Produktlinienübergreifende Wiederverwendung in einem modellgetriebenen Software-Entwicklungsprozess

Diplomarbeit im Fach Informatik

vorgelegt von

Christoph Elsner

geboren am 10. Oktober 1981 in Lichtenfels

Department Informatik
Lehrstuhl 4 (Verteilte Systeme und Betriebsysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat Dipl.-Inf. Daniel Lohmann
Beginn der Arbeit:	01.08.2007
Abgabe der Arbeit:	31.01.2008

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den

Christoph Elsner

Zusammenfassung

Software-Produktlinien ermöglichen es, mehrere ähnliche Endprodukte durch Komposition und Konfiguration einzelner Softwarebausteine effizient und parallel zu entwickeln. Die gemeinsame Verwendung der Softwarebausteine in unterschiedlichen Endprodukten macht hierbei den entscheidenden Synergieeffekt aus.
Haben wiederum verschiedene *Produktlinien* untereinander genügend Gemeinsamkeiten, kann die Wiederverwendung über Produktliniengrenzen hinweg zu weiteren, deutlichen Einsparungen im Entwicklungsaufwand der Produktlinien führen.
Im modellgetriebenen Kontext führt dies zunächst zu der Fragestellung, wie auch die Modellrepräsentationen und -transformationen sowie deren Einfluss auf die Artefakte eines Endprodukts wieder verwendet werden können.
Oft besteht auch der Bedarf, unter verschiedenen Produktlinien nicht nur die Gemeinsamkeiten, sondern auch ihre variablen Anteile sowie deren Konfigurierbarkeit wieder zu verwenden. Merkmalmodelle, die im Produktlinienkontext häufig Anwendung finden, verkörpern eine der möglichen Darstellungsformen dieser Variabilitäten.
Um Gemeinsamkeiten und Variabilitäten mit geringem Aufwand und trotzdem flexibel für mehrere Produktlinien nutzbar zu machen, wird im Rahmen dieser Arbeit das Konzept der *Produktlinienfamilien* vorgestellt. Mit dessen Hilfe ist es möglich, die zu teilenden Funktionalitäten unabhängig von den einzelnen Basisproduktlinien als so genannte *Produktlinienkomponenten* (*Product Line Components*, *PLiC*s) zu implementieren und weiterzuentwickeln.
Zunächst erfolgen Überlegungen, wie sich der Produktlinienfamilienansatz in den klassischen Referenzprozess für Produktlinienentwicklung einbetten lässt. Darauf aufbauend wird ein Realisierungskonzept entworfen. Der mehrschichtige Ansatz basiert auf der Erweiterung einer modellgetriebenen Entwicklungsumgebung namens *PLiC Framework*. Darauf aufbauend lassen sich Produktlinienfamilien entwerfen und implementieren, was hauptsächlich in der Definition einer Art gemeinsamer „Schnittstelle“ besteht, die alle Produktlinien der Familie zu implementieren haben. Von den für diese Familie entwickelten Produktlinienkomponenten können dann alle ihre Mitglieder gleichermaßen profitieren.
Mit einer prototypischen Implementierung des Konzepts wird daraufhin dessen praktische Umsetzbarkeit nachgewiesen. Eine bestehende modellgetriebene Entwicklungsumgebung wird zunächst um notwendige Fähigkeiten angereichert und auf deren Basis das PLiC Framework implementiert. Darauf aufbauend wird für eine bereits bestehende modellgetriebene Produktlinie eine Produktlinienfamilie definiert und die Entwicklung einer Beispiel-Produktlinienkomponente veranschaulicht.

Abstract

Software product lines allow the efficient development of many similar applications through composition and configuration of software modules. The reuse of modules among these applications represent the main advantage of this approach. If, in turn, several product lines have enough similarities, *inter-product-line reuse* of modules may help to reduce development efforts even more.
In the context of model-driven software development, this raises additional complexity, as models, model transformations, and code generators have to be reused as well. Furthermore, there may be needs for not only sharing but also the variability and configurability of product line artifacts. Some of these variabilities can even have a cross-cutting nature. Thus, an element on model level may influence several physical artifacts, and, vice versa, an artifact may be determined by more than one model element. Feature models, that are often used in product line context, only constitute one sort of variability representation. In model-driven software development, there are usually several models, each of them configuring the product line from a distinct viewpoint.
To reuse commonalities as well as variabilities in several model-driven product lines with reasonable efforts, this thesis presents the concept of *product line families*. It allows the disjoined development of diverse artifacts with reuse potential as so called *product line components* (*PLiC*s). The components can then be used in several of the family's product lines and thus may considerably increase productivity.
At first, the reference process of product line engineering is extended to support the development of product line families. Then a multi-layered approach is presented, which is based on the extension of a model-driven development environment called *PLiC Framework*. On top of this framework, product line families can be defined and implemented, which basically corresponds to the definition of a kind of "interface" that every product line of the family has to implement. All family members can then benefit from components which are developed for using this interface.
A prototype has been developed to prove the feasibility of the approach. An existing model-driven development environment is extended and the PLiC Framework is implemented on top of it. Then a product line family is defined for an already existing product line. Finally, a product line component is developed and integrated into the product line only by relying on the specified family interface.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Einführung

Aufgrund des hohen Bedarfs an leistungsfähiger Software sowohl für Arbeitsplatzrechner als auch für eingebettete Systeme stellt deren effiziente Entwicklung eine wichtige Voraussetzung für konkurrenzfähige Produkte dar.
Im Laufe der Jahre haben sich in der Softwaretechnik verschiedene Prinzipien als besonders leistungsfähig für diesen Zweck herausgestellt, wobei der Gedanke der systematischen *Wiederverwendung* häufig im Zentrum der Betrachtung steht. Weitere Prinzipien haben sich zu ihrer Ergänzung als besonders nÜtzlich erwiesen:

- Bei der *Abstraktion* werden bewusst Informationen verborgen, die in einer bestimmten Situation zur Erreichung eines bestimmten Zwecks als unerheblich gelten.
- Bei der *Dekomposition* hingegen geht es darum, bestehende Systeme in logisch abgetrennte und handhabbare Bestandteile zu zerlegen.

Die in dieser Arbeit angewandten softwaretechnischen Ansätze setzen diese drei Prinzipien alle auf verschiedene Weise und in unterschiedlichem Maße um. Dennoch werde ich sie zu ihrer Einführung vereinfachend entsprechend ihrem Schwerpunkt einem der genannten Prinzipien zuzuorden.

Wiederverwendung. Die *Produktlinienentwicklung* folgt dem Ansatz, ähnliche Softwareprodukte systematisch auf einer gemeinsamen Artefaktbasis zu entwickeln. durch die Wiederverwendung der Artefakte erhofft man sich hohe Produktivitätssteigerungen und Synergieeffekte, die durch die bisher von Unternehmen nach außen propagierten Erfolgsgeschichten [spl] auch bestätigt werden.

Abstraktion. Die *modellgetriebene Softwareentwicklung*, bei der Modelle als Elemente erster Klasse angesehen werden, setzt auf die Abstraktion vom Programmcode. Sie möchte damit zum einen die Automatisierung vieler repetitiver, manuell zu vollbringender Aufgaben erreichen. Zum anderen erhofft man sich durch die Darstellung als Modell Produktivitätsgewinne durch Verbergen von programmiersprachlichen Details, die es dann sogar Domänenexperten ermöglichen, Anwendungen zu erstellen, auch wenn sich ihr Fachgebiet nicht bis auf die Implementierungsebene erstreckt. Einer der Vorreiter auf diesem Gebiet ist die Steuergeräteentwicklung im Bereich Maschinen- und Automobilbau, in der sich die proprietäre Triade aus den Anwendungen MATLAB/Simulink[mat] und TargetLink [dsp] durchgesetzt hat.

Dekomposition. Dekompositionsverfahren stellen einen weiteren Teilbereich der Arbeit dar. Insbesondere soll es hierbei um die *komponentenbasierte Softwareentwicklung* gehen, bei der der Entwurf eines Systems üblicherweise durch dessen schrittweise Zerlegung in klare, durch explizite Schnittstellen voneinander abgegrenzte Bestandteile erfolgt. Dem Begriff Dekomposition kommt aber auch im Rahmen der aspektorientierten Softwareentwicklung Bedeutung zu. Diese erleichtert die Implementation gewisser Anforderungen, indem sie die Modularisierung von Funktionalitäten ermöglicht, die der klassisch objektorientierte Entwurf nicht gestattet. Aspektorientierte Softwareentwicklung wird im Grundlagenkapitel noch näher beschrieben.

1.2 Motivation und Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der Dekompositionsmöglichkeit von modellgetriebenen Produktlinien. Während der traditionelle Produktlinienansatz auf eine systematische Wiederverwendung von Artefakten innerhalb dieser Produktlinie abzielt (*intra-productline reuse*), soll es hier insbesondere um die Ermöglichungsbedingungen von *inter-productline reuse* gehen, also um die gemeinsame Verwendung von Artefakten über die Grenzen einer modellgetriebenen Produktlinie hinweg.
Bei der Entwicklung neuer Produktlinien wird häufig zwar nach systematischen Regeln vorgegangen, jedoch bleibt ihr Entwurf nicht selten ein Monolith, der sich, wenn einmal entwickelt, mit vertretbarem Aufwand kaum mehr in seine logischen Module zerlegen lässt. Dies gilt insbesondere für modellgetriebene Produktlinien, bei denen neben dem Anwendungscode auch die Modelle, Modelltransformatoren und Codegeneratoren zu den essentiellen Bestandteilen gehören.
Von bereits bestehenden modellgetriebenen Produktlinien kann daher bisher nur ad hoc und mit hohem manuellen Aufwand profitiert werden. Die Portierung einer Anwendungskomponente von der einen in die andere Produktlinie zieht dann diverse Änderungen von Modellen, Transformatoren und Generatoren mit sich. Je nach deren Entwurf in der Ursprungsproduktlinie macht dies eine manuelle Durchforstung diverser Artefakte nach den Anweisungen notwendig, die mit der zu portierenden Anwendungskomponente in Verbindung stehen.
Noch ungünstiger ist die Situation, wenn eine einmal portierte Anwendungskomponente in einer der Produktlinien weiterentwickelt oder verbessert wird, und dies sich ebenfalls bis auf die Modellebene erstreckt. Um davon zu profitieren zu können, müssen wiederum alle Änderungen manuell in die andere Produktlinie übertragen werden. Dies erzeugt einen erheblichen Zeit- und Kostenaufwand, der bei jeder neu einzuarbeitenden Änderung immer wieder auftritt.
Diese Arbeit beschäftigt sich mit der systematischen Wiederverwendung von Bestandteilen modellgetriebener Produktlinien, insbesondere von Modellen, Transformatoren, Generatoren und Anwendungscode. Bereits vom Beginn der Entwicklung einer Produktlinie werden daher die dafür nötigen Voraussetzungen, Entwurfspraktiken und die daraus entstehenden Konsequenzen beleuchtet. Nur so lässt sich sicherstellen, dass die Wiederverwendung planvoll abläuft und auch die Weiterentwicklung

von Produktlinienbestandteilen sich ohne manuelle Eingriffe auf alle beteiligten Produktlinien auswirken kann.
Das Ziel dieser Arbeit ist es schließlich, ein Konzept vorzustellen, das den besagten Anforderungen entspricht und für dieses einen konkreten Entwurf auszuarbeiten. Die prototypische Implementierung des Entwurfs für eine bestehende Produktlinie zeigt dann abschließend dessen Umsetzbarkeit.

