungsform eine aus Nutzersicht einfachere Bedienung beim Erstellen von Produktkonfigurationen als die Listendarstellung?

Da sich diese Fragestellungen nur schwer durch das Sammeln von Messwerten beantworten lassen, soll der Tree Configurator mit Hilfe einer qualitativen Evaluierung bewertet werden. Bei einer qualitativen Evaluierung werden nur die nicht messbaren Eigenschaften berücksichtigt. Dementsprechend eignet sie sich gut, um Eigenschaften wie Bedienbarkeit oder Aussehen zu bewerten.

Zur Durchführung dieser Evaluierung wird einer Gruppe von Nutzern ein Schema für ein spezielles Produkt einer Softwareproduktlinie vorgelegt. Anschließend sollen die Nutzer dieses Produkt mit Hilfe des klassischen Konfigurators von FeatureIDE zusammenstellen. Danach müssen die Nutzer ein weiteres Produkt erstellen, diesmal unter Verwendung des Tree Configurators. Zum Abschluss werden den Probanden verschiedene Fragen gestellt.

5.1.1 Fragestellungen

Mit Hilfe der folgenden Fragen soll es den Probanden ermöglicht werden, ein auswertbares Feedback zu liefern:

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Frage Q1 beschäftigt sich konkret mit den negativen Aspekten der Anwendung. Dadurch können mögliche Designschwachstellen erkannt werden. Außerdem wird hierdurch eine erste Gegenüberstellung der Konfiguratoren möglich. Mit Frage Q2 sollen die positiven Erfahrungen der Probanden bei der Nutzung des Tree Configurators zusammengetragen werden. Gleichzeitig sollen auch die negativen Aspekte des klassischen Konfigurators besprochen werden, um einen direkten Vergleich zu ermöglichen. Mit Frage Q3 soll herausgefunden werden, welchen Konfigurator die Probanden bevorzugen, da dies Informationen darüber liefern könnte, welcher der Konfiguratoren sich leichter bedienen lässt. Es ist wahrscheinlich, dass sich die Probanden für den Konfigurationseditor entscheiden, der eine bessere Bedienung ermöglicht.

5.1.2 Produkte

Für die Durchführung müssen die Probanden zwei Produkte mit Hilfe der verschiedenen Konfigurationseditoren erstellen. Als Produktlinie wird eine erweiterte Repräsentation des Pizzabestellservices verwendet, die in Abbildung 5.1 zu sehen ist. Die ursprüngliche Variante ist relativ simpel gehalten und daher ungeeignet für den Test. Deswegen weist die neue Produktlinie eine höhere Anzahl von möglichen Features auf. Eine vermehrte Zahl von Verzweigungen innerhalb des Baums und neu hinzugefügte cross-tree constraints sorgen für zusätzliche Komplexität.

Von den Probanden sollen folgende Produkte erstellt werden:

Produkt 1

Dieses Produkt wird mit dem klassischen Konfigurator erstellt.

- Soße: BBQ
- Fleisch: Salami und Schinken
- Käse: Cheddar
- Gemüse: Paprika und Tomaten
- Fisch: -
- Größe: 20 cm
- Extra Käse
- Geschnitten: Nein

