Entwicklung einer Software-Produktlinie mit Code-Generierung der Varianten: Fokus auf Entwurf - Entscheidungen

Masterarbeit

zur Erlangung des Grades Master of Science
des Fachbereichs Informatik und Medien der
Technischen Hochschule Brandenburg

vorgelegt von:
Mistra Forest Kuipou Tchiendja

Betreuer: Prof. Dr. rer. nat. Gabriele Schmidt
Zweitgutachter: Prof. Dr.-Ing. habil. Michael Syrjakow

Brandenburg an der Havel, 17. April 2023

Danksagung

An erster Stelle möchte ich Gott danken, dass ich in ihm jeden Tag die nötige Kraft und Motivation gefunden habe.
Anschließend möchte ich Frau Prof. Schmidt danken, die mich während der Arbeit mit hilfreichen Beiträgen, Feedbacks und konstruktiven Kritiken unterstützt hat. Ich bedanke mich weiterhin bei Herrn Prof. Syrjakow für die Betreuung während der gesamten Arbeit.
Mein Dank geht auch an Frau Renate und Herrn Raphael Spies für die Korrekturlesung.
Abschließend möchte ich meiner Verlobten und meinen Geschwistern für ihre tägliche und kontinuierliche Unterstützung während meines gesamten Studiums danken.
Ich widme diese Arbeit meiner Mutter.

Kurzfassung

Unter einer Software-Produktlinie versteht man eine Familie verwandter Softwareanwendungen. Verschiedene wissenschaftliche Publikationen ((Lopez-Herrejon & Batory, 2002), (Dubinsky et al., 2013)) beschäftigen sich bereits mit dem erfolgreichen Einsatz einer Softwareproduktlinie Entwicklung (oder Software Product Line Engineering, SPLE) in der Industrie. Diese Arbeit liefert nun einen weiteren Beitrag zur Entwicklung einer Software-Produktlinie mit dem besonderen Fokus auf Entwurf–Entscheidungen.

Anhand eines kleinen Beispiels in der Lehre wird die vorliegende Arbeit erarbeitet. Das Ziel der Arbeit ist zweistufig. Auf der ersten Ebene wird ein Konzept erarbeitet, um eine Produktlinie mit konfigurierbaren Varianten zu entwickeln. Auf der zweiten Ebene wird einen Code-Generator implementiert, der aufbauend auf dem erarbeiteten Konzept verschiedene Varianten eines Produkts erzeugt.

Im Vergleich zu der herkömmlichen Entwicklung von SPL wird der Fokus auf die zuvor in der Beispielanwendung getroffenen Entwurf-Entscheidungen gelegt werden, die der Konzeption und Umsetzung des Prototyps zugrunde liegt. Diese Entwurfsentscheidungen sollten im generierten Code wieder erkennbar sein.

Auf der Grundlage der mitgelieferten Beispielanwendung werden über eine systematische Analyse die relevanten Features extrahiert. Dabei wird die Methode der Bottom-up-Entwicklung verwendet. Mit dem Ergebnis der Analyse wird die Produktlinie-Anwendung konzipiert und mit bewährten Werkzeugen (FeatureIDE und Eclipse) implementiert.

In dieser Arbeit wird anhand dieser Beispiele dargestellt, welche Schritte für eine SPLE nötig sind und, wie Entwurf-Entscheidungen in der Beispielanwendung über die Varianten hinweg beibehalten werden können.

Schlüsselwörter

SPLE, MDSD, FOSD, AOP

Abstract

A software product line is a family of related software applications. Various scientific publications ((Lopez-Herrejon & Batory, 2002), (Dubinsky et al., 2013)) already deal with the successful use of software product line development (or SPLE) in industry.
This thesis now provides a further contribution to the development of a software product line with a special focus on design decisions.

The present work is developed based on a small example in teaching. The aim of the work is two-tiered. On the first level, a concept is worked out to develop a product line with configurable variants. On the second level, a code generator is implemented that generates different variants of a product based on the developed concept.

Compared to the traditional development of SPL, the focus will be on the design decisions previously made in the sample application that underlie the design and implementation of the prototype. These design decisions should be recognisable in the generated code.

Using the sample application provided as a basis, the relevant features are extracted via a systematic analysis. The bottom-up development method is used. With the result of the analysis, the product line application is designed and implemented with proven tools (FeatureIDE and Eclipse).

This thesis uses these examples to illustrate the steps required for an SPLE and how design decisions in the example application can be maintained across variants.

Keywords
SPLE, MDSD, FOSD, AOP

Inhaltsverzeichnis

1 Einleitung

An der technischen Hochschule Brandenburg werden Bestellsysteme in der Lehre benötigt. Lehrende wollen mit jedem neuen Semester, veränderte Varianten eines bestehenden Bestellsystems anbieten. Das Ziel der Lehrenden ist einerseits, verschiedene Systeme aus unterschiedlichen Domänen von einem einzigen Grundsystem abzuleiten. Andererseits geht es um das Vermitteln von Konzepten, die sich in Code-Strukturen widerspiegeln und weniger um die Anwendung als solche oder den Domäneninhalt. In diesem Kontext hat Grigarzik (2020) in seiner Abschlussarbeit ein bestehendes Bestellsystem nachimplementiert. Weitere Nachimplementierungen werden benötigt, um möglichst viele Variante zu bilden. Die manuelle Nachimplementierung ist zeitaufwendig und fehleranfällig. Eine mögliche Lösung besteht darin, eine Software-Produktlinie (SPL) zu entwickeln und zu verwalten.

Nach Siegmund et al. (2009) und McGregor, Monteith & Zhang (2011) wird unter SPL eine Gruppe von Produkten verstanden, die eine gemeinsame wiederverwendbare Grundstruktur aufweisen, welche nur einmal für alle Produktvarianten entwickeln wird. Darüber hinaus werden Unterschiede unter den Varianten innerhalb einer SPL dadurch gekennzeichnet, dass Features separat entwickelt und konfiguriert werden, um spezifischen Anforderungen einer bestimmten Domäne gerecht zu werden (vgl. Stahl et al. 2012, S. 35 und Lopez-Herrejon & Batory, 2002).

Die Herausforderung besteht erstens darin, ein Modell für die SPL aufzustellen. Vom Modell abgeleitet werden Varianten über eine Konfiguration ausgewählt. Zweitens werden Codes der Auswahl entsprechend generiert. Die generierten Codes müssen dann die Entwurfsvorgaben der zugrundeliegenden Anforderung erfüllen.

1.1 Aufgabenstellung

Angesichts dessen, was die bisher zitierten Autoren über SPL geschrieben haben, sind folgenden Fragen noch offen: Wie lässt sich eine SPL modellieren und entwickeln? Welche grundlegenden Prinzipien müssen dabei beachtet werden, um die Anforderungen zu erfüllen? Wie lassen sich Produktvarianten so generieren, dass getroffene Entscheidungen über das Design der zu entwickelnden Software sich in Codestrukturen widerspiegeln? Diese Fragen werden untersucht und zielorientiert beantwortet.

1.2 Ziel und Ergebnis

Das Ziel besteht auf zweierlei Bausteinen: Erstens, die Konzeption und die Entwicklung einer Produktlinie mit konfigurierbaren Varianten und zweitens die Implementierung eines Generators zur Generierung des Sourcecodes je nach Auswahl einer Variante. Dabei wird der Fokus besonders auf die Entwurfsentscheidungen gelegt. Diese Konzepte sollten im generierten Code wieder erkennbar sein. Das Endergebnis ist die Entstehung eines funktionierenden Prototyps.

1.3 Abgrenzung des Themas

Eine vollständige SPL-Lösung kann nicht innerhalb des Rahmens dieser Arbeit konzipiert und entwickelt werden. Deshalb wird nur ein Prototyp Anhang eines kleinen Beispiels implementiert. Auch die Auswahl der Werkzeuge, die für die Erfüllung des Zieles benötigt werden, wird nicht behandelt. Stattdessen werden ausgewählten Lösungen von anderen Autoren zur Hilfe genommen.

1.4 Aufbau der Arbeit

Beginnend im Kapitel 2 werden die benötigten Grundlagen eingeführt. Die Grundlage wird in drei Hauptteile unterteilt.
Im ersten Teil wird das Konzept der modellgetriebenen Software-Entwicklung (MDSD) betrachtet. Dabei werden das Vorgehen der (Meta-)Modellierung durch

die modellgetriebene Software-Architektur von der Erstellung von Modellen sowie deren Transformationen ineinander bis hin zur (Code-)Generierung beleuchtet (Unterkapitel 2.1 bis Unterkapitel 2.6). Am Ende des ersten Teils im Abschnitt 2.6.7 wird der Bezug auf die Produktlinien-Entwicklung hergestellt.

Im zweiten Teil wird der Produktlinien-Entwicklungsprozess (PLE) mit seinen Phasen vorgestellt (Unterkapitel 2.7). Nach einer kleinen Einleitung werden die erforderlichen Schritte im Domain-Engineering (Abschnitt 2.7.3), Applikation-Engineering (Abschnitt 2.7.4) sowie Problemraum und Lösungsraum (Abschnitt 2.7.6), das Feature-Konzept (Abschnitt 2.7.7) und die Feature-orientierte Software-Entwicklung (Abschnitt 2.7.8) für die Umsetzung von Produktlinien erläutert.
Im dritten Teil der Grundlagen werden einige bewährten Programmierpraktiken im Unterkapitel 2.8 beschrieben, die die Konzeption und Implementierung des Prototyps unterstützen.

Nachdem die Grundlagen zu MDSD und PLE beschrieben wurden, wird zunächst die Pizza-Pronto-Anwendung im nachfolgenden Kapitel 3 (Analyse) vorgestellt, welche als Grundlage für die Umsetzung der Produktlinien-Anwendung dient. Danach werden diverse Anforderungen an die Produktlinien-Anwendung gestellt. In diesem Kapitel werden auch die Technologie und die Werkzeugunterstützung zur Erfüllung der Anforderungen beschrieben.

Anknüpfend an die Analyse wird die Konzeption der zu entwickelten Produktlinien-Anwendung im Kapitel 4 vorgenommen. Dabei wird die Vorgehensweise zur Erstellung eines Feature-Modell mit Hilfe den im vorangegangenen Kapitel ausgewählten Werkzeugen und Techniken beschrieben. Anschließend werden die verschiedenen Komponenten der Anwendung identifiziert und unter Berücksichtigung der Anforderungen entworfen. Hierbei werden das Paketdiagramm und das Komponentendiagramm zur Hilfe genommen.

Das Kapitel 5 „Umsetzung" wird sich um die Implementierung der zuvor konzipierten Komponenten kümmern. Anhand einer konkreten Anforderung aus dem Kapitel „Analyse" wird die Implementierung der einzelnen Komponenten durchgenommen.

Nach der Implementierung wird im Kapitel 6 die Arbeit evaluiert. Die Anwendung wird getestet und das Testergebnis wird interpretiert. Hinzu kommt eine Gegenüberstellung der Anforderungen mit dem Ergebnis. Darauffolgend werden mögliche Abweichungen enumeriert.

Abschließend wird die Arbeit im Kapitel 7 zusammengefasst und das Ergebnis vorgestellt. Daraufhin wird eine kritische Betrachtung folgen und einen Ausblick ausgegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu den Konzepten und Methoden elaboriert, welche für den Weg zum vorgestellten Ziel wichtig sind. Es handelt sich um die modellgetriebene Software-Entwicklung und die Softwareproduktlinien-Entwicklung mit den dazu gehörigen Konzepten. Das nähere Betrachten beider Verfahren wird helfen, die verschiedenen Meilensteines im Verlauf dieser Arbeit aufzustellen, um den Prototyp zu konzipieren. Für die Umsetzung werden zusätzliche bewährte Programmierpraktiken vorgestellt.

2.1 Modellgetriebene Software-Entwicklung

Modelle ermöglichen die Funktion, die Struktur und das Verhalten eines Systems in einem bestimmten Kontext und aus einem bestimmten Bezugspunkt formal zu beschreiben oder zu spezifizieren (Cephas Consulting Corp, 2006). Modelle können auch als Abstraktion bei der Entwicklung von (Software-) Systemen dienen, sodass aus einem Modell unterschiedliche Ausprägungen abgeleitet werden können.
Die anfängliche Nutzung von Modellen in der Softwareentwicklung war auf die Dokumentation von Systemen beschränkt. Laut Stahl et al. (2012) bestand lediglich eine gedankliche Verbindung zwischen der Implementierung und dem Modell. Dieser Ansatz ist sehr anfällig bezüglich der Modifizierbarkeit. Stahl et al. (2012) bezeichnen diese Art der Nutzung von Modellen als „modellbasiert".

Bei modellgetriebener Software-Entwicklung (MDSD, Model Driven Software Development) gibt es einen stärkeren Zusammenhang zwischen Modell und Code. MDSD ist ein Sammelbegriff für Verfahren, welche die vollständige oder teilweise Generierung von ausführbarerem Code aus formalen Modellen sicherstellen (vgl. Stahl et al. 2012, S. 11). Der Begriff „formal" heißt an dieser Stelle nicht, dass alles beschrieben wird. Jedoch werden formale Modelle benutzt, um bestimmte Aspekte mithilfe eines Regelwerks vollständig zu beschreiben (Stahl et al. 2012, S. 11).

Dieses Regelwerk ragt über eine einfache Darstellung aus Diagrammen und Texts hinaus. Es basiert vielmehr auf einer Sprache, welche eine klar definierte

semantische Bedeutung aufweist, die mit jedem ihrer Konstrukte verbunden ist (Cephas Consulting Corp, 2006). Mit formalen Modellen wird z.B. die Architektur einer Software beschrieben, woraus Codes generiert werden können. Auf dieser Weise lässt sich ein einheitliches Design über ein Softwaresystem hinweg für eine Zielplattform erzeugen.

Zusammengefasst bietet der Einsatz von MDSD folgende Vorteile:

- Die Erhöhung der Entwicklungsgeschwindigkeit
- Die Wiederverwendbarkeit: Insbesondere bei der Entwicklung von Softwareproduktlinien sehr nützlich
- Die Interoperabilität und die Plattformunabhängigkeit
- Die Reduzierung der Komplexität durch Abstraktionen: Änderungen werden bei Bedarf nur auf einer Stelle (z.B. im Modell) gemacht und nicht mehr im Quellcode. Diese Änderungen werden dann in Instanzen des Modells übernommen und bis zum Programmcode durchpropagiert. Dadurch wird die Modifizierbarkeit erhöht (vgl. Stahl et al. 2012).

Ein formales Modell aufzustellen ist nützlich und wichtig. Allerdings betonen (Petrasch & Meimberg, 2006) in ihrer Ausführung, dass die Modellierung gepaart mit der Formalisierung nicht ausreichend ist, um die Lücke zwischen Modell und Implementierung zu schließen, denn auch Zielpattformen müssen formal vorliegen. An dieser Stelle knüpft die Standardisierungswerkzeuge von OMG (Object Management Group) wie die modellgetriebene Architektur (MDA, Model Driven Architecture) an, um z.B. Plattformen zu standardisieren.

2.2 Object Management Group und Modellgetriebene Architektur

Die OMG ist ein Konsortium, welches international anerkannte Standards (z.B. UML, XML, CORBA, XMI, etc…) für verschiedene Aspekte der Softwareentwicklung bereitstellt (Colomb, et al., 2006). Um die modellgetriebene Entwicklung umzusetzen, hat die OMG die MDA entwickelt.

Mit MDA werden Weichen für folgende Zwecke aufgestellt: Erstens für die Spezifikation eines Systems unabhängig von der Plattform, die es unterstützt;

Zweitens für die Spezifikation von Plattformen; Drittens für die Auswahl einer bestimmten Plattform für das zu entwickelte System und Viertens für die Umwandlung der Systemspezifikation in eine Spezifikation für eine bestimmte Plattform (MDA Guide, 2003).

Bei der Modellierung in Sinne der MDA werden Modelle höherer Abstraktionsebenen in Modelle niedrigerer Abstraktionsebenen bis hin zu ausführbaren Codes unter Nutzung geeigneter Abbildungsregeln überführt. Die Abbildung 1 visualisiert die verschiedenen Modelle und die Transformationen ineinander (vgl. Becker, et al., 2007).

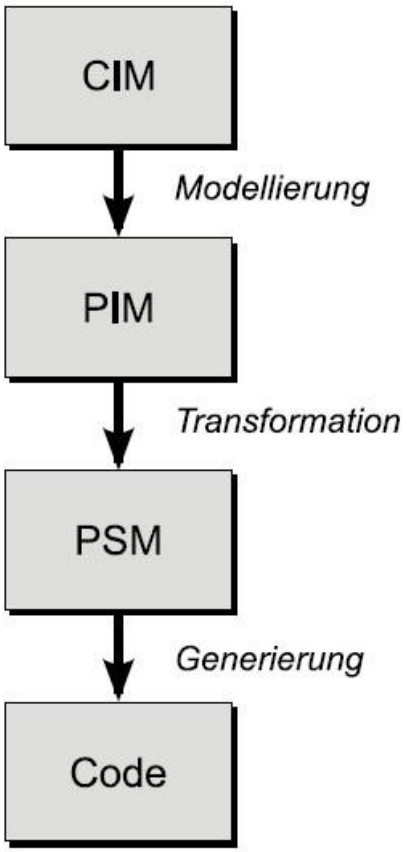

Abbildung 1: Einfache Darstellung von MDA-Modellen. (Auszug aus Becker, et al., 2007, S. 2)

CIM

Das computerunabhängige Modell (CIM, Computer Independent Model) beschreibt die Geschäftsprozesse des zu entwickelten Systems aus der Sicht der Fachseite, ohne dabei die internen Details zu beachten (MDA Guide, 2003). Darüber hinaus entspricht CIM dem Endergebnis der Anforderungsanalyse und dient einerseits hauptsächlich dazu, das System besser zu verstehen.

Andererseits kann das CIM als Quelle eines gemeinsamen Fachwortschatzes zur Wiederverwendung in anderen Modellen genutzt werden (MDA Guide, 2003).

PIM

Im plattformunabhängigen Modell (PIM, Platform Independent Model) werden die grundlegenden Funktionalitäten, das Verhalten, und die Struktur des Systems beschrieben, ohne dass die Details oder die Eigenschaften der endgültigen Zielplattform gekannt werden (Kuznetsov, 2007). Das heißt, das System wird im PIM ausschließlich aus der fachlichen Perspektive betrachtet und modelliert (Kempa & Mann, 2005).

PSM

Im plattformspezifischen Modell (PSM, Platform Specific Model) wird das PIM konkret angewendet und die Details zur Realisierung der Anforderungen im PIM für ein bestimmtes Plattform-Modell werden angegeben. Aus PSM kann den Quelltext bereits generiert werden und dient zugleich als Abstraktion für den auszuführenden Code (Kempa & Mann, 2005).

Ein Plattform-Modell definiert eine Sammlung von technischen Modullen, aus denen eine Plattform besteht und die von der Plattform bereitgestellten Dienste sowie Konzepte zur Nutzung der Plattform durch eine Anwendung im PSM-Kontext (MDA Guide, 2003). Beispiele für ein Plattform-Modell können die Beschreibung von J2EE oder .NET sein. Das PIM ist in diesen Fällen die Beschreibung eines Anwendungssystems, ohne die Details von J2EE bzw. von .NET zu kennen.
Im dazugehörigen PSM wird die Spezifikationen von J2EE bzw. von .NET eingeführt. Einschließend wird je nach Auswahl einer konkreten Plattform-Modell Java-Codes oder C#-Codes generiert.

Code

Auch Codemodell oder plattformspezifische Implementierung (PSI, Platform Specific Implementation) genannt ist der (generierte) Code. Es ist das Produkt des letzten Transformationsschritts beim MDA-Transformationsprozess, welcher dem Meta Object Facility zugrunde liegt.

2.3 Meta Object Facility und Hierarchie Ebene

Nach (Tang, 2018) ist die Meta Object Facility (MOF) ein weiterer Standard der OMG-Gruppe zur Metamodellierung und Metadaten-Repository. Es handelt sich um ein erweiterbares, modellgesteuertes Integrationsframework zur Definition, Bearbeitung und Integration von Metadaten und Daten in einer plattformunabhängigen Weise.

Das heißt, Modelle aus einer Anwendung können exportiert und in eine andere importiert werden. Dafür werden sie über ein Netzwerk transportiert. Dabei werden sie in einem Repository gespeichert und stehen zum Abruf bereit. In verschiedenen Formaten können die Modelle der Zielumgebung entsprechend umgewandelt werden (OMG/MOF, 2023).

Diese Umwandlung dient als Grundlage für die Generierung von Anwendungscodes. Ein geeignetes und verbreitetes Format für die Umwandlung ist das XMI (XML metadata interchange). Es handelt sich um ein XML-basierten Standardformat der OMG für die Übertragung und Speicherung von Modellen (vgl. OMG/MOF, 2023).

Bei der Transformation können die Modelle auf derselben oder auf unterschiedlichen Modellebenen sein. Mit der Modellebene oder Metaebene wird die hierarchische Sichtweise der verschiedenen Abstraktionen zwischen Modellen dargestellt. Die Abstraktionsebenen werden, wie folgt, unterteilt (Siehe Abbildung 2 zur Veranschaulichung).

- **M3: Die Metametamodell-Ebene**

Sie repräsentiert die MOF. In der MOF werden alle Metamodelle der OMG entwickelt. Die MOF wird als Urmodell aller Metamodell gesehen. Jedoch wird die MOF selbst mit Hilfe einer Untermenge der UML (UML-Metamodelle befindet sich auf der M2-Ebene) zum Ausdruck gebracht. Daher kann MOF in sich selbst zum Ausdruck gebracht werden (Colomb, et al., 2006, S. 22).

Ein Metametamodell ist in der Lage die gültigen Bestandteile eines Metamodells zu beschreiben. Eine wichtige Komponente auf dieser Ebene ist die Definierung der Modellierungssprachen (Metasprache). Jede Metasprache definiert Konstrukte sowie die Bedeutung dieser Konstrukte zur Erstellung von Modellen

für eine Domäne (auch als DSL, Domain Specific Language bekannt, vgl. Stahl et al. 2012, S. 216). Einige Beispiele sind die erweiterte Backus-Naur-Form (EBNF) oder Das ER[1]-Diagramm (ERD). Das ERD definiert Klassen mit ihren Attributen sowie die Beziehung zwischen Klassen, um ER[1]-Modell (ERM) zu erstellen. Analog wird die EBNF für Textbasierte Sprachen verwendet (Aßman, Zschaler, & Wagner, 2006).

- **M2: Die Metamodelle-Ebene.**

Einige Beispiele sind UML und OCL (Object Constraint Language). Der Begriff Metamodell kann sowohl „relativ" als auch „absolut" verwendet werden.
Die MOF kann z.B. als Metamodell der UML aufgefasst werden, denn die MOF ist das Urmodell, das sich auf M3-Ebene befindet, während die UML auf M2 steht. Die UML-Metamodelle sind folglich Instanzen von MOF.
Ein Metamodell kann auch „absolut" verwendet werden: Wird aus einem PIM ein PSM gewonnen, so lässt sich aus dem entstandenen PSM, den Code generieren. In dieser Konstellation kann das PIM als Metamodell „absolut" betrachtet werden (Vgl. Stahl et al., 2012).

- **M1: Die Modell-Ebene.**

Hier befinden sich konkrete Modelle basierend auf UML-Metamodelle. Sie umfasst Elemente wie z.B. das ERM, UML-Diagramme, Datenbanktabellen-Deklarationen, XML-Schemata oder Klassen-, Modul- und Typdeklarationen (Poernomo, 2006).

- **M0: Die Instanz-Ebene.**

Sie repräsentiert die Real-Welt, z.B. eine konkrete Instanz eines ER-Modells, instanziierte Datenbanktabellen, Algorithmen oder einen generierten Code.

2.4 Modellierung mit MOF

Wie in der Abbildung 2 aufgezeigt, werden Modellierungen mit Hilfe der vier Ebenen-Hierarchie umgesetzt. Die MOF-Sprache wird durch eine Menge von zusammenhängenden Modellelementen auf der Ebene M3 festgelegt. Anschließend wird ein Metamodell durch eine Menge von MOF-Objekten auf der M2-Ebene erzeugt, welches die MOF-Modellbausteine instanziieren. Die Ebenen sind dann durch eine objektorientierte Klasse/Objekt-Instanziierungsbeziehung

[1] Entity-Relationship

miteinander verbunden. Klassenelemente der Ebene M_{E+1} liefern Typbeschreibungen von Objekten der Ebene M_E. M_E-Objekte instanziieren M_{E+1}-Klassen (Poernomo, 2006).

Diese MOF-Objekt-Darstellung eines Metamodells kann auch als M2-Metamodell umgeschrieben werden, das Typbeschreibungen über eine Menge von Modellelementen bereitstellt. Ein Modell der M1-Ebene wird als eine Menge von Elementen verstanden, welche die Klassifikatoren eines Metamodells der M2-Ebene instanziieren. Schließlich können diese Elemente der Ebene M1 auch umgeschrieben werden, um Modell-Klassifikatoren der Ebene M1 zu bilden, welche die erforderliche Struktur einer Modellinstanziierung der Ebene M0 spezifizieren (Poernomo, 2006).

Bei der Modellierung können UML-Metaelemente (Klassen, Klassenassoziationen oder Klassenobjekte) in jeder Ebene der Hierarchie verwendet werden. So können z.B. Klassen unterschiedliche Aufgaben erfüllen. Auf der Ebene M3 werden Klassen zur Typisierung von Modellierungssprachen verwendet, während sie auf der Ebene M2 innerhalb von Modellierungssprachen zur Typisierung von Modellen genutzt werden.
Type können sowohl als Klassifikationen als auch als Datensätze behandeln werden. Der Grund dafür ist die Möglichkeit, Informationen in der MOF-Hierarchie auf den Ebenen M1 und M2 auf zwei verschiedenen Arte zu kodieren, nämlich als Objektinstanzen oder als Modellelemente (Poernomo, 2006).