1.3 Aufbau der Arbeit

Im Kapitel 2 wird zunächst auf die zum Verständnis dieser Arbeit nötigen **Grundlagen** eingegangen, namentlich Software-Produktlinien und modellgetriebene Softwareentwicklung.
Anschließend gibt Kapitel 3 einen **Überblick dieser Arbeit** und führt in die für sie neu entwickelte Terminologie ein.
Basierend auf den Anforderungen durch die **Anpassung des Referenzprozesses für Produktlinienentwicklung** in Kapitel 4 wird im Kapitel 5 ein Konzept für eine **Referenzarchitektur für Produktlinienfamilien** entworfen.
Ab Kapitel 6 erfolgt dann die **Implementation** der Architektur auf Basis einer modellgetriebenen Entwicklungsumgebung. Anhand eines Beispiels wird in Kapitel 10 die Erstellung einer neuen Produktlinienfamilie sowie die Implementation einer Produktlinienkomponente verdeutlicht.
Das Kapitel 11 geht schließlich auf themenverwandte Forschung ein, bewertet die Ergebnisse der Arbeit und gibt einen Ausblick für ihre mögliche Fortsetzung.

2 Grundlagen

Diese Arbeit stellt ein Konzept vor, mit dessen Hilfe Komponenten für mehrere modellgetriebene Produktlinien parallel entwickelt werden können. Dieses Kapitel enthält daher zunächst einen kurzen Überblick über die notwendigen Grundlagen und führt in die in der Arbeit verwendete Terminologie ein. Zunächst werden Software-Produktlinien und modellgetriebene Softwareentwicklung separat vorgestellt. Danach folgt eine Schilderung der Besonderheiten, die eine Verknüpfung der beiden Ansätze mit sich bringt.

2.1 Software-Produktlinien

Bei der Software-Produktlinienentwicklung wird versucht, auch im Standardsoftwarebereich die Produkte nach Kundenwünschen zu differenzieren. Dabei wird ein größtmögliches Maß an Wiederverwendung angestrebt, indem man die Gemeinsamkeiten und Unterschiede der Produkte zusammen in einer so genannten *Produktlinie* verwaltet.
Eine weit verbreitete Definition findet sich in [NC01, Seite 5]:

> „A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

Die in der Definition erwähnten Merkmale (*Features*) beziehen sich hierbei auf eine spezielle Domäne, den *Problemraum* (siehe Abbildung 2.1).
Unter dem Begriff *Domäne* wird nach [CE00] ein Bereich von Fachwissen verstanden, der für spezifische Anforderungen ausgewählt wird und dessen Konzepte und Begriffe durch Experten dieses Fachbereichs verstanden werden. Er schließt auch das Wissen ein, wie Softwaresysteme, oder Teile davon, für diesen Bereich zu erstellen sind.
Die bereits erwähnten Merkmale können sich auf funktionale oder nicht funktionale Eigenschaften beziehen. Die domänenspezifischen Merkmale stehen in Beziehung zu den im *Lösungsraum* zu erstellenden extensionalen Architektur- und Implementierungsartefakten. Eine Gruppe logisch zusammengehöriger Implementierungsartefakte wird im Folgenden auch als Anwendungskomponente bezeichnet.
Bisher wurde mit dem Problem und dem Lösungsraum nur die Modellebene betrachtet. Die Anwendungsentwicklung geschieht hingegen auf Instanzebene und erfolgt

im Idealfall allein durch Konfiguration einer Menge von Merkmalen und der Wiederverwendung der mit diesen in Relation stehenden Artefakte. Das daraus entstehende, lauffähige Endprodukt wird Variante genannt.

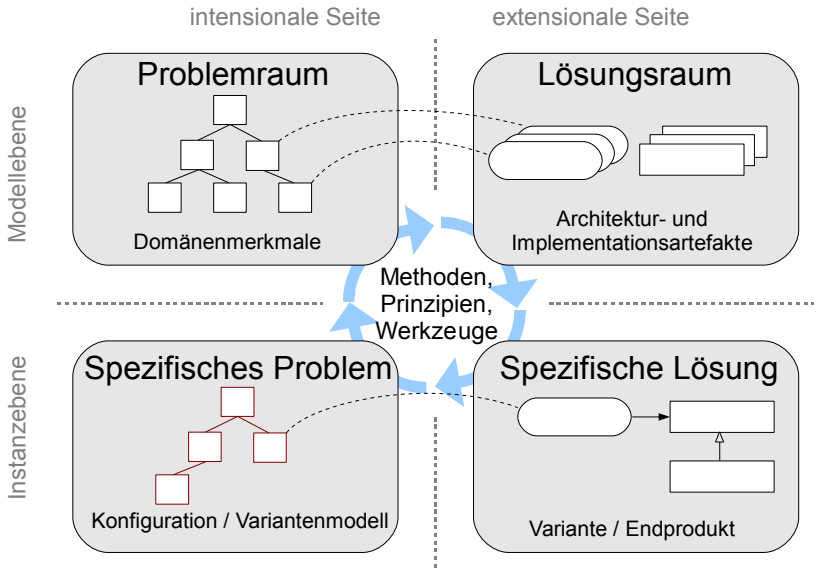


Abbildung 2.1: Problemraum-Lösungsraum-Modell für Software-Produktlinien

Durch die Wiederverwendung gemeinsam benötigter Artefakte ergeben sich für alle Phasen der Softwareentwicklung deutliche Einsparungspotentiale. Im Idealfall entsteht nur für solche Eigenschaften separater Implementierungsaufwand, in welchen die Endprodukte voneinander abweichen.
Das Synergiepotential durch Wiederverwendung hängt entscheidend davon ab, auf welche Weise die Merkmale auf Artefakte abgebildet werden können. Bei Merkmalen handelt es sich zunächst einmal nur um Anforderungen, die von den Implementierungen der Varianten umzusetzen sind. Die von unterschiedlicher Merkmalauswahl induzierte Variabilität der Endprodukte soll sich idealerweise in der gemeinsamen Softwarearchitektur der Endprodukte, der so genannten *Produktlinienarchitektur* widerspiegeln. In diesem Fall besteht dann die Erstellung einer Variante einfach nur in einer Auswahl an Artefakten. Häufig lässt es sich aber nicht vermeiden, dass ein Artefakt von mehreren Merkmalen beeinflusst wird. Dann ist es nötig, dass dieses je nach Merkmalauswahl entsprechend intern konfiguriert werden kann.

2.1.1 Referenzprozess der Produktlinienentwicklung

Der Entwicklungsprozess von Softwareproduktlinien ist im Gegensatz zum klassischen Softwareentwurf zweistufig. Zunächst erfolgt die Entwicklung der gemeinsamen und variablen Artefakte im *Domain Engineering*. Die Erstellung einzelner Produkte geschieht dann durch Wiederverwendung der Artefakte im zweiten Schritt (*Ap-*

plication Engineering). Beide Phasen sind wiederum dreistufig gegliedert in Analyse, Entwurf und Implementierung (siehe Abbildung 2.2). Da der Referenzprozess im Rahmen dieser Arbeit angepasst und erweitert wird, um Softwareproduktlinien*familien* zu unterstützen, wird er im Folgenden angelehnt an [BKPS04] tiefergehend beschrieben.

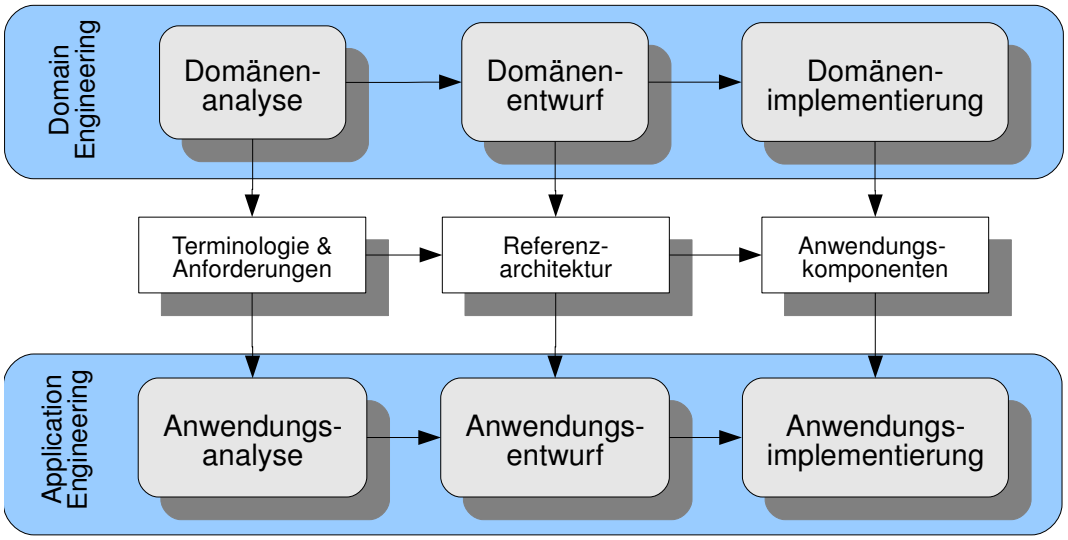


Abbildung 2.2: Der Referenzprozess für Software-Produktlinien-Entwicklung

2.1.1.1 Domain Engineering

Das **Domain Engineering** dient der Erstellung der Architektur- und Implementierungsartefakte und gliedert sich in die Phasen **Domänenanalyse, -entwurf und -implementierung**

Domänenanalyse. Die Domänenanalysephase beschäftigt sich mit der Erhebung aller domänenrelevanten Informationen und dient dazu, die Anforderungen an die Endprodukte und damit an die Produktlinie selbst zu erheben.

- Sammeln relevanter Informationen

  Zunächst ist das domänenspezifische Wissen zu sammeln und zu bündeln, beispielsweise durch Miteinbeziehung von Domänenexperten oder Analyse existierenden Programmcodes.

- Bestimmung und Abgrenzung der Domäne

  Anhand der Gesichtspunkte Kundenbedarf und Wiederverwendungspotential ist die „Breite“ und „Tiefe“ der Zieldomäne der Produktlinie genau festzulegen.

- Analyse der gemeinsamen und unterschiedlichen Merkmale

  Danach können die Einzelmerkmale der Endprodukte genauer beleuchtet werden. Hierbei sind insbesondere auch die Abhängigkeiten zwischen den Merkmalen zu dokumentieren, damit sich für den Kunden sinnvolle Merkmalkombinationen ergeben.