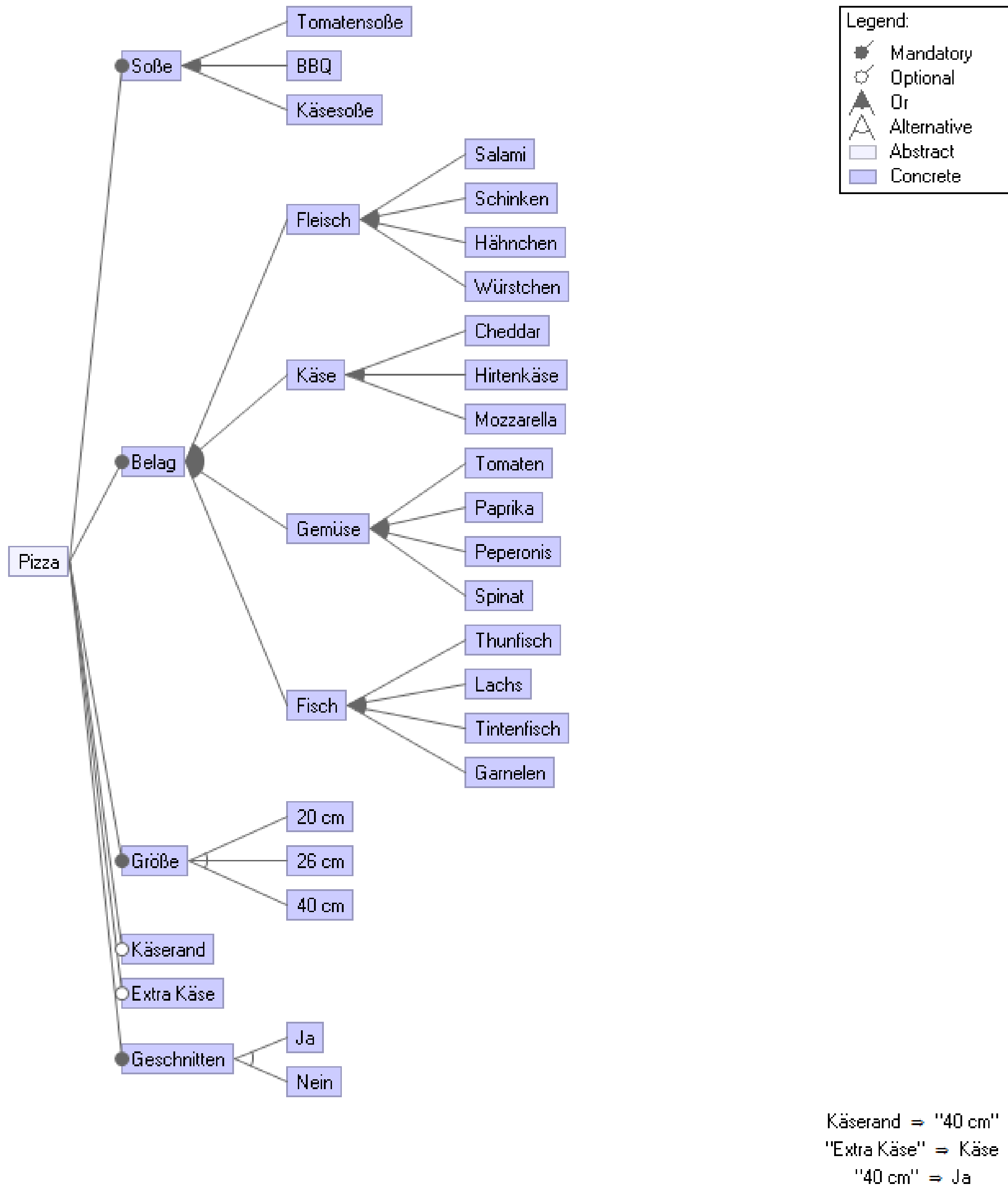

Abbildung 5.1: Erweitertes Feature-Diagramm eines Online-Pizzabestellservices

Produkt 2

Dieses Produkt wird mit dem Tree Configurator erstellt.

- Soße: Käsesoße
- Fleisch: -
- Käse: Hirtenkäse und Mozzarella
- Gemüse: Spinat
- Fisch: Lachs und Garnelen
- Größe: 40 cm
- Käserand
- Geschnitten: Ja

5.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Evaluation vorgestellt. Das Sammeln der Informationen geschah in Form eines Interviews, bei dem die zuvor aufgestellten Fragen Q1, Q2 und Q3 als Leitfaden dienten. Die Aussagen von fünf Probanden wurden handschriftlich protokolliert. Die hier aufgeführten Ergebnisse sind Zusammenfassungen der Protokolle und enthalten die Kernaussagen der Interviews.

Proband 1

Student, Bachelor Informatik, Erfahrung mit Softwareproduktlinien und FeatureIDE

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Der Proband hätte sich gewünscht, dass die Eltern-Knoten beim Anwählen eines Features automatisch angewählt worden wären, wie es beim Standardkonfigurator der Fall ist. Auch der Wurzelknoten sollte standardmäßig angewählt sein. Des Weiteren hätte er sich eine Textzeile oder Auflistung gewünscht, die die aktuelle Konfiguration anzeigt.

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Dem Probanden hat die Baumdarstellung sehr gut gefallen. Er empfand sie als übersichtlicher und war der Meinung, dass er mit ihr bei größeren Produktlinien schneller Produkte erstellen könnte, als mit dem Standardkonfigurator. Außerdem fand er, dass die Abhängigkeiten zwischen den Features im Baumdiagramm deutlich besser erkennbar waren, als in der Listendarstellung. Das Vorhandensein einer Legende empfand er ebenfalls als positiv.

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Der Proband würde sich in Zukunft immer für den Tree Configurator entscheiden, da ihm besonders wichtig ist, dass er während des Konfigurationsprozesses einen Überblick über die Abhängigkeiten und Beziehungen der Features hat. Des Weiteren glaubt er, dass die Übersichtlichkeit der Baumdarstellung den Konfigurationsprozess beschleunigen würde.

Proband 2

Student, Bachelor Informatik, keine Erfahrung mit Softwareproduktlinien

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Der Proband empfand es als umständlich, dass die restlichen Features einer Alternativen-Beziehung nicht abgewählt wurden, sobald ein Feature angewählt wurde. Außerdem hätte er sich ein automatisches Anwählen von Eltern-Knoten gewünscht.

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Der Proband fand die Baumdarstellung optisch ansprechend und zudem übersichtlicher als die Listendarstellung des Standardkonfigurators. Des Weiteren bemängelte er beim Standardkonfigurator, dass keine Abhängigkeiten ersichtlich waren.

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Der Proband würde sich bei weitere Konfigurationen für den Standardkonfigurator entscheiden, da beim Tree Configurator kein automatisches An- beziehungsweise Abwählen möglich ist.

Proband 3

Student, Bachelor Informatik, keine Erfahrung mit Softwareproduktlinien

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Der Proband hätte sich eine Funktion gewünscht, mit der alle Features, die sich im undefinierten Zustand befinden, in den negativen Zustand übergehen. Dadurch müsste er nur die gewünschten Features auswählen und könnte den Rest mit einem Befehl abwählen.

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Dem Probanden gefiel, dass die Abhängigkeiten zwischen den Features sofort ersichtlich waren. Dementsprechend empfand er das Baumdiagramm des Tree Configurators übersichtlicher als die Listendarstellung des Standardkonfigurators. Er glaubte außerdem, dass die automatischen Konfigurationsschritte im Standardkonfigurator bei größeren Modellen für Verwirrung sorgen könnten.

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Der Proband würde in Zukunft den Tree Configurator nutzen, da er ihn optisch ansprechender findet als den Standardkonfigurator. Beispielsweise fand er die farbigen Rahmen zur Darstellung der Zustände angenehmer als die Checkboxen im Standardkonfigurator.