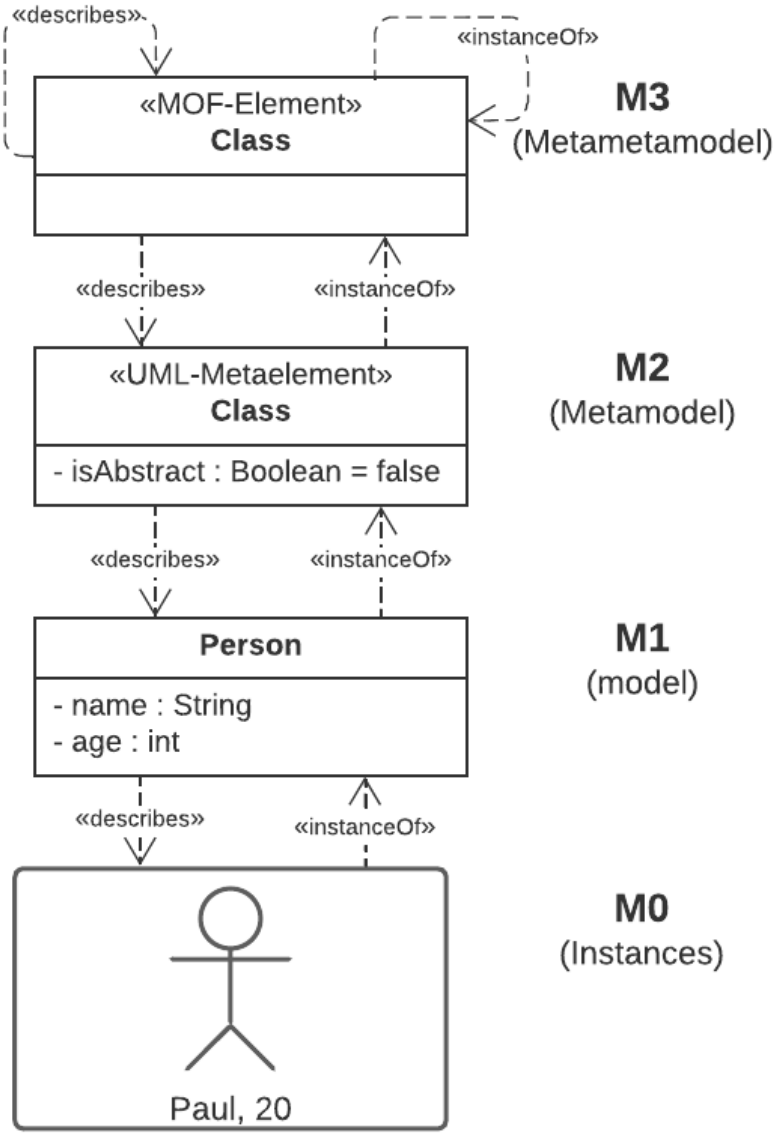

Abbildung 2: Illustration der MOF. in Anlehnung an Stahl et al. (2012, S. 62) und an Petrasch & Meimberg (2006, S. 49)

2.5 UML als Modellierungssprache

Die UML (Unified Modeling Language) ist eine standardisierte Modellierungssprache zur Dokumentation, Spezifikation und Visualisierung von Softwaresystemen. Die OMG hat die UML fortlaufend überarbeitet und erweitert. Die Spezifikation von UML ab Version 2.0 sieht zwei Teile vor (Petrasch & Meimberg, 2006): Die „Superstructure" und die UML „Infrastructure".

Die Infrastructure dient in der UML v2.4.1 als Basis zur Erstellung aller Metamodelle. Im Mittelpunkt der Infrastructure steht die Infrastructure Library (Ebene M3), denn sie wird sowohl vom MOF auf der Ebene M3 als auch von UML auf der Ebene M2 wiederverwendet.
Die UML Superstructure definiert Module zur Modellierung in drei Segmente:

- Der erste Block ist die Strukturmodellierung, um z.B. Komponenten-, Objekt- oder Klassendiagramme darzustellen.

- Der zwei Block besteht aus Paketen, welche zur Verhaltensmodellierung dienen. Mit diesen Paketen lassen sich z.B. Aktivitäts-, Zustands- oder Sequenzdiagramme erstellen.

- Der dritte und letzte Bereich ist das Ergänzungsmodul. Darin werden andere Konstrukte beschreiben, die bei der Modellierung notwendig sind und sich außerhalb des Definitionsbereichs der ersten beiden Module befinden. Profile zur Erweiterung der UML und weiterer Mechanismen werden z.B. in diesem Modul definiert. Es sei auf die Ausführungen in (Petrasch & Meimberg, 2006) und in (OMG UML, 2017) verwiesen, um tiefere Einblicke in UML zu bekommen. Darüber hinaus wird in der UML Superstructure die konkrete und die abstrakte Syntax festgelegt.

In der neuen Version von UML 2.5.1 vom Dezember 2017 wurde die UML Infrastructure abgelöst. Stattdessen verfügen die UML-Metamodelle auf der Ebene M2 über eine abstrakte und eine konkrete Syntax, und die MOF der Ebene M3 verwendet die abstrakte Syntax von UML (Vgl. OMG UML, 2017).

2.5.1 Konkrete Syntax und abstrakte Syntax

Die Superstructure (die neue UML) stellt grafische Symbole bereit, um die visuelle Modellierung zu gestalten (Konkrete Syntax).
Zu den visuellen Elementen werden Kombinationsregeln festgelegt und letztendlich die Semantik. Die Semantik wird darüber hinaus mittels OCL angereicht, um die Ausdruckkraft bei der Modellierung zu verfeinert (Abstrakte Syntax).

2.5.2 Object Constraint Language

Die Object Constraint Language (OCL) ist eine deklarative Sprache, die zusätzlich zur UML definiert ist. Mit der OCL können Konstrukte beschrieben werden, um Restriktionen einem System hinzuzufügen. Diese Beschränkungen liefern die Grundlage dafür, ein Modell zu validieren (Petrasch & Meimberg, 2006).
Sie nehmen deshalb einen wichtigen Platz bei einer Transformation an, denn Generatoren oder Interpreter sollten sich auf die Konformität des zu transformierenden Modells verlassen können.

OCL kann nicht wie eine reine Programmiersprache eingesetzt werden. Sie ist typisiert und kann nur für die Spezifikation von Sprache verwendet werden. Für einen tieferen Einstieg um OCL sei auf die Spezifikation der OMG für OCL in (OMG/OCL, 2014) verwiesen.

2.6 Transformationen in der modellgetriebenen Architektur

Innerhalb dieses Unterkapitel werden zwei grundlegenden Prinzipien über die Transformation von Modellen in der MDA beschrieben: Die Modell-zu-Modell-Transformation und die Modell-zu-Code-Transformation. In dem MDA-Guide (MDA Guide, 2003) werden diese ausführlich dokumentiert. Hier wird nur einen Überblick verschafft.

2.6.1 Modell-zu-Modell-Transformation

Transformation durch Markierungen oder/und Pattern

Um ein Modell A in ein Modell B (z.B. von PIM zu PSM oder von PSM zu Code) zu überführen, wird eine Abbildung mit entsprechenden Regeln zwischen den Modellen spezifiziert. Diese Abbildung, auch Mapping genannt, gewährleistet eine kontinuierliche Transformation auf eine neue auszuwählende Plattform.

Die Mappings beinhalten Marken (die sogenannte „Marks"), welche für die Markierung des Modells (PIM) verwendet werden, um die Transformation zu leiten. Mit Hilfe von Mappings wird das markierte PIM elementweise in PSM

übersetzt (Vgl. Colomb, et al., 2006, S. 238 und MDA Guide, 2003). Analog wird die Transformation durch Pattern umgesetzt.

Transformation mit Hilfe von UML-Profilen

Bei diesem Vorgang wird ein PIM unter Verwendung eines plattformunabhängigen UML-Profils vorbereitet. Die Umwandlung in PSM kann stattfinden und wird durch ein zweites, plattformspezifisches UML-Profil ausgedrückt. Es ist nicht ausgeschlossen, dass zusätzliche Markierungen vom plattformspezifischen UML-Profil intervenieren (MDA Guide, 2003).

Andere MDA-Transformationen

Darüber hinaus gibt es in MDA die manuelle und die automatische Transformation. Bei der manuellen Transformation müssen Entwurfsentscheidungen getroffen werden, welche in Einklang mit den technischen Anforderungen sind. Bei der automatischen Transformation kann vom Model eine direkte Übersetzung in Code erfolgen, ohne dass das Modell Markierungen oder Profile verwendet. In diesem Fall sind keine Kenntnisse über das PSM nötigt, denn das erstellte PIM in komplett und hat alle Information innen.

Zu einer Transformation gehören ein oder mehrere Quellmodelle und ein Zielmodell. Dabei werden mindestens zwei Arten von Beziehungen zwischen den Modellen verzeichnet: Eine „Meta"- und eine „Abstraktion"-Beziehung. Die Abbildung 3 illustriert die Meta- und die Abstraktionsbeziehungen zwischen Modellen.

Meta-Beziehung

Bei der „Meta"-Beziehung übernimmt das Quellmodell die Rolle des Metamodells. Ein Metamodell kann einfach als Modell eines Modells definiert werden (MDA Guide, 2003) und bildet eine Elementmenge, welche als Grundlage dient, abgeleitete Modelle zu definieren. Die resultierenden Modelle beziehen sich auf ihr Metamodell. In diesem Fall liegen Quell- und Zielmodelle auf unterschiedlichen Metaebenen z.B. M2 und M1.

Abstraktion-Beziehung

Bei der „Abstraktion"-Beziehung ist das Quellmodell eine Abstraktion des Zielmodells. Analog das Zielmodell ist eine Konkretisierung des Quellmodells (Stahl et al. 2012, S. 59). Ein Modell ist eine strukturelle und maschinell Datensammlung, die verarbeitet werden kann. Ein Metamodell definiert und

beschreibt die syntaktischen Regeln und ihre Beziehung zueinander zur Darstellung des Modells (Stahl et al. 2012, S. 59).

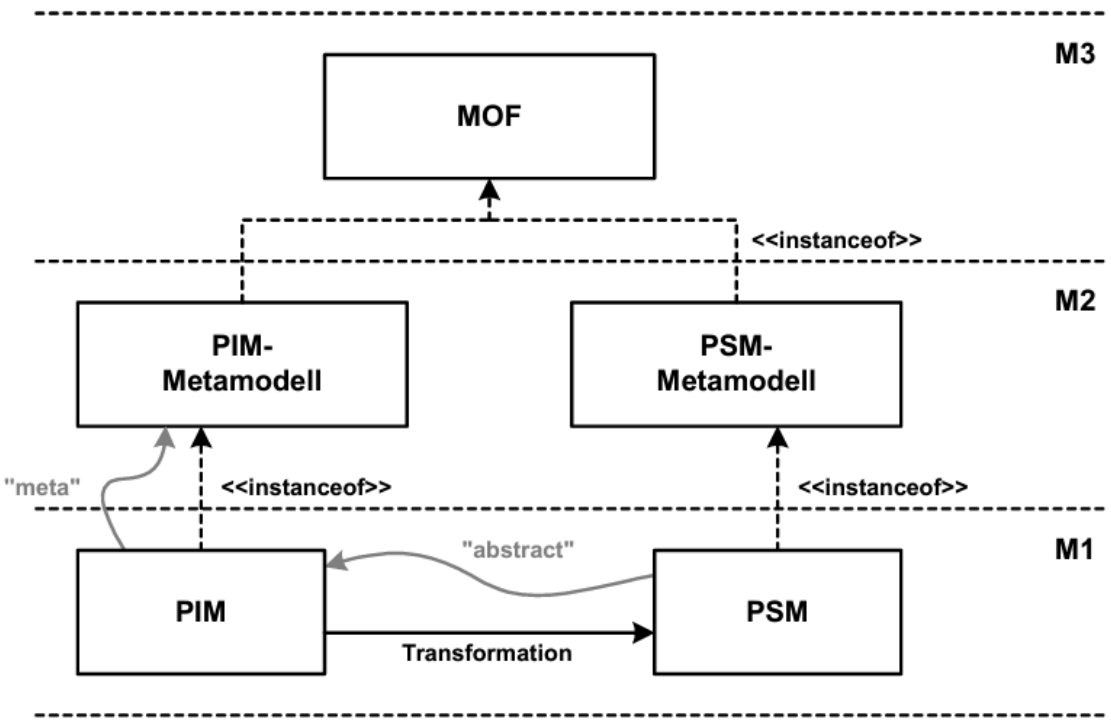

Abbildung 3: Meta- und Abstraktionsbeziehungen bei der Modellierung. Auszug aus Stahl et al. (2012, S. 63)

2.6.2 Modell-zu-Code-Transformation

Die Modell-zu-Code-Transformation (kurz, M-2-C-Transformation) ist die Überführung eines Modells in Quelltext (z.B. von PSM zu Code). Hierzu wird der Code mit Hilfe eines Werkzeugs generiert. Der Template-orientierte Ansatz ist ein typisches Beispiel für die M-2-C-Transformation

2.6.3 Template-basierte Codegenerierung

Die Generierung unter Verwendung von Template benötigt kein Metamodell. Für die Transformation wird eine Regel aufgestellt, welche den Rahmen schafft, um ein Modell z.B. Java in einem Java-Programm zu erzeugen. Eine Regel ist typisch wie folgt aufgebaut: Eine linke und eine rechte Seite der Bedingung. Dabei verweist die linke Bedingung auf das Modell und die rechte auf das Metamodell. Die rechte Bedingung wird beim Template-basierten Ansatz durch das Template selbst ersetzt (Petrasch & Meimberg, 2006).

Der Template-Ansatz ist automatisch und die Transformation wird durch ein Template-Engine gesteuert. Das Template definiert Platzhalter, welche von einem Generator mit Werten zur Laufzeit befüllt werden, und dient als Vorlage des zu erzeugenden Codes.

2.6.4 Generator

Generatoren sind Programme, die selber weitere Programme erzeugen. Sie können als Metaprogramme bezeichnet werden. Die Grundprinzipien eines Generators sind: Eingabe, Verarbeitung und Ausgabe. Die Eingabe kann ein Modell oder andere Spezifikationen sein, die dann in eine Berechnung eingegeben wird und anschließend ein Quellcode erzeugt wird (Stahl et al. 2012, S. 143).
Je nach Ausführungszeit eines Generators bezüglich des generierten Programms (auch Basisprogramm genannt) können Generatoren in drei Kategorien unterteilt werden:

- Vor der Erstellung des Codes
- während der Erstellung des Codes
- zur Laufzeit des Codes.

Problem beim Re-Generieren
Bei Eingabe wird ein Modell eingelesen und zur Weiterverarbeitung übergeben. In der Verarbeitungsphase werden Modelle miteinander verlinkt. Danach findet eine Validierung statt. Unter Einhaltung der Transformationsregeln wird das Eingangsmodell transformiert. Anschließen wird der Code generiert (Stahl et al. 2012, S. 139).

Eine Software lässt sich nicht immer vollständig generieren. Deshalb muss in vielen Fällen (z.B. bei großen Projekten) dem Produkt (generiertem Code) ein manuell erzeugter Code hinzugefügt werden. Bei der Software-Entwicklung mit den Techniken von MDSD werden Modell inkrementell erstellt. Das heißt ein Modell wird dabei öfter verändert. Dies führt erneut zu einer Generierung. Das manuell erzeugte Teil wird bei Regenerieren überschrieben. Zu diesem Problem stellen Petrasch & Meimberg (2006, S. 135-138) zwei Lösungsansätze vor.

Der erste Ansatz besteht darin, „Protected Regions" einzuführen. Das heißt bestimmte Bereiche vom Code werden bei der ersten Generierung durch spezielle Markierungen gekennzeichnet. Die markierten Bereiche werden bei einer

erneuten Generierung erkannt und nicht überschrieben. Ein möglicher Nebeneffekt ist die Reduzierung der Lesbarkeit und Wartbarkeit, welche durch die Vermischung von manuellen (im geschützten Bereich) und generierten Codes begünstigt wird (Petrasch & Meimberg 2006, S. 135-138).

Der zweite Ansatz löst das Problem der Vermischung auf. Hierbei wird das Produkt vom manuell erzeugten Code getrennt. Das heißt, jedes Mal, wenn eine manuelle Implementierung benötigt wird, wird sie in eine separate Datei geschrieben. Die wachsende Anzahl an Dateien kann später zu einer Unübersichtlichkeit führen. (Petrasch & Meimberg 2006, S. 135-138).

2.6.5 Alternative zu Generatoren

Metaprogramme

Neben dem klassischen Generator gibt es andere Metaprogrammierungstechniken wie Präprozessor, Integrierte Metaprogrammierfunktionalitäten oder Interpreter als Alternative. Ein Präprozessor agiert, wie bei Generatoren, vor dem Compiler. Bei einer M-2-C-Transformation wird der Code in die Zielsprache generiert. Ein wesentlicher Vorteil eines Präprozessors gegenüber einem Generator ist die Wiederverwendung, ohne die Wirkung der Zielsprache zu berücksichtigen (Stahl et al. 2012, S. 143).

Einer Programmiersprache können zusätzliche Funktionalitäten hinzugefügt werden, um sie mit einer dynamischen Metaprogrammierung zu befähigen. Dies hat jedoch den Nachteil, dass Typen nicht statisch geprüft werden können. Außerdem sind Metaprogramme für Plattformen, die auf mehreren Sprachen basieren, nicht geeignet (Stahl et al. 2012, S. 144).

Interpreter

Der Einsatz eines Interpreters sieht ein wenig anders aus. Er hat die Aufgabe ein Programm, welches in Form eines abstrakten Syntaxbaums (AST, Abstract Syntax Tree) vorliegt, nach Anweisungen zur Laufzeit zu durchsuchen und sie auszuführen. Einer der wesentlichen Unterschiede zu einem Generator liegt darin, dass Generatoren eher eingesetzt werden, wenn es um strukturelle Aspekte eines Systems geht. Wenn aber das Verhalten eines Systems im Vordergrund steht, kommen Interpreter zum Einsatz (Stahl et al. 2012, S. 144-174).

2.6.6 Top-Down und Bottom-Up Entwicklung

Mit Top-Down und Bottom-Up werden zwei Strategien bezeichnet, die zwei entgegengesetzte Wirkrichtungen in Prozessen aufweisen. Top-Down bezeichnet die Richtung: von Abstraktion zu Konkretisierung oder vom Allgemeinen zum Speziellen. Bei der Modellierung entspricht diese Strategie der Entwicklungsrichtung: Metamodell → Modell → Code.

Bei Bottom-Up wird entlang der entgegengesetzten Richtung entwickelt. Das heißt vom Speziellen zum Allgemeinen oder von der Konkretisierung zur Abstraktion. Bei der Modellierung werden Modellen ausgehend von Bekannten Quellcodes erstellt.

2.6.7 MDSD, Produktlinie und Software-Systemfamilie.

Als Software-Systemfamilie wird eine Menge von Produkten bezeichnet, welche mit Hilfe einer geeigneten und wiederverwendbaren Domänenarchitektur, die zugleich für Unterschiede in den verschiedenen Produkten flexibel ist, hergestellt werden kann. Die Domänenarchitektur beschreibt den Weg vom Modell zum Produkt und fasst alle notwendigen Elemente zusammen, welche für die Realisierung benötigt wird. Dazu gehören das Metamodell, die Plattform sowie die Transformationen (Stahl et al. 2012).

Unter (Software-)Produktlinien wird eine Menge individueller (Software-) Produkte verstanden, die fachlich aufeinander zugeschnitten sind und inkrementell sowie iterativ entwickelt werden.
Der Entwicklungsprozess einer Produktlinie unterliegt verschiedenen Phasen, wie z.B. der Definition der Domäne und deren wichtigsten Kernkomponenten oder die Analyse der Unterschiede und Gemeinsamkeiten zur Wiederverwendung. Mehr dazu im Unterkapitel 2.7.
Eine Software-Systemfamilie eignet sich gut als Grundlage für die Realisierung einer Produktlinie. Wiederum dient die Wiederverwendung von Gemeinsamkeiten bei der Entwicklung von Produktlinien als Basis im MDSD (Stahl et al. 2012).

Weiterhin betonen Stahl et al. (2012) in ihrer Ausführung, dass MDSD und Produktlinien-Entwicklung zwei voneinander zu trennende Verfahren, die jeweils von den Techniken und der Methodik des anderen profitieren können. So kann

MDSD als Mittel zur Bewerkstelligung einer Produktlinien-Entwicklung fungieren, während die Produktlinien-Entwicklung als Analysemethode für MDSD betrachtet werden kann (Stahl et al. 2012).

2.7 Überblick: Software-Produktlinien-Entwicklung

MDSD wurde als Verfahren zur Generierung von ausführbarem Code vorgestellt. Darin ist das Konzept der Wiederverwendung von gemeinsamen Software-Artefakten und den Mechanismen der agilen Software-Entwicklung integriert (Petrasch & Meimberg, 2006, S. 157).
Die agile Methode wird im Folgenden nicht näher betrachte. Stattdessen wird der Fokus auf die verschiedenen Etappen in SPLE gelegt. Wie die agile Produktentwicklung funktioniert, wurde in (Schröder, 2018) betrachtet. Für einen praxisorientierten Ansatz wird auf (Steinbrecher, 2020) verwiesen.

2.7.1 SPL-Definition

Produktlinien sind in vielen Bereichen des Lebens zu finden: Von der Flugzeug-, Auto-Industrie bis hin zu mobilen Endgeräten oder Restaurants.
Eine Software-Produktlinie ist nach Clements und Northrop (2002, S. 5) eine Menge von Softwaresystemen mit einem gemeinsamen, verwalteten Funktionsumfang, die auf spezifische Anforderungen einer Domäne oder einer bestimmten Aufgabe angepasst wurden und auf der Grundlage eines gemeinsamen Kernbestands (Core Assets) in einer vorgegebenen Weise entwickelt werden (Vgl. Siegmund et al., 2009 und McGregor, Monteith & Zhang, 2011).

Kernbestand
Der Kernbestand repräsentiert allgemein die Menge der Funktionalitäten und Eigenschaften, welche sich in aller Derivaten vorhanden ist.

2.7.2 Illustration: Pizzeria-Beispiel

Im Pizzeria-Beispiel können der Teig, die Pizzasauce und die verschiedenen Zutaten zum Kernbestand bei der Produktion von Pizzen in einer Pizzeria angeordnet werden.

Von diesen Grundzutaten ausgehend können verschiedenen Varianten durch Kombinierung der Zutaten unter Aufsicht vom Chefkoch hergestellt werden. Dabei werden andere Zutaten wie Teig oder Pizzasauce in hergestellten Pizzen immer wiederverwendet.

Aus diesem unvollständigen Beispiel können zwei Entwicklungsschritte bei der SPL entnommen werden: Die Entwicklung von Kernkomponenten für Produktfamilien oder Produktlinien (Domain Engineering-Phase) und die Entwicklung jedes Produkts mit den Kernkomponenten (Application Engineering-Phase). Hinzu kommt das Management für eine reibungslose Entwicklung der beiden Phasen.

Die Abbildung 4 fasst die verschiedenen Phasen bei SPLE zusammen. Die Kernkomponentenentwicklung (Core Asset Development) ist die Aktivität der Domain Engineering, während Die Produktentwicklung (Product Development) die Aktivität der Applikation-Engineering Phase ist. Die dritte Aktivität ist das Management. Alle drei sind iterativ und miteinander verzahnt, was auf die gegenseitige Abhängigkeit zwischen den Aktivitäten hinweist.

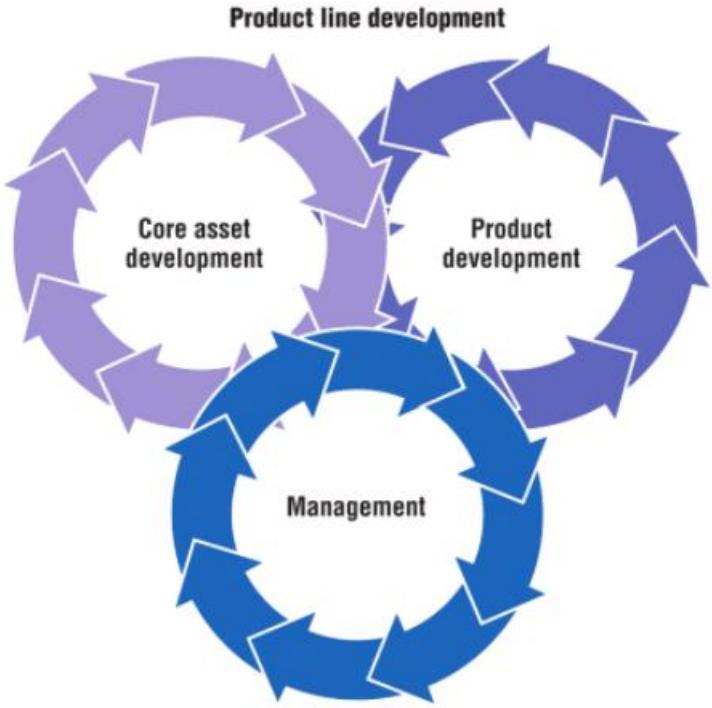

Abbildung 4: Iterative Phasen in SPLE nach Northrop (2002, S. 30)

2.7.3 Kernkomponenten-Entwicklung (Domain Engineering)

In dieser Phase findet die eigentliche Entwicklung wichtiger Kernkomponenten des Produkts statt. Dafür wird ein Mechanismus in Gang gesetzt, welcher aus eine Eingangsmenge von aus der Anforderungsanalyse kommenden Eingaben eine Menge von Ausgaben produziert. Die Elemente der Ausgabe werden für die weitere Phase der Produktentwicklung eingesetzt. Im Folgenden wird ein Überblick über die Ein- und Ausgaben dargestellt.

1) Elemente der Eingabe

Die Eingangselemente können als Erwartungen oder Anforderungen zusammengefasst werden. So ergeben sich folgende Konstellationen (Clements & Northrop, 2002, S. 29-54):

- Die Anforderungen an das Produkt
- Einsatz von Frameworks, Entwurfsmuster, Architekturmuster (-Styles)
- Die Anforderungen an die Produktion
- Die Strategie der Produktion
- Implikation der existierenden Kernkomponenten

2) Elemente der Ausgabe

Nachdem die Eingabeelemente bereits grob beschrieben sind, werden Ausgabeelemente nach der Verarbeitung erwartet. Es betrifft folgende Bausteine (Clements & Northrop, 2002):

- Umfang der Produktlinie oder „Scoping"
    - Eingrenzung der Domäne
    - Relevante Features ermitteln
    - Wirtschaftliche Entscheidungen treffen
- Kernkomponenten
- Produktionsplan

2.7.4 Produktentwicklung (Applikation-Engineering)

Die Ausgangsprodukte des Domain Engineerings bilden die Basis der Produktentwicklung. Da Produkte in ihren Eigenschaften und Funktionalitäten voneinander abweichen, fließen produktspezifische Anforderung mit in die Entwicklung ein (Siehe Abbildung 4 oben). Jede auf die Anforderungen

zugeschnittene Produktvariante wird auf der Grundlage des Ergebnisses des Domain Engineering in Lösungsraum (Siehe Abbildung 5 unten) hergeleitet (Kästner & Apel, 2013).

2.7.5 Management

Domain und Applikation Engineering unterliegen dem unverzichtbaren Management, das weiterhin in technischem und organisatorischem Management unterschieden wird. Beide sind maßgeblich in der Entwicklung der Kernkomponenten sowie der Produkten beteiligt.

Organisatorisches Management

Das organisatorische Management koordiniert und verteilt die Ressourcen und das Budget, die für Erfüllung der verschiedenen Aufgaben notwendig sind. Eine weitere Aufgabe besteht darin, die Aktivitäten in den Iterationen harmonisch zusammenzufügen. Während der Entwicklung wird Ausschau danach gehalten, ob alle Konzepte sorgfältig dokumentiert werden. Darüber hinaus stellt sich das organisatorische Management als Ansprechpartner dar für externe Kommunikation mit anderen Organisationen, die unmittelbar mit der Produktlinien-Entwicklung in Verbindung stehen (Clements & Northrop, 2002).

Technisches Management

Das technische Management hat die Aufgabe, eine reibungslose Entwicklung der Kernkomponenten und der Produkte zu gewährleisten. Es wird sichergestellt, ob die richtigen Produkte korrekt entwickelt werden, indem die zuständigen Prozesse für Kernkomponenten und Produkte überwacht werden (Clements & Northrop, 2002).