- Erstellung des Domänenmodells

  Beim Domänenmodell handelt es sich um eine formale und explizite Ausarbeitung der bisherigen Analysephase. In der **Domänendefinition** wird die Domäne textuell beschrieben und eingegrenzt, wobei bereits dort auf Merkmale und deren Abhängigkeiten eingegangen wird. Das **Domänenlexikon** definiert das Vokabular der Domäne. Diverse **Konzeptmodelle** beschreiben statische und dynamische Strukturen der Endprodukte (Klassen-, Zustands-, Interaktionsdiagramme, ...). Schließlich dokumentiert ein **Merkmalmodell** (siehe Abschnitt 2.3.1.1) die Menge aller gültigen Merkmalkombinationen.

Domänenentwurf. Der Domänenentwurf dient der Ableitung einer Anwendungsreferenzarchitektur, die sowohl den Anforderungen an die einzelnen Endprodukte genügt als auch die Wiederverwendung von Anwendungskomponenten innerhalb der Produktlinie unterstützt. Zudem wird hier der Prozess definiert, auf welche Weise im die Erstellung des Endprodukts zu erfolgen hat.

- Entwurf einer Anwendungsreferenzarchitektur
  - In der **Analysephase** werden zunächst die für die Modellierung der Anwendungsreferenzarchitektur notwendigen Informationen erhoben:
    - Zu Beginn wird untersucht, welche Produktanforderungen tatsächlich architekturrelevanten Einfluss haben, die so genannten *Architekturtreiber*. Dabei handelt es sich häufig um nicht-funktionale Eigenschaften wie z. B. Performanz oder Betriebssicherheit.
    - Die verschiedenen in Frage kommenden Architekturentwurfsmechanismen (Komponenten, Schichten, Model-View-Controller (MVC), ...) sind gegeneinander abzuwägen und in Hinblick auf die Vereinbarkeit mit den Architekturtreibern zu analysieren.
    - Schließlich lässt sich die Darstellung der Architektur anhand konkreter Sichten formulieren. Um bei der folgenden Modellierung alle Architekturtreiber im Blick zu behalten, kann für jeden eigens eine Sicht definiert werden, welche die für diesen Punkt entscheidenen Aspekte besonders heraushebt (statische Klassendiagramme, kommunikationsorientierte Ablaufdiagramme, ...).
  - In der **Modellierungsphase** sind dann die tatsächlichen Entwurfsentscheidungen zu treffen, und die Architektur ist schrittweise zu verfeinern und zu dokumentieren. Um die Architekturtreiber dabei nicht aus den Augen zu verlieren, helfen die im Vorfeld definierten Architektursichten.

  - Die anschließende **Evaluierungsphase** dient dazu, anhand von Szenarien die Einhaltung der Anforderungen insbesondere auch für kritische Anwendungsfälle zu überprüfen.

- Entwurf des Anwendungserstellungsprozesses

  Häufig wird davon ausgegangen, dass der Vorgang der Anwendungsentwicklung im Kern einer „Konfiguration der Merkmale“ entspricht [BKPS04]. Da dies einer Automation des Anwendungserstellungsprozesses gleichkommt, ist dieser bereits im Vorfeld entsprechend auszulegen. Die Automation kann auf verschiedene Weisen erfolgen. Das Produktlinienwerkzeug pure::variants [Beu03] beispielsweise ermöglicht die Generierung beliebiger Textdateien, also insbesondere auch Make-, Ant- oder IDE-Projektdateien. Zudem liefert es Unterstützung bei der Auswahl von Implementierungsartefakten und ermöglicht die Vorverarbeitung von Einzeldateien über Präprozessoren.

Domänenimplementierung. Die Implementierung der Domäne setzt die Architektur mit dem Fokus auf der Erstellung wieder verwendbarer Anwendungskomponenten um. Dabei können Techniken eingesetzt werden, die das Trennen der Belange in separate Implementierungsartefakte unterstützen, wie Aspektorientierung [KLM+97, KHH+01].

2.1.1.2 Application Engineering

Die Beschreibung der eigentlichen Anwendungsentwicklung orientiert sich an der Definition in [BKPS04, Seite 5]:

> „Im Application Engineering werden einzelne Produkte der Produktlinie entwickelt bzw. abgeleitet. Die Produkte werden so weit wie möglich aus den Artefakten der Plattform zusammengefügt (konfiguriert), so dass nur im geringen Maße produktspezifische Softwareentwicklung notwendig wird.“

Der folgende Ablauf geht vom Idealfall aus, dass die produktspezifischen Anforderungen an die zu erstellende Anwendung keinerlei Anpassungen oder Erweiterungen an den bereits zur Verfügung stehenden Artefakten notwendig machen.

Anwendungsanalyse. In der Anwendungsanalysephase werden die produktspezifischen Anforderungen des Kunden auf eine Merkmalkombination des Merkmalmodells der Produktliniendomäne abgebildet. Die konkrete Merkmalauswahl betreffende Artefakte wie Teile des Domänenlexikons oder Konzeptmodelle können wieder verwendet werden.

Anwendungsentwurf. Der Anwendungsentwurf besteht ebenfalls primär aus der Wiederverwendung von Entwurfsartefakten der gewählten Architektur. Im Idealfall ist bereits hier die Dokumentation so modular aufgebaut, dass der Kunde maßgeschneiderte Dokumente zur Verfügung gestellt bekommt.

Anwendungsimplementierung. Die Implementierung selbst erfolgt durch das Wiederverwenden und Konfigurieren der Anwendungskomponenten sowie schließlich der Übersetzung der Quellartefakte in eine maschinenausführbare Anwendung. Dieser Prozess sollte im Rahmen des Domänenentwurfs bereits so weitgehend spezifiziert sein, dass nur noch wenige Benutzereingriffe notwendig sind.

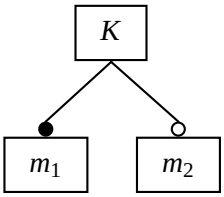
2.1.2 Beschreibung des Problemraums und der Varianten

Unter dem Problemraum versteht man die verschiedenen möglichen Anforderungskombinationen innerhalb einer Domäne. Er charakterisiert also die Variabilität der Merkmale innerhalb des angestrebten Marktes.
Um ihn zu beschreiben, können verschiedene Ansätze gewählt werden. Sie unterscheiden sich je nach Bedarf an Flexibilität und Verständlichkeit, wobei sie alle den Anspruch erheben, von reinen Domänenexperten verstanden werden zu können.
Während bereits eine Liste an Konfigurationsvariablen mit ihren Wertebereichen einen Problemraum aufspannen kann, sind komplexe Merkmalabhängigkeiten damit kaum auszudrücken. Für solche Fälle eignen sich sogenannte *Merkmalmodelle*, die einen Kompromiss zwischen Verständlichkeit und Mächtigkeit im Bezug auf die Formulierung von Abhängigkeiten darstellen.
In anderen Fällen geht es darum, ein System aus einer prinzipiell frei wählbaren Anzahl an Anwendungskomponenten mit spezifischen Eigenschaften zusammenzustellen. Durch die Erweiterung des Merkmalmodell-Konzepts [CHE05, CK05] lassen sich zwar auch solche Zusammenhänge ausdrücken, aus dem modellgetriebenen Umfeld wird für solche Fälle hingegen die Definition eigener Metamodelle sowie domänenspezifischer Sprachen vorgeschlagen [SV05], die in Abschnitt 2.2 noch näher erläutert werden.

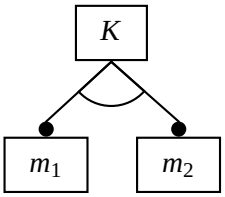
2.1.2.1 Merkmalmodelle

Dieser Abschnitt erklärt nun die Grundzüge, die zum Verständnis von Merkmalmodellen notwendig sind. Für eine ausführliche Darstellung des Themas siehe [KCH+90].

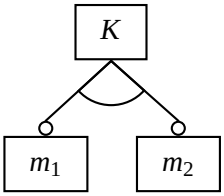
Merkmaldiagramm. Das Kernstück eines jeden Merkmalmodells ist das Merkmaldiagramm. Dabei handelt es sich um einen gerichteten azyklischen Graphen, üblicherweise ein Baum, an den von der Wurzel (dem *Konzept*) ausgehend die verschiedenen funktionalen und nicht-funktionalen Merkmale, Submerkmale usw. annotiert werden. Das Merkmalmodell spannt dabei einen Raum auf, der die gültigen Auswahlen an Merkmalskombinationen beschreibt. Die Kanten des Graphen haben dabei je nach ihrer Darstellung andere Bedeutung (siehe Abbildung 2.3).



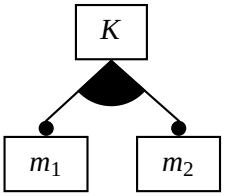
Das **Pflichtmerkmal** m_1 muss und das **optionale Merkmal** m_2 kann ausgewählt werden, falls ihr Elternmerkmal K gewählt ist.



Genau eines der **alternativen Merkmale** m_1 oder m_2 muss ausgewählt werden, falls K gewählt ist.



Höchstens eines der **optional alternativen Merkmale** m_1 oder m_2 darf ausgewählt werden, falls K gewählt ist.



Mindestens eines der **kumulativen Merkmale** m_1, m_2 muss ausgewählt werden, falls K gewählt ist.

Abbildung 2.3: Die Basiskonstrukte von Merkmaldiagrammen

Anhand dieser Basiskonstrukte lassen sich bereits aussagekräftige Merkmalsräume aufspannen (siehe Abbildung 2.4).

Da das Merkmaldiagramm einen ganzen Raum von Möglichkeiten aufspannt, stellt es eine Art Metamodell dar, das instantiiert werden kann, indem man sich für eine konkrete Kombination entscheidet. Das dadurch entstehende Modell wird nachfolgend **Variantenmodell** genannt.

Kompositionsregeln. Bereits die grafischen Basiskonstrukte von Merkmaldiagrammen stellen Kompositionsregeln dar, die festlegen, welche Merkmalkombinationen nach dem Konzept erlaubt sind und welche nicht. Komplexere Bedingungen können jedoch mit der bewusst einfach gehaltenen Syntax der Diagramme nicht mehr dargestellt werden. Für diesen Zweck müssen daher zusätzliche, mächtigere Sprachen verwendet werden, wie beispielsweise OCL [Obj03].

Merkmaldefinition. Für jedes Merkmal muss textuell genau festgelegt werden, was unter ihm zu verstehen ist.

Entscheidungsrichtlinien zur Merkmalauswahl. Schließlich ist es noch möglich, für jedes Merkmal Empfehlungen und Richtlinien anzugeben, um den Produktlinienkonfigurator bei seiner Entscheidungsfindung zu unterstützen.