Proband 4

Student, Bachelor Informatik, Erfahrung mit Softwareproduktlinien

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Der Proband hätte sich gewünscht, dass Features nicht per Doppelklick, sondern per Einfachklick angewählt worden wären. Außerdem bemängelte er, dass Eltern-Knoten von ausgewählten Features nicht automatisch angewählt werden.

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Dem Probanden gefiel die grafische Oberfläche des Tree Configurators. Er sagte außerdem, dass die Anzeige der Abhängigkeiten im Diagramm den Konfigurationsprozess einfacher gestaltet und er demzufolge schneller arbeiten könnte.

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Der Proband bevorzugt den Tree Configurator, da ihm wichtig ist, dass die Abhängigkeiten zwischen den Features im Editor angezeigt werden.

Proband 5

Student, Bachelor Computervisualistik, keine Erfahrung mit Softwareproduktlinien

Q1 Gibt es Dinge, die am Tree Configurator verbessert werden könnten?

Dem Probanden hätte es gefallen, wenn der Tree Configurator Features automatisch anwählt, wie es beim Standardkonfigurator der Fall ist. Er hätte sich außderm eine Funktion gewünscht, mit der er mit einem Befehl alle Features im undefinierten Zustand an- oder abwählen kann.

Q2 Gibt es Dinge, die der Tree Configurator besser macht als der Standardkonfigurator?

Der Proband empfand den Tree Configurator als übersichtlicher, da im Standardkonfigurator nicht immer klar war, welche Features miteinander in Beziehung stehen. Außerdem half die Legende ihm die verschiedenen Abhängigkeiten des Diagramms zu verstehen.

Q3 Welchen Konfigurator würden Sie in Zukunft für das Erstellen von Konfigurationen verwenden und warum?

Der Proband würde in Zukunft den Tree Configurator verwenden, da ihm der Standardkonfigurator zu unübersichtlich war und keine Abhängigkeiten zwischen den Features darstellt.

5.3 Auswertung

5.3.1 Interpretation der Ergebnisse

Nach der Befragung zeigt sich, dass sich die Probanden in vielen Punkten einig waren. Alle Testpersonen empfanden die Baumdarstellung optisch ansprechender als die Listendarstellung. Dies könnte sich unter anderem auch darauf zurückführen lassen, dass sie das Baumdiagramm insgesamt viel übersichtlicher fanden als die Liste des Standardkonfigurators. Der Hauptgrund dafür war das Vorhandensein der Abhängigkeiten im Editor. Den Probanden fiel es im Standardkonfigurator deutlich

schwerer Beziehungen zwischen den Features zu erkennen. Ein großer Kritikpunkt war jedoch das Fehlen von automatisierten Konfigurationsschritten. Jeder Proband hätte sich gewünscht, dass die Eltern-Knoten eines Features automatisch angewählt werden, sobald dieses Feature angewählt wurde. Zwei Probanden wünschten sich zusätzlich eine Funktion, mit der Features im undefinierten Zustand automatisch abgewählt werden können.

Mit Hilfe der gesammelten Ergebnisse lassen sich nun die wissenschaftlichen Fragen R1 und R2 beantworten. Zur Beantwortung der Frage R1 sollten die Vor- und Nachteile des Tree Configurators ermittelt werden. Ein wichtiger Vorteil gegenüber dem Standardkonfigurator ist die bessere Übersichtlichkeit des Tree Configurators. Aufgrund der Baumdarstellung wird sofort deutlich, welche Features in einer Eltern-Kind-Beziehung stehen. Die bessere Übersichtlichkeit ist eng mit einem weiteren Vorteil verknüpft: dem Darstellen von Abhängigkeiten. Ein weiterer Vorteil wäre das grafische Interface. Die Probanden empfanden die grafische Darstellung des Editors als sehr angenehm und hoben unter anderem die Legende und die farbliche Kennzeichnung der Zustände hervor. Der einzige Nachteil, der sich aus den Ergebnissen ableiten lässt, ist die fehlende Automatisierung von Konfigurationssschritten.

Frage R2 lässt sich nur bedingt beantworten. Alle Probanden fanden die Baumdarstellung übersichtlicher als die Listendarstellung und zwei der Probanden glaubten, dass der Tree Configurator den Konfigurationsprozess beschleunigen würde. Eine bessere Übersichtlichkeit resultiert letztlich auch in einer einfacheren Bedienbarkeit. Dagegen spricht allerdings einer der Hauptkritikpunkte des Tree Configurators: das Fehlen von automatisierten Konfigurationssschritten. Im Tree Configurator müssen Features manuell ausgewählt werden, deren Zustand sich sonst automatisch geändert hätte. Dies kann den Konfigurationsprozess wiederum verlangsamen. Da vier der fünf Probanden den Tree Configurator weiterhin verwenden würden, überwiegen die Vorzüge des Tree Configurators diesen Nachteil. Für eine eindeutigere Antwort auf diese Frage wäre es allerdings sinnvoll, eine weitere Evaluation durchzuführen, bei der ein neuer Prototyp verwendet wird. Dieser Prototyp sollte, wie im Konzept beschrieben, ein automatisches An- und Abwählen ermöglichen.

5.3.2 Aussagekraft der Ergebnisse

Die Evaluation verschafft einen ersten Eindruck darüber, wo die Stärken und Schwächen des Tree Configurators liegen. Es ist allerdings nicht möglich eine endgültige Aussage darüber zu treffen, ob der Tree Configurator anderen Konfiguratoren mit Listendarstellung überlegen ist. Dies hat verschiedene Gründe.