2.7.6 Problemraum und Lösungsraum

Problem- und Lösungsraum sind zwei Konzepte zur Trennung der Nutzeranliegen (das Problem) von den entsprechenden Lösungswegen zur Erfüllung der Anliegen. Der Problemraum „beschreibt" und der Lösungsraum „konkretisiert". Diese Modularisierung erinnern an die Modell-zu-Code-Transformation (Siehe Abbildung 2 oben)

Problemraum (Problem Space)

Der Problemraum setzt sich aus Abstraktionen der Domäne zusammen, welche das erwartete Verhalten des zu entwickelten Software-Systems und seine Anforderungen beschreiben. In diesem Bereich wird die Domäne-Analyse durchgeführt. Das Ergebnis ist in Features zusammengefasst (Kästner & Apel, 2013). Erst wenn das Problem gut verstanden und definiert wurde, können Lösungen ausgearbeitet werden (Siehe Abbildung 5 unten).

Lösungsraum (Solution Space)

Der Lösungsraum umfasst implementierungsorientierte Abstraktionen, wie z.B. Code-Artefakte oder Modelle. Die Artefakte des Lösungsraums werden auf die Features des Problemraums zugeordnet. Das ist die Mapping (Siehe Abbildung 5 unten).

Sie kann eine Namenskonvention sein oder ein Regelwerk, welches maschinell verarbeiten werden kann und in Kompositionswerkzeugen oder Generatoren oder auch Präprozessoren kodiert werden (Kästner & Apel, 2013). Produktvarianten werden sowohl aus den gemeinsamen als auch aus den variablen Teilen zusammengesetzt.

Anforderungsanalyse

Die Anforderungsanalyse einer Produktlinie erfasst die erforderlichen Funktionalitäten in einer Menge von Modellen, wie z. B. einem Anwendungsfallmodell, einem Objektmodell (Kang, Lee, & Donohoe, 2002.). In diesem Fall handelt es sich um ein Feature-Modell. Das Resultat der Anforderungsanalyse fließt als Eingangselement zur Entwicklung der Kernkomponenten im Domain Engineering ein.

Daraus entstehen Features, welche als Grundlage zur Erstellung des Feature-Modells dienen (Siehe Abbildung 5 unten). Diese Features stehen für die verschiedenen Anwendungsfälle zur Auswahl (Application Engineering).

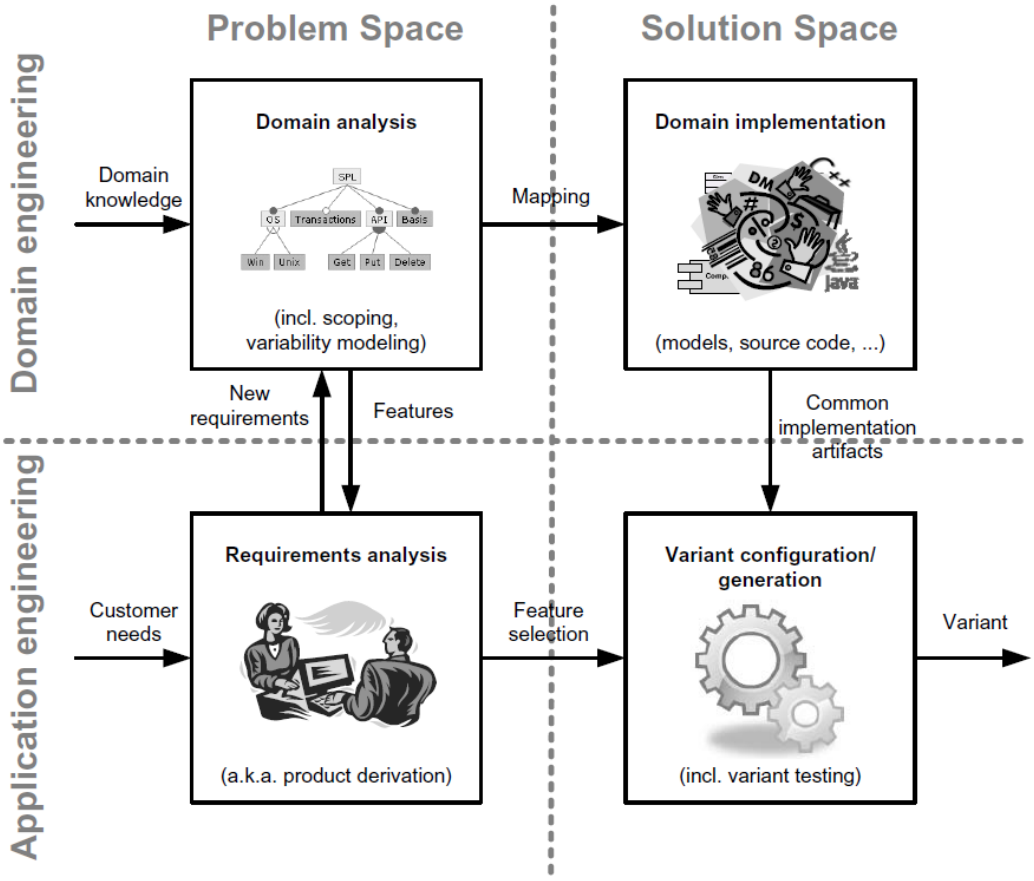

Abbildung 5: Übersicht in SPL als 4-Entwicklungsquadrant (Kästner & Apel, 2013, S. 351).

Variantenkonfiguration

Nach der Fertigstellung des Modells können Features gemäß einem spezifischen Anwendungsfall anhand einer Konfigurationsoption ausgewählt und generiert werden. Vor dem Generieren muss die Produktkonstellation durch das Feature-Modell mit Hilfe von eingebauten Einschränkungen (Constraints) validiert werden (Jin, 2018). Diese Einschränkungen funktionieren analog zu OCL (Siehe Abschnitt 2.5.2).

2.7.7 Feature-Konzept in Produktlinien

Durch die Definition der Kernkomponenten ist eine Software-Produktlinie der Inbegriff einer strategischen, geplanten Wiederverwendung in einer Domäne. Das Ergebnis der Analyse dieser Domäne wird Feature genannt. Features fassen alle Gemeinsamkeiten und relevanten Anforderungen sowie die Unterschiede zur

Produktdiversifizierung zusammen, die von besonderer Bedeutung der Domäne sind. Sie fungieren auch als gemeinsamer Sprachgebrauch zwischen den Stakeholdern.

Apel & Kästner (2009) definieren in ihrer Studie, dass ein Feature eine Funktionalitätseinheit eines Softwaresystems ist, welche eine Anforderung erfüllt, eine Entwurfsentscheidung darstellt und eine mögliche Option für die Konfiguration bietet. Diese Definition entspricht dem bereits betrachteten Feature-Entstehungsprozess.
Software-Produktlinie werden Feature-orientiert entwickelt. Diese Feststellung führt zur Betrachtung der feature-orientierten Software-Entwicklung (FOSD, Feature Oriented Software Development).

2.7.8 Feature-Orientierte Software-Entwicklung (FOSD)

Nach den Ausführungen von Kästner & Apel (2013, S. 346) wird FOSD als ein Paradigma für die Konstruktion, Anpassung und Synthese von umfangreichen und variablen Software-Systemen mit Schwerpunkt auf Struktur, Variation und Wiederverwendung bezeichnet.

FOSD eignet sich gut für die SPLE, denn die Phasen des FOSD-Prozesses decken sich mit den Phasen der SPLE. So wird in der Domäne-Analyse Phase die Dimension der Domäne festgelegt. Dabei werden irrelevante Funktionalitäten aussortiert und relevante Funktionalitäten festgehalten sowie die verschiedenen Konstellationen der Features festgelegt (Vgl. Apel & Kästner, 2009).

Entwurfs- und Implementierungsinformationen werden in der Entwurfs- und Implementierungsphase durch eine Menge von Feature-Artefakten gekapselt. Es findet eine Mapping zwischen Features und Artefakten statt und durch eine valide Selektion von Features werden Varianten generiert (Vgl. Apel & Kästner, 2009).
FOSD lässt sich in drei wesentliche Aktivitäten zusammenfassen: das Modellieren der Features, die Interaktion der Features untereinander und die Implementierung (Vgl. Apel & Kästner, 2009).

a) Erstellung des Feature-Modells

Die Features-Modellierung ist die wichtigste Aktivität der Domäne-Analyse. Das Feature-Konzept bietet die Möglichkeit nicht nur die Gemeinsamkeiten und die Variabilität der SPL zu beschreiben, sondern auch den Problemraum formal darzustellen. Dieses Konzept wird im Feature-Modell ausgedrückt, mit dem Ziel Beziehungen und Abhängigkeiten von Features untereinander aufzuzeigen.

Das dafür benötigte Ausdrucksmittel ist das Feature-Diagramm, welches die visuelle Modellierung anhand der grafischen Symbole ermöglicht (konkrete Syntax). Die Bedeutung und die Kombinierungsmöglichkeiten der grafischen Elemente werden beim Modellieren beachtet. So ergibt sich ein Feature-Syntax-Baum (FST, Feature-Syntax-Tree), wobei die Wurzel das zu modellierende Konzept darstellt und die anderen Blätter die Features. Eine Menge von logischen Ausdrücken steht für die Syntax zu Verfügung.

Darüber hinaus können Features miteinander verknüpft werden, um bestimmte Einschränkungen oder Abhängigkeiten auszudrücken. In der Abbildung 6 wird das Zusammenspiel der FST mit seiner Syntax sowie die Möglichkeit zur Einschränkung beispielhaft im Werkzeug FeatureIDE aufgezeigt. Es geht um eine Graph-Produktlinie (die Wurzel) mit seinen Features (die Blätter). Die Symbole und ihre Bedeutungen (Alternative, Or, etc…) rechts neben dem Feature-Modell bilden die gesamten Verknüpfungsmöglichkeiten. Einschränkungen und weitere Abhängigkeiten werden durch beigefügte logische Ausdrücke umgesetzt.

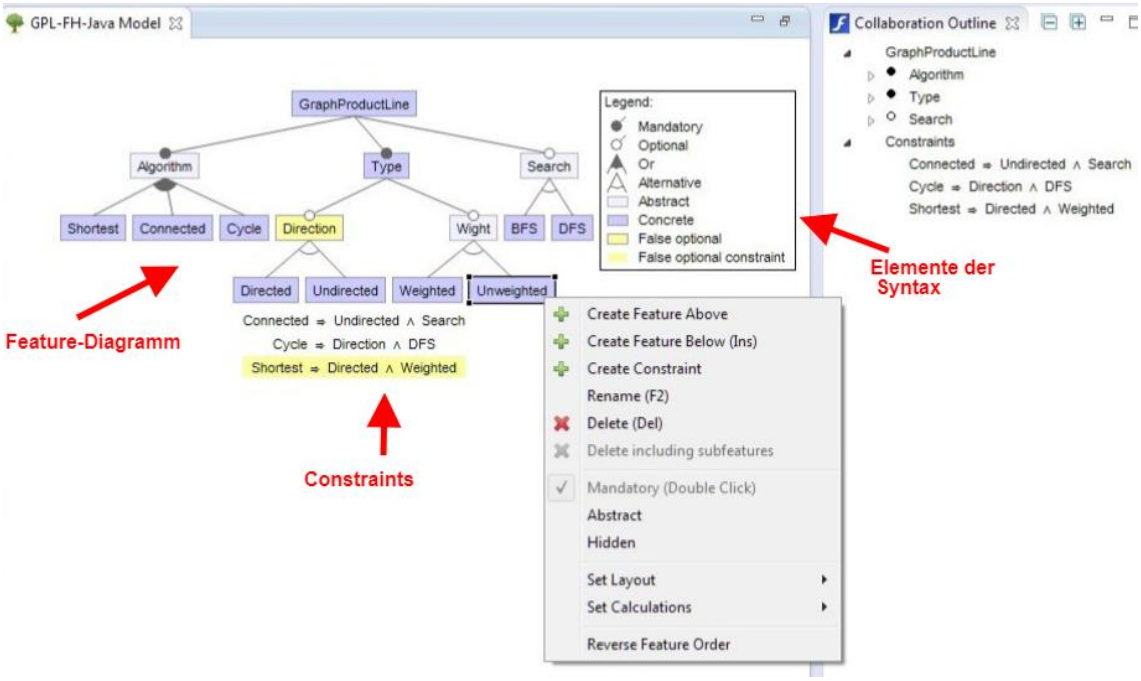

Abbildung 6: Feature-Modell eines Graphen in FeatureIDE (METOP-FeatureIDE, 2020)

b) Interaktion der Features

Die Features als Code-Artefakte in der Domäne-Implementierung umzusetzen, erfolgt nach der Domäne-Analyse. Ein höherer Aufwand muss allerdings betrieben werden, um die Features miteinander zu verknüpfen. In vielen Anwendungsfällen reicht eine einfache Verknüpfung nicht aus. Manuell erzeugte Codes müssen in diesen Fällen hinzugefügt werden, um das Gebilde inhaltlich konsistent aufrechtzuerhalten. Apel & Kästner (2009) veranschaulichen diese Tatsache in ihren Ausführungen anhand einer konkreten Beispielimplementierung.

Es geht in diesem Beispiel um zwei isolierte Features, die einzeln funktionsfähig sind: die Implementierung einer verketteten Liste (Linked List) und die invertierte verkettete Liste. Bei der Vereinigung dieser Features kommt eine doppelt verkette Liste zustande. Wird ein Element am Anfang oder am Ende der neuen doppelt verketteten Liste hinzugefügt, bleiben die Verbindungen zwischen Listenelementen nicht erhalten. Denn die Rückwärtsreferenz wird nicht entsprechend angepasst und die Referenz auf das letzte Element nicht korrekt gesetzt, wenn die Liste leer ist. Das unerwartete Verhalten nach der Verknüpfung der beiden Features muss manuell behoben werden.

c) Feature-Implementierung

Features werden einzeln und separat in der Domäne-Implementierung in Code umgesetzt und mittels Kompositionswerkzeuge oder Generatoren zusammengesetzt, um ein Softwareprodukt zu erstellen. Diese Programmierparadigma wird unter feature-orientierten Programmierung (FOP, Feature Oriented Programming) zusammengefasst.

d) Besonderheit der FOP

Anders als bei OOP lässt sich das „Single Responsibility Principle" von SOLID in der feature-orientierten Programmierung nur schwer bzw. nicht umsetzen. Features sind „Cross-Cutting Concern" (querschnittliche Belange) und durchdringen mehrere Schichten eine Softwarearchitektur. FOP kann als Methodik für FOSD verstanden werden, denn das Ziel der FOP besteht darin, dass Entwurf und Code eines Softwaresystems entlang der Features zerlegt werden, die es bietet. Die Features werden anschließend verschiedenartig zusammengefügt, um eine Produktdiversifizierung zu ermöglichen (Kästner & Apel, 2013). Die Abbildung 7 illustriert den Kerngedanken von FOP.

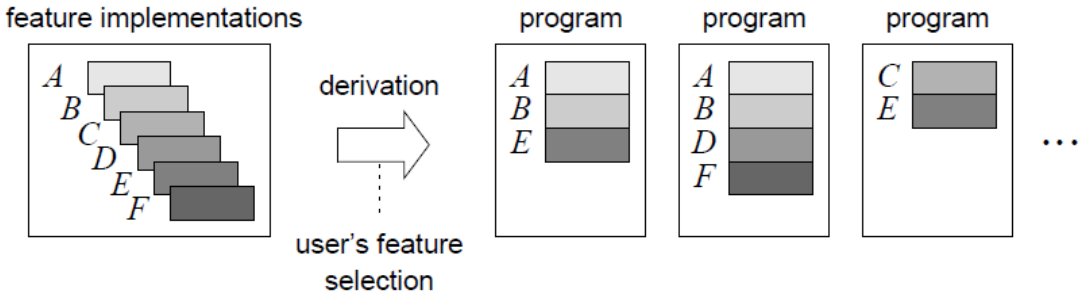

Abbildung 7: Kernidee der FOP (Apel & Kästner, 2009, S. 10)

e) Feature-Modul

Ein Feature-Modul stellt eine Einheit dar, welche ein Feature implementiert. Es ist mit einem Paket in einem Java-Projekt vergleichbar, wobei jedes Paket möglichst eine Funktionalität behandelt.

2.8 Bewährte Entwicklungsvorgehensweise

Neben den Prinzipien von FOSD für die SPLE werden einige bewährten Programmierpraktiken (S.O.L.I.D und „Clean Code") für eine bessere Wartbarkeit, Wiederverwendbarkeit oder Lesbarkeit des Codes beschrieben. Ferner wird das Entwurfsmuster Fabrik-Methode vorgestellt.

2.8.1 Die S.O.I-Prinzipien von SOLID

Die Designprinzipien helfen dabei, wartbare, verständliche und flexible Software zu entwickeln. So kann eine unnötige Steigerung der die Komplexität bei Weiterentwicklung an einem bestehenden Software Stack verhindert werden. Das S in steht für „Single Responsibility", O für „Open/Closed", L für „Liskov Substition", I für Interface Segregation und D für „Dependency inversion".
Innerhalb dieser werden folgende Prinzipien relevant sein: Single Responsibility, Open/Closed und Interface Segregation (zusammen S.O.I)

- **Single-Responsibility-Prinzip (SRP)**

Dieses Prinzip besagt, dass jede Klasse nur eine Aufgabe oder einen einzigen Zweck haben sollte. Das bedeutet, dass eine Klasse nur eine Aufgabe erfüllt und sollte nur einen Grund haben sollte, sich zu ändern. Um nach dem SRP-Prinzip entwickeln zu können, sollten sich Gedanken über den Problembereich, die geschäftlichen Anforderungen und die Anwendungsarchitektur gemacht werden. Mit diesem Prinzip können eine hohe Kohäsion sowie die Robustheit erreicht werden, welche zusammen die Fehlerquote senken (Janschitz, 2015 und Millington, 2022).

- **Open/Closed-Prinzip (OCP)**

Dass Klassen für Erweiterungen offen und für Änderungen geschlossen sein sollten, ist der Grundsatz dieses Prinzips. Auf diese Weise werden neue Fehler im Code vermieden, die durch Änderungen eintreten. OCP kann mit Vererbungen oder durch Implementierung von Schnittstellen realisiert werden (Janschitz, 2015 und Millington, 2022).

- **Interface Segregation**

Größere Schnittstellen sollten in kleinere Schnittstellen aufgeteilt werden. Auf diese Weise kann sichergestellt werden, dass sich implementierende Klassen nur

um die Methoden kümmern müssen, die für sie von Interesse sind (Janschitz, 2015 und Millington, 2022).

2.8.2 Clean Code

Neben den SOLID-Prinzipien sollte auch geachtet werden, dass Code gut und „sauber" geschrieben wird. Mit Clean Code soll die Wartbarkeit über längeren Zeitraum erhalten bleiben. Martin Robert C. (2013) zeigt in seinem Werk, wie ein sauberer Code auszusehen hat und sogenannte „Code-Smells" im Code beseitigt werden können. Einige wichtige Empfehlungen sind in der folgenden Tabelle verfasst.

Tabelle 1: Auszug aus den Clean Code Prinzipien (Martin, 2013)

Aussagekräftige Namen	Distinkte, aussprechbare, absichtserklärende Namen sollten bei der Benennung von Variablen, Methoden oder Klassen verwendet werden. Methodennamen sollten Verben sein, während Klassennamen Nomen sind.
Eine Abstraktionsebene pro Funktion	Um sicherzustellen, dass eine Funktion genaue „eine Sache" tut, soll dafür sorgt werden, dass alle Anweisungen innerhalb der Funktion auf der gleichen Abstraktionsebene liegen.
DRY (Don't Repeat Yourself)	Code-Duplizierungen sollten vermieden werden, denn der Code wird dadurch aufgebläht und erfordert mehr Aufwand bei Änderungen
Kommentar	sollten nicht benutzt werden, um schlecht implementierten Code zu verschönen. Vielmehr sollten sie einen informativen, erläuternden, warnenden Charakter haben.

Diese Maßnahmen repräsentieren nur einige Beispiele davon, was der Autor in seinem Werk aufgelistet hat. In dieser Arbeit werden auf weitere nützliche bewährte Praktiken des Autors Rücksicht genommen, die nicht in der Tabelle aufgelistet sind.

2.8.3 Entwurfsmuster: Fabrik-Methode

Der Untersuchung von Zhart (2023, zitiert nach Gamma et al., 1995) zufolge wird Entwurfsmuster (Design Pattern) sind typische Muster für häufig auftretende Probleme beim Softwareentwurf. Sie sind vorgefertigte Entwürfe, die angepasst werden können, um ein wiederkehrendes Entwurfsproblem im Code zu lösen. Entwurfsmuster werden in drei Kategorien unterteilt (DigitalOcean 2022, zitiert nach Gamma et al., 1995):

- Kreation- oder Erzeugungsmuster: Die Entwurfsmuster befassen sich mit der Erstellung von Objekten.
- Strukturelle Entwurfsmuster: Sie befassen sich mit Klassenstrukturen wie die Komposition oder Vererbung.
- Verhaltensorientierte Muster: Diese Art von Entwurfsmustern bietet allgemein Lösungen für die bessere Interaktion zwischen Objekten, wie eine lose Kopplung und Flexibilität für eine einfache Erweiterung in der Zukunft erreichen werden kann.

Die Factory-Methode gehört zu der Kategorie der „Kreationsmuster" und besagt, dass die Erstellung von Objekten an Unterklassen delegiert wird. Hierbei wird die Objekterstellung durch den „Kreator" abstrakt definiert. In diesem Fall ein Produkt (Siehe Abbildung 8). Die unterschiedlichen konkreten Produkte werden von den abgeleiteten konkreten „Kreators" individuell erzeugt.

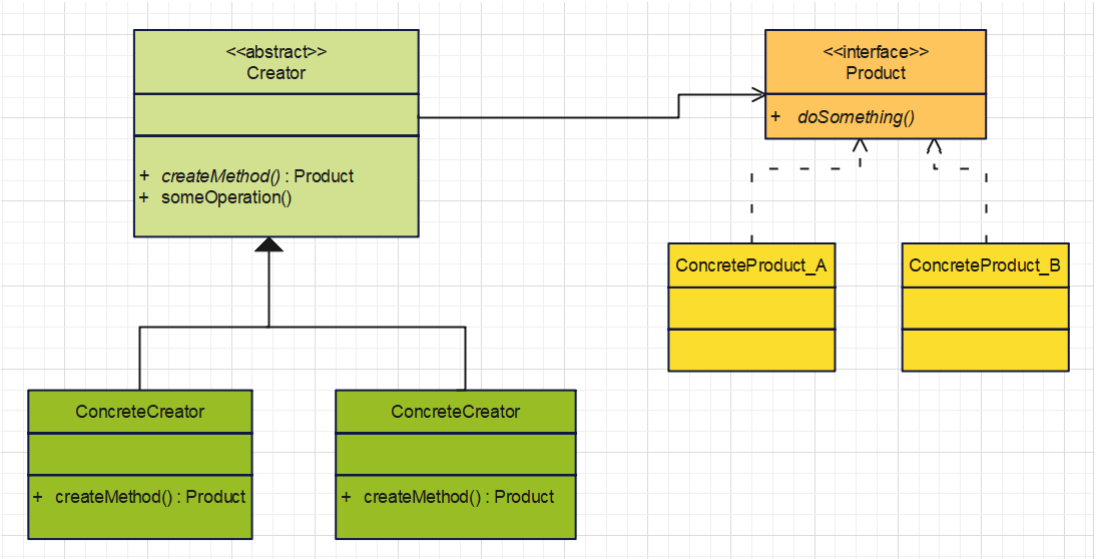

Abbildung 8: Factory Method design Pattern

Die Objekterzeugung in einer Klasse, die das instanziierte Objekt selbst verwendet, bindet die Klasse an dieses Objekt und macht es unmöglich, die Instanziierung unabhängig von der Klasse zu ändern. Das Factory-Pattern zielt darauf ab, die lose Kupplung durch mehr Flexibilität zu erreichen, indem die Objekterzeugung separat behandeln wird.

Darüber hinaus können neue Klassen hinzugefügt werden, ohne die bestehenden zu ändern. Es reicht aus, dass die neuen Klassen die Schnittstelle „Produkt" (Siehe Abbildung 8) implementieren. Danach sollen die entsprechenden konkreten „Kreators" eingebunden werden. Neben der Modifizierbarkeit wird mit diesem Entwurfsmuster die Testbarkeit der Anwendung erhöht (IONOS, 2021).

Da die Produktlinie-Anwendung iterativ und inkrementell wird, werden mit jedem Inkrement neue Features der Anwendung hinzugefügt. Hinter jedem Feature steht eine Aspekt-Klasse. So können mit der Fabrik-Methode neue Features der Anwendung addiert werden, ohne die vorherigen zu verändern.

3 Analyse

Nachdem die grundlegenden Konzepte der MDSD und SPLE im vorhergehenden Kapitel erarbeitet wurden, werden in diesem Kapitel die Anwendung vorgestellt, die in dieser Arbeit entwickelt werden soll. Danach werden die Anforderungen an den zu erstellenden Prototyp beschrieben. Anschließend wird die Werkzeugunterstützung zur Erfüllung der Anforderungen vorgestellt.

3.1 Die „Pizza-Pronto" Anwendung

Die „Pizza-Pronto" Anwendung ist ein der viele Bestellsysteme, welche in der Lehre an der technischen Hochschule Brandenburg benützt wird. Sie besteht aus mehreren aufeinander aufbauenden Teilen. Für die vorliegende Arbeit werden die drei ersten Teile berücksichtigt: Übung1, Übung2 und Übung3. Jede Übung besteht aus POJO-Klassen, Testklassen für die POJO-Klassen sowie eine Main-Klasse zur Ausführung und eine GUI-Anwendung zum Testen.

Die Übung1 und Übung2 haben drei POJO-Klassen: *ChefVO*, *PizzaVO* und *CustomerVO* sowie die entsprechenden TestKlassen: *JUnitTestChefVO*, *JUnitTestPizzaVO* und *JUnitTestCustomerVO*. Eine vierte Klasse kommt in der Übung3 zusätzlich dazu: *OrderVO* mit der Testklasse *JUnitTestOrderVO*.
Hinzu kommen die Main-Klasse und die GUI-Anwendung für die jeweiligen Teile.

Die Pizza-Pronto Anwendung verfolgt vier Ziele, die auf Übungen verteilt sind und Schritt für Schritt erreicht werden sollen (Schmidt & Buchholz, 2022):

- Objektorientierte Programmierung in Java
- Kennenlernen und verstehen von Teilen der Java-API
- Kennenlernen und verstehen von objektorientiertem Design
- Realitätsnähe durch eine größere Aufgabe

Darüber hinaus wurden die Implementierungen sowie deren Klassendiagrammen samt den technischen Anforderungen mitgeliefert, sodass der Fokus auf die SPLE gelegt wird. Die getroffenen Entwurfsentscheidungen werden vom ersten bis zum letzten Teil der „Pizza-Pronto" Anwendung fortlaufend verfeinert und sollten nicht geändert werden.