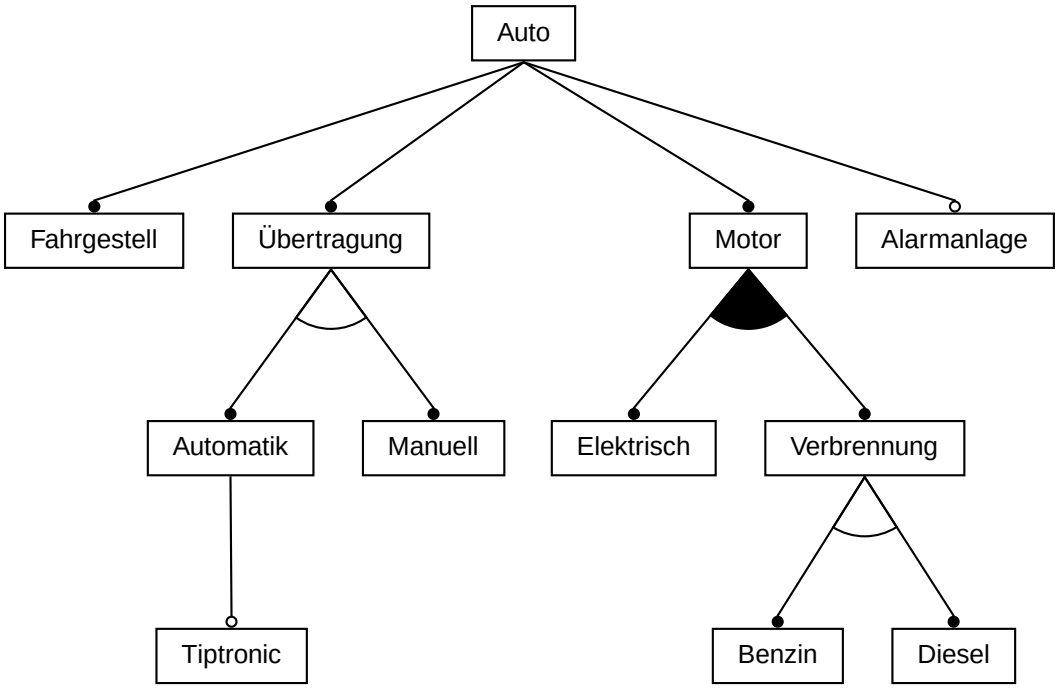


Abbildung 2.4: Merkmaldiagramm

2.2 Modellgetriebene Softwareentwicklung

Diese Arbeit orientiert sich an folgender Definition [SV05, Seite 11]:

> „Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."

Ein **formales Modell** hat im Gegensatz zu reinen „Modellen zur Dokumentation" eine eindeutige Syntax und Semantik. Das Modell beschreibt einen bestimmten Aspekt einer Anwendung also vollständig und interpretationsfrei [SV05]. Auf diese Weise kann es als Vorlage dafür dienen, automatisiert einen Teil des Anwendungscodes zu generieren.
Modelle stellen dabei bewusst nur einen bestimmten Ausschnitt der Anwendung dar, wie z. B. ein Daten- oder Komponentenmodell, und abstrahieren von den aus dieser Sicht unnötigen Details. Sie eignen sich insbesondere zur Modellierung von Gesichtspunkten, deren manuelle Programmierung repetitiv und dadurch fehleranfällig ist und garantieren durch den automatisierten Generierungsvorgang eine eindeutige und unverwässerte Architektur.

2.2.1 (Meta-)*Modelle

Modellierungsarchitekturen sind meistens, wenn auch nicht zwingend, vierstufig (siehe Abbildung 2.5, links). Die obersten drei Ebenen M1 bis M3 stellen Modelle dar, während die unterste Ebene M0 eine „physische“ Instanz des Modells der darüber liegenden Ebene M1 repräsentiert. Enthält die M1 Ebene beispielsweise das Modell einer Anwendung für ein Kassensystem, so befindet sich die Anwendung selbst auf der Ebene M0.

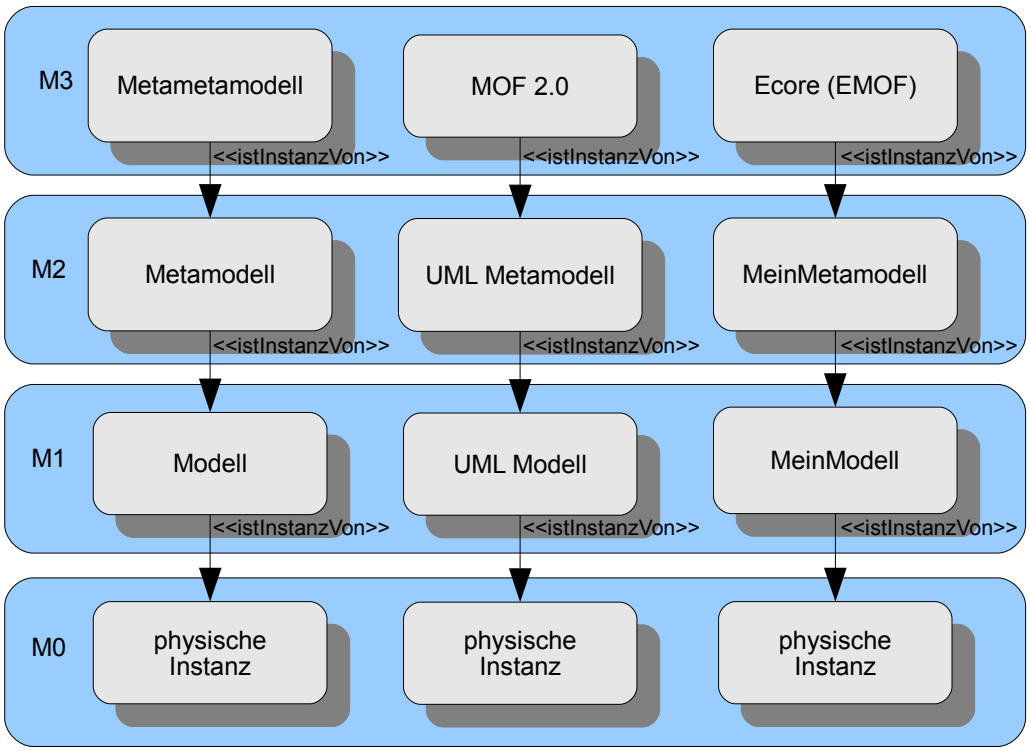


Abbildung 2.5: Hierarchie der Metamodellierung

Alle Modelle der Ebene M1 müssen nach bestimmten Regeln geformt sein, die das Metamodell auf Ebene M2 festlegt. Somit spannt ein M2-Modell eine Domäne auf, mit der sich beispielsweise alle Anwendungen für Kassensysteme beschreiben lassen. Da es für viele Interessengebiete den Bedarf an maßgeschneiderten (Domänen-)Metamodellen gibt, müssen auch dieses Metamodelle irgendwie definiert werden, was das Metametamodell auf M3-Ebene ermöglicht.
Während so prinzipiell beliebig hohe Hierarchien entwickelbar wären, ist mit dieser Stufe üblicherweise der Bedarf gedeckt. Das M3-Modell nutzt die Konzepte, die es *für* die M2-Schicht definiert *um* diese zu definieren und zieht sich damit sozusagen selbst aus dem Sumpf.
Es sei darauf hingewiesen, dass die Begriffe Metamodell, Modell und Instanz jeweils

relativ sind. Das Metamodell kann beispielsweise auch als Instanz des Metametamodells aufgefasst werden. Im Folgenden wird aber wenn immer möglich versucht, in dieser vierstufigen Hierarchie zu bleiben.
Das prominenteste Beispiel einer solchen Hierarchie ist die Meta Object Facility (MOF) [Obj02] in Zusammenspiel mit der UML [Obj07c, Obj07b] (siehe Abbildung 2.5, Mitte). Softwareentwickler erstellen in diesem Zusammenhang üblicherweise auf dem UML-Metamodell basierende M1-Modelle. Beim UML-Metamodell selbst handelt es sich im Prinzip um nichts anderes als ein Domänenmetamodell, dessen Entwicklungsfokus auf der möglichst kompletten Abdeckung aller Eventualitäten im Softwareentwicklungsbereich (und teilweise noch weiter) abzielt.
Neben MOF gibt es noch weitere Metamodellierungskonzepte. Das Eclipse Modeling Framework (EMF) [emf] beispielsweise bietet die Möglichkeit, mit der Metamodellierungssprache ECore recht einfach eigene domänenspezifische Metamodelle zu erstellen, die genau auf ein spezielles Anwendungsgebiet zugeschnitten sind.

2.2.2 Domänenspezifische Sprachen

Domänenspezifische Sprachen (DSLs) definieren neben einem domänenspezifischen Metamodell insbesondere noch eine konkrete Syntax. Mit deren Hilfe können dann mehr oder weniger komfortabel Instanzen der Metamodelle, also Modelle, spezifiziert werden. Die konkrete Syntax kann grafisch oder textuell sein und lässt sich somit als „Programmiersprache für eine Domäne“ interpretieren [SV05].

UML-basierte DSLs. Die UML in der Version 2.0 bietet im Wesentlichen zwei Mechanismen zur Spezifikation von DSLs: UML-Profile und die explizite Änderung des UML-Metamodells. Der zweitgenannte erfordert aufgrund der Komplexität des Metamodells Werkzeugunterstützung, die bisher erst in Entwicklung ist. UML-Profile hingegen sind leichter zu handhaben, arbeiten jedoch nur mit Stereotypen und sind daher nicht für Fälle geeignet, in denen komplett neue Metamodellelemente eingeführt werden sollen [FGDTS06].

Andere DSLs. Andere Konzepte zur Metamodellierung wie z. B. ECore, MetaGME [LMBea01] oder XMLSchema [ea00a] sind weniger ausdruckskräftig. Da es für viele Domänenkonzepte jedoch gar nicht des vollen UML-Umfangs bedarf und sie sich dadurch als deutlich leichter handhabbar erweisen, werden sie im MDSD-Umfeld dennoch häufig eingesetzt.

2.2.3 Der modellgetriebene Softwareerstellungsprozess

Für die Erstellung einer ausführbaren Anwendung auf Basis von Modellen gibt es keinen allgemein gültigen Referenzablauf. Je nach Bedarf kann die Anzahl der Modelle und Metamodelle, der Transformationen zwischen diesen und die Codegenerierung stark variieren.

Die OMG hat mit der Model Driven Architecture (MDA)[ea00b] einen umfangreichen MDSD-Standard definiert, der sich sehr gut zur Orientierung eignet und bereits viele nützliche Konzepte beinhaltet. Auch wenn die MDA nur für Metamodelle auf MOF-Basis und für UML-Profile spezifiziert ist, lässt sich über sie ein sehr guter Eindruck über die Möglichkeiten im modellbasierten Softwareentwurf gewinnen.
Die nächsten Abschnitte erklären die Themen Prozessablauf, Modelltransformationen und Codegenerierung zunächst aus MDA-Sicht, um dann auf alternative Ansätze und Besonderheiten hinzuweisen.

2.2.3.1 Prozessablauf

Die MDA geht von verschiedenen MOF-basierten Modelltypen aus, wobei die Modelle durch Transformationen immer mehr angereichert werden, bis im Idealfall eine komplett lauffähige Anwendung daraus generiert werden kann (siehe Abbildung 2.6).

- Computational Independent Business Model (Domänenmodell, CIM)

  Beim CIM handelt es sich um die implementierungsunabhängige Problemraumdarstellung der Anwendung. Es ist von der OMG aufgrund der Vielfalt möglicher Domänen nicht standardisiert. Dessen Vorhandensein jedoch ist ein, wenn auch nicht zwingend notwendiger, Bestandteil der MDA.