Zum Einen ist die für die Evaluation erstellte Produktlinie verglichen mit in der Industrie verwendeten Produktlinien eher klein. Das Ziel des Tree Configurators war es aber vor allem bei größeren Produktlinien eine bessere Bedienbarkeit zu ermöglichen.

Zum Anderen war es aufgrund des Zeitrahmens der Arbeit nicht möglich alle im Konzept beschriebenen Funktionen zu implementieren. Beispielsweise wurde von mehreren Probanden bemängelt, dass im Tree Configurator keine automatisierten Konfigurationsschritte möglich sind. Die Evaluation hätte möglicherweise andere Ergebnisse geliefert, wenn diese Funktion bereits implementiert gewesen wäre.

Ein weiterer Grund für die eingeschränkte Aussagekraft der Evaluation ist die Anzahl von Probanden. Für die Evaluation wurden fünf Probanden befragt. Auch wenn sich die Meinungen der Probanden größtenteils gleichen, wäre mit einer höheren Zahl deutlicher geworden,ob und in welcher Hinsicht der Tree Configurator anderen Konfiguratoren überlegen ist.

5.4 Zusammenfassung

In diesem Kapitel wurde der zuvor entwickelte Prototyp des Tree Configurators evaluiert. Ziel der Evaluation war es, das Konzept des Tree Configurators zu bewerten und die Forschungsfragen R1 und R2 zu beantworten.

Als Teil einer qualitativen Evaluation mussten fünf Probanden zwei Produkte einer erweiterten Variante der Pizza-Produktlinie erstellen. Für ein Produkt sollte der Standardkonfigurator von FeatureIDE und für das andere Produkt der Tree Configurator verwendet werden. Anschließend wurden die Probanden interviewt. Die Fragen Q1, Q2 und Q3 dienten dabei als Leitfaden. Die Antworten der Probanden wurden handschriftlich protokolliert. Anschließend wurden die Kernaussagen zusammengetragen.

Aus den Ergebnissen ließen sich verschiedene Vor- und Nachteile des Tree Configurators ableiten. Der wichtigste Vorteil des Tree Configurators ist seine Übersichtlichkeit. Hingegen wurde vor allem bemängelt, dass Features nicht automatisch an- oder abgewählt werden. Mit Hilfe dieser Ergebnisse ließ sich die Forschungsfrage R1 beantworten. Für Frage R2 konnte keine endgültige Antwort gefunden werden. Die Durchführung einer weiteren Evaluation mit einem verbesserten Prototypen wäre daher hilfreich, um eine eindeutige Antwort auf diese Frage zu erhalten.

6. Verwandte Arbeiten

Im Verlauf dieser Arbeit wurde der Konfigurationseditor von FeatureIDE vorgestellt und eine zusätzliche Editoransicht, der Tree Configurator, entwickelt. In diesem Kapitel werden drei weitere Themen vorgestellt, die sich ebenfalls mit Konfigurationen auseinandersetzen.

6.1 S2T2

Bei S2T2 [2] handelt es sich um ein Tool zur Konfiguration von Softwareproduktlinien. Ähnlich dem Tree Configurator können über eine grafische Repräsentation des Feature-Modells Produktkonfigurationen erstellt werden. Im Diagramm des Konfigurationseditors lassen sich die verschiedenen Abhängigkeiten, wie OR und Alternative sowie notwendige und optionale Features, ablesen. Zusätzliche Abhängigkeiten werden allerdings nicht durch cross-tree constraints dargestellt. Stattdessen gibt es zusätzliche Verbindungselemente zwischen den Knoten. Eine grüne Verbindung mit Pfeilende repräsentiert eine Implikation. Eine Verbindung von Feature A nach Feature B würde bedeuten, dass Feature B ausgewählt werden muss, wenn auch Feature A ausgewählt wurde. Zwei Features können sich auch gegenseitig ausschließen, das heißt, sie können nicht gleichzeitig aktiv sein. Diese Art der Abhängigkeit wird durch eine rote Verbindung dargestellt.

Im Gegensatz zu den drei Zuständen des Tree Configurators (positiv, negativ und undefiniert), können die Features in S2T2 vier verschiedene Zustände besitzen: *ausgewählt*, *eliminiert*, *nicht entschieden* und *nicht erfüllbar*. Zusätzlich erhält jedes Feature eine Kennzeichnung, die darüber Auskunft gibt, was der Ursprung der Konfiguration ist. Dabei wird zwischen *Model (M)*, *ModelConsequence (MC)*, *User (U)* und *UserConsequence (UC)* unterschieden.

Mit M werden alle Features markiert, deren Zustand bereits durch die Beschaffenheit des Modells vorgegeben wird. Ein Beispiel hierfür wäre ein notwendiges Feature, da es aufgrund des Modells immer den Zustand ausgewählt erhält. Falls solch ein durch das Modell gegebener Konfigurationsschritt einen weiteren Konfigurationsschritt zur Konsequenz hat, wird das entsprechende Feature mit MC markiert. Dies

kann beispielsweise der Fall sein, wenn ein mit M markiertes Feature das Auswählen eines anderen Features impliziert. Die Markierung U kennzeichnet Features, deren Zustand durch eine Entscheidung des Nutzers hervorgerufen wurde. UC markiert, ähnlich wie MC, Features, deren Zustandsänderung die Konsequenz einer Nutzerentscheidung ist.

Der S2T2-Konfigurator verfügt außerdem über eine zusätzliche Funktionalität, die es erleichtern soll, automatisch getroffene Entscheidungen nachzuvollziehen. Im Kontextmenü befindet sich eine so genannte *Explain-Funktion*. Durch visuelle Hilfsmittel sollen dem Nutzer automatische Zustandsänderungen erklärt werden. Sobald die Explain-Funktion auf ein Feature angewendet wird, werden durch Highlighting die Features und Abhängigkeiten markiert, die zur Zustandsänderung des Features geführt haben.