Diese Verfeinerungen werden sich später in dieser Arbeit als Features herauskristallisieren. Features oder Komponenten die sich wiederverwenden lassen (Innerhalb einer Übung oder Übungsübergreifend), werden den Kernbestand bilden.

3.2 Anforderungen

Dieses Unterkapitel dient dazu die Anforderungen an die zu erstellende Produktlinie zu formulieren. Die wesentlichen Funktionalitäten werden beschrieben, welche einen Mehrwert beizutragen haben. Aus einem kleinen Beispiel in der Lehre an der technischen Hochschule Brandenburg soll ein Prototyp einer Produktlinie entwickelt werden.

Es geht darum eine Übung, die fortlaufend während des Semesters entsteht, als Produktlinie darzustellen. Mit „fortlaufend entstehen" ist gemeint, dass neue Aufgaben in späteren Versionen der Übung hinzugetan oder ältere Aufgaben ersetzt oder verändert werden. Die „Pizza-Pronto" Anwendung sowie die dazu gehörigen Klassendiagramme, welche die Entwurf-Entscheidungen beinhalten, wurden mitgeliefert und müssen nicht neu skizziert und implementiert werden.

Die Anforderungen werden zweistufig formuliert und nummeriert, um spätere Verlinkungen zu vereinfachen. Zunächst wird eine grobe Übersicht über die Erwartungen an das System dargestellt und danach eine detaillierte Sicht der Anforderungen vorgestellt.

3.2.1 Grobe Anforderungen

Hier werden grob die Erwartungen an die zu erstellende Produktlinie-Anwendung erfasst. Sie dient zur Übersicht über das Ziel, welches eingangs der Arbeit formuliert wurde und fasst die wichtigsten Anforderungen zusammen.

Anforderungen A-01:

Von einem Grundprodukt ausgehend sollen weitere Produkte entwickelt werden, sodass eine Produktlinie entsteht. Dabei soll jedes Produkt im Einklang mit den vorgegebenen Entwurf-Entscheidungen sein.

Anforderungen A-02:
Jedes Produkt soll über eigene Eigenschaften verfügen, welche durch eine sinnvolle Konfiguration zusammengesetzt werden.

Anforderungen A-03:
Nach der Selektion in der Konfiguration soll der Quellcode für diese Konfiguration generiert werden können.

Anforderungen A-04:
In den generierten Produkten bzw. Codes sollten sich die zuvor getroffenen Entwurfsentscheidungen widerspiegeln.

3.2.2 Funktionale Anforderungen

Neben den groben Erwartungen werden die einzelnen Funktionalitäten der Produktlinien-Anwendung beschrieben. Diese funktionalen Anforderungen werden aus der Sicht eine(r)s Benutzer(in)s der Anwendung wiedergegeben. Deshalb werden diese als „User-Stories" erfasst.
Die Anwendung wird in drei Haupt-Features bestehen: Die Features „Uebung1", „Uebung2" und „Uebung3". Deshalb werden diese verschieden in drei Gruppen unterteilt wie folgt:

a. Anforderungen in „Uebung1"

Anforderung A-05:
Als Benutzer(in) möchte ich zwischen den verschiedenen Darstellungen des Klassen-Konstruktors auswählen.

Anforderung A-06:
Als Benutzer(in) möchte ich zwischen der Setter-Methode und der eventuell nötigen Konsistenzprüfung auswählen.

Anforderung A-07:

Als Benutzer(in) möchte ich immer die Features mit Hilfe der dazugehörigen Test-Klassen testen.

Anforderung A-08:
Als Benutzer(in) möchte ich Anwendung über die GUI testen.

b. Anforderungen in „Uebung2"

Anforderung A-09:
Als Benutzer(in) möchte ich die Varianten der „toString-Methode" sehen.

Anforderung A-10:
Als Benutzer(in) möchte ich die Varianten der „hash-Methode" sehen.

Anforderung A-11:
Als Benutzer(in) möchte ich die Varianten der „equal-Methode" sehen.

Anforderung A-12:
Als Benutzer(in) möchte ich die „clone-Methode" sehen.

Anforderung A-13:
Als Benutzer(in) möchte ich immer die Features mit Hilfe der dazugehörigen Test-Klassen testen.

Anforderung A-14:
Als Benutzer(in) möchte ich Anwendung über die GUI testen.

c. Anforderungen in „Uebung3"

Anforderung A-15:
Als Benutzer(in) möchte ich die Gestaltung der Assoziation zwischen zwei einer existierenden und einer neuen erstellten Klasse kennenlernen.

Anforderung A-16:
Als Benutzer(in) möchte ich die effiziente bzw. die nicht- effiziente Implementierung der „toString-Methode" entdecken.

Anforderung A-17:

Als Benutzer(in) möchte ich die Implementierung der neu erstellten Klasse aus der Assoziation entdecken.

Anforderung A-18:
Als Benutzer(in) möchte ich immer die Features mit Hilfe der dazugehörigen Test-Klassen testen.

Anforderung A-19:
Als Benutzer(in) möchte ich die Anwendung über die GUI testen.

3.3 Technologie und Werkzeugunterstützung

Wie eingangs dieser Arbeit angekündigt, wird es keine komplette Werkzeuganalyse und -vergleich geben. In diesem Unterkapitel werden nur bewährte Werkzeuge bzw. Technologien zum Thema SPLE vorgestellt, welche für die Bewältigung der Aufgaben benötigt werden.
Thüm, et al. (2014) erwähnen in ihren Ausführung führende kommerzielle Tools wie *pure::variants* und *Gears* sowie open-source Variante wie *KConfig*, *CDL*, *GUIDSL*, *S.P.L.O.T.* und *FAMA* und geben dabei nur eine knappe Beschreibung an.

Über FeatureIDE wird ausführlicher geschrieben und es wird als Werkzeug zur Unterstützung der FOSD bezeichnet, welches alle Phasen (Domänenanalyse, Anforderungsanalyse, Domänenimplementierung und Software-Generierung. Siehe Abschnitt 2.7.3 u. ff.) der SPLE berücksichtigt (vgl. FeatureIDE-METOP, 2020). Ferner gilt FeatureIDE nach Gerspacher & Weber (2013) als stabil und „hervorragend".
FeatureIDE ist auch mit dem bekannten und sehr verbreiteten Eclipse IDE sehr gut kompatibel. Deshalb wie FeatureIDE im Zusammenhang mit Ecplise für den weiteren Verlauf verwendet.

3.3.1 Problem Space in FeatureIDE

FeatureIDE ist dafür geschaffen, Produktlinien zu konzipieren und zu implementieren. Die Unterstützung der Domänenanalyse und -Implementierung

in FeatureIDE hilft dabei, eine domänenspezifische Abstraktion zu bilden (FeatureIDE-METOP, 2020).

a) Domänenanalyse

In der Domänenanalyse werden Variabilität und Gemeinsamkeiten einer Softwaresystem-Domäne erfasst und ein Feature-Modell erstellt und als Diagramm gespeichert (Feature-Diagramm). Das Feature-Modell (Siehe Abbildung 6 oben) kann graphisch konstruiert werden. Dabei besteht die Möglichkeit neue Features zu erstellen, bestehende zu modifizieren und oder zu entfernen (vgl. Thüm, et al., 2014).

Der integrierte graphische und textuelle Modelleditor sorgt für eine interaktive Gestaltung (Syntaxhervorhebung, Erkennung von Inkonsistenzen) des Feature-Diagramms (Thüm, et al., 2014). Die Namenkonvention der Features im Feature-Diagramm ist ähnlich der Namenkonvention für Java-Klassen. Darüber hinaus verfügt der Editor über ein Kontextmenü, um einem Feature zusätzliche Eigenschaft beizufügen. Ein Feature kann z.B. abstrakt oder konkret sein. Dies ist über das Kontextmenü einstellbar.

Der Constraint-Editor ist ebenso interaktiv (die Entscheidungspropagierung wird unterstützt) und hilft bei der Validierung des Feature-Modells und einzelner Features durch Einfügen von Einschränkungen mit Hilfe von logischen Ausdrücken. Anomalien wie tote und falsch-optionale Features werden automatisch erkannt und angezeigt (Thüm, et al., 2014).

b) Domänenimplementierung

In dieser Phase unterstützt FeatureIDE die gleichzeitige Implementierung aller Software-Systeme der Domäne sowie die Zuordnung der Code-Assets zu den Features. FeatureIDE unterstützt verschiedene Kompositionswerkzeuge (Siehe Abschnitt 3.3.3) (Thüm, et al., 2014).

3.3.2 Solution space in FeatureIDE

Neben dem Problemraum (Problem Space) wird in FeatureIDE auch der Lösungsraum (Solution Space) unterstützt, um eine implementierungsorientierte Abstraktion zu schaffen (vgl. FeatureIDE-METOP, 2020).

a. Anforderungsanalyse

Diese Phase wird durch einen Konfigurationseditor umgesetzt und unterstützt. Nachdem das Feature-Modell aus der Domänenanalyse entstanden ist, können Feature über den Konfigurationseditor ausgewählt werden. Mit dem Editor können mehrere selbst definierte Konfigurationen erstellt und miteinander verglichen werden. Features, die als obligatorisch markiert sind, werden automatisch selektiert. Darüber hinaus werden Konfigurationen automatisch auf ihre Gültigkeit geprüft, in dem die erstellten Constraints angewendet werden. Obsolete Features werden auch erkannt (Thüm, et al., 2014).

b. Generierung

Unterschiedliche Kompositionswerkzeuge können integriert werden, um Quelldateien für die Kompilierung zusammenzufügen. Der Build-Prozess kann manuell oder automatisch ausgelöst werden (Thüm, et al., 2014). In den Source-dateien des Projekts werden alle Feature-Dateien automatisch generiert, sobald das Feature-Modell fertiggestellt und gespeichert ist.

3.3.3 Kompositionswerkzeuge in FeatureIDE

FeatureIDE unterstützt die Integration von vielen Kompositionswerkzeugen, die auch die SPLE unterstützt. Die verschiedenen Kompositionswerkzeuge können in vier Gruppen eingeteilt werden:

- Gruppe 1: Die Gruppe FeatureHouse (Lengauer, Apel, & Kastner, 2009), AHEAD (Gerspacher & Weber, 2013), FeatureC++ (Rosenmüller, et al., 2005) für die feature-orientierte Programmierung.
- Gruppe 2: Die Gruppe Munge und Antenna für annotationsbasierte Implementierung mit Präprozessoren (Thüm, et al., 2014).
- Gruppe 3: Die Gruppe DeltaJ für die delta-orientierte Programmierung (Strocco, et al., 2010).
- Gruppe 4: Die Gruppe AspectJ (Siehe Abschnitt 3.3.4) für die Aspekt-orientierte Programmierung.

Auswahl des Kompositionswerkzeugs
Da die „Pizza-Pronto" Anwendung in Java geschrieben wurde, wird eins der javabasierten Kompositionswerkzeuge ausgewählt. Es handelt sich um die Gruppen 2, 3 und 4 sowie FeatureHouse in der Gruppe 1.

Featurehouse ist ein Framework für die Softwarekomposition (Aktuell werden Java 1.5, C#, C, Haskell, JavaCC, Alloy, und UML als Kompositionssprache unterstützt) auf der Grundlage von Überlagerungen, zu dem nach Bedarf neue Sprachen hinzugefügt werden können (Lengauer, Apel, & Kastner, 2009). FeatureHouse ist eher geeignet mehrere Sprachen integrieren. Die „Pizza-Pronto" Anwendung besteht nur aus einer Sprache. Deshalb wird featureHouse aussortiert und eine einfache Variante gesucht.

Die übrigen stellen die Wahl zwischen Präprozessoren (bedingte Kompilierung mit Präprozessormakros wie z. B. ***#ifdef*** und ***#endif*** des C-Präprozessors CPP, vgl. Thüm, et al., 2014) und Nicht-Präprozessoren-Nutzung. Die Entscheidung fällt auf die Nicht-Präprozessoren-Gruppe aus Gründen des Komforts bei der Benutzung. Dies führt auf die Wahl zwischen AspectJ und DeltaJ.
Da über AspectJ mehr Dokumentation vorhanden sind, wird dieses endgültiges Kompositionswerkzeug ausgewählt. Wie die anderen Kompositionswerkzeuge eingesetzt werden, wird auf die jeweiligen Untersuchung der Autoren verwiesen.

3.3.4 AspectJ: Kompositionswerkzeug in FeatureIDE

In diesem Abschnitt wird ein kleiner Einblick in die AspectJ-Welt gegeben, welcher für die spätere Implementierung ausreichend ist. So dient dieser Abschnitt nur zum Verständnis der späteren Benutzung von AspectJ im Kapitel 5. Deshalb wird auf das Dokumentationshandbuch für tiefere Einblicke verwiesen (The AspectJTM Programming Guide - Xerox Corporation, 2003).

Mit dem AspectJ-Entwicklungstool (AJDT – AspectJ Development Tool), das in der Eclipse Entwicklungsumgebung integrierbar ist, kann die aspektorientierte Programmierung (AOP) mit AspectJ betrieben werden. AspectJ ist weit verbreitet und wurde von der Firma „Xerox PARC" entwickelt, um aspektorientierte Sprache für die Java-Plattform umzusetzen (Kanis, 2008). Die Sprache zielt darauf ab, übergreifende Belange (cross-cutting concerns) zu modularisieren, indem ein

replizierter, verstreuter und verworrener Code in Aspekte gekapselt wird (Batory, Apel, & Kästner, 2007).

Mit AspectJ kann eine Implementierung von Aspekt-orientierter Programmierung für Java realisiert werden, denn das AspectJ-Metamodell erweitert das Java-Metamodell, wie es auf der Abbildung 9 aufgezeigt wird. Darüber hinaus zeigen Lopez-Herrejon & Batory (2002) in ihrer Untersuchung anhand eines Beispiels, wie SPL mit AspectJ entwickelt werden kann.

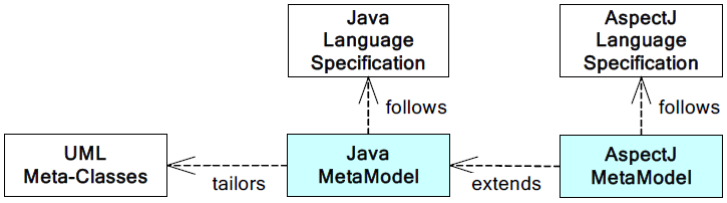

Abbildung 9: Das AspectJ-Metamodell erweitert das Java-Metamodel (Han, Kniesel, & Cremers, 2004)

Mit AspectJ können neue Funktionalitäten einem Programm hinzugefügt werden. Dafür werden Kontextübergaben nötig. Die übergreifende Struktur (cross-cutting concerns) der Kontextübergabe stellt eine erhebliche Quelle der Komplexität in Java-Programmen dar. Mit AspectJ lässt sich diese Art der Kontextübergabe auf modulare Weise implementieren und dies bringt einige Vorteile mit sich (The AspectJTM Programming Guide - Xerox Corporation, 2003):

- Die Struktur des übergreifenden Anliegens wird explizit anhand des Schlüsselwort *pointcut* deklariert und festgehalten.

- Die Modifizierbarkeit wird erhöht: Bei Änderungsbedarf wird nur eine Stelle angefasst, nämlich die Datei, in das Querschnittsanliegen deklariert wurde.

- Die Funktionalität lässt sich leicht ein- und ausbauen: Wird ein Feature zu einem bestimmten Zeitpunkt nicht mehr benötigt, kann es einfach ausgeblendet und bei Bedarf wieder eingeblendet werden.

- Die gesamte Implementierung wird stabiler sein: Die Verwendung von AspectJ führt zu sauberen, gut modularisierten Implementierungen von übergreifenden Belangen, welche explizit und einfach zu verstehen sind.

a) Einige Schlüsselworte in AspectJ

Mit nur wenigen neuen Konstrukten bietet AspectJ Unterstützung für die modulare Implementierung einer Bandbreite von Crosscutting Concerns.

- Join Point

Ein *Join Point* repräsentiert einen wohldefinierten Punkt im Programmfluss, an dem ein neuer Code (Aspekt) eingefügt werden kann. *Join Points* kommen unterschiedlich zum Einsatz. Sie können Methoden- oder Konstruktorausführungen, Attributzugriffe, Objekt- und Klasseninitialisierungen, Loop- oder If-Anweisungen (Kanis, 2008) sein.

- Pointcuts

Ein *Pointcut* stellt eine Menge von *Join Points* dar und dient dazu bestimmte Joint Points zu selektieren, an welchen ein Querschnittsverhalten ausgeführt werden soll. Die Signatur eines *Pointcuts* besteht aus dem Schlüsselwort *pointcut*, dem Bezeichner, ggf. einem Parameter(-Liste) und einer Pointcut-Expression. Pointcut-Expressionen können aus mehreren Ausdrücken bestehen, die mit **„&&"-** oder **„||"**- Verknüpfung miteinander verbunden sind. Die Signatur eines Pointcuts enthält den Schlüsselwort *pointcut*, den Pointcut-Bezeichner mit der dazugehörigen Parameterliste sowie den Pointcut-Ausdruck. Sie sieht wie folgt aus:
pointcut pointcutBezeichner (parameter): pointcutExpression;
Der Einsatz von *Pointcuts* ist vielfältig. Die typischen Verwendungeg sind (Kanis, 2008):

Call (Signatur):
Der Call-Joinpoint geschieht beim Aufruf einer Methode und wählt die Methode nach ihrer Signatur aus.

Execution (Signatur):
Der wesentliche Unterschied zu Call-Pointcut besteht darin, dass sich der *Join Point* beim *Call* vor der Ausführung einer Methode befindet, bei der *Execution* befindet er sich nach der Ausführung.

Get (Signatur) / set (Signatur):

Hier werden *Join Points* selektiert, die einen lesenden/schreibenden Zugriff auf Variablen ermöglichen.

Handler (TypPattern):
Mit "TypePattern" kann definiert werden, welcher Typ von Ausnahmen behandelt wird. Nur Exception-Handler, die diesen Typ behandeln, werden ausgewählt.

- Advice

Ein *Advice* kombiniert ein *Pointcut* und ein auszuführendes Code-Ausschnitt, um ein übergreifendes Verhalten zu implementieren. Die Signatur eines *Advice*s sieht wie folgt aus:

adviceSpezifikation [throws TypList]: Pointcut { Body }

Der *Body* repräsentiert einen normalen Java-Code und *Pointcut* bezeichnet einen zuvor definierten *Pointcut*. Unter „*adviceSpezifikation*" werden die Bedingungen definiert, unter welchen den Programmcode ausgeführt werden soll. Eine „adviceSpezifikation" kann eine der folgenden Ausdrücke sein: before-, around- und after-Advice.

Before() :
Die Ausführung geschieht unmittelbar vor dem *Join Point*.

after () [returning | throwing]:
Dieser Advice wird ausgeführt, nachdem der *Join Point* im Programmfluss erreicht wurde, genau direkt nach dem Methodenrumpf.

around () [returning | throwing]:
Der *Around advice* wird ausgeführt, wenn der *Join Point* erreicht wird, und hat explizite Kontrolle darüber, ob das Programm mit dem *Join Point* fortfährt (The AspectJTM Programming Guide - Xerox Corporation, 2003).
Die bereit vorgestellten *advices* eignen sich sehr gut für die Integration von neuen Features in ein Programm.

- Inter-type declarations

„Inter-type declarations" sind Konstrukte, die sich über Klassen und deren Hierarchien erstrecken. Sie können Member deklarieren, die sich über mehrere Klassen erstrecken, oder die Vererbungsbeziehung zwischen Klassen ändern. Im

Gegensatz zu *advice*, das hauptsächlich dynamisch arbeitet, arbeiten Inter-type declarations statisch, zur Kompilierzeit.

- Aspects

Aspects können wie eine Java-Klasse mit dem Schlüsselwort *aspect* (Bei Java-Klassen heißt der Schlüsselwort „class") definiert werden und bestehen aus Methoden, Variablen, Konstruktoren. Hinzu kommen Aspekt-Member, um Querschnittsanliegen über Klassenhierarchien hinweg darzustellen (The AspectJTM Programming Guide - Xerox Corporation, 2003). Zu einem *Aspect* gehören *poincuts*, *advice* und *inter-type declation*, die eine modulare Einheit der übergreifenden Implementierung bilden
Über AspectJ Weitere *Pointcut* sind ausführlich in der Dokumentation aufgelistet (The AspectJTM Programming Guide - Xerox Corporation, 2003).

b) Kompilierungstechniken in AspectJ

Um die auf die Quelltext-Ebene getrennten Implementierung von „Crosscutting Concern" wieder zu einem Programm mit seinem neuen Feature zusammenzufügen (Siehe Abbildung 10), wird in AspectJ eine Strategie namens „Bytecode Weaving" (das Weben der Bytecodes nach der Kompilierung) verwendet (Kanis, 2008). Es wird zwischen drei verschieden Varianten unterschieden: „Compile-Time Weaving", „Post-Compile Weaving" und „Load-Time Weaving".

- Compile-Time Weaving

Angenomen, es gibt zwei Quellcodes: ein Aspect-Code und ein Code, in dem der Aspect verwendet wird. Im „Compile-Time Weaving" kompiliert der AspectJ-Compiler aus dem Quellcode und erzeugt eine gewebte Klassendatei als Ausgabe. Anschließend wird bei der Ausführung des Codes, welcher den Aspect verwendet, die Ausgabeklasse des Weaving-Prozesses als normale Java-Klasse in die JVM geladen (Baeldung, 2023).

- Post-Compile Weaving

Post-Compile Weaving (manchmal auch Binary Weaving genannt) wird verwendet, um bestehende Klassendateien und JAR-Dateien zu weben. Wie beim Weaving zur Kompilierungszeit können die für das Weaving verwendeten Aspekte in Quell- oder Binärform vorliegen und selbst durch Aspekte gewebt werden (Baeldung, 2023).

- Load-Time Weaving

Das Weaving zur Ladezeit ist einfach ein binäres Weaving, das bis zu dem Punkt aufgeschoben wird, an dem ein Klassenloader eine Klassendatei lädt und die Klasse in der JVM definiert. Um dies zu unterstützen, werden ein oder mehrere "Weaving Class Loader" benötigt. Diese werden entweder explizit von der Laufzeitumgebung bereitgestellt oder durch einen "Weaving Agent" aktiviert (Baeldung, 2023).

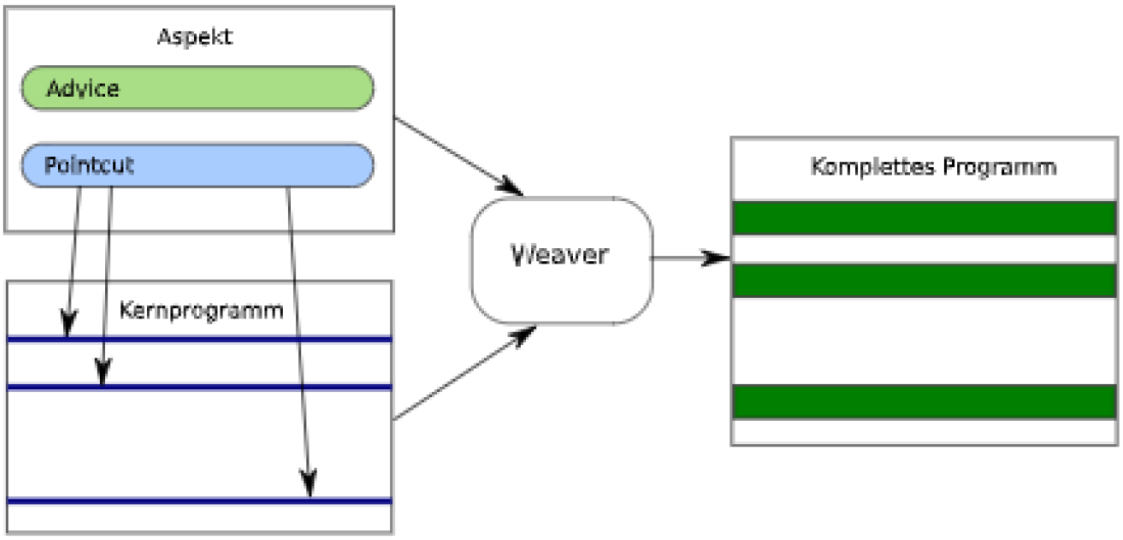

Abbildung 10: Der AspectJ Weaver ("Weber") (Kanis, 2008)

3.3.5 Besonderheit der Codegenerierung in FeatureIDE

Das Erstellen einer Anwendung in AspectJ erfordert mindestens zwei Schritte. Erstens wird die in Java implementierte „Pizza-Pronto" Anwendung benötigt. Zweitens werden die Implementierungen der Querschnittsbelange (cross-cutting), die in Aspekt-Dateien gespeichert werden, zur „Pizza-Pronto" Anwendung hinzugefügt. Dies führt dazu, dass neue Features in die „Pizza-Pronto" Anwendung generiert werden. Das heißt, dass das entstandene Ausgabeprodukt wird nicht in eine separate Datei erzeugt.

Um die im Unterkapitel 3.2 (Anforderungen) eingeführten Entwurf-Entscheidungen zu respektieren, die in der eingelieferten „Pizza-Pronto" Anwendung getroffen wurden, soll eine Alternative zur Generierung in mehreren Dateien gefunden werden. So kann z.B. die Übersichtlichkeit und die Modularisierung nach der Generierung bewahrt werden.

3.4 Externer Generator

Um dem bereits vorgestellten Problem der Generierung in der Pizza-Pronto-Anwendung zuvorzukommen, wird ein externer Generator benötigt. Mit der Anbindung eines externen Generators können generierte Produkte in separaten Dateien geschrieben werden.

Auf dem Markt gibt es zahlreiche Varianten von kommerziellen Angeboten bis zur Open-Source Varianten. Im Folgenden werden nur zwei Generatoren „FreeMarker" und „JavaParser" vorgestellt, die die Aufgaben erfüllen. Andere Generatoren wie JavaPoet (Square Open-Source, 2020), Picocog (Ainsley, 2020), Spoon (Pawlak, et al., 2015) oder JavaForger (Heuvel, 2022) könnten genauso angebunden werden.

3.4.1 Apache FreeMarker

a) Definition

FreeMarker ist ein Template-orientierter Generator, der es ermöglicht komplette Anwendungen zu generieren und kann einfach in Build-Werkzeuge oder Konfigurationsdateien oder als JAR-Datei in einem Projekt angebunden werden. Eine Template-Engine kombiniert statische Daten mit dynamischen Daten, um Inhalte zu erzeugen.
Ein Template lässt sich mit einer Vorlage oder Schablone vergleichen, die eine Zwischendarstellung des zu erzeugenden Inhalts aufzeigt. Mit der Schablone wird die Struktur der Ausgabe entworfen.

Die Template-Dateien werden in FreeMarker Template Language (FTL) geschrieben. Die Sprache enthält Konstrukte wie FreeMarker-Interpolationen, welche als Platzhalter für die dynamischen Inhalte darstelle (Apache Software Foundation, 2015).

b) Struktur des Templates

Ein Template ist ein Programm, das in FreeMarker Template Language (FTL) geschrieben ist. Ein FTL-Programm enthält folgende Komponenten:

- Texte: Sie werden unverändert auf die Ausgabe übertragen.