- Platform Independent Model (PIM)

  Das PIM stellt die durch Modelltransformation (siehe unten) aus dem CIM abgeleitete Anwendung auf Modellebene dar. Es bezieht sich sowohl auf die statische Anwendungsarchitektur als auch auf das dynamische Verhalten und abstrahiert dabei von plattformspezifischen Details. Hier sind Standardisierungsbestrebungen zur Beschreibung bestimmter Anwendungsgruppen über UML-Profile im Gange.

- Platform Specific Model (PSM)

  Das PSM ist eine mit plattformspezifischen Details angereicherte Verfeinerung des PIMs, z. B. für Corba oder J2EE. Die OMG schlägt vor, die Abbildung vom PIM auf das PSM über ein separates Platform Description Model (PDM) zu bewerkstelligen, das die Zielplattform in Metamodellform beschreibt. Auf dieses wird jedoch im Folgenden nicht näher eingegangen. Aus dem PSM wird dann schließlich der eigentliche Code generiert.

CIM, PIM und PSM sind nicht zwingend als Einzelmodelle aufzufassen. So kann jede dieser Schichten verschiedene Modelle und Modelltypen enthalten, die ein System aus unterschiedlichen *Viewpoints* beschreiben. So hat das PIM z. B. häufig einen statischen Gesichtspunkt, der die Artefakte wie Klassen oder Komponenten einer Anwendung beschreibt (*Anwendungsmodell*), sowie einen dynamischen Gesichtspunkt, der beschreibt, welche der Elemente auf welche Weise instantiiert werden sollen (*Instantiierungsmodell*).

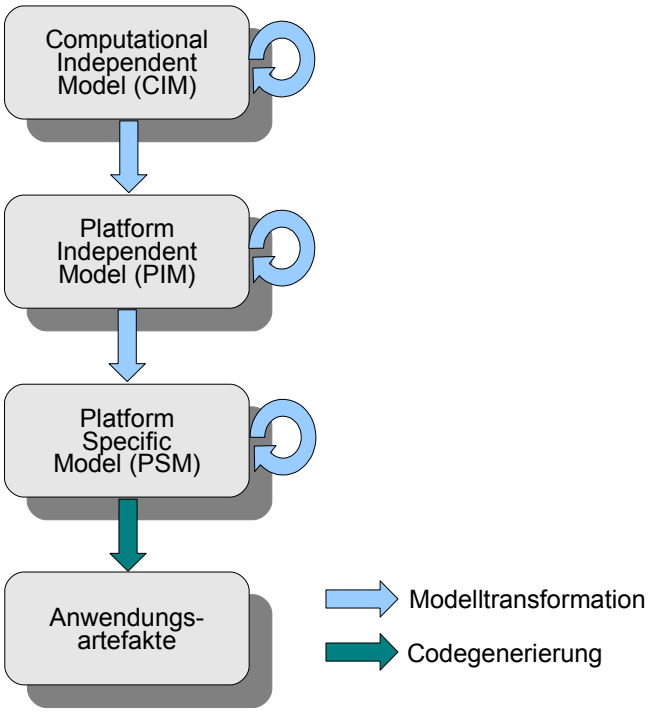


Abbildung 2.6: Die Model Driven Architecture

Auch müssen Transformationen nicht immer zwingend in eine tiefere Modellebene führen. So könnte beispielsweise das PIM verfeinert oder um bestimmte Modellelemente erweitert werden, ohne seine Plattformunabhängigkeit zu verlieren.
Der hier beschriebene komplexe Ablauf stellt ein sehr umfangreiches und mächtiges Konzept vor, das jedoch eine hohe Komplexität mit sich bringt. Durch Verzicht auf ein plattformunabhängiges Modell oder auf Domänenmodelle und durch bewusste Beschränkung auf einen Viewpoint pro Schicht kann der Ansatz, bei gleichzeitigem Kompatibilitätsverlust zur MDA, auf eigene Bedürfnisse zugeschneidert werden. Die MDA basiert zwingend auf der relativ umfangreichen MOF. Mit anderen, leichtgewichtigeren Alternativen wie ECore und MetaGME lässt sich daher keine MDA-Kompatibilität erreichen. Der eben dargestellte Prozessablauf lässt sich aber durchaus auch mit ihnen verwirklichen.
Die folgenden zwei Abschnitte beschreiben nun die verschiedenen Typen von Modelltransformationen und die verfügbaren Transformationssprachen.

2.2.3.2 Typen von Modelltransformationen

In [SV05, Seite 199 f] werden drei verschiedene Arten von Modelltransformationen unterschieden:

- Modellmodifikation

  Hierbei wird das Quellmodell selbst verändert, so dass dessen ursprüngliche Version überschrieben wird (Abbildung 2.7). Dies geschieht üblicherweise bei Verfeinerungen auf Modellebene, beispielweise bei einem Zustandsmodell durch Hinzufügen eines „Not-Aus“-Zustands und von Transitionen zu diesem von jedem weiteren Zustand aus. Da es sich somit nur um einen Seiteneffekt handelt, ist auch das Ergebnis vom selben Metamodell wie das Eingabemodell.

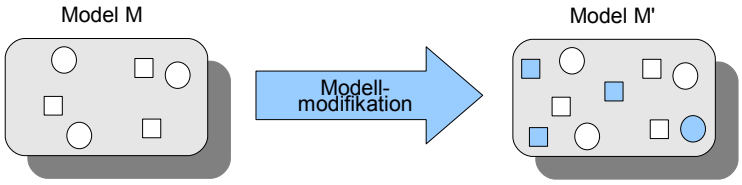


Abbildung 2.7: Modellmodifikation

- Modelltransformation

  Bei der Modelltransformation im engeren Sinne bleibt das Quellmodell unverändert und im Rahmen der Transformation wird ein neues Zielmodell erstellt (Abbildung 2.8). Üblicherweise besitzt dies ein anderes Metamodell, wie bei der Transition vom plattformunabhängigen ins plattformabhängige Modell bei der MDA. Im Unterschied zur Modellmodifikation muss in diesem Fall für jedes Modellelement, das im Zielmodell auftauchen soll, eine explizite Regel im Transformator bestehen, das es erzeugt.

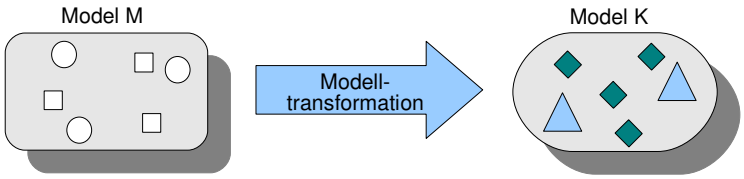


Abbildung 2.8: Modelltransformation

- Modellweben

  Im engeren Sinne versteht [SV05] unter Modellweben das reine „Verlinken“ mehrerer Quellmodelle, so dass Modellelemente auf Elemente des jeweils anderen Modells verweisen können (Abbildung 2.9, oben). Die Referenzierungs*fähigkeit* muss dabei explizit in den Metamodellen der Quellmodelle vorgesehen sein. Ist dies nicht der Fall, müssen diese zuvor durch Modifikation entsprechend abgeändert werden. Dadurch wachsen die Metamodelle zusammen und es bildet sich konzeptionell ein integriertes Metamodell.

In der Praxis ist der Begriff aber häufig weiter gefasst. So fallen darunter auch Modellmodifikationen und Modelltransformationen, wenn mehrere Quellmodelle zu einem Zielmodell verschmelzen (Abbildung 2.9, unten). Ein Beispiel hierfür sind so genannte *Mixin-Modelle*. Sie dienen als zusätzliches Quellmodell bei einer Modelltransformation und ermöglichen es, Modelle mit Implementierungsdetails wie beispielsweise über die Plattform anzureichern, die vorher aus Gründen der Abstraktion noch nicht in diesen enthalten waren.

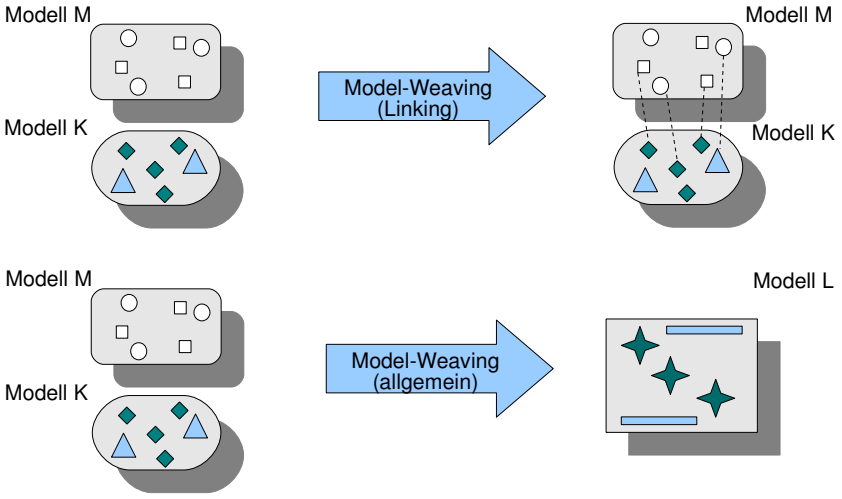


Abbildung 2.9: Modellweben

Hinweis Modelltransformationen sind nicht auf die M1-Ebene beschränkt. Auch Metamodelle lassen sich entsprechend obiger Verfahren transformieren, so der Bedarf besteht.

2.2.3.3 Sprachen für Modelltransformationen

Für Modelltransformationen gibt es viele unterschiedliche Sprachen mit stark unterschiedlichen Ansätzen. Daher kann diese Arbeit nur einen groben Abriss über dieses Thema geben. Nach einer Untersuchung [CH06] lassen sich die bestehenden Sprachen nach dutzenden verschiedener Merkmale kategorisieren. Im Folgenden werden daher nur drei verschiedene Ansätze beispielhaft vorgestellt.
Die Query/View/Transformation (QVT) [Obj07a] Language der MDA beinhaltet sowohl zwei deklarative (*Relations*, *Core*) als auch eine imperative Programmiersprache (*Operational Mappings*). Tatsächlich gibt es bisher aber kein einziges Werkzeug, das die Sprachen vollständig unterstützt. Teilweise tut dies z. B. bereits Borland mit dem Together Architect [bor] und France Telecom R&D mit SmartQVT [sma].
Andere Sprachen setzen auf so genannte Graphentransformationsregeln wie beispielsweise GReAT (Graph Rewriting and Transformation) [Agr03]. Sie basieren auf der Er-

setzung von bestimmten Graphmustern im Quellmodell. Dabei ist die konkrete Syntax der Transformationssprache üblicherweise ebenfalls ein Graph, was für Domänenexperten intuitiver sein kann.

Die MDSD-Umgebung openArchitectureWare (oAW) [oawb], die auch in dieser Arbeit verwendet wird, basiert wiederum auf einer funktionalen Modellmanipulationssprache namens XTend.