6.2 GEARS

GEARS [13, 14] ist ein Framework für Softwareproduktlinienentwicklung. Für das Erstellen von Softwareproduktlinien nutzt GEARS drei grundlegende Konzepte: *Software Assets*, *Product Feature Profiles* und den *GEARS Configurator*.

Bei den Software Assets handelt es sich um wiederverwendbare Softwareartefakte, die in den verschiedenen Produkten der Softwareproduktlinie eingesetzt werden können. Dabei kann es sich beispielsweise um Code oder Testfälle handeln. In den Product Feature Profiles wird festgelegt, welche Features in einem bestimmten Produkt enthalten sein sollen. Der GEARS Configurator erstellt schließlich aus den Software Assets die jeweiligen Produkte, welche zuvor durch die Product Feature Profiles definiert wurden.

Der Feature-Modellierungs-Prozess in GEARS beinhaltet drei Komponenten. *Feature declarations* beschreiben die Vielfältigkeit der Produktlinie und repräsentieren somit einzelne Features. Sie besitzen außerdem einen Datentypen. Wird ein Feature als Teil eines Produkts ausgewählt, muss ihm ein Wert zugewiesen werden, der mit seinem Datentypen kompatibel ist. *Feature assertions* definieren die Abhängigkeiten und Beziehungen zwischen den einzelnen Features. Es wird dabei zwischen zwei Abhängigkeiten unterschieden. Zum Einen können sich zwei Features gegenseitig ausschließen, das heißt, dass sie nicht gleichzeitig Teil eines Produkts sein können. Zum Anderen kann die Auswahl eines Features dazu führen, dass ein weiteres Feature ausgewählt werden muss, um ein gültiges Produkt erstellen zu können. Diese Abhängigkeiten können nicht nur zwischen zwei Features bestehen, sondern auch zwischen Gruppen von Features. Die letzte Komponente sind *feature profiles*. Sie werden verwendet, um den feature declarations Werte zuzuweisen. Dadurch definieren sie, welche Features ein Teil des Produkts sein sollen.

Der Konfigurationsprozess in GEARS unterscheidet sich stark von FeatureIDE, da GEARS sich *Multistage Configuration* [6] zu Nutze macht. Bei einer hohen Anzahl von Produkten innerhalb einer Produktlinie kann es zu verschiedenen Problemen kommen. Beispielsweise wird es für die Softwareentwickler schwieriger die Produkte zu implementieren und für den Kunden wird es schwieriger ein passendes Produkt auszuwählen, was seinen Anforderungen entspricht. Um diese Probleme zu lösen,

können die Produkte einer Produktlinie in einem *Produkt-Stammbaum* organisiert werden. Ein solcher Baum besitzt mehrere Stufen, in denen ein Produkt immer weiter spezifiziert wird. Mit Hilfe des eigens dafür entwickelten *Multistage Configuration Trees* [15] werden Entscheidungen, die in den ersten Stufen des Baumes getroffen werden, bis zu den Blättern durchgereicht. So lassen sich durch eine Reihe von Entscheidungen Produkte immer weiter spezifieren. Dies kann ein fertiges Produkt oder aber eine spezifische Produktfamilie zur Folge haben. Im Falle einer Produktfamilie erzeugt GEARS automatisch alle möglichen Kombinationen von Features innerhalb der Familie. Aus diesen lässt sich ebenfalls ein finales Produkt generieren.

6.3 KConfig

Die KConfig-Sprache wird genutzt, um Konfigurationen des Linux-Kernels und zwischen ihnen bestehende Abhängigkeiten zu spezifizieren [8]. Das Konfigurations-Tool *xconfig* stellt die Konfigurationsoptionen als Liste dar, ähnlich wie S.P.L.O.T. oder der Standardkonfigurator von FeatureIDE. Xconfig verfügt außerdem über cross-tree constraints, mit denen weitere Abhängigkeiten definiert werden können.

Die verschiedenen Konfigurationsoptionen werden configs genannt. Es ist möglich die configs zu verschachteln. Jede config verfügt zudem über einen bestimmten Datentyp. Zulässige Datentypen sind hierbei *Boolean*, *Int*, *Hex*, *Tristate* und *String*. Configs mit dem Typen Boolean können dementsprechend zwei verschiedene Werte annehmen: true oder false. Tristate configs hingegen können die Werte n, y und m annehmen. n bedeutet, dass die ausgewählte Option kein Teil des resultierenden Systems ist. Im Gegensatz dazu sind Features, die mit y gekennzeichnet werden, Teil des Systems. Der dritte Zustand m bedeutet, dass das Feature als Modul kompiliert wird. Dadurch kann es zur Laufzeit geladen und wieder entfernt werden. Diese beiden Arten von configs werden auch *switch configs* genannt, da zwischen verschiedenen Zuständen hin- und hergeschalten werden kann.

Verschiedene configs können zusammen in *choice groups* gruppiert werden. Diese Gruppen müssen immer vom Typ Boolean oder Tristate sein. Erhält die Gruppe den Wert y, stehen die configs innerhalb der Gruppe in einer XOR-Beziehung. Wird der Gruppe hingegen der Wert m zugewiesen, besteht eine OR-Beziehung zwischen den einzelnen configs. Durch den Wert n werden die configs der Gruppe optional.