- Interpolationen: Diese Komponenten werden in der Ausgabe durch einen berechneten Wert ersetzt. Interpolationen werden durch *${* und *}* abgegrenzt.
- FTL-Tags: FTL-Tags sind ähnlich wie HTML-Tags und sind Anweisungen an FreeMarker. Sie werden nicht in die Ausgabe weitergeleitet.
- Kommentare: Wie bei HTML-Kommentare werden sie ignoriert und nicht in die Ausgabe geschrieben. In FreeMarker aber werden sie durch *<#--* und *-->* getrennt.

Weitere Direktive und zahlreiche Ausdrückmöglichkeiten sind in der Apache FreeMarker Dokumentation ausführlich erklärt (Apache Software Foundation, 2015).

c) Funktionsweise

Im ersten Schritt der Codeerzeugung wird eine FreeMarker-Konfiguration-Instanz erstellt und passend eingestellt. Die Konfiguration-Instanz dient nicht nur als zentraler Ort zum Speichern der Einstellungen auf Anwendungsebene, sondern auch für die Erstellung und das Zwischenspeichern von vorgefertigten Templates. Das Datenmodell wird einschließend erstellt.
Es handelt sich um einen Container (z.B. Java-Klasse) zur Datenhalterung. Danach wird ein vorgefertigtes Template von der Konfiguration gelesen, geparst und instanziiert. Einschließend folgt die Verschmelzung des Datenmodells mit dem Template Apache FreeMarker um die gewünschte Ausgabe zu erzeugen.

Der Apache FreeMarker Konzept ähnelt sich dem Prinzip des Entwurfsmuster MVC (Model-View-Controller), was die Trennung der Informationsdarstellung und -Bereitstellung angeht. Das Template (die View in MVC) ist für die Darstellung der Daten zuständig, während das Datenmodell (Das Modell in MVC) die Daten bereitstellt. Der Apache FreeMarker verbindet das Ganze für die Ausgabe (Apache Software Foundation, 2015). Die Abbildung 11 veranschaulicht die Funktionsweise von FreeMarker.

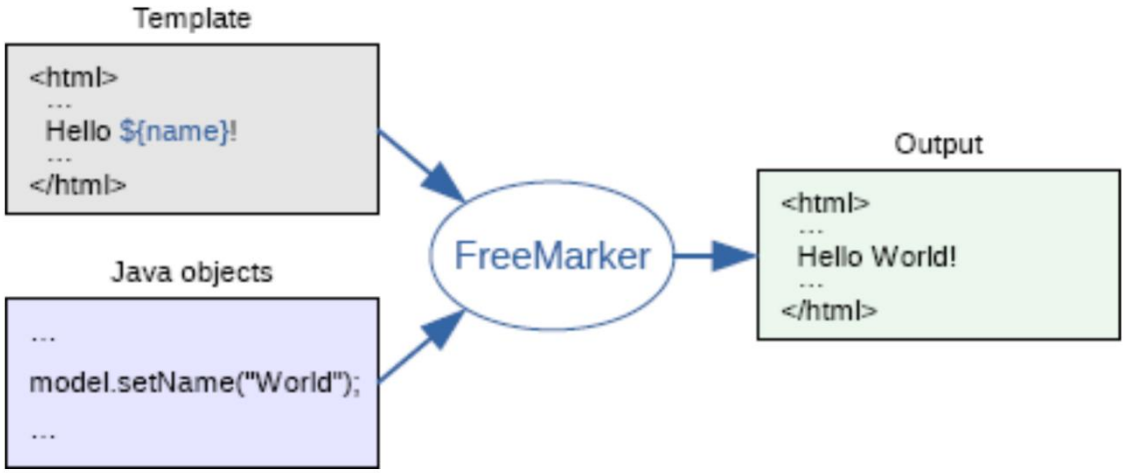

Abbildung 11: Funktionsweise von FreeMarker (Apache Software Foundation, 2015)

3.4.2 JavaParser

JavaParser ist ein weit verbreitete Open-Source Parser für die Java-Sprache. Im Jahr 2008 entstand die erste Version, die stets weiterentwickelt wurde. Inzwischen gibt es mehr als fünfzig Bibliotheken, die von JavaParser abhängig sind. Hunderte von Projekten integrieren JavaParser, darunter auch kommerzielle Projekte.

Der Zweck von JavaParser limitiert sich nicht nur auf das Parsen, womit eine Repräsentation des Codes eines Java-Objekt erstellt werden kann. Das Ergebnis des Parsen kann für weitere Analysen und Transformationen benutzt werden. Nach jedem dieser Schritte können Quellcodes „im Handumdrehen" generiert werden.

JavaParser im Zusammenspiel mit AspectJ kann gut mit Apache FreeMarker kombiniert werden, um die *Anforderungen A-04* besser zu erfüllen. Mit AspectJ können gewünschte Stellen in der „Pizza-Pronto" Anwendung gekennzeichnet werden. Danach kann mit JavaParser die „Pizza-Pronto" Anwendung analysiert, geeignete Transformationen vorgenommen und anschließend in gewünschtem Format und in der richtigen Anordnung in einem Verzeichnis mit Apache FreeMarker-Template generiert werden.

3.4.3 Zusammenspiel der externen Generatoren mit FeatureIDE

FeatureIDE hat die Aufgabe, die Aspekt-Klassen durch den integrierten Generator zu erzeugen, welcher nicht von außen beeinflusst werden kann.
Nachdem die Anwendung zur Ausführung gebracht wird, werden alle in der Konfiguration ausgewählten Features aktiviert.
Die zuständigen Klassen der „Pizza-Pronto" Anwendung werden zu einem gegebenen Zeitpunkt (durch AspectJ-Methoden definiert) geladen.
Diese werden durch die JavaParser-Bibliothek in einen AST transformiert und entsprechend verändert.
Um die Klassen wieder in neu generierten Dateien auszugeben, wird das FreeMarker-Framework eingesetzt.

4 Konzeption

Unter der Berücksichtigung der im vorangegangenen Kapitel erzielten Ergebnisse, wird innerhalb dieses Kapitels das Konzept der zu implementierenden Anwendung errichtet.
Zunächst wird nach dem Konzept von MDSD ein Modell für die Produktlinien-Anwendung erstellt. Nachdem die Entstehung des Modells beschrieben wird, werden die wichtigsten Komponenten und ihre Beziehung zueinander aufgelistet. Danach wird jede Komponente detailliert vorgestellt. Darüber hinaus wird das Entwurfsmuster, dem die Umsetzung am ehesten entspricht, präsentiert.

4.1 Vom Modell zu Features: Erstellung des Feature-Modells

Zu Beginn wird die Erstellung des Feature-Modells benötigt. Dies geschieht nach dem Prinzip des Bottom-Up-Entwicklung. Die eingelieferte „Pizza-Pronto" Anwendung wird gründlich analysiert, um alle relevanten Features herauszuarbeiten. Die Art der Features variiert. Es gibt ganze Klassen, Methoden und Methodenaufrufe, Klassenattribute, Assoziationen sowie diverse Veränderungen in den Komponenten.
In der Abbildung 12 wird ein Ausschnitt des Feature-Modells als Ergebnis der Erfassten Features aufgezeigt.

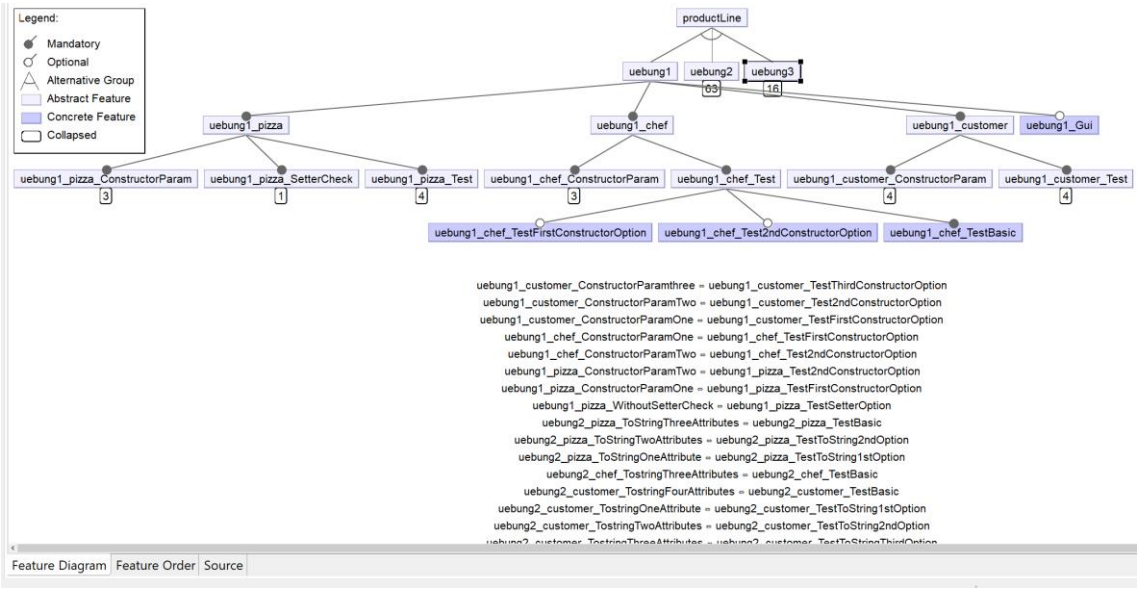

Abbildung 12: Das Feature-Modell als Grundlage für die Implementierung im Editor.

4.1.1 Die M-2-C-Transformation und das Paketdiagramm

Die Legende des Feature-Diagramms in der Abbildung 12 zeigt unterschiedliche Parameter im Modell auf. Zum einem wird zwischen abstrakte (z.B. *Uebung1_chef_Test*) und konkrete (z.B. *Uebung1_chef_TestFirstConstructorOption*) Features unterschieden. Zum anderen werden Features in obligatorische, optionale oder alternative Gruppen unterteilt. Zusammengehörige und logisch sinnvolle Features werden in Feature-Gruppen ausgedrückt.

Die M-2-C-Transformation von FeatureIDE

In FeatureIDE wird aus dem erstellten Modell zum einem die Codes (Nur die Klassen-Köpfe als AspectJ-Code) aller Features in ihre zugehörigen Pakete automatisch generiert. Zum anderen wird eine Grundkonfiguration automatisch generiert. Diese Konfiguration wird dann repliziert und gemäß den Anforderungen verändert. Es handelt sich jeweils um ein Beispiel für die M-2-C-Transformation. Diese Aufgabe wird vom integrierten Generator in FeatureIDE übernommen.

Das Paketdiagramm

Hierbei werden nur konkrete Features generiert. Es wird z.B. Für das Feature „Uebung1_chef_TestFirstConstructorOption" eine leere Aspekt-Klasse „*TestFirstConstructorOption*" in das Paket „Uebung1" und in Unterpaket „chef" generiert.

Analog werden alle erarbeiteten Features aus dem Feature-Modell (Siehe Abbildung 12) in ihre jeweiligen Pakete (Uebung1, Uebung2 und Uebung3) bzw. Unterpakete generiert. Auf dieser Weise wird ein Teil des Paketdiagramms erstellt. Die Abbildung 13 stellt die Übersicht über alle Pakete und ihre Abhängigkeit zueinander als Paketdiagramm dar. Die wichtigsten sind:

- Das Paket „Feature-Auswahl"

Das Paket „Feature-Auswahl" wird aus dem Feature-Modell automatisch generiert und fasst alle ermittelten Features zusammen. Die anderen Pakete werden manuell erstellt, um die Komponenten sinnvoll zu gruppieren.

- Das Paket „JavaParser"

Die JavaParser Bibliothek kann genutzt werden, um eine beliebige Java-Klasse als Abstrakt-Syntax-Baum darzustellen.

- Die Pakete „AST-Handler" und „Transformation"

In diesen Paketen befinden sich Komponenten für die Definition und Implementierungen der Schnittstelle zur Modifizierung des Abstrakt-Syntax-Baumes der Pizza-Pronto-Anwendung.

- Das Paket „Parser"

Die Komponenten in diesem Paket werden eingesetzt, um eine als AST-dargestellte Klasse in eine normale Java-Klassen zu transformieren.

- Das Paket „Template"

Mithilfe der FreeMarker Bibliothek werden Komponenten dieses Pakets eingesetzt, um die modifizierten Java-Klassen zu formatieren und in neue Dateien zu generieren.

- Das Paket „DirExplorator"

Das Paket beinhaltet Komponenten, die für die Aufbereitung der Dateien sowohl beim Lesen als auch beim Schreiben zuständig sind.

- Das Paket „Generator"

Im „Generator"-Paket befinden sich Komponenten, die für die Aufbereitung der Features, die Transformation des Abstrakt-Syntax-Baums der Pizza-Pronto-Anwendung und die Generierung des Quellcodes zuständig sind. Zu diesem

Zweck beinhaltet das „Generator"-Paket weiteren Pakete. Die wichtigsten davon sind: „Controller" „Visitor" und „Konfigurator".

Die Komponenten der beschriebenen Pakete werden im Unterkapitell 4.2 näher betrachtet.

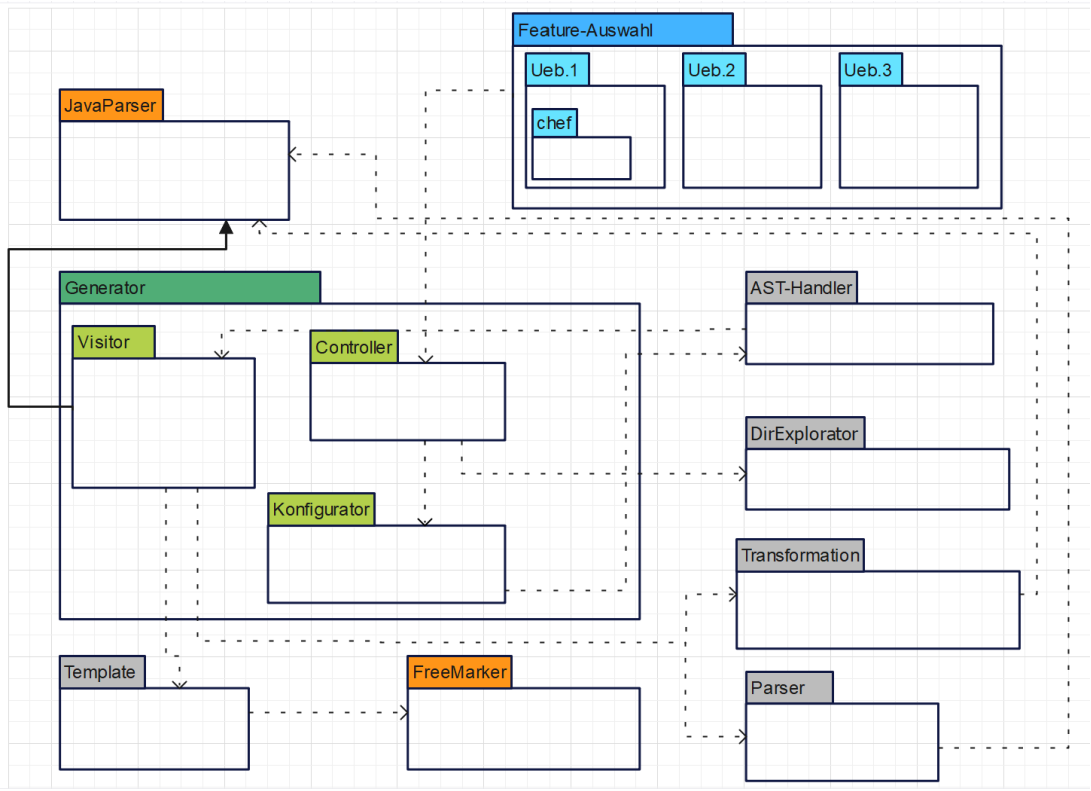

Abbildung 13: Das Paketdiagramm

4.1.2 Constraints

Die verknüpften Ausdrücke unter dem Feature-Diagramm (Siehe Abbildung 12, oben) repräsentieren Constraints. Sie bringen zusätzlich Informationen zwischen den Features(-Gruppe) entlang der Feature-Syntax-Tree (FST). Es handelt sich um Restriktionen oder Abhängigkeiten.
Bei der Erstellung eines Constraints im Contraints-Editor (Siehe Abbildung 14) wird der ganze FST automatisch auf die logische Korrektheit geprüft. Darüber hinaus werden mit Hilfe von Constraint im Zusammenspiel der logisch verknüpften Features(-Gruppen) geprüft, ob eine Konfiguration im Konfigurationseditor valide ist.

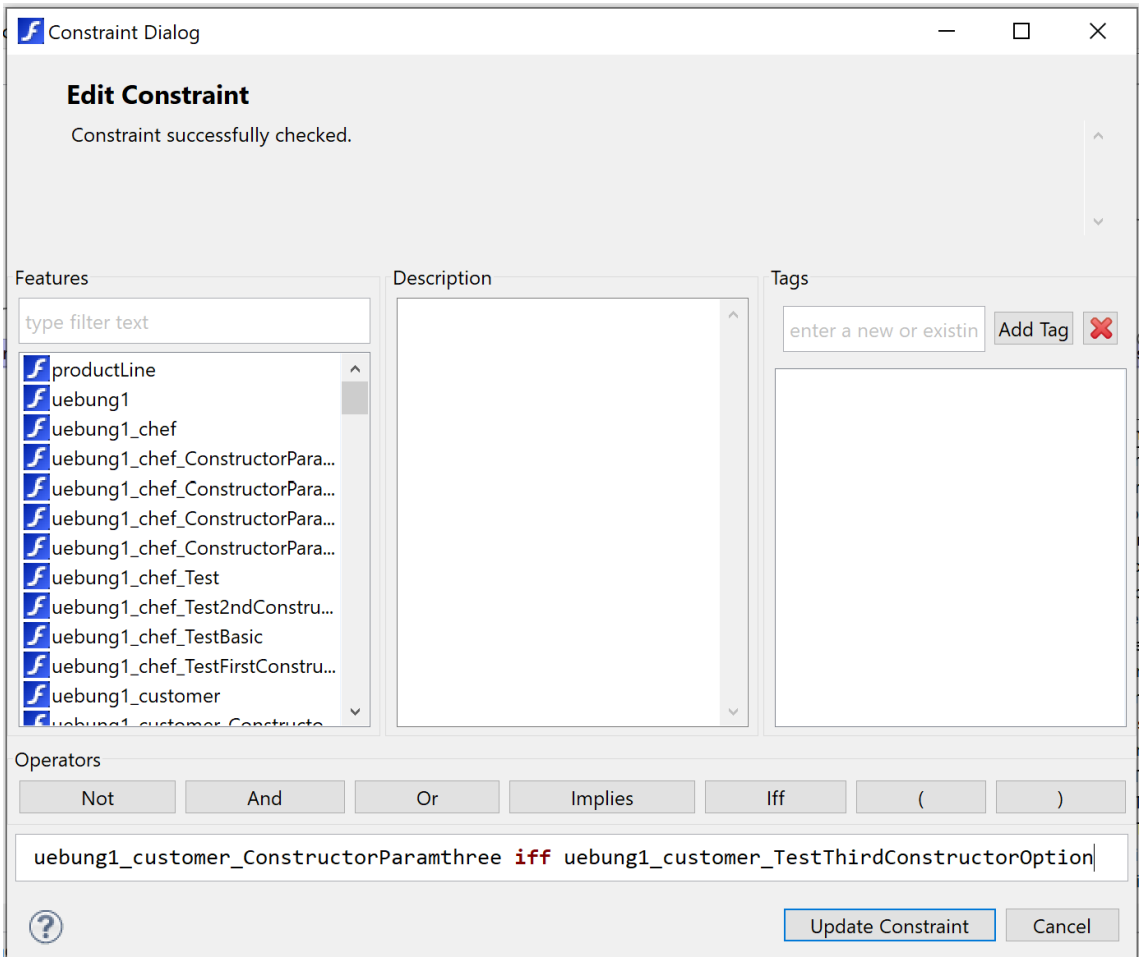

Abbildung 14: Constraint-Editor für zusätzliche Informationen und Prüfungen

4.2 Komponenten-Überblick

Die wichtigsten Funktionalitäten der Anwendung werden sein: Die Feature-Auswahl in der Konfigurationseditor und die Generierung des Programm-Codes. Daraus ergibt sich folgende Komponenten: Konfiguration, Konfiguration-Handler, Generator, Template-Handler und Verzeichnis-Explorator.

4.2.1 Grober Ablaufplan der Produktlinie-Anwendung

Der Programmablauf wird wie folgt aussehen: Nach der Ausführung der Anwendung wird die selektierte Konfiguration mitsamt den entsprechenden Komponenten der „Pizza-Pronto" Anwendung zur Weiterverarbeitung an den Generator übergeben. Durch den Generator-Kontroller werden die zuständigen Konfigurationen aufgerufen.

In der Konfigurator-Komponente werden die nötigen Veränderungen vorgenommen. Dazu wird die Visitor-Komponente durch die Benutzung der Schnittstelle „TransformationHandler" zur Hilfe genommen.
Die Visitor-Komponente selbst erweitert die VisitorAdapter-Komponente von JavaParser. Jede Java-Klasse wird in JavaParser als AST dargestellt. Die Visitor-Komponente wird den AST durchlaufen, um einen neuen Knoten hinzufügen oder einen bestehen Knoten zu verändern.

Nach der Transformation wird mit Hilfe des Templates die ASTs in Java-Dateien zurücktransformiert und in die neu durch den Verzeichnis-Explorator erstellten Verzeichnisse geschrieben. Die Abbildung 15 stellt das Komponentendiagramm dar.

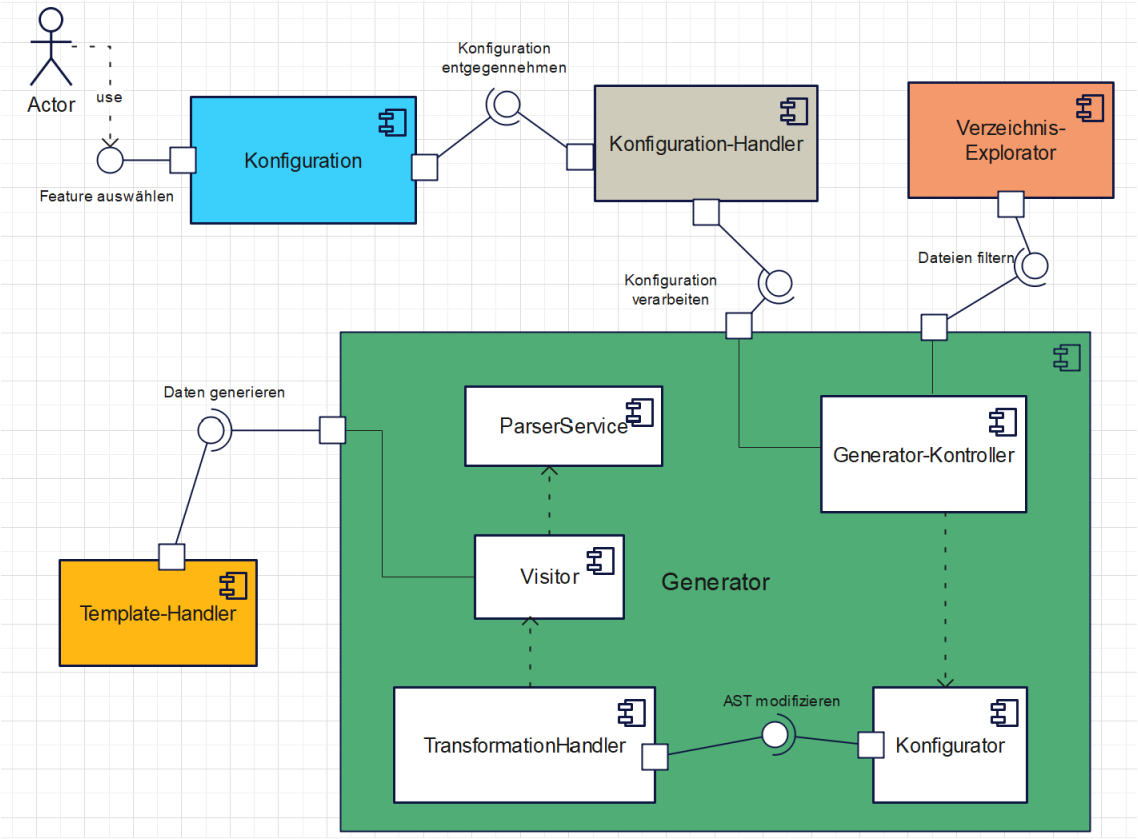

Abbildung 15: Komponentendiagramm

4.2.2 Einsatz der Fabrik-Methode

Die Produktlinien-Anwendung wird nach den Prinzipien der Fabrik-Methode entwickelt. Die Produktlinien-Anwendung entsteht dadurch, dass neue Features der Pizza-Pronto-Anwendung hinzugefügt werden.

Wie im vorangegangenen Abschnitt eingeführt, werden individuelle Transformationseinheiten benötigt. Die Fabrik-Methode wird dafür eingesetzt, konkrete und individuelle Transformationseinheiten für jedes Feature zu erstellen. Auf diese Weise werden neue Features erzeugt.

4.2.3 Die Stärke der Wiederverwendbarkeit in SPLE

Die Produktlinien-Anwendung besitzt drei Hauptfeatures, „uebung1", „uebung2" und „uebung3" (Siehe Abb. oben). Die Anwendung wurde so konzipiert, dass nur eines der drei Features gleichzeitig ausgewählt wird. Innerhalb eines selektierten Features können alle darunter hängenden Features ausgewählt werden, soweit die Constraints die ausgewählte Konstellation zulässt.

Ferner wird die Auswahlreihenfolge der Features sequenziell festgelegt, zuerst „uebung1" dann „uebung2" und zuletzt „uebung3". Folglich kann kein Code aus dem Feature „uebung2" generiert werden, solange der Code für „uebung1" noch nicht erzeugt wurde usw...

Darüber hinaus werden die Artefakte aus dem Feature „uebung1" in den anderen Features unter Berücksichtigung der Anforderungen im Unterkapitel 3.2 wiederverwendet. Der generierte Code aus der Feature-Gruppe „uebung1" wird als Kernbestand für die Feature-Gruppe „uebung2" verwendet usw.…

4.2.4 Die Konfigurationskomponente

Die Konfiguration (Siehe das blaue Viereck in der Abbildung 15) spielt eine zentrale Rolle, denn sie bietet die Möglichkeit Features gemäß der Anforderungen A-02 auszuwählen. Eine konsistente Konfiguration impliziert ein konsistentes Feature-Modell. Die Grundkonfiguration wird durch FeatureIDE automatisch generiert, nachdem das Feature-Modell gespeichert wurde. Weitere Konfigurationen können manuell erstellt werden. Es geschieht eine Transformation (M-2-C. Siehe Abschnitt 2.6.2), wobei der FST in FeatureIDE-XML-Datei übersetzt wird. So ist eine Konfigurationsdatei eine der vielen Abbildungen des FSTs.