2.2.4 Codegenerierung aus Modellen

Enthalten die Modelle schließlich ausreichend Information, lassen sich aus ihnen Anwendungscode, Konfigurationsdateien, Deskriptoren und ähnliches generieren. All dies kann in verschiedenem Umfang geschehen. Von der alleinigen Generierung von programmiersprachlichen Schnittstellen über Zustandsmaschinen-implementierenden Code bis hin zur vollständigen Erstellung der Anwendung aus den Modellen ist alles vorstellbar.

Für den letztgenannten Fall sieht die UML Action Semantics [Obj01] vor, eine Erweiterung für OCL um Verhalten spezifizieren zu können. Es handelt sich hierbei um eine vollständig plattformunabhängige Programmiersprache, die prinzipiell die komplette Generierung der Anwendung aus Modellen erlaubt. Bisher mangelt es ihr jedoch noch an einer standardisierten konkreten Syntax und an Verwendung in größeren Projekten. Zudem ist noch nicht geklärt, in wie weit die erhofften Vorteile die befürchteten Nachteile wie mangelnde Effizienz zur Laufzeit und Unterstützung durch IDEs aufwiegen können.

Während in gewissen Bereichen wie im Elektrotechnik- und Maschinenbau-Umfeld die Modellierung und vollständige Codegenerierung oft mittels Software wie MATLAB [mat], Simulink [mat] und TargetLink [dsp] erfolgt, wird im allgemeinen IT- und Embedded-Bereich die eigentliche Funktionalität zum großen Teil noch von Hand programmiert. So bieten sich dem Programmierer die größtmögliche Flexibilität und ausgereifte Entwicklungstools (IDEs, Debugger, Bibliotheken, ...). Hier dient die Codegenerierung dazu, von repetitiven und fehlerträchtigen Aufgaben zu befreien und eine unverwässerte Systemarchitektur zu gewährleisten.

Die am häufigsten eingesetzten Sprachen zur Generierung von Code sind „Template-Sprachen“. Dabei werden die Befehle zur Texterzeugung in einfache Textdateien geschrieben. Bei Erzeugung der Zieldatei erfolgt dann die Ersetzung der Befehle durch dynamisch generierten Text. Im Web-Bereich werden textgenerierende Programmiersprachen wie JSP, ASP oder PHP seit langem eingesetzt. Sie sind häufig aber nur umständlich zu benutzen, wenn es um die Traversierung von Modellen auf Basis selbst entworfener Metamodelle geht. Daher eignen sich für solche Fälle speziell auf dieses Anwendungsfeld zugeschnittene „Template-Sprachen“ besser, wie JET aus dem Eclipse Modeling Framework und Xpand aus openArchitectureWare.

2.3 Modellgetriebene Produktlinienentwicklung

Durch die Implementation von Produktlinien mittels modellgetriebener Softwareentwicklung erhofft man sich, die Vorteile beider Ansätze zu verbinden. Während die Produktlinienentwicklung die Herangehensweise und Methodologie vorgibt, dient MDSD dann der Modellierung und Generierung des Endprodukts, also der eigentlichen Anwendungserstellung. Gleichwohl wirkt sich MDSD auf alle Phasen des Produktlinien-Referenzprozesses aus.
Die im Folgenden dargestellte Verschmelzung von MDSD und Referenzprozess ist im Rahmen dieser Arbeit entstanden, da bisher ausführliche Literatur zu diesem Thema fehlt. Sie hat daher keinen Allgemeingültigkeitsanspruch, sondern geht vornehmlich auf die Begrifflichkeiten und Zusammenhänge ein, die für spätere Kapitel benötigt werden.

2.3.1 Domänenanalyse

Bei der Domänenanalyse spielen sowohl Merkmal- als auch Konzeptmodelle eine Rolle.

2.3.1.1 Merkmalmodell

Beim Merkmalmodell handelt es sich bereits um eine Art Metamodell, dessen Instanz (das Variantenmodell) als Eingabeparameter des modellbasierten Anwendungserstellungsprozesses dienen kann. Es handelt sich nach MDA Terminologie dabei somit um ein CIM, also um ein Domänenmodell, aus dem durch schrittweise Verarbeitung mittels Transformationen dann letztendlich die Anwendung generiert wird.
Für Fälle, bei denen es um die *Konfiguration* globaler Eigenschaften eines Systems geht, eignen sich Merkmalmodelle sehr gut. Leider unterstützen sie klassischerweise keine multiple Instantiierung von Merkmalen. So ist es beispielsweise nicht möglich, eine beliebige Anzahl von Tasks eines eingebetteten Systems mit Hilfe einer Merkmalmodellinstanz zu konfigurieren, da die maximale Anzahl bei diesen jeweils von vornherein feststehen muss. In der Forschung existieren bereits Formalisierungen für Merkmalmodelle, die auch dies unterstützen [CHE05]. Jedoch stoßen auch diese erweiterten Merkmalmodelle an ihre Grenzen, wenn es beispielsweise darum geht, ein verteiltes System mit dessen gegenseitigen Referenzen zu konfigurieren.

2.3.1.2 Konzeptmodelle

Konzeptmodelle können auch weiterhin während der Analysephase alleinig zur Dokumentation von erwünschtem Verhalten oder zur Darstellung anderer konzeptioneller Aspekte der zu erstellenden Anwendung verwendet werden. Nachfolgend sollen aber nur solche Konzeptmodelle explizit betrachtet werden, die sich besonders für den Einsatz im modellgetriebenen Umfeld eigenen.

Insbesondere interessant ist daher der Fall, wenn Konzeptmodelle vom Anwendungsingenieur selbst erstellt werden können. Er bekommt damit eine weitere Möglichkeit, Einfluss auf die Anwendung zu nehmen.
Handelt es sich beispielsweise bei der Anwendungserstellung eher um eine *Konstruktion* als eine Konfiguration, gibt es geeignetere Modelltypen als Merkmalmodelle. Für solche Fälle von „struktureller Variabilität“ [GV07a] lassen sich im MDSD-Kontext mit relativ wenig Aufwand domänenspezifische Sprachen (also ein Metamodell mit zugehöriger konkreter Syntax) definieren, die genau auf den jeweiligen Anwendungsfall, beispielsweise die Struktur eines verteilten eingebetteten Systems, zugeschnitten sind. Solche domänenspezifischen Modelle repräsentieren dann die Anwendungsstruktur besser, als dies Variantenmodelle tun könnten, und sind somit besser als Ausgangsmodell für Transformationen geeignet.
Wenn im Folgenden von Konzeptmodellen oder -metamodellen gesprochen wird, sind primär solche, für die modellgetriebene Anwendungserstellung geeignete Modelle gemeint.

2.3.2 Domänenentwurf

Beim Domänenentwurf sind sowohl die Anwendungsreferenzarchitektur als auch die Definition des Erstellungsprozesses von Bedeutung.

2.3.2.1 Entwurf der Anwendungsreferenzarchitektur

Bei modellgetriebenen Produktlinien erfolgt die Erstellung der Anwendungsreferenzarchitektur im Gegensatz zum klassischen Anwendungsentwurf über formale Modelle. Das heißt, die Modellierung der Anwendung muss über eindeutig definierte Metamodelle geschehen.

Analysephase. Zunächst ist unter Berücksichtigung der Architekturtreiber der Architekturentwurfsmechanismus zu wählen, sich also z. B. für einen schichtenbasierten oder einen MVC-Entwurfsansatz für die Erstellung der Anwendung auf PIM-Ebene zu entscheiden. Dieser legt damit fest, wie die schrittweise Verfeinerung des Anwendungsentwurfs in der darauf folgenden Modellierungsphase abzulaufen hat.
Damit die eigentliche Anwendungsmodellierung dann formal vonstatten gehen kann, ist es nötig, bereits hier Metamodelle zur Darstellung der statischen und dynamischen Gesichtspunkte der Anwendung zu wählen oder zu entwerfen. An dieser Stelle empfiehlt sich das UML-Metamodell mit breiter Werkzeugunterstützung und mächtigen Ausdrucksmitteln, während einfache, selbst erstellte Metamodelle zur Anwendungsentwicklung sich durch höhere Flexibilität bei geringerer Komplexität auszeichnen.
Die Beschreibung der statischen Struktur einer Anwendung erfolgt über die Instanz eines *Anwendungsmetamodells*, während die Möglichkeiten, das dynamische Anwendungsverhalten zu definieren, im *Instantiierungsmetamodell* festgelegt sind.
Für den Fall, dass UML verwendet wird, entspricht das Anwendungsmetamodell einem Klassen- oder Komponentendiagramm. Entscheidet man sich hingegen dafür,

ein eigenes Anwendungsmetamodell aufzustellen, sind insbesondere die folgenden Punkte zu klären:

- Darstellung von Architekturelementen

  Architekturelemente wie Schichten und Subschichten oder die drei MVC-Hauptmodule können auf zwei verschiedene Arten repräsentiert werden.

  – Eine Möglichkeit ist es, für sie *explizite* Modellelemente im Metamodell vorzusehen, beispielsweise ein `MVCModel`, ein `MVCView` und eine `MVCController` Element, die dann beliebig andere Elemente enthalten können.
  – Des Weiteren kann die Dekomposition auch *implizit* anhand des Paketnamens eines Modellelements erfolgen. In diesem Fall deklariert dann der Paketname `layerX.sublayerY` die Zugehörigkeit der enthaltenen Elemente zu Subschicht Y in Schicht X.

  Während erstgenannte Möglichkeit semantisch ausdrucksstärker und eindeutiger ist, ermöglicht die andere eine einfachere Wiederverwendung des Metamodells für andere Anwendungsgebiete.

- Darstellung der atomaren Elemente

  Da in der anschließenden Modellierungsphase das Modell einer (plattformunabhängigen) Anwendung entstehen soll, ist ein in gewisser Weise „atomares Element“ nötig, das auf ein abgeschlossenes Programmierkonstrukt abgebildet werden kann. Bei feingranularer objektorientierter Modellierung benötigt das Metamodell beispielsweise ein klassenartiges Metamodellelement inklusive Methoden und Feldern. Erfolgt die Modellierung nicht ganz so detailliert, lässt sich dies durch komponentenartige Metamodellelemente ausdrücken, die eine bestimmte Anzahl Dienste zur Verfügung stellen. Da der komponentenbasierte Produktlinienentwurf durch seinen höheren Abstraktionsgrad viele Vorteile bietet, werden diese atomaren Elemente analog zu anderen modellorientierten Produktlinienentwurfsansätzen [Atk01] [SV05] im Folgenden als *Anwendungskomponenten* aufgefasst.