Eine weitere Möglichkeit der Gruppierung von configs sind *menus*. KConfig bietet die Möglichkeit menus zu verstecken oder sichtbar zu machen. Die Sichtbarkeit hängt dabei von cross-tree constraints, die dem menu zugewiesen werden, ab. Ist die cross-tree constraint nicht erfüllt, wird das menu in xconfig ausgegraut und lässt sich nicht bearbeiten. Solche menus werden auch *conditional menus* genannt.

Da es in der Forschung nur wenige realistische und zugleich ausreichend komplexe Variabilitätsmodelle gibt, wurde versucht mit Hilfe der KConfig-Sprache ein Modell zu erstellen, welches die Konfigurationsoptionen eines Linux-Kernels repräsentiert [21]. Mit Hilfe von choice groups, switch configs, Verschachtelung und cross-tree constraints ist es somit möglich aus Ausdrücken der KConfig-Sprache ein Feature-Modell zu generieren.

7. Zusammenfassung

Mit Hilfe von Softwareproduktlinienentwicklung können die Stärken von Massenprodukten und individualisierten Einzelprodukten vereint werden. Damit soll vor allem eine höhere Zeit- und Kosteneffizienz erreicht werden. Die einzelnen Funktionen der Produktlinie werden in Features aufgeteilt. Aus diesen Features kann dann ein fertiges Produkt konfiguriert werden. Da das Auswählen der Features per Hand sehr mühsam ist, wurden mit der Zeit verschiedene Tools entwickelt, die den Konfigurationsprozess erleichtern können. Viele dieser Konfigurationseditoren, zum Beispiel FeatureIDE oder fmp, bieten die Möglichkeit, den Konfigurationsprozess teilweise zu automatisieren. Die Features werden in den Konfigurationseditoren häufig als Liste dargestellt. Diese Art der Darstellung bringt allerdings Probleme mit sich. Oftmals lassen sich die Abhängigkeiten zwischen den Features nur schwierig oder gar nicht erkennen. Besonders bei großen Produktlinien kann der Nutzer beim Konfigurieren schnell den Überblick verlieren. Außerdem kann es passieren, dass der Nutzer automatisch durchgeführte Konfigurationsschritte aufgrund der Listendarstellung nicht nachvollziehen kann. Diese Probleme können den Konfigurationsprozess verlangsamen oder in ungewünschten Produktkonfigurationen resultieren. Um solche Probleme zu adressieren, wurde ihm Rahmen dieser Arbeit ein Konzept für einen Konfigurationseditor entwickelt, der die Features in Fom eines Baumdiagramms darstellt.

Für den neuen Konfigurationseditor, den so genannten Tree Configurator, wurden zunächst verschiedene Anforderungen definiert. Da es sich ausschließlich um funktionale Anforderungen handelt, beschreiben sie die einzelnen Funktionalitäten, die der Tree Configurator besitzen soll. Damit das Konzept umgesetzt werden konnte, musste außerdem entschieden werden, ob der Editor als eigenständige Anwendung entwickelt wird oder auf einem bestehenden Konfigurationstool aufbaut. Es wurde sich dazu entschieden den Tree Configurator in FeatureIDE zu integrieren. FeatureIDEs Konfigurationseditor ist leicht erweiterbar und bietet aufgrund der verschiedenen Editorseiten eine gute Möglichkeit, die verschiedenen Darstellungsformen gegenüberzustellen. Zur Darstellung der Features verwendet der Editor ein Baumdiagramm. Dieses Baumdiagramm orientiert sich optisch am Feature-Diagramm von

FeatureIDE. Dementsprechend sind auch die Abhängigkeiten zwischen den Features im Diagramm ersichtlich. Das Erstellen eines Produkts erfolgt direkt über das Diagramm. Durch Anwählen der Knoten des Baums kann die Konfiguration angepasst werden. Farbige Rahmen signalisieren dabei den Zustand des Features. Außerdem werden Features unter Berücksichtigung des Feature-Modells automatisch an- oder abgewählt. So werden beispielsweise die übrigen Features einer Alternativen-Beziehung automatisch abgewählt, sobald ein Feature angewählt wurde. Diese Anforderung konnte allerdings aufgrund des Zeitrahmens der Arbeit nicht mehr in den entwickelten Prototypen integriert werden.

Unter Verwendung des Prototyps wurde das Konzept des Tree Configurators in Form einer qualitativen Evaluation bewertet. Dazu wurden fünf Probanden interviewt, welche zuvor zwei Produkte einer Softwareproduktlinie mit Hilfe des Standardkonfigurators von FeatureIDE und des Tree Configurators erstellt hatten. Bei der Produktlinie handelte es sich um eine erweiterte Variante des Online-Pizzabestellservices, der in Kapitel 2 vorgestellt wurde. Sie war etwas komplexer als das ursprüngliche Beispiel, um den Konfigurationsprozess schwieriger zu gestalten.

Aus den Ergebnissen konnten verschiedene Vor- und Nachteile des Tree Configurators abgeleitet werden, welche zur Beantwortung der ersten Forschungsfrage notwendig sind. Alle Probanden empfanden den Tree Configurator als übersichtlicher. Das lag unter anderem daran, dass die Feature-Beziehungen im Baumdiagramm klar erkennbar waren. Ein Teil der Probanden war außerdem der Meinung, dass diese Darstellungsform den Konfigurationsprozess beschleunigen könnte. Unabhängig davon, dass das Baumdiagramm übersichtlicher war, als die Listendarstellung, empfanden viele Probanden die Oberfläche des Tree Configurators auch optisch ansprechender. Ein Kritikpunkt ist allerdings das Fehlen automatischer Konfigurationsschritte. Alle Probanden hätten sich gewünscht, dass die Eltern-Knoten von bereits ausgewählten Features automatisch angewählt werden.