Der XML-Datei ist der Konfigurationseditor als ein grafisches Userinterface zugeordnet. Der Editor kann auch als Fassade für die dahinterstehenden Aspekt-Codes und die dazugehörige Logik betrachtet werden. Ohne gültige Konfiguration kann kein Vorgang gestartet werden. Darüber hinaus ist der Editor intuitiv. Neben der Validierung erkennt der Editor bei der Auswahl, wieviele mögliche Konstellationen noch vorhanden sind und gibt Hilfestellung durch Verfärbungen (Siehe Abbildung 16).

Die Aspekt-Codes gehören zu der Konfiguration und sind eng mit den Features verbunden (Ein Feature = ein Aspekt-Code). Diese Codes enthalten AspectJ-spezifische Anweisungen, die kurz nach der Kompilierung aufgerufen, ausgeführt und in die „Pizza-Pronto" Anwendung integriert werden. Dazu werden Anhaltpunkte definiert, um dieser Vorgang auszulösen.

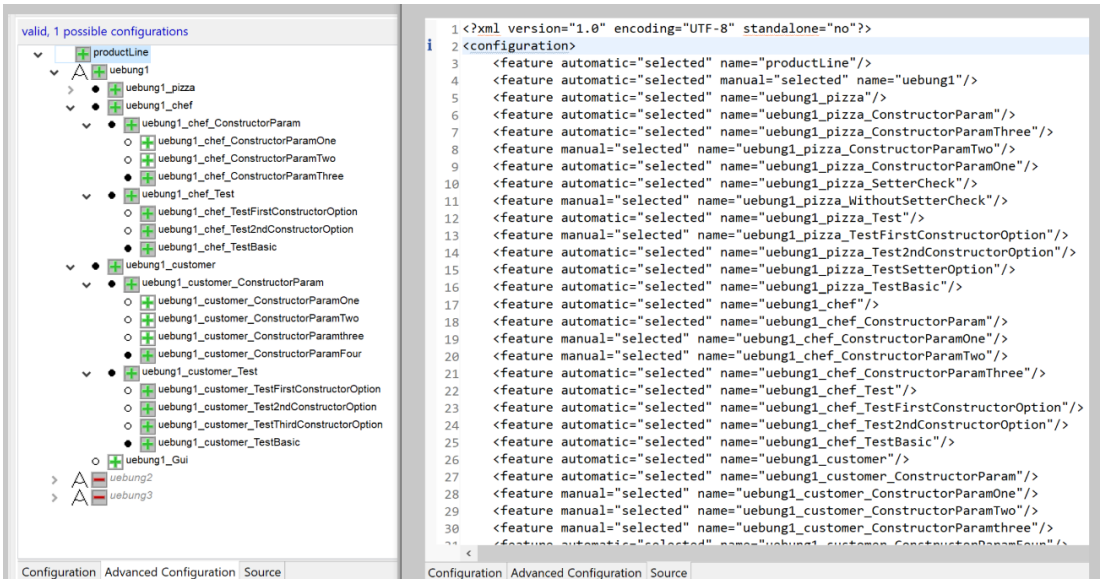

Abbildung 16: Ausschnitt aus der Konfiguration (Links der Editor, rechts die XML-Version)

4.2.5 Die Verarbeitung der Konfigurationsauswahl

Ab dieser Stelle gibt es keine Interaktion mehr über eine grafische User-Interface mit einem Editor (Das graue Viereck in der Abbildung 15). Hier werden die jeweiligen Features aus der Konfiguration entgegengenommen für die Weiterverarbeitung. Das heißt, die leeren generierten AspectJ-Klasse werden implementiert.
Allerdings findet die Verarbeitung an sich hier nicht statt, sondern sie wird an die nächste Komponente (Generator) delegiert. Genauer genommen stellt die

„Konfigurationshandler-Komponente" die Schnittstelle zwischen Generator und Konfiguration dar.

4.2.6 Die Generator-Komponenten der Anwendung

Der Generator fasst mehrere Komponenten zusammen: Der in der Abbildung 17) skizzierte „Generator-Kontroller", der „Konfigurator", der „Transformation-Handler", der „Visitor" und der „ParserService".

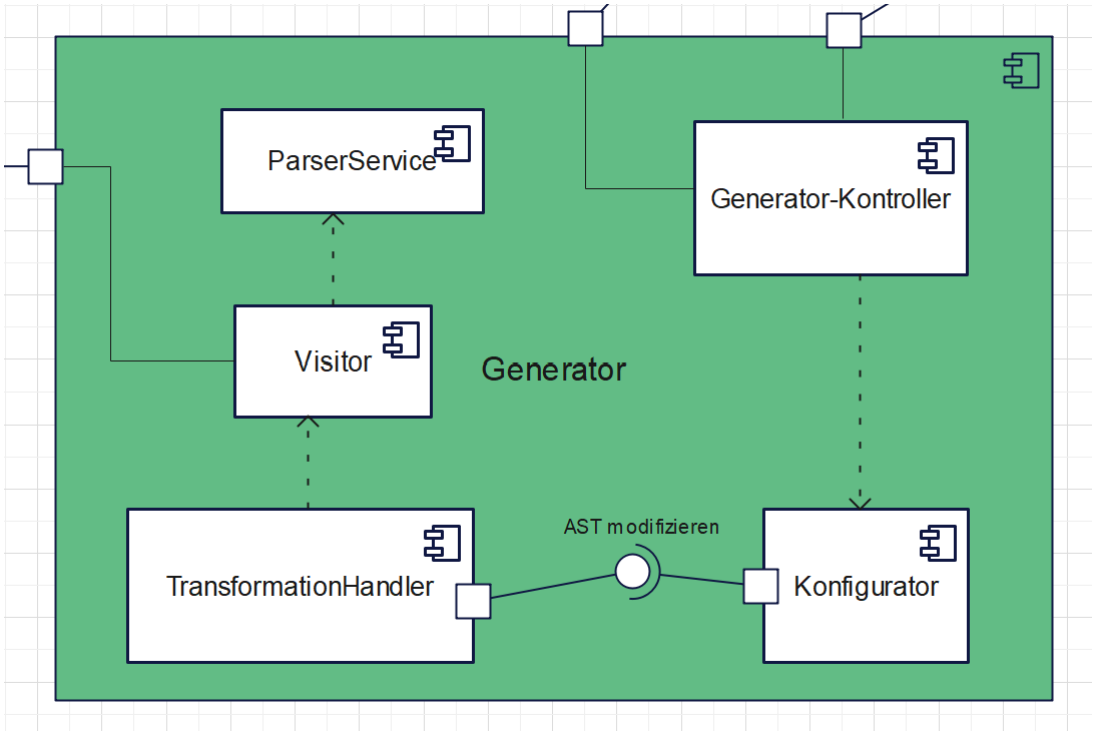

Abbildung 17: Die Generator-Komponenten der Anwendung (Ausschnitt aus der Abbildung 15)

a. Der Generator-Kontroller

Dem Kontroller wird der Pfad der zu modifizierenden „Pizza-Pronto" Anwendung (Ab der zweiten Generation, spricht „Uebung2", wird der Pfad des erzeugten Artefakts) übergeben. Nachdem die relevanten Dateien herausgefiltert wurden, werden die zuständigen Logiken in der Konfigurator-Komponente aufgerufen und angewendet. Der Kontroller übernimmt an dieser Stelle die Rolle eines „Aufgaben-Verteilers" (Siehe Abbildung 17).

b. Die Konfigurator-Komponente

Die Aufgabe dieser Komponente besteht darin, Feature-relevante Aspekte zusammenzusetzen und für die Transformation bereitzustellen. Da sich alle Feature-Konstellationen voneinander unterscheiden, werden separate Implementierungen benötigt. Deshalb ist die Konfigurator-Komponente eigentlich eine Sammlung vieler kleinerer Komponenten. Jede der Komponenten muss dann die Schnittstelle „Transformation-Händler" implementieren (Siehe Abbildung 17).

c. Der Transformation-Handler

Das von der vorherigen Komponente vorbereitete Paket wird an den Transformation-Handler zur Verarbeitung weitergegeben. Diese Komponente spielt die Rolle eines „Vermittlers". Deshalb findet die tatsächliche Transformation hier nicht statt, sondern dafür wird die Zuständige Transformationseinheit aufgerufen (Siehe Abbildung 17).

d. Die Visitor-Komponente der Anwendung

Der „Visitor" aus dem Paketdiagramm (Siehe Abbildung 17) erweitert den „VoidVisitorAdapter" aus der JavaParser-Bibliothek. und übernimmt die nötigen Transformationen in der „Pizza-Pronto" Anwendung (bzw. im erzeugten Artefakt), die für die Erstellung von Featrues notwendig sind. In der Visitor-Komponente wird jede Java-Klasse als „Compilation Unit" (Kompilierungseinheit) dargestellt.

Eine Kompilierungseinheit ist mit einem AST vergleichbar. Der Root-Knoten ist die Kompilierungseinheit selbst und die Blatt-Knoten sind beispielsweise die Packlagedeklaration, die Importstatements und die Klasse. Alle Member einer Klasse sind Blatt-Knoten dieser Klasse (Siehe Abbildung 18).

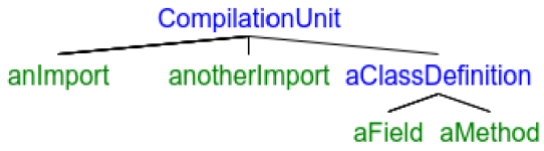

Abbildung 18: Kompilierungseinheit (Compilation Unit) als AST (Smith, van Bruggen, & Tomassetti, 2021)

Die Rolle des Visitors besteht darin, die Knoten der Kompilierungseinheit (AST) zu „besuchen" (durchlaufen) und je nach Anwendungsfall Veränderungen vorzunehmen. Angenommen ein „Refactoring" eines Attributs soll als neuer Feature in einer bestimmten Klasse hinzugefügt werden. Der Visitor wird in

diesem Fall die Kompilierungseinheit der Klasse holen, den AST besuchen und den entsprechenden Knoten des Attributs suchen. Falls der Knoten gefunden wird, wird das Attribut wie gewünscht verändert.

e. Der ParserService

Der Parser-Vorgang findet vor der Transformation statt. Dieser Schritt ist notwendig, denn der ParserService nimmt die zu modifizierenden Dateien entgegen und überführt diese in Kompilierungseinheiten (Siehe Abbildung 17).

4.2.7 Template-Handler

Anders als der ParserService hat das Template die Aufgabe die Kompilierungseinheiten wieder in normale Java-Klasse zurückzukonvertieren. Die neu erstellten Dateien werden in ihre jeweiligen Verzeichnisse geschrieben.
Zunächst wird mit Hilfe der Apache FreeMarker ein Template instanziiert. Ein Datenmodell wird erstellt und mit der Kompilierungseinheit befüllt. Gemäß der Formatierung im Template wird eine Java-Datei erzeugt (orangefarbenes Viereck, siehe Abbildung 15).

4.2.8 Verzeichniserzeuger

Der Verzeichnis-Explorator (hell-rotes Viereck, siehe Abbildung 15) wird bei der ersten Ausführung der Anwendung alle benötigen Verzeichnisse erzeugen. Ab der zweiten Ausführung werden bestehenden Verzeichnisse erhalten bleiben und wenn notwendig, werden neue Dateien hinzugefügt oder bestehende aktualisiert. Die Importstatements in den generierten Java-Dateien werden so konfiguriert, dass die erzeugten Dateien einfach in eine geeignete Umgebung importiert und sofort ausgeführt werden können.

4.3 Einsatz von S.O.I und Clean Code Prinzipien

Neben der Fabrik-Methode werden diese Prinzipien eingesetzt, um den Quellcode besser zu strukturieren, damit einen Mehrwert erschaffen werden kann.

4.3.1 Die S.O.I Prinzipien

Single-Responsibility-Prinzip

Mithilfe des Single-Responsibility-Prinzips wird sichergestellt, dass die Verarbeitung des Informationsflusses der einzelnen Konfigurationskonstellation von der Auswahl über die Generator-Kontroller bis zur Generierung für jede der drei Feature-Gruppen (Übung1, Übung2 und Übung3) getrennt. So wird beispielsweise einen Generator-Kontroller für jede der drei Hauptfeatures implementiert. Analog werden die bereits eingeführten Transformationseinheiten umgesetzt (Siehe Kapitel 5 für die konkrete Umsetzung).

Außerdem wird die Trennung der Verantwortung in der Implementierung jeder einzelnen Feature sichtbar. Für die Implementierung der einzelnen konkreten Features im Feature-Modell wird genau eine Komponente zuständig sein. Dies wird auch durch die automatische Generierung der konkreten Features (jede Feature in ihrer eigenen Datei) begünstigt.

Open/Close-Prinzip

Durch den Einsatz der Fabrik-Methode wird eine abstrakte Komponente sowie mindestens eine Schnittstelle entworfen (Siehe Abbildung 8, oben). Darüber hinaus werden einigen Schnittstelle integriert, wie das Komponentendiagramm aufzeigt (Siehe Abbildung 15, oben). Die Implementierung dieser Schnittstellen bzw. die Umsetzung der Abstraktion dienen als Grundlage dafür, dass Klassen für Erweiterungen offen und für Änderungen geschlossen sind.
In der Feature-Gruppe „Übung3" z.B. soll gemäß der Anforderung A-17 eine Assoziationsklasse erstellt werden. Die entsprechenden verarbeitenden Komponenten (z.B. eigene Generator-Kontroller) für die neue Feature können hinzugefügt werden, ohne die bestehenden zu beeinflussen.

Interface-Segregation-Prinzip

Die Schnittstellen der Produktlinien-Anwendung werden „schmal" gehalten. Sie werden mit einer verarbeitenden Methode ausgestattet. Auf dieser Weise kann das Interface-Segregation-Prinzip umgesetzt werden. Mit dieser Maßnahme wird sichergestellt, dass die implementierenden Komponenten genau eine bestimmte Aufgabe verarbeiten.
Im Falle, dass eine Komponente erweitert werden sollte, kann die Komponente eine neue Schnittstelle implementieren, statt eine bestehende durch eine weitere

Methode zu vergrößern. Auf dieser Weise wird das Open/Close-Prinzip nicht verletzt.

4.3.2 Einsatz von „Clean Code"

Aussagekräftige Namen

Eine der wichtigsten Bestandteile der Produktlinien-Anwendung besteht darin, Features aus einer Konfiguration zu auszuwählen. Die Grundkonfiguration wird automatisch aus dem Feature-Modell generiert. Aus diesem Grund ist es wichtig, eine aussagekräftige Namengebung bereits vor der Erstellung des Feature-Modells in der Domänenanalyse zu treffen. Eine sinnvolle Benennung der Entitäten (Komponente, Variables, etc…) wird sich wie „ein roter Faden" durch die Produktlinien-Anwendung ziehen.

Eine Abstraktionsebene pro Funktion

Dieses Prinzip wird überall eingesetzt, wo eine komplexe Berechnung stattfindet. Hierbei werden Teile der Berechnung in Hilfsklassen oder -methoden übergeben. Dies betrifft vor allem Komponenten, die die Transformation eines AST vornehmen (z.B. die Visitor-Komponente).

DRY

Die Delegierung bei komplexen Berechnungen in Verarbeitungseinheiten wie Hilfsmethoden haben einen weiteren Vorteil. Die Wiederverwendbarkeit wird erhört. Die bereits vorgestellte Anforderung A-05 soll innerhalb der Feature-Gruppe „Uebung1" umgesetzt werden. Hierfür wird eine Abstraktion (gleich ein Modell) gebildet, die vorgibt, wie einen Konstruktor erzeugt werden kann. Ferner kann dieselbe Abstraktion benützt werden, um (individuelle) Implementierungen innerhalb der Feature-Gruppe zu erzielen. Auf diese Weise werden sich wiederholende Codestücke vermieden.

Kommentar

Das Zusammenspiel der bisher beschrieben Maßnahmen werden dazu beitragen, dass die Umsetzung mit so viel Kommentaren wie nötig bewerkstelligt wird.

5 Umsetzung

Im nachfolgenden Kapitel wird das Vorgehen zur konkreten Umsetzung des im vorangegangenen Kapitel vorgestellten Konzeptes beschrieben. Hierbei wird die Umsetzung der verschiedenen Komponenten beleuchtet.

5.1 Entwicklungsumgebung

Die Entwicklung der Produktlinie-Anwendung wird in den Sprachen Java und AspectJ verfasst. AspectJ ist von Java abgeleitet und ist für die SPLE geeignet. Dazu wird „Eclipse IDE" als Entwicklungsumgebung verwendet. Die in den vorangegangenen Kapiteln vorgestellte FeatureIDE lässt sich gut in Eclipse integrieren.

FeatureIDE kann einfach auf dem Eclipse Marketplace herunterladen werden. Es wird empfohlen, die älteren Versionen von Eclipse für die Installation von FeatureIDE zu benützen In FeatureIDE wird mindestens Java 8 benötigt. Dazu ist die Eclipse Version 4.16 am besten geeignet (FeatureIDE Team, 2014).

Zur Versionierung wird Git verwendet, um Änderungen im Entwicklungsverlauf nachzuvollziehen. Zu jeder Zeit kann eine ältere Version der Anwendung hergestellt werden. Außerdem gibt Git bei der Entwicklung Sicherheit, denn der Code der Anwendung wird in einem GitHub Repository online repliziert. Falls der lokale Rechner ausfällt, ist auf dieser Weise die Ausfallsicherung gewährleistet.

Darüber hinaus werden zwei Bibliotheken als Abhängigkeit benötigt: JavaParser und Apache FreeMarker. Da FeatureIDE als Plugin in Eclipse heruntergeladen wird, lässt sich die Anwendung von keinen Build-Tools wie „Maven" oder „Gradle" bauen. Deshalb werden die JAR-Dateien der beiden Abhängigkeiten heruntergeladen und in das Projektverzeichnis importiert.
Eine komplette Anleitung zum Aufsetzen des Projektes für die Produktlinien-Anwendung wird am Ende der vorliegenden Arbeit als Anhang angehängt.

5.2 Umsetzung des Konzepts

Bezugnehmend auf die im Kapitel 4.2 konzipierten Komponenten wird im folgenden Kapitel die konkrete Umsetzung der Komponenten veranschaulicht. Grundsätzlich erfolgt die Umsetzung gemäß dem erstellten Feature-Modell im Unterkapitel 4.1 einerseits und dem Entwurfsmustern im Abschnitt 4.2.2 sowie die bewährten Praktiken aus dem Unterkapitel 4.3 andererseits. Die Umsetzung wird bis auf ein paar Änderungen nahezu identisch sein, wie es das Entwurfsmuster vorschreibt. Die Abbildung 19 zeigt das konkrete Klassendiagramm mit den wichtigsten Methoden auf.

5.2.1 Umsetzung der Fabrik-Methode: Klassendiagramm

Die Klasse „FeatureGenerator" spielt die Rolle des Creators im Fabrik-Methode-Entwurfsmuster. Aus dieser Klasse werden konkrete Creators abgeleitet. Diese konkreten Creators werden in „FeatureConfigurator" abgebildet.
Die FeatureGenerator-Klasse benutzt die Schnittstelle „AstTransformationHandler", welche von mehreren „ConfigurationHandler" implementiert wird. Jede Klasse „ConfigurationHandler" stellt ein konkretes „Produkt" dar (vgl. Abbildung 8, oben).

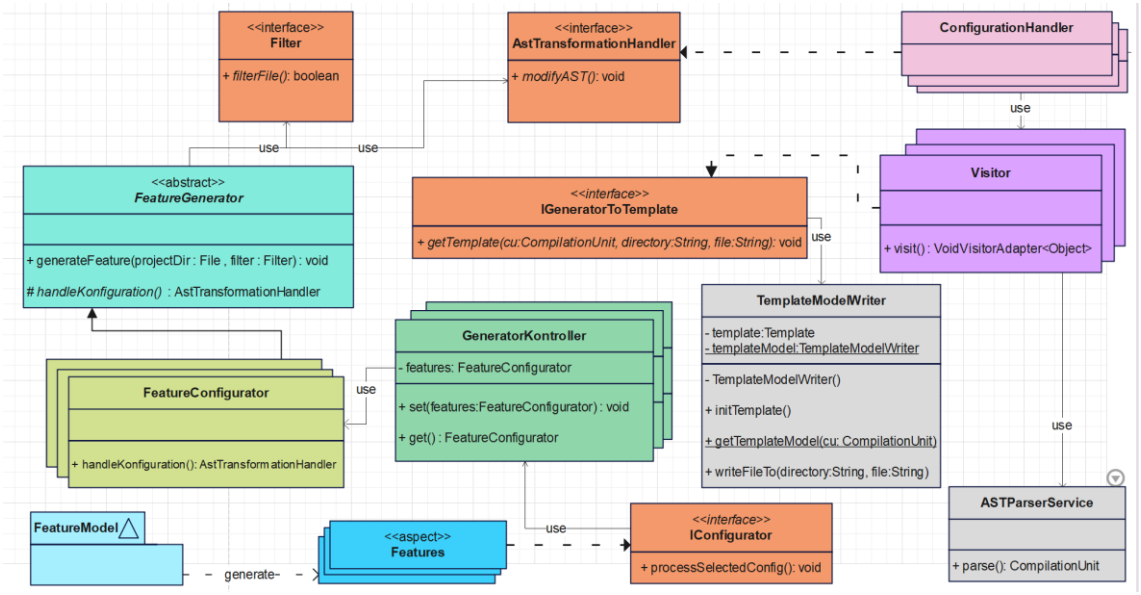

Abbildung 19: Vereinfachtes Klassendiagramm als Gesamtsicht.

5.2.2 Konfiguration

Das Feature-Modell fungiert als Eingabe für die Konfiguration. Die Konfigurationskomponente besteht aus Konfigurationsdateien für die Auswahl der Features und die Aspekt-Dateien, welche die verschiedenen Features als Codes darstellen.
Nach dem Erstellen des Feature-Modells werden leere Aspekt-Klassen generiert. Danach werden die Aspekt-Klasse nur ausgeführt, wenn die entsprechende Konstellation in dem Konfigurationseditor selektiert wurden.

Beispiel: Angenommen die Anforderung A-05 soll erfüllt werden.
Dieses Beispiel soll in den darauffolgenden Komponenten immer wieder aufgegriffen, um die Implementierung der einzelnen Komponenten besser und konkreter zu beschreiben.

Jede Aspekt-Klasse implementiert die Schnittstelle `IConfigurator`, welche die Methode „`void processSelectedConfig()`“ nach außen anbietet. Wenn die Anwendung ausgeführt wird, ruft diese Methode die zuständige Verarbeitungseinheit auf. Darüber hinaus fixiert jede Aspekt-Klasse ein wohldefinierter Punkt im Programmfluss, an dem der Aspekt-Code eingefügt bzw. ausgeführt werden kann.

Die Aspekt-Klasse „`aspect ConstructorParamTwo implements IConfigurator{}`“ wird die Pojo-Klassen in der „Pizza-Pronto“ Anwendung mit einem Zwei-Parameter-Konstruktor als neue Feature bereichern. Mit den definierten „pointcuts“

- „`pointcut classesTwo() : within(de.thb.dim.pizzaPronto.ChefVO)`“ und
- „`pointcut withTwoParam() : execution(new(..))`“

wird der Punkt im Programmfluss definiert, an welchem eine Aktion getriggert wird. Diese Aktion ist im folgenden „Advice“ definiert:

```
after() : classesTwo() && withTwoParam() {
          processSelectedConfig();
}
```

Listing 1: Advice - Bedingung zur Ausführung der Methode „processSelectedConfig()“

Die Methode „processSelectedConfig()“ soll nur dann aufgerufen werden, nachdem die Klasse „de.thb.dim.pizzaPronto.ChefVO“ im Programmfluss instanziiert wird (Siehe Listing 1)

5.2.3 Die Verarbeitung der Konfigurationsauswahl

Es handelt sich um die Schnittstelle IConfigurator. Ihre Methode „void processSelectedConfig()“ hat die Aufgaben den GeneratorKontroller dazu zu triggern, die entsprechenden FeatureKonfigurator aufzurufen. Das Listing 2 zeigt auf, wie der Inhalt der Methode „processSelectedConfig()“ aussieht. Das zu erstellende Feature des Beispiels im vorangegangenen Kapitel wird weiter behandelt.

```
@Override
public void processSelectedConfig() {
//
4      ConstructorConfigGeneration featureCreator =
5             new ChefGenerator(CHEF_VO).getConstructorConfigGeneration();
6             featureCreator.setNumberOfParameter(NUMBER_OF_PARAMETER);
7             featureCreator
              .generateFeature(new FileNameFilter(CHEF_VO), projectDirectory);
}
```

Listing 2: Eine konkrete Implementierung von IConfigurator

Zunächst wird einer Instanz vom GeneratorKontroller die zu modifizierende Datei übergeben. In diesem Fall handelt es sich um die Klasse ChefGenerator. Mit der Methode getConstructorConfigGeneration() wird eine Instanz der Klasse ConstructorConfigGeneration erstellt. Die Klasse ConstructorConfigGeneration ist ein FeatureGenerator (Zeile 4 bis 7 im Listing 2).

Da es darum geht, die Parameteranzahl des Konstruktors in den Pojo-Klassen der „Pizza-Pronto“ Anwendung zu modifizieren, wird dies in der Zeile 6 ausgeführt. Ab der nächsten Zeile bis zum Ende des Methodenrumpfs wird die Methode generateFeature(Filter, File) aufgerufen, um das mit neuem Feature versehenen Programm in eine neue Datei zu generieren.

5.2.4 Generator-Kontroller

Der Generator-Kontroller erhält die Informationen über alle konkreten Implementierungen der abstrakten Klasse `FeatureGenerator`. So kann jede valide Konstellation aus der Konfiguration bedient werden.
Die im vereinfachten Klassendiagramm (Siehe Abbildung 19 oben) dargestellten Attribute und Methoden in der `GeneratorKontroller`-Komponente sind Platzhalter für die verschiedenen konkreten Feature-Generatoren. So wird die `GeneratorKontroller` für das bisher betrachtete Beispiel folgendermaßen aussehen (Abbildung 20):

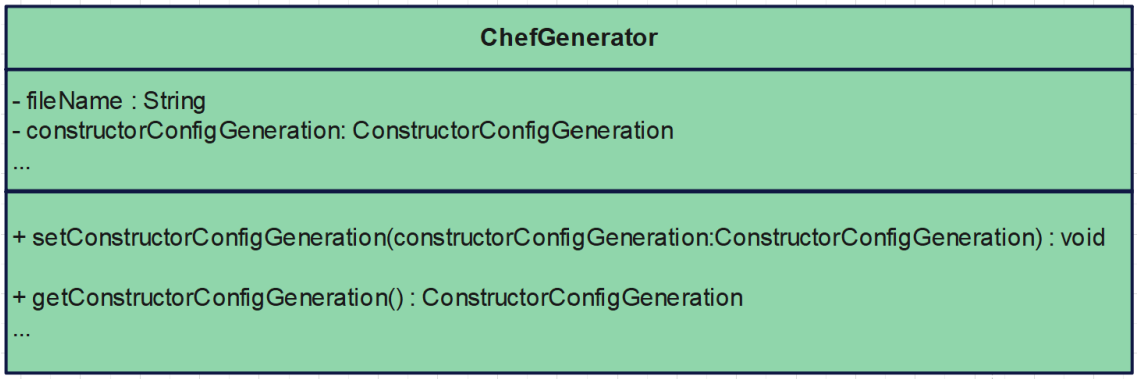

Abbildung 20: Ein Ausschnitt eines Generator-Kontrollers: „ChefGenerator"

Der Generator-Kontroller wird die Instanziierung eines konkreten Feature-Generators vornehmen und bei Bedarf aufrufen. Ein konkretes Beispiel wurde bereits im Listing 2 erläutert.