Die Modellelemente können in verschiedenen Beziehungen zueinander stehen. So stellt beispielsweise eine Modul-Hierarchie die tatsächlichen Aufrufbeziehungen dar, während eine funktionale Hierarchie die logischen Abhängigkeiten in den Vordergrund rückt [Par76]. Je nach Bedarf können beliebig viele dieser Beziehungstypen in einem selbst erstellten Metamodell integriert werden. Zumindest die Darstellung der Modul-Hierarchie sollte aber berücksichtigt werden, um sich ein Bild über die Abhängigkeiten im daraus entstehenden Code zu machen. Diese lassen sich dann auswerten, um eine konfigurierte Anwendung auf Konsistenz zu prüfen oder um beispielsweise Kapselungsklassen (*Wrapper*) für verteilte Systeme zu erzeugen.
Zu einer tatsächlichen Anwendung fehlt noch das dynamische Verhalten. Es basiert häufig auf den Spezifikationen, die in den CIM-Modellen vorgenommen wurden. Wurde in einem Konzeptmodell z. B. ein System von Sensoren, Steuergeräten und Aktoren eines verteilten Systems spezifiziert, bildet das Instantiierungsmodell dieses auf

konkrete zu instantiierende Anwendungskomponenten ab. Da das Anwendungsmodell selbst nur die statische Modulstruktur beschreibt, kann es solche dynamischen Gesichtspunkte nicht von sich aus abbilden. Zwei Alternativen bieten sich zu ihrer Modellierung:

- Instantiierung basierend auf dem Anwendungsmodell

  Die UML ermöglicht die Instantiierung der Anwendungskomponenten durch Objektdiagramme. Diese können in MDSD interpretiert werden um daraus Code zu generieren, der die entsprechenden Anwendungskomponenten (Klassen oder Komponenten) instantiiert. Das Instantiierungs*meta*modell entspricht in diesem Fall dem Anwendungsmodell.

  Dies gilt im Prinzip auch für selbst erstellte Anwendungsmetamodelle, so die Metamodellierungsumgebung dies unterstützt und die Metamodelle entsprechend formuliert sind. Hierbei ist anzumerken, dass somit praktisch eine weitere Schicht in die eigentlich vierstufige MDA-Hierarchie eingefügt wird. Da das Anwendungsmodell eigentlich auf der M1-Schicht angesiedelt ist, befindet sich das Objektdiagramm dann auf Ebene M0 und die tasächliche, in Ausführung befindliche Anwendung somit auf Ebene M-1 („M minus eins“).

- Instantiierung durch eigenes Instantiierungsmetamodell

  Um die Instantiierung von bestimmten Elementen zu bewirken, kann auch ein eigenes, separates Instantiierungsmetamodell erstellt werden. Dies ist insbesondere nötig, wenn die Modellierungsumgebung keine Objektmodelle wie die UML unterstützt.

  Es muss aber sichergestellt sein, dass das Metamodell ausreichend mächtig ist, um eine vollständige Instantiierung (inklusive einer möglichen Parametrisierung der Anwendungskomponenten) zu ermöglichen. Dafür ist es nötig, dass das Anwendungsmetamodell und das Instantiierungsmetamodell in einer gewissen Verbindung zueinander stehen. Nur so kann sichergestellt werden, dass eine Instanz genau so parametrisiert wird, wie es ihre „Schablone“ die Anwendungskomponente spezifiziert hat. In der Praxis ist dies entweder über Modellweben im strengeren Sinne oder mit physischer Zusammenlegung in ein einziges Metamodell zu erreichen.

Wenn im Folgenden von Instantiierungs(-meta)modellen gesprochen wird, umfasst dies beide der genannten Alternativen und dient nur der vereinfachenden Beschreibung.

Modellierungsphase. In der Modellierungsphase wird die Produktlinienanwendung entsprechend des vorher definierten Metamodells ausgearbeitet und schrittweise bis hinunter zu den atomaren Modellelementen verfeinert.

Evaluierungsphase. Da MDSD eine formale und eindeutige Modellierung verlangt, eignet sich das Anwendungsmodell sehr gut, um es nach Kennzahlen zu analysieren oder es durch Model-Checking-Verfahren zu verifizieren.

2.3.2.2 Entwurf des Anwendungserstellungsprozesses

Beim Entwurf des Anwendungserstellungsprozesses für modellgetriebene Produktlinien sind alle notwendigen Metamodelle auszuwählen oder zu entwerfen und der Ablauf der Transformatoren und Generatoren zu planen. Bildet man das Varianten- und ein Konzeptmodell auf die CIM- und das Anwendungs- und Instantiierungsmodell auf die PIM-Ebene ab, und fügt noch eine PSM-Schicht ein, so erhält man einen der MDA sehr ähnlichen Ablauf (siehe Abbildung 2.10).

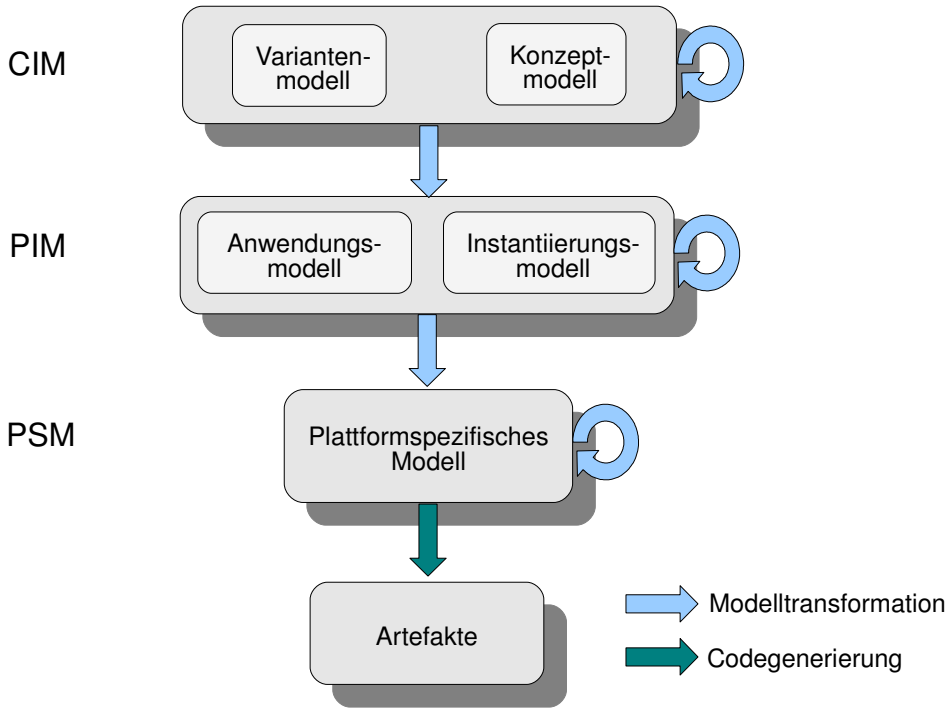


Abbildung 2.10: Möglicher Anwendungserstellungsprozess einer modellgetriebenen Produktlinie

Eine Besonderheit in dem Ablauf stellt das Variantenmodell dar. Prinzipiell lässt es sich wie jedes andere Domänenmodell behandeln. Einige MDSD-Umgebungen wie oAW bieten jedoch auch die Möglichkeit, die Ausführung ganzer Transformations- und Generatormodule von der Merkmalauswahl abhängig zu machen. Das Variantenmodell hat in diesem Fall eher den Charakter eines globalen Konfigurationshalters als eines gleichberechtigten Modells.

2.3.3 Domänenimplementierung

Während der Domänenimplementierungsphase erfolgt die programmiersprachliche Erstellung der eigentlichen Anwendungslogik sowie der Transformatoren und Generatoren.

2.3.3.1 Anwendungslogik

Zunächst sind die einzelnen Anwendungskomponenten des Anwendungsmodells zu implementieren. Dies kann eine weitere Verfeinerung der Modellelemente der Referenzarchitektur erforderlich machen und bis hin zur Spezifikation des Verhaltens über weitere Modelle oder Action Semantics gehen. In der Praxis wird das Kernverhalten der Anwendungskomponenten jedoch weiterhin häufig als programmiersprachlicher Code hinterlegt, z. B. in Java. In diesem Fall kann das bereits bestehende Anwendungsmodell zumindest zur Generierung von Codegerüsten und Basisfunktionalitäten verwendet werden.

2.3.3.2 Generatoren und Transformatoren

Das Verhalten der Generatoren und Transformatoren hängt vom Inhalt des Variantenmodells und der Konzeptmodelle ab, die als Eingabemodelle auf CIM-Ebene dienen. Geht man von einem MDA-artigen Ablauf wie in Abbildung 2.10 aus, sind folgende Module zu implementieren:

- Der **CIM-PIM-Transformator** besorgt die Übersetzung des Varianten- und der Konzeptmodelle in das Anwendungs- und Instantiierungsmodell.
- Der **PIM-PSM-Transformator** reichert das PIM an, damit im plattformspezifischen Modell ausreichend Informationen vorhanden sind, um daraus die Generierung der Anwendung zu ermöglichen.
- Der **Generator** wertet das PSM schließlich aus, um entsprechend der „Template-Dateien“ Code zu erstellen.
- Weitere Transformationen (z. B. **PIM-PIM** oder **PSM-PSM**) werden, falls Bedarf besteht, ebenfalls an dieser Stelle implementiert.

Von besonderer Bedeutung ist die vom Variantenmodell induzierte Variabilität, weil es sich dabei oftmals um Belange handelt, die klassisch-objektorientierte Architekturen quer schneiden. Diesem kann man auf der strukturellen Ebene durch Transformatoren und Generatoren oder auf der Implementierungsebene durch aspektorientierte Programmierung entgegentreten.

2.3.4 Application Engineering

Im Idealfall besteht das Anwendungserstellung einzig und allein darin, die Domänenmodelle (Variantenmodell und Konzeptmodelle) aufzustellen und den Erstellungs-

prozess in Gang zu setzen. Als dessen Ausgabe erhält der Anwendungsingenieur dann die komplett fertig gestellte Anwendung.

2.4 Zusammenfassung

Dieses Kapitel beleuchtete die Grundlagen, die zum weiteren Verständnis der Arbeit notwendig sind. In der Einführung in die Entwicklung von Software-Produktlinie grenzte es Problemraum und Lösungsraum voneinander ab und ging auf den Referenzprozess für Produktlinienentwicklung ein. Danach wurde der Aufbau von Merkmalmodellen erläutert.
Der anschließende Abschnitt behandelte die modellgetriebene Softwareentwicklung. Neben Grundbegriffen stand hier vor allem der modellgetriebene Anwendungserstellungsprozess mit Modelltransformationen und Codegenerierung im Vordergrund.
Schließlich wurde mit der modellbasierten Produktlinienentwicklung die Verknüpfung der beiden Ansätze vorgestellt. Während die Produktlinienentwicklung hier die Herangehensweise und Methodologie vorgibt, dient die modellgetriebene Softwareentwicklung der eigentlichen Erstellung des Endprodukts auf Basis von Modellen.

3 Die Arbeit im Überblick

Während das vorherige Kapitel bereits die nötigen Grundlagen aufgearbeitet hat, führt dieses nun in die spezielle Terminologie dieser Arbeit ein und bietet einen Überblick über deren weiteren Verlauf. Wie in Abschnitt 11.2 noch gezeigt werden wird, stellt die produktlinienübergreifende Wiederverwendung in der Forschung noch einen weißen Fleck dar, so dass neue Begriffe eingeführt werden:

> Unter **Produktlinienfamilien** verstehe ich eine Menge von Produktlinien, bei denen es trotz Unterschieden im Problemraum Wiederverwendungspotential im Lösungsraum sowie möglicherweise auch im Problemraum der Produktlinien gibt.
>
> Die innerhalb einer Produktlinienfamilie wieder verwendbaren Bestandteile nenne ich **Produktlinienkomponenten** .