Die zweite Forschungsfrage ließ sich nur bedingt beantworten. Hierfür sollte entschieden werden, ob der Tree Configurator eine erleichterte Konfiguration gegenüber einem Konfigurator mit Listendarstellung bietet. Dafür spricht, dass vier der fünf Probanden auch in Zukunft den Tree Configurator verwenden würden. Sie empfanden den Tree Configurator als übersichtlicher, was auch in einer besseren Bedienbarkeit resultiert. Dagegen spricht allerdings das Fehlen der automatischen Zustandsänderungen. Viele Probanden empfanden manuelle Auswählen der Eltern-Knoten als mühsam. Dementsprechend schränkt dieser Kritikpunkt die Bedienbarkeit ein. Da diese Funktionalität aber im Konzept vorgesehen war, könnte mit Hilfe eines verbesserten Prototyps eine eindeutigere Antwort auf diese Forschungsfrage gefunden werden.

Die Arbeit hat gezeigt, dass es möglich ist einen Konfigurationseditor mit einer alternativen Darstellungsform zu erstellen. Es wurde deutlich, dass sich die Anordnung der Features als Baum positiv auf die Übersichtlichkeit innerhalb des Editors auswirkt, da vor allem die Feature-Abhängigkeiten wichtig für den Konfigurationsprozess sind. Die Baumdarstellung kann sich daher vor allem für größere Produktlinien als nützlich erweisen. Verschiedene Faktoren weisen außerdem darauf hin, dass die Baumstruktur die Bedienung des Editors verbessern könnte. Allerdings sind die Ent-

wicklung eines besseren Prototyps und eine erneute Evaluation notwendig, um dies vollständig zu bestätigen.

8. Zukünftige Arbeiten

In diesem Kapitel wird erläutert, wie man eine eindeutigere Antwort auf Forschungsfrage R2 erhalten kann. Des Weiteren werden verschiedene Funktionen vorgestellt, um die der Tree Configurator erweitert werden könnte.

Klärung der Forschungsfrage R2

Da Forschungsfrage R2 nicht vollständig beantwortet werden konnte, wäre es interessant, eine weitere qualitative Evaluation durchzuführen. Das Problem beim Beantworten der Frage war, dass das Argument gegen eine leichtere Bedienung die fehlende Automatisierung des Konfigurationsprozesses war. Diese Funktion war allerdings im Konzept vorgesehen, konnte aber nicht rechtzeitig implementiert werden. Die Verwendung eines verbesserten Prototyps, der alle im Konzept beschriebenen Anforderungen erfüllt, würde die Stärken und Schwächen des Tree Configurators besser widerspiegeln. Eine erneute Evaluation würde dann zu besseren Ergebnissen führen, mit denen Frage R2 beantwortet werden kann.

Alternativ könnte auch eine quantitative Evaluation durchgeführt werden. Eine messbare Größe, mit deren Hilfe Aussagen über die Bedienbarkeit getroffen werden können, wäre beispielsweise die Dauer des Konfigurationsprozesses. Ähnlich wie bei der qualitativen Evaluation müssten die Probanden mit jedem der beiden zu vergleichenden Konfigurationseditoren ein Produkt erstellen. Dabei würde die Zeit gemessen werden, die die Probanden benötigen, um besagtes Produkt zu konfigurieren. Wenn ein Proband mit einem Editor signifikant länger braucht, als mit dem anderen, ist es wahrscheinlich, dass er bei diesem Editor Probleme mit der Bedienung hatte. Mit einer ausreichend großen Gruppe von Testpersonen wäre es somit möglich, diese Forschungsfrage zu beantworten.

Erweiterungsmöglichkeiten

Es gibt verschiedene Funktionen, um die der Tree Configurator erweitert werden könnte:

Konfigurationsschritthistorie

Eine Möglichkeit zur Verbesserung des Tree Configurators wäre das Hinzufügen einer Historie. Diese Historie enthält dabei alle manuell durchgeführten Konfigurationsschritte. In jedem Eintrag wäre zusätzlich vermerkt, welche automatischen Konfigurationsschritte durch diesen Schritt durchgeführt wurden. Die Historie wäre vor allem nützlich für besonders große Feature-Modelle, da man so besser verfolgen kann, welche Feature-Zustände sich automatisch geändert haben. Zusätzlich könnte die Historie über eine Undo- und Redo-Funktion verfügen, um zwischen einzelnen Konfigurationsssschritten hin- und herwechseln zu können. Außerdem könnte jeder Eintrag anzeigen lassen, wieviele Konfigurationsschritte minimal nötig sind, um eine vollständige Produktkonfiguration zu erhalten.

Layout-Einstellungen

Das Baumdiagramm des Tree Configurators verfügt bisher nur über eine geordnete Top-Down-Darstellung. Im Feature Diagram Editor ist es allerdings möglich, das Modell anders anzuordnen, sodass sich der Baum beispielsweise von links nach rechts auffächert. Diese Funktion könnte man auch für den Tree Configurator implementieren. Dadurch kann jeder Nutzer die Darstellung des Baumes nach eigenen Wünschen anpassen.