5.2.5 Konfigurator-Komponente

Die Konfigurator-Komponente wird im Klassengramm als `FeatureConfigurator` dargestellt. Diese Komponente steht stellvertretend für alle weiteren konkreten `FeatureGenerator`.
Ähnlich wie bei der Komponente `GeneratorKontroller` wird die Klasse `ConstructorConfigGeneration` anstelle von `FeatureConfigurator` im Klassendiagramm geschrieben werden.

Die Klasse `ConstructorConfigGeneration` erweitert die Klasse `FeatureGenerator`, um die abstrakte Methode `handleKonfiguration()` der Anforderung gemäß zu

implementieren. Die Implementierung des bisher vorgestellten Beispiels wird fortgesetzt.

In diesem Fall gibt die Methode `handleKonfiguration()` die konkrete Instanziierung einer Transformationseinheit zur Modifizierung des AST zurück. Dies erfolgt nach dem Prinzip des Fabrik-Methode-Entwurfsmusters (Siehe dazu Listing 3). Wie die Transformation tatsächlich vorgenommen wird, übernimmt die „Transformation-Handler-Komponente" (Siehe Abschnitt 5.2.7).

```
@Override
protected AstTransformationHandler handleKonfiguration() {
	return new ConstructorConfiguration(numberOfParameter, javaFileName);
}
```

Listing 3: Instanziierung einer konkreten AST-Transformationseinheit gemäß der Fabrik-Methode

Die Klasse `ConstructorConfiguration` ist ein Repräsentant der nächsten Komponente, der Transformationshandler.

5.2.6 Umsetzung der Constraints

Mit Hilfe der Constraints wird sichergestellt, dass zusätzlich zu den neuen hinzugefügten Konstruktoren, die dazugehörigen Test-Methoden automatisch erzeugt werden. Wiederum es nicht nicht möglich eine Test-Methode zu erzeugen, ohne die Hauptmethode generiert zu haben.

Die Abbildung 21 zeigt auf, wie Constraints gesetzt werden können. In der Abbildung sind die zuständigen Constraints für die Konstruktoren gelb markiert. Es betrifft die roten und orangefarbenen Features im Feature-Diagramm. Eins der Features kann im Test-Zweig nur ausgewählt werden, wenn das Pendant auf dem anderen Zweig ausgewählt ist.

Im Constraint-Editor wird dies wie folgt ausgedruckt:

```
uebung1_chef_ConstructorParamOne iff uebung1_chef_TestFirstConstructorOption
```

Abbildung 21: Umsetzung der Constraints

5.2.7 Transformation-Handler

Die folgende Komponente fasst mehrere konkrete Komponenten zusammen. Es handelt sich um die Klassen `ConfigurationHandler` im Klassendiagramm (Siehe Abbildung 19, oben). Eine Klasse `ConfigurationHandler` wird hier als Sammelbegriff verwenden und steht stellvertretend für z.B. eine Klasse wie `ConstructorConfiguration`.

Jede Komponente (z.B. `ConstructorConfiguration`) implementiert die Schnittstelle `AstTransformationHandler`, um die Transformationseinheit umzusetzen. Dies geschieht durch die Implementierung der Methode `modifyAST(…)`, welche die Instanziierung des dazugehörigen Visitor vornimmt. Die Fortsetzung des Beispiels für Implementierung dieser Methode wird im Listing 4 aufgezeigt.

```
@Override
public void modifyAST(int level, String path, File file) {

	try {
		new ConstructorVisitor(paramNumber, javaFileName)
			.visit(ASTParserService.parse(file), null);

	} catch (FileNotFoundException e) {
		e.printStackTrace();
	}
}
```

Listing 4: Eine Implementierung der Methode „modifyAST(..)"

5.2.8 Visitor- und IGeneratorToTemplate-Komponente

Diese Komponenten stellt die transformierende Einheit dar (Siehe Abbildung 19, oben). Eine Java-Klasse der „Pizza-Pronto" Anwendung wird entgegengenommen. Anschließend wird der dazugehörige AST nach einem bestimmten Knoten durchsucht (in dem vorgestellten Beispiel wird der relevante Knoten der Konstruktor der Klasse sein). Der gefundene Knoten soll gemäß der Anforderung modifiziert werden. Für das Beispiel wird es die Anzahl der Parameter sein. Für die Modifizierung des AST werden diverse Hilfsklasse bzw. -methoden geschrieben.

Die Komponente selbst stellt keine Methoden für die Transformation her, sondern erbt die Methode `visit(..)`-Methode von der Klasse `VoidVisitorAdapter<Object>` aus der JavaParser-Bibliothek.
Darüber hinaus implementiert jede Visitor-Komponente die Schnittstelle `IGeneratorToTemplate` durch die Implementierung der Methode `getTemplate(..)`, um die modifizierte Klasse wieder formatiert in eine Datei auszugeben.

5.2.9 Modell für die Erzeugung von Konstruktoren

In der Abbildung 22 wird aufgezeigt, wie die zu erzeugenden Konstruktoren konstruiert werden können. Hierbei wird ein neuer Konstruktor aus einem bestehenden abgeleitet.

Die „ClassOrInterfaceDeclaration" der JavaParser-Bibliothek repräsentiert in diesem Fall eine Java-Klasse als AST, wobei die Elemente der Klasse Knoten sind. So ist ein Konstruktor ein Knoten, der wiederum Knoten enthält, die Parameter oder die Anweisungen (Statements) im Konstruktor-Rumpf.

In den Zeilen 57 bis 59 werden die Parameter des zu konstruierenden Konstruktors gesetzt. In den Zeilen 61-62 werden die Anweisungen in den Konstruktor-Rumpf gebildet. Danach wird einen Konstruktor-Knoten erzeugt und der Klasse hinzugefügt.

```java
/**
 *
 * @param numberOfParameter    positive integer less than the number of parameter of the
 *             constructor with the highest number of parameters. It represent
 *             the number of parameter to build the new constructor
 * @param classAsAST the given ClassOrInterfaceDeclaration
 * @throws InvalidParameterNumber Thrown when n is out of range.
 */
public static void addNewConstructor(int numberOfParameter, ClassOrInterfaceDeclaration classAsAST) throws InvalidParameterNumber {

    ConstructorDeclaration constructorDeclaration = highestParamNumberConstructor(classAsAST);

    String[] parameterTypes = getParameterTypes(constructorDeclaration);

    Optional<ConstructorDeclaration> constructorByParameterTypes = classAsAST
            .getConstructorByParameterTypes(parameterTypes);
    if (constructorByParameterTypes.isEmpty()) {
        return;
    }
    ConstructorDeclaration constructorHelper = constructorByParameterTypes.get();
    BodyDeclaration<?> afterThisNode = constructorHelper;

    if (numberOfParameter < 0 || numberOfParameter > parameterTypes.length) {
        throw new InvalidParameterNumber("the number of parameter " + numberOfParameter + " is invalid");
    }

    Parameter[] parametersArray = new Parameter[numberOfParameter];
    NodeList<Expression> arguments = buildThisArguments(numberOfParameter, parameterTypes, constructorHelper, parametersArray);
    NodeList<Parameter> parameters = buildParameters(parametersArray);

    NodeList<Statement> statements = buildStmts(arguments);
    BlockStmt body = new BlockStmt(statements);

    BodyDeclaration<?> newNode = buildNewNode(constructorHelper, parameters, body);
    classAsAST.getMembers().addAfter(newNode, afterThisNode);
}
```

Abbildung 22: Modell zu Generierung eines Konstruktors

5.2.10 ParserService-Komponente

Der `ParserService` ist eine kleine, aber sehr wichtige Komponente. Durch die Transformation in einen AST wird der `ParserService` jede übergebene Datei für die Modifizierung bereithalten. Dazu wird die Methode

```
CompilationUnit parse(File file) throws FileNotFoundException
```

der JavaParser-Bibliothek verwendet, welche eine „Compilation Unit" (Kompilierungseinheit) zurückliefert. Die Kompilierungseinheit ist die Darstellung einer Java-Klasse als AST.

5.2.11 Anbindung des Templates

Mit der Komponente `TemplateModelWriter` wird einen externen Generator angebunden. Gemäß der Einleitung über Apache FreeMarker im Unterkapitel 3.4, wurde eine Template-Datei erstellt und vorkonfiguriert. Nach der Initialisierung der FreeMarker-Konfiguration mit

```
Configuration configuration = new Configuration(new Version(2, 3, 31));
```

wird ein Template-Datei geladen. Dies geschieht mit den folgenden Schritten:

```
templateLoader = new FileTemplateLoader(new File("src/utils/template"));
configuration.setTemplateLoader(templateLoader);
```

Anschließend wird das Template aus der Konfiguration unter Angabe des Template-Datei wie folgt erstellt

```
template = configuration.getTemplate(TEMPLATE_FILE).
```

Das Datenmodell:

Das Datenmodell wird aus der Kompilierungseinheit erstellt. Für den Datentransfer in das Template, wird den dazugehörigen Template-Platzhalter miteingeliefert. So formiert sich ein Key-Value-Pair aus dem Platzhalter und der Kompilierungseinheit, das von der Methode

```
TemplateModelWriter getTemplateModel(CompilationUnit compilationUnit)
```

bereitgestellt wird. Das Datenmodell wird folgendermaßen gefüllt:

```
compilationModel.put("compilationUnit", compilationUnit).
```

Dabei ist `"compilationUnit"` der Platzhalter in der Template-Datei, welche durch die Markierung `${compilationUnit}` erkennbar ist.

Wiederverwendung von Komponenten

Neben der Wiederverwendbarkeit von Code-Artefakten (Siehe Abschnitt 4.2.3) wurden die bereits beschriebenen Implementierungen so angelegt, dass die Vorgehensweise bei der Umsetzung der Komponenten für alle Features mit möglichst minimalen Änderungen (gemäß der Anforderung) wiederverwendet werden.

5.3 Umsetzung der Clean Code und S.O.I-Prinzipien

Die Umsetzung dieser Prinzipien lässt sich mit den bisher beschriebenen Implementierungen nicht trennen. Deshalb wird auf die bereits beschriebenen Komponenten bezuggenommen.

5.3.1 S.O.I-Prinzipien

Für die Umsetzung des beschriebenen Beispiels im Abschnitt 5.2.2 wird gemäß der Single-Responsability-Prinzip, drei „GeneratorKontroller" (ChefGenerator, CustomerGenerator und PizzaGenerator) aufgestellt, denn die vorgestellte Anforderung betrifft genau drei Klassen (ChefVO, CustomerVO und PizzaVO).

Für alle Features der jeweiligen Klassen wird den dazugehörigen „GeneratorKontroller" in den Aspekt-Klassen getriggert. Wie das Triggern umgesetzt wird, wurde im Abschnitt 5.2.3 erklärt.

Ebenso wird im Transformationsschritt in der Visitor-Komponente der Produktlinien-Anwendung mehrere „Visitors" für den jeweiligen Fall implementiert. Dafür wurde die Schnittstelle „AstTransformationHandler" aufgesetzt (Siehe Abschnitt 5.2.7), die als Schablone für die Umsetzung aller benötigten Transformationen abstrahiert.
Jede weitere Komponente, die an dieser Schnittstelle interessiert ist, muss die Methode `public void modifyAST()` implementieren. Komponente, die an mehrere Schnittstellen interessiert sind, können diese einfach zusätzlich implementieren. Auf diese Weise wird sowohl das Open/Closed-Prinzip als auch das Interface-Segregation-Prinzip (aufgrund seiner Schlankheit) umgesetzt.

5.3.2 Umsetzung von Clean Code

Um die „Abstraktionsebene pro Funktion" zu realisieren, werden Komplexe Berechnungen moduliert und in verschiedenen kleineren Berechnungseinheiten unterteilt. Wie in dem Listing 5 aufgezeigt, werden die Implementierungen der Methoden *addNewConstructor()* (Zeile 29), *buildDir()* (Zeile 34) und *createDir()* (Zeile 35) in eigene Komponente delegiert. In der Zeile 29 wird die Methode *addNewConstructor()* aufgerufen, die in der Abbildung 22 (Abschnitt 5.2.9) ausführlich dokumentiert ist.

Die Implementierung der drei Methoden, die ausgelagert wurde, werden bei Bedarf in den anderen Komponenten wiederverwendet. Auf diese Weise die das DRY-Prinzip umgesetzt.

Die Klasse `ConstructorVisitor` im Listing erweitert den `VoidVisitorAdapter` von JavaParser. Mit den `VoidVisitorAdapter` kann der AST einer Java-Klasse mit Hilfe der `visit()`-Methode durchlaufen (besucht) werden. In diesem Fall soll einen neuen Konstruktor-Knoten hinzugefügt werden (Zeile 29 im Listing 5). Deshalb besteht die Klassenbezeichnung aus „Konstruktor" und „Visitor". Analog werden anderen Bezeichnungen hergeleitet. So lässt sich das Programm einfacher lesen und warten und überflüssigen Kommentaren werden dadurch vermieden.

```
13 public class ConstructorVisitor extends VoidVisitorAdapter<Object>
                                implements IGeneratorToTemplate{

      ...
      ...

23    @Override
24    public void visit(ClassOrInterfaceDeclaration classAsAST, Object arg)
{
25          super.visit(classAsAST, arg);

27          CompilationUnit compilationUnit =
                                classAsAST.findCompilationUnit().get();
28          try {
29                ConstructorUtil
                                .addNewConstructor(paramNumber, classAsAST);
30          } catch (InvalidParameterNumber e1) {
31                e1.printStackTrace();
32          }

34          String directory = DirCreatorUtil.buildDir(compilationUnit);
35          DirCreatorUtil.createDir(directory);

37          getTemplate(compilationUnit, directory, javaFileName + ".java");
38    }

      ...
      ...

44}
```

Listing 5: Umsetzung von Clean Code

6 Evaluation

In diesem Kapitel werden die Erwartungen mit dem Ergebnis verglichen. Dafür wird die Produktlinien-Anwendung getestet und das Testergebnis interpretiert. Ferner werden die Herausforderungen bei der Implementierung sowie mögliche Abweichungen zur Konzeption gekennzeichnet und eine Schlussfolgerung gezogen.

6.1 Testbetrieb

Im Testbetrieb wird kontrolliert, ob die Erwartungen erfüllt werden können. Bezugnehmen auf dem **Beispiel:** Angenommen die Anforderung A-05 soll erfüllt werden wird überprüft, ob die Basis Programm nach der Ausführung der Anwendung die neue Feature aufweist.

6.1.1 Einrichten der Testumgebung

Dazu werden die generierten Klassen in ein neu initiiertes Eclipse-Projekt importiert. Im neuen Eclipse-Projekt wird die Abhängigkeit von JUnit zunächst importiert: *Recht-Klick auf dem Projekt > Build Path > Add Librairies > JUnit*.

6.1.2 Testdurchführung

Ist die Testumgebung richtig eingerichtet, können die mitgenerierten Testklasse ausgeführt werden. Mit der folgenden Anweisung: „*Recht-Klick auf eine Klasse > Run as > JUnit Test*" kann jede Testklasse individuell ausgeführt werden.

6.1.3 Testergebnis und Interpretation

Die Resultate erscheinen grün markiert (Siehe Abbildung 23). Es ist auch zu sehen, dass die Tests für die Konstruktoren mit einem und zwei Parameter(n) als Argument auch erfolgreich durchgeführt wurden.
Die erfolgreiche Durchführung deutet darauf hin, dass die entsprechenden Konstruktoren tatsächlich in der zu testenden Klasse vorhanden sind. Auf diese

Weise kann bestätigt werde, dass die neuen Features generiert und dem Basispragramm hinzugefügt wurden, wie erwartet.

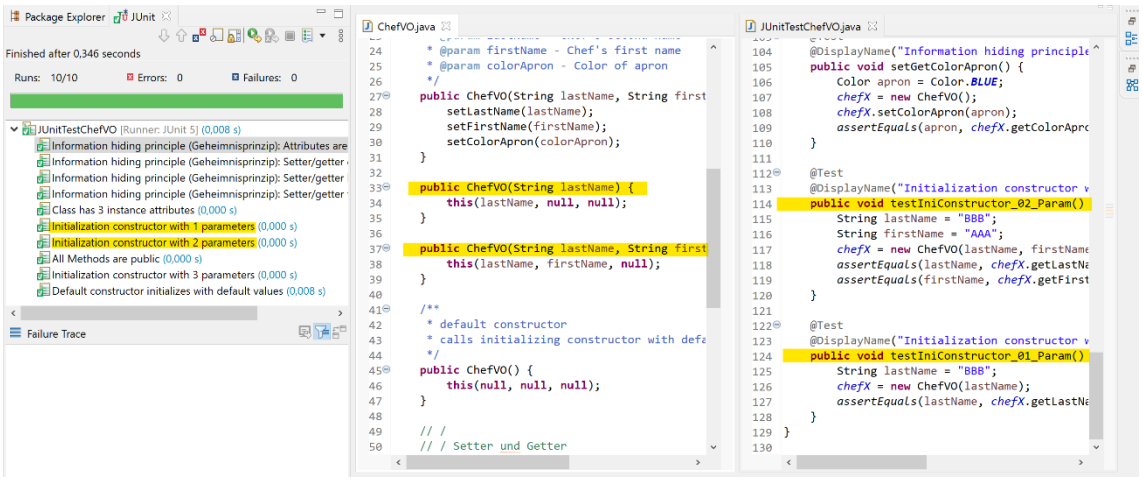

Abbildung 23: Produktlinie-Anwendung im Testbetrieb

6.2 Erfüllung der Erwartungen

In der Aufgabenstellung (Siehe Unterkapitel 1.1) eingangs der Arbeit wurden Fragen gestellt, die im Verlauf der Arbeit beantwortet wurden. Die Antworten auf die Fragen:

- „Wie lässt sich eine SPL modellieren und entwickeln?" und
- „Welche Grundlegenden Prinzipien müssen dabei beachtet werden, um die Anforderungen zu erfüllen?"

wurden mit dem Vorgehen der SPLE in FeatureIDE mit AspectJ im Kapiteln 3 mit den Konzepten Problem- und Lösungsraum gegeben.

Die Beantwortung der Frage „Wie lassen sich Produktvarianten so generieren, dass getroffene Entscheidungen über das Design der zu entwickelten Software sich in Codestrukturen widerspiegeln?" wurde mit Hilfe der Anbindung von externen Generatoren umgesetzt. Diese Maßnahme war nötig, um die generierten Codes von der mitgelieferten „Pizza-Pronto" Anwendung zu trennen. Die neu generierte Anwendung konnte auf dieser Weise portierbar gemacht werden.

Neben den Erwartungen waren auch detaillierte Anforderungen an das System gestellt worden. Diese konnten durch die Identifikation, die Konzeption und Implementierung der verschiedenen Komponenten umgesetzt werden. Daraus ist

ein funktionierender Prototyp der Produktlinien-Anwendung entstanden, welcher alle Anforderungen erfüllt.

6.3 Herausforderungen und Abweichungen

Der Prototyp der Produktlinien-Anwendung besteht aus drei sich aufeinanderfolgenden Hauptfeatures. Diese Entscheidung beruht auf folgenden Gründen: Erstens die verschiedenen Teile der Pizza-Pronto-Anwendung bauen aufeinander auf und zweitens die Wiederverwendbarkeit kommt mehr zum Tragen. Ansonsten müssten die gemeinsamen Artefakte jedes Mal von neue generiert werden, statt bestehende zu benutzen.
Im Moment ist der Prototyp nicht so implementiert, dass die zweite Feature-Generation ausgeführt wird, ohne vorher die erste durchgeführt zu haben. An dieser Stelle kann eine Alternative ausgearbeitet werden.

Außerdem ist es in FeatureIDE vorgesehen, dass die Generation von Varianten automatisch nach der Auswahl der Features im Konfigurationseditor erfolgt. Dies erwies sich als schwierig umzusetzen auch in Bezug auf die Generierung auf externen Dateien.
Deshalb weicht die Implementierung des Prototyps von der klassischen SPLE in FeatureIDE ab. Aus diesem Grund mussten neue Lösungswege gegangen werden.

Eine große Herausforderung bestand darin, die Features an die Pizza-Pronto-Anwendung anzuheften, um neue Varianten zu bilden. Die erste Überlegung, Features mit der Java-Reflexion-API zur Laufzeit zu verändern (eine Feature z.B. ein Methodenrumpf), war auf Anhieb zum Scheitern verurteilt. Denn nach der Kompilierung wird einen Byte-Code ausgegeben und kann mit der Reflexion-API nicht manipuliert werden.

Eine Alternative wurde mit der JavaParser-Bibliothek gefunden. Im Zusammenspiel mit AspectJ konnte jede Java-Klasse zur Laufzeit modifiziert werden. Dafür musste eine neue Herangehensweise gefolgt werden (Siehe dazu Abschnitt 3.4.3).

Schussfolgerung

Der funktionierende Prototyp wurde anhand des im Grundlagen-Kapitel ausgearbeiteten Wissens entworfen und umgesetzt. Die gestellten Anforderungen konnten erfüllt werden. Dennoch hat die Produktlinien-Anwendung noch Verbesserungspotenzial, dass im letzten Kapitel aufgezeigt werden soll.

7 Ergebnis und Ausblick

Die vorliegende Arbeit wird abschließend in diesem Kapitel zusammengefasst. Nach einer Darstellung des Ergebnisses dieser Arbeit, werden die identifizierten Kritikpunkte zusammengefasst. Danach wird die Arbeit mit einer Ausblick auf die nächsten Schritte „Blick über den Tellerrand" abgeschlossen.

7.1 Zusammenfassung

In der vorliegenden Masterarbeit wurde eine Produktlinien-Anwendung entworfen und prototypisch implementiert, mit der Möglichkeit Features zu konfigurieren und verschiedene Varianten als Quellcode zu generieren. Mit der Produktlinien-Anwendung wird eine Alternative zu der vorangegangenen manuellen Implementierung der Varianten angeboten.

Zu diesem Zweck wurden Grundwissen zu MDSD und SPLE angeeignet. Das Konzept der Modell-Transformation sowie der Feature-orientierte Software-Entwicklung wurden beleuchtet. Mit diesem Wissen wurden die verschiedenen Phasen zur Erstellung einer SPL gezeigt. Für die Identifizierung der Features wurde die Pizza-Pronto-Anwendung vorgestellt und analysiert, welche in der weiteren Arbeit als Grundlage für Konzeption und Implementierung diente. Bei der Umsetzung des gewonnenen Wissens konnten die eingangs Fragen zur Theorie der SPLE, fortlaufend beantwortet werden.

Konkrete Anforderungen wurden anschließend an das zu entwickelnde System gestellt. Folgend wurden die Technologien sowie die Werkzeuge zur Unterstützung des SPL-Entwicklungsprozesses beschrieben. Hierbei wurde das bewährte und für die SPLE dedizierte Werkzeug „FeatureIDE" vorgestellt und verwendet.
Die FOSD wurde durch das aspektorientierte Kompositionswerkzeug AspectJ von FeatureIDE umgesetzt. Es wurde gezeigt, wie „Anhaltspunkte" mittels Aspekt-Direktiven definiert werden können, um neue Features der Pizza-Pronto-Anwendung zur Laufzeit hinzufügen. Darüber hinaus wurden einige bewährten Programmierpraktiken für eine bessere Wartbarkeit, Wiederverwendbarkeit oder Lesbarkeit des Codes beschrieben.

Darauffolgend wurden die Komponenten der Produktlinien-Anwendung identifiziert und konzipiert. Dazu gehören das mit dem Feature-Syntax-Tree erstellte Feature-Modell mit den dazugehörigen Constraints, die Konfigurationsdateien mit den entsprechenden Features als Aspekt-Dateien sowie das Paket- und Komponentendiagramm. Die Konzeption galt als Grundlage für die Implementierung.

Bei der Umsetzung der Produktlinie-Anwendung kam es durch Einschränkungen durch FeatureIDE zu einigen nötigen Anpassungen. Deshalb wurde eine Hybridlösung aus klassischer Entwicklungsidee in FeatureIDE und eigenen Anpassungen und Erweiterungen durch Anbindung eines externen Generators umgesetzt.

Mit dieser Lösung konnten die Entwurfsentscheidungen, die bei Pizza-Pronto-anwendung getroffen wurden, in den generierten Codes der Features beibehalten werden. Zum Schluss wurde ein funktionierender Prototyp erfolgreich entwickelt, welcher die gestellten Anforderungen erfüllt und auch mittels implementierter Tests bestätigt.

7.2 Kritische und offene Punkte

Trotz der erfolgreichen Implementierung eines funktionierenden Prototyps der Produktlinien-Anwendung können einige Kritiken angeführt werden.

In der Implementierung der Produktlinien-Anwendung wird vorausgesetzt, dass die Generierung der Features sequenziell ablaufen. Die Generierung der Feature-Gruppe „Uebung2" hängt vom Ergebnis der Feature-Gruppe „Uebung1" ab. Eine Alternative ohne diese sequenzielle Abhängigkeit wäre in dieser Hinsicht von Vorteil. Die Möglichkeit jede der drei Haupt-Features unabhängig voneinander zu generieren und gleichzeitig die gemeinsamen Artefakte effizienter wiederzuverwenden, könnte angestrebt werden.

Dafür sollten einige der erstellten Modelle bis zu einem gewissen Grad abstrakter gestaltet werden. Diese Modelle erfüllen den Zweck, wofür sie gebaut wurden, aber können aufgebessert werden (das Modell zur Erstellung eines Konstruktors

ist ein gutes Beispiel, siehe Abbildung 22). Dies war keine explizite Anforderung an das System, aber eine sinnvolle Ergänzung.

Ferner kann zu diesem Entwicklungsstand des Prototyps der Diskursbereich nicht einfach ersetzt werden. Der Code müsste stellenweise überarbeitet oder teilweise umgeschrieben und generischer gestaltet werden.

7.3 Ausblick

Nachdem die Arbeit evaluiert und Kritik an dieser geübt wurde, werden ein paar Verbesserungsvorschläge für eine zukünftige Weiterimplementierung gegeben. Einige davon wurden im vorangegangenen Kapitel bereits angedeutet.

Für eine generische Implementierung sollte der Gebrauch der JavaParser-Bibliothek weiter ausgebaut werden. Dazu ist es nötig alle validen Varianten zu untersuchen, um ggf. die Struktur der Ausgaben abzubilden. Jedoch ist eine komplette Abbildung nicht einfach zu realisieren und bringt einiges an Aufwand mit sich. Eine DSL (Domainspezifische Sprache) zu entwickelt kann als Alternative berücksichtigt werden. In beide Fällen (Mit JavaParser oder DSL) wird oft der generierte Code mit manuell erzeugtem Code befüllt werden.

Darüber hinaus kann die Performanz der Anwendung verbessert werden. Die Vorteile von AspectJ können noch weiter ausgebaut werden. Bisher werden einige „*pointcut*" so gesetzt, dass die dazugehörige „*advice*" mehrere Male ausgeführt werden. Zum Beispiel der „Pointcut"

```
pointcut withTwoParam(): execution(new(..))
```

wird jedes Mal ausgeführt, wenn die gezielte Klasse instanziiert wird. Bei einer Mehrfach-Instanziierung wird der „Pointcut" jedes Mal neu ausgeführt.

An dieser Stelle kann eine Lösung implementiert werden, welche einen „Pointcut" auswählt, die möglichst nur einmal oder nicht so oft im Programmfluss vorkommt. Sollte solch ein „Pointcut" nicht gefunden werden, könnte eine Überlegung sein, einen künstlichen Punkt (z.B. eine Methode) so einzuführen, dass dieser bei der Ausführung der Produktlinien-Anwendung nicht mehrmals vorkommt.

Literaturverzeichnis

Ainsley, C. (2020, Januar 17). *Introducing Picocog: A Lightweight Code Generation Library*. (DZone, Editor) Retrieved März 02, 2023, from Introducing Picocog: A Lightweight Code Generation Library: https://dzone.com/articles/introducing-picocog-a-lightweight-code-generation

Apache Software Foundation. (2015, September 02). *Apache FreeMarker*. Retrieved März 04, 2023, from https://freemarker.apache.org/

Apel, S., & Kästner, C. (2009). An overview of feature-oriented software development. (E. Zurich, Ed.) *Journal of Object Technology, 8*(4), 49-84.

Aßman, U., Zschaler, S., & Wagner, G. (2006). Ontologies, Meta-Models, and the Model-Driven Paradigm. In C. Calcero, F. Ruiz, & M. Platini (Eds.), *Ontologies for software engineering and software technology* (pp. 249-273). Berlin: Springer.

Baeldung. (2023, Februar 23). *Intro to AspectJ*. Retrieved März 01, 2023, from Intro to AspectJ: https://www.baeldung.com/aspectj

Batory, D., Apel, S., & Kästner, C. (2007). A Case Study Implementing Features Using AspectJ. *11th International Software Product Line Conference (SPLC 2007)*, 223-232. doi:doi:10.1109/SPLINE.2007.12

Becker, S., Goldschmidt, T., Groenda, H., Happe, J., Jacobs, H., Janz, C., . . . Veliev, B. (2007). *Transformationen in der modellgetriebenen Software-Entwicklung* (ISSN 1432-7864 ed., Vol. 9). (I. f. Fakultät für Informatik, Ed.) Karlsruhe: Fakultät für Informatik, Universität Karlsruhe. Retrieved from http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007100

Cephas Consulting Corp. (2006, Januar). *The Fast Guide to Model Driven Architecture.* (F. Truye, Ed.) Retrieved November 11, 2022, from The Basics of Model Driven Architecture (MDA): https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf

Clements, P., & Northrop, L. (2002). *Software Product Lines - Practices and Patterns.* Boston: Addison-Wesley.

Colomb, R., Raymond, K., Hart, L., Emery, P., Welty, C., Tong Xie, G., & Kendall, E. (2006). The Object Management Group Ontology Definition Metamodel. In C. Calero, M. Piattini, F. Ruiz, C. Calero, M. Piattini, & F. Ruiz (Hrsg.), *Ontologies for Software Engineering and Software Technology* (S. 217-

247). Berlin, Heidelberg: Springer. doi:https://doi.org/10.1007/3-540-34518-3_8

DigitalOcean. (2022, August 3). *Gangs of Four (GoF) Design Patterns*. Retrieved März 28, 2023, from https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns

Dubinsky, Y. R., Berger, T., Duszynski, S., Becker, M., & Czarnecki, K. (2013). An Exploratory Study of Cloning in Industrial Software Product Lines. *2013 17th European Conference on Software Maintenance and Reengineering*, 25-34. doi:10.1109/CSMR.2013.13

FeatureIDE Team. (2014). *FeatureIDE*. Retrieved März 10, 2023, from An extensible framework for feature-oriented software development: https://featureide.github.io/

FeatureIDE-METOP. (2020). *FeatureIDE*. (M. GmbH, Editor, & METOP GmbH) Retrieved November 26, 2022, from FeatureIDE: https://www.featureide.de/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software.* Deutschland : Pearson Deutschland GmbH.

Gerspacher, J., & Weber, F. (2013). Featureorientierte Softwareentwicklung mit FeatureIDE. *informatikJournal 2013*, 159-167.

Grigarzik, P. (2020, Dezember 7). Analyse eines Bestellsystems für zukünftige Variantenbildung in der Lehre. *Unveröffentlichte Bachelorarbeit am Fachbereich Informatik und Medien. Brandenburg an der Havel: Technische Hochschule Brandenburg.* Brandenburg.

Han, Y., Kniesel, G., & Cremers, A. B. (2004, October). A meta model and modeling notation for AspectJ. *The 5th AOSD Modeling with UML Workshop.*

Heuvel, D. v. (2022). *JavaForger*. (A.-2. license, Producer) Retrieved März 02, 2023, from JavaForger: https://github.com/daanvdh/JavaForger

IONOS. (2021, Februar 17). *Digital Guide IONOS*. (I. Inc., Editor) Retrieved März 09, 2023, from Web development: https://www.ionos.com/digitalguide/websites/web-development/what-is-a-factory-method-pattern/

Janschitz, M. (2015, Juni 27). *Prinzipien der Software-Entwicklung einfach erklärt: SOLID*. (y. m. 2023, Editor) Retrieved März 28, 2023, from t3n

digital pioneers: https://t3n.de/news/prinzipien-software-entwicklung-solid-615556/

Jin, Z. (2018). Chapter 6 - Feature Model of Domain Environment. In Z. Jin (Ed.), *Environment Modeling-Based Requirements Engineering for Software Intensive Systems* (pp. 85-98). Morgan Kaufmann. doi:https://doi.org/10.1016/B978-0-12-801954-2.00006-6

Kang, K., Lee, J., & Donohoe, P. (2002., Juli-August). Feature-oriented product line engineering. *IEEE software, 19*(4), 58-65.

Kanis, M. (2008). *Von Objektorientierung zu Aspektorientierung AspectJ.* Technische Universität München.

Kästner, C., & Apel, S. (2013). Feature-Oriented Software Development: A Short Tutorial on Feature-Oriented Programming, Virtual Separation of Concerns, and Variability-Aware Analysis. Generative and Transformational Techniques. (R. S. Lämmel, Ed.) *Software Engineering IV: International Summer School, GTTSE 2011. Lecture Notes in Computer Science, 7680*(Revised Papers (2013)), 346-382. Retrieved from https://doi.org/10.1007/978-3-642-35992-7_10

Kempa, M., & Mann, Z. Á. (2005). *Model Driven Architecture*. (M. Kempa, Z. Á. Mann, Herausgeber, G. f. Informatik, Produzent, & sd&m AG software design & management) Abgerufen am 10. September 2022 von https://gi.de/informatiklexikon/model-driven-architecture/

Kuznetsov, M. (Febuar 2007). UML Model Transformation and Its Application to MDA Technology. (L. © Pleiades Publishing, Hrsg.) *Programming and Computer Software, 33*(1), S. 44-53. doi:https://doi.org/10.1134/S0361768807010069

Lengauer, C., Apel, S., & Kastner, C. (2009). FEATUREHOUSE: Language-independent, automated software composition. *2009 IEEE 31st International Conference on Software Engineering*, 221-231. doi:10.1109/ICSE.2009.5070523

Lopez-Herrejon, R. E., & Batory, D. (2002). *Using AspectJ to Implement Product-Lines: A Case Study.* (C. Science, Ed.)

Martin, R. (2013). *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe.* MITP-Verlags GmbH & Co. KG.

McGregor, J. D., Monteith, J. Y., & Zhang, J. (2011). Quantifying Value in Software Product Line Design. (A. f. Machinery, Ed.) *Proceedings of the 15th International Software Product Line Conference, 2* ((SPLC '11)), Article 40, 1-7. doi:https://doi.org/10.1145/2019136.2019182

MDA Guide. (2003, Juni 12). *MDA Guide Version 1.0.1.* (J. Mille, & J. Mukerj, Eds.) Retrieved 11 12, 2022, from MDA Guide Version 1.0.1: http://www.omg.org/cgi-bin/doc?omg/03-06-01

Millington, S. (2022, November 23). *A Solid Guide to SOLID Principles*. (Baeldung, Editor) Retrieved März 28, 2023, from Baeldung: https://www.baeldung.com/solid-principles#d

OMG UML. (2017, Dezember). *OMG® Unified Modeling Language®.* (O. U. Language®, Ed.) Retrieved November 10, 2022, from OMG® Unified Modeling Language®: https://www.omg.org/spec/UML/2.5.1/PDF

OMG/MOF. (2023). *METAOBJECT FACILITY*. (O. Object Management Group®, Editor, & Object Management Group®, OMG®) Retrieved Februar 19, 2023, from OMG'S METAOBJECT FACILITY™: https://www.omg.org/mof/

OMG/OCL. (2014, Februar). *Object Constraint Language.* (OMG, Ed.) Retrieved November 11, 2022, from OMG: https://www.omg.org/spec/OCL/2.4/PDF

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., & Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. . *Software: Practice and Experience, 46*(8), 1155-1179.

Petrasch, R., & Meimberg, O. (2006). *Model Driven Architecture - eine praxisorientierte Einführung in die MDA.* Heidelberg: dpunkt.verlag GmbH.

Poernomo, I. (2006, April). The Meta-Object Facility Typed. (A. f. Machinery, Ed.) *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1845-1849. doi:https://doi.org/10.1145/1141277.1141710

Rosenmüller, M., Apel, S., Leich, T., & Saake, G. (2005). FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. *Proc. Int'l Conf. Generative Programming and Component Engineering, GPCE*, 125–140.

Schmidt, G., & Buchholz, S. (2022). Lehrveranstaltung Programmierung II. *Unveröffentliche Lehrveranstaltung vom Fachbereich Informatik und Medien*. Brandenburg an der Havel: Technische Hochschule Brandenburg.

Schröder, A. (Ed.). (2018). *Agile Produktentwicklung: Schneller zur Innovation–erfolgreicher am Markt* (2., überarbeitete ed.). Carl Hanser Verlag GmbH Co KG.

Siegmund, N., Pukall, M., Soffner, M., Köppen, V., & Saake, G. (2009). Using software product lines for runtime interoperability. (N. Y. Association for Computing Machinery, Ed.) *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, Article 4, 1–7. doi:https://doi.org/10.1145/1562860.1562864

Smith, N., van Bruggen, D., & Tomassetti, F. (2021). *JavaParser: Visited - Analyse, transform and generate your Java code base.* (leanpub, Ed.) leanpub e-book. Retrieved from http://leanpub.com/javaparservisited

Square Open-Source. (2020). *Javapoet*. (S. O. Source, Editor) Retrieved März 02, 2023, from Javapoet: https://github.com/square/javapoet

Stahl, T., Efftinge, S., Haase, A., & Völter, M. (2021). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt. verlag.

Steinbrecher, W. (Ed.). (2020). *Agile Einführung der E-Akte mit Scrum.* Berlin,Heidelberg: Springer Gabler. doi:https://doi.org/10.1007/978-3-662-59705-7

Strocco, F., Meglio, D., Bettini, L., Damiani, F., & Schaefer, I. (2010). *DeltaJ*. Abgerufen am 26. Februar 2023 von https://deltaj.sourceforge.net/

Tang, W. (2018). Meta Object Facility. In L. Liu, & M. Özsu (Eds.), *Encyclopedia of Database Systems* (pp. 2238-2239). New York, NY: Springer New York. doi:10.1007/978-1-4614-8265-9_914

The AspectJTM Programming Guide - Xerox Corporation. (2003). *The AspectJ Programming Guide*. (2.-2. P. Copyright (c) 1998-2001 Xerox Corporation, Editor) Retrieved März 01, 2023, from The AspectJ Programming Guide: https://www.eclipse.org/aspectj/doc/released/progguide/

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming, 79*, 70–85. doi:https://doi.org/10.1016/j.scico.2012.06.002

wikipedia. (2022, Juni 16). *AspectJ*. Retrieved März 19, 2023, from AspectJ: https://en.wikipedia.org/wiki/AspectJ

Zhart, D. (2023). *DESIGN PATTERNS*. Retrieved März 28, 2023, from Refactoring Guru: https://refactoring.guru/design-patterns

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

A

AJDT 99,
AspectJ Development Tools (AspectJ-Entwicklungstool)

AOP 42,
Aspect Oriented Programming (aspektorientierte Programmierung)

AspectJ 42, 43, 45, 46, 47, 48, 51, 52, 54, 58, 59, 63, 76, 77, 79, 81,
AspectJ ist eine Erweiterung der aspektorientierten Programmierung (AOP), die am PARC für die Programmiersprache Java entwickelt wurde. Sie ist in Open-Source-Projekten der Eclipse Foundation verfügbar, sowohl als eigenständige Lösung als auch integriert in Eclipse. AspectJ hat sich zu einem weit verbreiteten De-facto-Standard für AOP entwickelt, da es auf Einfachheit und Benutzerfreundlichkeit für den Endbenutzer setzt. Es verwendet eine Java-ähnliche Syntax und enthält seit seiner ersten öffentlichen Veröffentlichung im Jahr 2001 IDE-Integrationen zur Anzeige von Querschnittsstrukturen.

AST 56, 60, 69, 72, 73,
Abstract Syntax Tree: ist eine Art Baumdarstellung der abstrakten syntaktischen Struktur von Quellcode, der in einer Programmiersprache geschrieben wurde. Jeder Knoten des Baums bezeichnet ein im Quellcode vorkommendes Konstrukt.

C

CIM 13,
Computer Independent Model (Computerunabhängige Modell)

F

FOP 36,
Feature Oriented Programming (Feature-orientierte Programmierung)

FOSD 33, 34, 36, 40, 48,
Feature Oriented Software Development (Feature-orientierte Software-Entwicklung)

M

MDA 13, 14, 20, 21,
Model Driven Architecture (Modellgetriebene Architektur)

MDSD 10, 11, 12, 23, 25, 26, 37, 79,
Model Driven Software Development (modellgetriebenen Software-Entwicklung)

MOF 15, 16, 18, 19,
Die MetaObject Facility Specification™ (MOF™) ist die Grundlage der OMG-Standardumgebung, in der Modelle aus einer Anwendung exportiert, in eine andere importiert, über ein Netzwerk transportiert, in einem Repository gespeichert und dann abgerufen, in verschiedene Formate (einschließlich XMI™, dem XML-basierten Standardformat der OMG für die Übertragung und Speicherung von Modellen) umgewandelt und zur Generierung von Anwendungscode verwendet werden können. Diese Funktionen sind nicht auf Strukturmodelle oder sogar auf in UML definierte Modelle beschränkt - Verhaltensmodelle und Datenmodelle nehmen ebenfalls an dieser Umgebung teil, und auch Modellierungssprachen, die nicht auf UML basieren, können daran teilnehmen, solange sie MOF-basiert sind (OMG/MOF, 2023).

O

OCL 19, 20, 33,
Object Constraint Language: OCL ist eine reine Spezifikations- und Formalsprache, die zur Beschreibung von Ausdrücken in UML-Modellen verwendet wird (OMG/OCL, 2014).

OMG 12, 15, 18, 20,
Object Management Group: ist ein Konsortium, welches international anerkannte Standards (z.B. UML, XML, CORBA, XMI, etc…) für verschiedenen Aspekte der Softwareentwicklung bereitstellt (Colomb, et al., 2006).

P

PIM 14,
Platform Independent Model (Plattformunabhängigen Modell)

PSI 14,
Platform Specific Implementation (der Code)

PSM 14,
Platform Specific Model (Plattformspezifischen Modell)

S

SPL 8, 9, 26, 27, 32, 34, 42, 76, 79,
Software Product Line (Software Produktlinie)

SPLE ii, iii, 27, 33, 37, 40, 47, 57, 63, 76, 77, 79,
Software Product Line Engineering (Software Produktlinie Entwicklung

Listingsverzeichnis

Komplete Implementierung des Beispiels

```
public aspect ConstructorParamOne implements IConfigurator {

	private static final String CHEF_VO = "ChefVO";
	Logger log =
		Logger.getLogger(ConstructorParamOne.class.getCanonicalName());
	File projectDirectory = new
				File("generated/de/thb/dim/pizzaPronto/u1/");
	private static final int numberOfParameter = 1;

	pointcut classesOne() : within(de.thb.dim.pizzaPronto.ChefVO);
	pointcut withOneParam() : execution(new(..));

	after() : classesOne() && withOneParam() {
		log.config(thisJoinPointStaticPart.toShortString());
		processSelectedConfig();
	}

	@Override
	public void processSelectedConfig() {
		ConstructorConfigGeneration featureCreator =
				new ChefGenerator( CHEF_VO)
					.getConstructorConfigGeneration();

		featureCreator.setNumberOfParameter(numberOfParameter);
		featureCreator
			.generateFeature(
				new FileNameFilter(CHEF_VO),projectDirectory
			);
	}
}
```

Listing 6: Die komplette AspectJ-Klasse „ConstructorParamOne“

```
public class ChefGenerator {
	...
	private ConstructorConfigGeneration constructorConfigGeneration;
	...
	public ChefGenerator(String javaFileName) {
	...
	this.setConstructorConfigGeneration(constructorConfigGeneration);
	...
	}
	...

	public ConstructorConfigGeneration getConstructorConfigGeneration() {
		return constructorConfigGeneration;
	}
```

```
	public void setConstructorConfigGeneration(
		ConstructorConfigGeneration constructorConfigGeneration) {

		this.constructorConfigGeneration = constructorConfigGeneration;
	}
	...
}
```

Listing 7: Der Generator-Kontroller „ChefGenerator"

```
public class ConstructorConfigGeneration extends FeatureGenerator{

	private int numberOfParameter;

	public int getNumberOfParameter() {
		return numberOfParameter;
	}
	public void setNumberOfParameter(int numberOfParameter) {
		this.numberOfParameter = numberOfParameter;
	}

	@Override
	protected AstTransformationHandler handleKonfiguration() {
		return new ConstructorConfiguration(
					numberOfParameter, javaFileName
		);
	}

}
```

Listing 8: Die konkrete Feature-Fabrik: Der Konfigurator „ConstructorConfigGeneration"

```
public class ConstructorConfiguration implements AstTransformationHandler{

	private int paramNumber;
	private String javaFileName;

	public ConstructorConfiguration(int paramNumber, String javaFileName)
{
		this.paramNumber = paramNumber;
		this.javaFileName = javaFileName;
	}

	@Override
	public void modifyAST(int level, String path, File file) {

		try {
			new ConstructorVisitor(paramNumber, javaFileName)
				.visit(ASTParserService.parse(file), null);
```

```
			} catch (FileNotFoundException e) {
				e.printStackTrace();
			}
		}
}
```

Listing 9: Der Transformation-Handler: ConstructorConfiguration

```
package generator.astvisitor;

import com.github.javaparser.ast.CompilationUnit;
import com.github.javaparser.ast.body.ClassOrInterfaceDeclaration;
import com.github.javaparser.ast.visitor.VoidVisitorAdapter;

import utils.ConstructorUtil;
import utils.DirCreatorUtil;
import utils.TemplateModelWriter;
import utils.exceptions.InvalidParameterNumber;
import utils.generator.handler.IGeneratorToTemplate;

public class ConstructorVisitor extends VoidVisitorAdapter<Object> implements
IGeneratorToTemplate{

	private int paramNumber;
	private String javaFileName;

	public ConstructorVisitor(int paramNumber, String javaFileName) {
		this.paramNumber = paramNumber;
		this.javaFileName = javaFileName;
	}

	@Override
	public void visit(ClassOrInterfaceDeclaration coid, Object arg) {
		super.visit(coid, arg);

		CompilationUnit cu = coid.findCompilationUnit().get();
		try {
			ConstructorUtil.addNewConstructor(paramNumber, coid);
		} catch (InvalidParameterNumber e1) {
			e1.printStackTrace();
		}

		String directory = DirCreatorUtil.buildDir(cu);
		DirCreatorUtil.createDir(directory);

		getTemplate(cu, directory, javaFileName + ".java");
	}

	@Override
	public void getTemplate(CompilationUnit cu, String directory, String
file) {
		TemplateModelWriter.getTemplateModel(cu).writeFileTo(directory,
file);
	}
```

}
Listing 10: Der Visitor „ConstructorVisitor"

```
public class TemplateModelWriter {

	Logger logger = Logger.getLogger(this.getClass().getSimpleName());

	private static TemplateModelWriter templateModel =
					new TemplateModelWriter();
	private Template template;

	private static Map<String, Object> compilationModel =
					new HashMap<String, Object>();
	private final String TEMPLATE_FILE = "compilationUnit.ftl";

	private TemplateModelWriter() {
		initTemplate();
	}

	public void initTemplate() {

		Configuration configuration =
				new Configuration(new Version(2, 3, 31));

		configuration
			.setIncompatibleImprovements(new Version(2, 3, 31));
		configuration.setDefaultEncoding("UTF-8");
		configuration.setLocale(Locale.GERMAN);
		configuration.setTemplateExceptionHandler(
				TemplateExceptionHandler.RETHROW_HANDLER);

		FileTemplateLoader templateLoader = null;

		try {
			templateLoader = new FileTemplateLoader(new
File("src/utils/template"));
			configuration.setTemplateLoader(templateLoader);

			template = configuration.getTemplate(TEMPLATE_FILE);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public static TemplateModelWriter getTemplateModel(
				CompilationUnit compilationUnit) {
		compilationModel.put("compilationUnit", compilationUnit);
		return templateModel;
	}

	public void writeFileTo(String dir, String clazzName) {

		Writer file = Writer.nullWriter();
```

```
                new File(dir).mkdirs();
                try {
                      file = new FileWriter(new File(dir+clazzName));
                } catch (IOException e) {
                      e.printStackTrace();
                }
                try {
                      template.process(compilationModel, file);
                } catch (TemplateException e) {
                      e.printStackTrace();
                } catch (IOException e) {
                      e.printStackTrace();
                }
                try {
                      file.flush();
                } catch (IOException e) {
                      e.printStackTrace();
                }
          }

}
```

Listing 11: Das Template-Modell: TemplateModelWriter

Betriebshandbuch

Systemvoraussetzungen

Um die Produktlinie-Anwendung in Betrieb zu nehmen, folgende Vorrausetzung müssen erfüllt werden:

- Java 17
- JUnit
- Eclipse IDE for Java Developers, Version 2020-06 (4.16)
- AspectJ Development Tools
- FeatureIDE 3.9.1
- Die Java Archive File (JAR) von
  - JavaParser
    - javaparser-core-3.24.2.jar
    - javaparser-symbol-solver-core-3.24.2.jar
  - FreeMarker
    - freemarker-gae-2.3.31.jar
- Plattform-Unterstützung:
  - Windows,
  - Mac,
  - Linux/GTK

Nachfolgend wird die Vorbereitung der Entwicklungsumgebung beschrieben.

Einrichten der Entwicklungsumgebung

Eclipse

Zuerst soll die Eclipse IDE mit der Version 2020-06 (4.16) installiert werden. Dazu bietet die offizielle Seite unter https://www.eclipse.org/downloads/packages/release/2020-09/r die Möglichkeit frühere Versionen herunterzuladen. Bei der Installation soll die Option Eclipse IDE for Java Developers gewählt werden.

AspectJ Development Tools (AJDT)

Mit der richtigen Version von Eclipse gestattet kann der AJDT mit dem Suchwort "AspectJ"gesucht werden. Dies wird in Eclipse unter: Help > Eclipse Marketplace gemacht. Nach diesem Schritt wird „AspectJ Development Tools" auf der Liste der Suchergebnis erscheinen und kann mit dem Button „install" installiert werden.

FeatureIDE

bietet sich drei Möglichkeiten FeatureIDE in Eclipse zu installieren.
Die erste Option erfolgt über den Eclipse Marketplace: Help > Eclipse Marketplace und nach FeatureIDE suchen.

Die zweite Variante wird mit „Update Site" heruntergeladen.

1. Eclipse öffnen und gehen zu:
   Hilfe > Neue Software installieren... (Eclipse 3.5 und älter: Hilfe > Software-Updates... > Verfügbare Software > Add Site...)

2. Eine der folgenden Update-Sites hinzufügen:
   - FeatureIDE v3.x: http://featureide.uni-ulm.de/update/v3/
   - FeatureIDE v2.6 und v2.7:
   http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy/
   - FeatureIDE v2.5 und frühere Versionen: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy/2.5/

- Nightly Builds: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/nightly/p2-updateSite/ (Tipp: Die Option Elemente nach Kategorie gruppieren im Update-Assistenten deaktivieren).

3. Die Features FeatureIDE, Feature Modeling und die benötigten FeatureIDE Erweiterungen auswählen.
4. Weiter mit Installieren...

Diese Schritte sind unter der Webseite FeatureIDE | An extensible framework for feature-oriented software development zu finden.

Die dritte Möglichkeit FeatureIDE zu installieren, umfasst die Installation von Eclipse, AJDT und FeatureIDE in einem Schritt. Es wird nicht nötig sein vorher Eclipse oder AJDT installiert zu haben. Siehe dazu die vorverpakten Versionen unter FeatureIDE | An extensible framework for feature-oriented software development.

Die Bibliotheken

Falls die Bibliotheken nicht mit dem Projekt mitgeliefert wurden, können die folgenden Anweisungen gefolgt werden.

JUnit
Die Anbindung von JUnit geht über Recht-Klick auf dem Projekt > Build Path > Add Librairies > JUnit.

Für die folgenden Bibliotheken werden die JAR-Dateien benötigt.
JavaParser

- javaparser-core-3.24.2.jar: https://mvnrepository.com/artifact/com.github.javaparser/javaparser-core/3.25.1
- avaparser-symbol-solver-core-3.24.2.jar https://mvnrepository.com/artifact/com.github.javaparser/javaparser-symbol-solver-core/3.25.1

FreeMarker
freemarker-gae-2.3.31.jar
https://mvnrepository.com/artifact/org.freemarker/freemarker-gae/2.3.32

Import und Ausführung der Anwendung

Nach der Einrichtung der Entwicklungsumgebung kann das Projekt in Eclipse importiert. In Eclipse über File > Import… und danach den Importvorgang folgen.

Falls die mitgelieferte Bibliothek (JavaParser, FreeMarker) schon installiert sind, kann die Anwendung wie folgt als JUnit Test ausgeführt werden.

1. Recht Klick auf dem Projekt in Eclipse
2. Run as
3. JUnit Test

Konfigurationswechsel
Die Anwendung besitzt drei Konfigurationsdateien. Der Wechsel der Konfiguration erfolgt über einen recht Klick auf die Datei. Danach öffnet sich ein Kontextmenü. Die Maus soll bis auf die Höhe der Zeile „FeatureIDE" bewegt werden. Es öffnet sich ein weiteres Menü und die Option „set as current configuration" soll ausgewählt werden.

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Ort, Datum
Brandenburg, 17.04.2023

Unterschrift
Mistra Forest Kuipou Tchiendja