Entscheidungsvorschau

Eine weitere Möglichkeit den Tree Configurator zu verbessern, wäre eine Funktion, die den Nutzer darüber informiert, welche Konsequenzen ein manueller Konfigurationsschritt hat. Bisher werden zum Beispiel im Standardkonfigurator von FeatureIDE nach dem manuellen Auswählen eines Features weitere Features automatisch aktiviert oder deaktiviert. Dabei kann es passieren, dass dem Nutzer nicht klar ist, warum ein automatischer Konfigurationsschritt durchgeführt wurde. Mit Hilfe der neuen Funktion könnte dem Nutzer vor Auswahl eines Features angezeigt werden, welche Features automatisch ihren Zustand ändern werden. Das Ausführen der Funktion könnte beim Mouseover über ein Feature gestartet werden. Anschließend würden die Feature, die ihren Zustand ändern würden, farblich markiert werden. Ein rotes Highlighting würde die Features hervorheben, die automatisch abgewählt werden. Ein grünes Highlighting hingegen kennzeichnet Features, die automatisch angewählt werden.

Tree Configurator Outline

Der Tree Configurator besitzt momentan keinen Support für die Outline. Man könnte daher eine Outline hinzufügen, die sich an der Outline des Feature Diagram Editors orientiert. In ihr werden alle Features des Feature-Diagramms in einer Liste zusammengefasst. Über die Outline können außerdem alle Aktionen durchgeführt werden, die auch im Editor möglich sind. Es können zum Beispiel Kind-Knoten über die Outline ein- oder ausgeklappt werden. Die Outline des Tree Configurators würde ähnlich aussehen. Auch in ihr wären alle Features des Diagramms aufgeführt. Zusätzlich könnten in der Outline die Zustände der einzelnen Features angezeigt werden,

zum Beispiel durch verschiedene Schriftfarben oder spezielle Icons. Es wäre außerdem möglich, über die Outline die Zustände zu verändern und dadurch das Produkt zu konfigurieren. Die Listendarstellung der Outline ähnelt dabei dem Standardkonfigurator im Konfigurationseditor von FeatureIDE. Man könnte daher versuchen, die Funktionen des Standardkonfigurators in die Outline des Tree Configurators zu übernehmen. So könnte man den Tree Configurator und den Standardkonfigurator kombinieren.

Literaturverzeichnis

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented Software Product Lines: Concepts and Implementation.* Springer, 2013. (zitiert auf Seite 1, 5, 6 und 7)

[2] Goetz Botterweck and Andreas Pleuss. S2T2-Configurator: Interactive Support for Configuration of Large Feature Models. *European Conference on Modelling Foundations and Applications*, 2012. (zitiert auf Seite 39)

[3] Krzysztof Czarnecki, Michal Antkiewicz, CHP Kim, S Lau, and K Pietroszek. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange*, 2004. (zitiert auf Seite 17)

[4] Krzysztof Czarnecki, Ulrich W Eisenecker, G Goos, J Hartmanis, and J van Leeuwen. Generative Programming. 2000. (zitiert auf Seite 1 und 6)

[5] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. Workshop on Variability Modeling of Software-intensive Systems*, 2012. (zitiert auf Seite 1)

[6] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 2005. (zitiert auf Seite 7 und 40)

[7] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product Derivation in Software Product Families: A Case Study. *Journal of Systems and Software*, 2005. (zitiert auf Seite 10)

[8] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the Kconfig Semantics and its Analysis Tools. In *ACM SIGPLAN Notices*, 2015. (zitiert auf Seite 41)

[9] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M Jensen, Henrik R Andersen, Jesper Møller, and Henrik Hulgaard. Fast Backtrack-free Product Configuration Using a Precompiled Solution Space Representation. *PETO Conference*, (1), 2004. (zitiert auf Seite 10)

[10] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented Domain Analysis: Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990. (zitiert auf Seite 2 und 7)

[11] Kyo C Kang and Hyesun Lee. Variability Modeling. In *Systems and Software Variability Management*. Springer, 2013. (zitiert auf Seite 7 und 8)

[12] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. An Efficient Algorithm for Feature-Model Slicing. Technical report, Technical Report FIN-001-2016, University of Magdeburg, Germany, 2016. (zitiert auf Seite 8)

[13] Charles Krueger and Paul Clements. Systems and Software Product Line Engineering with BigLever Software Gears. In *Proc. Software Product Line Conference*, 2013. (zitiert auf Seite 40)

[14] Charles W Krueger. BigLever Software Gears and the 3-tiered SPL Methodology. In *OOPSLA Companion*, 2007. (zitiert auf Seite 40)

[15] Charles W Krueger. Multistage Configuration Trees for Managing Product Family Trees. In *Proc. Software Product Line Conference*, 2013. (zitiert auf Seite 41)

[16] Marcilio Mendonca, Moises Branco, and Donald Cowan. SPLOT: Software Product Lines Online Tools. In *Proc. Conf. on Object-oriented Programming, Systems, Languages and Applications*, 2009. (zitiert auf Seite 1)

[17] Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *International Conference on Software Reuse*, 2016. (zitiert auf Seite 2, 9 und 10)

[18] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (zitiert auf Seite 1, 5 und 6)

[19] Klaus Schmid and Martin Verlage. The Economic Impact of Product Line Adoption and Evolution. *IEEE software*, 2002. (zitiert auf Seite 1)

[20] Christoph Seidl, Tim Winkelmann, and Ina Schaefer. A Software Product Line of Feature Modeling Notations and Cross-Tree Constraint Languages. In *Modellierung*, 2016. (zitiert auf Seite 8)

[21] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The Variability Model of The Linux Kernel. *VaMoS*, 2010. (zitiert auf Seite 41)

[22] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Science of Computer Programming*, 2014. (zitiert auf Seite 1)

[23] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proc. Conf. on Software Engineering*, 2009. (zitiert auf Seite 9)

[24] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer Science & Business Media, 2007. (zitiert auf Seite 5)

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

Magdeburg, den 03.04.2017

Daniel Püsche