3.1 Produktlinienfamilien

Der zentrale Gedanke von Produktlinienfamilien ist der, Gemeinsamkeiten zwischen Produktlinien auszunutzen, um doppelten Entwurfs- und Implementationsaufwand zu vermeiden und in allen Produktlinien direkt von Verbesserungen und Weiterentwicklungen profitieren zu können.

3.1.1 Arten von Produktlinienfamilien

Die Wiederverwendung bei *nicht-modellgetriebenen Produktlinienfamilien* ist aufgrund des häufig geringeren Anteils an generativer Codeerzeugung meist auf die Anwendungslogik beschränkt. Sobald sich eine Anwendungskomponente als geeignet herausgestellt hat, ist sie üblicherweise ad hoc und manuell in die andere Produktlinie zu integrieren.[1]
Bei *modellgetriebenen Produktlinienfamilien* ist einerseits ein deutlicher Mehraufwand nötig, da hier neben Implemetierungscode auch Modelle, Modelltransformationen und Codegeneratoren wieder verwendbar zu halten sind. Gerade wegen diesen besteht aber auch ein höherer Wiederverwendungsbedarf der zahlreichen Artefakte. Des Weiteren bietet MDSD eine größere Anzahl und abstrahiertere Angriffspunkte zur Einflussnahme auf die Anwendung. Änderungen der Anwendungsstruktur lassen

[1] Mir ist keine Arbeit bekannt, die einen solchen Fall systematisch aufarbeitet.

sich beispielsweise auf Modellebene sehr leicht ausdrücken, während in nicht modellgetriebenen Umgebungen der Anwendungscode jeweils an zahlreichen Stellen zu ändern wäre.
Gewisse hier vorgestellte Ansätze lassen sich auf alle Arten von Produktlinienfamilien verallgemeinern, insbesondere was die Domänenanalyse betrifft. Trotzdem beschäftigt sich diese Arbeit vornehmlich mit modellgetriebenen Produktlinienfamilien.

3.1.2 Gemeinsamkeiten innerhalb von Produktlinienfamilien

Die *Gemeinsamkeiten zwischen Produktlinien* können einerseits stark domänenspezifisch sein, so dass sie letztendlich durch tatsächliche Verwandtschaft der Domänen der Produktlinien bedingt sind. Wenn im Folgenden gemeinsame domänenspezifische Konzepte angenommen werden, dann jeweils nur in einem begrenzten Ausmaß, so dass eine Verschmelzung der Produktlinien trotz alledem nicht in Frage kommt.
Im Gegensatz dazu können die Produktlinien auch domänenübergreifende oder sogar domänenunabhängige Gemeinsamkeiten besitzen. Die Produktlinienkomponenten erfüllen in diesem Fall eher generische oder implementierungsspezifische Funktionen, und sind potentiell für eine Vielzahl von Domänen von Interesse, sofern sie auf ähnlichen Implementierungskonzepten aufbauen.
Der folgende Abschnitt beleuchtet daher die Kapselungseinheit der Gemeinsamkeiten, die Produktlinienkomponente, genauer.

3.2 Produktlinienkomponenten

Der Begriff Produktlinienkomponente ist umfassender zu verstehen als die alleinige, produktlinienübergreifende Wiederverwendung von Anwendungslogik. Vielmehr ist es so, dass eine Produktlinienkomponente auf die gesamte modellgetriebene Produktlinie und damit auf den gesamten Erstellungsprozess des Produkts Einfluss nehmen kann. Sie kann sich also auf Modelle, Transformatoren, Generatoren und im Code auf das Endprodukt auswirken.
Der Begriff Komponente stellt in diesem Zusammenhang klar, dass die Interaktion mit der Produktlinie über wohldefinierte Schnittstellen erfolgt. Dadurch, dass alle Mitglieder einer Produktlinienfamilie diese Schnittstellen implementieren müssen, wird die Wiederverwendung der Produktlinienkomponenten schließlich ermöglicht.

3.2.1 Der Komponentenbegriff

In dieser Arbeit wird zwischen Produktlinienkomponenten und Anwendungskomponenten unterschieden:

- Unter **Produktlinienkomponenten** verstehe ich eine umfangreiche Sammlung von Artefakten, die auf diverse Phasen des modellgetriebenen Produkterstellungsprozesses und damit letztendlich die Endprodukte selbst Einfluss nehmen

können. Ihr Wirkungsfeld erstreckt sich dabei auf Modelle, Transformatoren, Codegeneratoren und manuell implementierte Anwendungslogik.

Insbesondere sind sie auch in der Lage, auf die Ausdruckskraft der *Domänenmodelle* von Produktlinien Einfluss zu nehmen und für die von ihnen hinzugefügten Konzepte je nach konkreter Konfiguration spezifische Lösungen bereitzustellen. Daher erstreckt sich ihr Einfluss auf alle Bereiche des Problemraum-Lösungsraum-Modells, sowohl auf intensionaler als auch auf extensionaler Seite der Modell- und Instanzebene. Abbildung 3.1

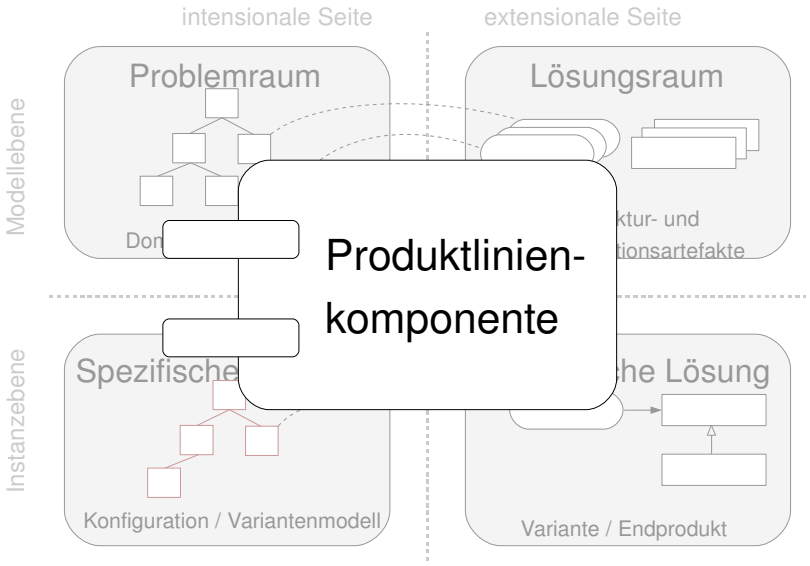


Abbildung 3.1: Problemraum-Lösungsraum-Modell mit Produktlinienkomponente

- **Anwendungskomponenten** hingegen stellen das Basiselement des Produktlinienlösungsraums da. Zu ihnen gehören sowohl die sie implementierenden Artefakte selbst als auch deren Beschreibung auf Modellebene. Eine Produktlinienkomponente *kann* eine Produktlinie mit weiteren Anwendungskomponenten anreichern, muss dies aber nicht. Diese Arbeit unterscheidet zwischen *variabilitäts- und gemeinsamkeitsbasierten Anwendungskomponenten*. Erstgenannte hängen von der Konfiguration der Domänenmodelle ab, zweitgenannte hingegen nicht. Die Unterscheidung hat zunächst die Bedeutung, dass für variabilitätsbasierte Komponenten ein Mechanismus vorgesehen werden muss, der für ihre Einbindung je nach Konfiguration zuständig ist, was bei den anderen nicht der Fall ist. Des Weiteren hat dies einen praktischen Vorteil, da gemeinsamkeitsbasierte Anwendungskomponenten wegen ihrer unkonfigurierbaren Natur während des Anwendungserstellungsprozesses nicht zwingend eine Modelldarstellung benötigen. Bei ihnen könnte es sich also beispielsweise um umfangreiche Bibliotheken oder kommerzielle Komponenten handeln. Auf die Modellierung zu verzichten

kann in diesen Fällen durchaus einen Effizienzgewinn bedeuten.

Falls es aus dem Zusammenhang klar hervorgeht, werden beide Komponentenarten, Produktlinien- und Anwendungskomponenten, vereinfachend als *Komponenten* bezeichnet.

3.2.2 Beispiele für Einsatzgebiete von Produktlinienkomponenten

Produktlinienkomponenten können Produktlinien und damit indirekt auch die Endprodukte auf verschiedenste Weisen beeinflussen. Je nach der Verwandtschaftsnähe ihrer Domänen ist auch die gemeinsame Nutzung von stark domänenspezifischen Bestandteilen möglich. Die Unterscheidung zwischen mobilen MP3- und Videoabspielgeräten, PDAs, Navigationsgeräten und Mobiltelefonen beispielsweise schwindet immer mehr, und so überschneiden sich auch ihre Domänen und es ergibt sich ein hohes Wiederverwendungspotential.
Im implementierungsbezogenen Bereich sind die Chancen auf Synergieeffekte noch weitaus höher. Alle Bestandteile mit infrastrukturellem Charakter, seien es Betriebssysteme, Middleware oder Bibliotheken, eignen sich gut dafür, wieder verwendet zu werden. Auch für quer schneidende und nicht funktionale Belange kann Wiederverwendungsbedarf entstehen, sowohl zur Laufzeit (Sicherheitsmechanismen, für „Angriffs-“ oder „Betriebssicherheit“) als auch zur Entwicklungszeit („Debugging-Infrastruktur“).

3.2.3 Dimensionen von Produktlinienkomponenten

Wichtig ist es, die neuen Dimensionen zu verstehen, die durch die Einführung von Produktlinienkomponenten erschlossen werden können. Klassische Wiederverwendung beschränkt sich auf die Übernahme funktionaler Gemeinsamkeiten durch die Übernahme von Anwendungskomponenten.
Produktlinienkomponenten sind jedoch weitaus mächtiger. So können sie über Modelltransformatoren neben der *Funktionalität* auch Einfluss auf die *Struktur* der bestehenden Anwendung nehmen. Außerdem müssen sich die Änderungen, die sie an der Produktlinie vornehmen, nicht zwingend auf alle Endprodukte gemeinsam auswirken. Durch Beeinflussung der Metamodelle auf Domänenebene können sie auf die Variabilität einer Produktlinie Einfluss nehmen.
In diesem Fall spreche ich von *variabilitätsbasierten Produktlinienkomponenten*, ansonsten von *gemeinsamkeitsbasierten Produktlinienkomponenten*. Schließlich kann der Einsatz einer solchen Komponente für die Basisproduktlinie *obligatorisch* oder *optional* sein, je nachdem ob sie für die Erstellung eines lauffähigen Endprodukts unbedingt nötig ist oder nicht. Abbildung 3.2 verdeutlicht die neu eingeführten Dimensionen.
In der Praxis kann es häufiger vorkommen, dass eine Produktlinienkomponente mehrere oder sogar alle der benannten Dimensionen abdeckt. Auch können Transformatoren funktionale Erweiterungen an einer Produktlinie vornehmen, beispielsweise durch Manipulation von Zustandsmodellen, aus denen später Anwendungscode generiert

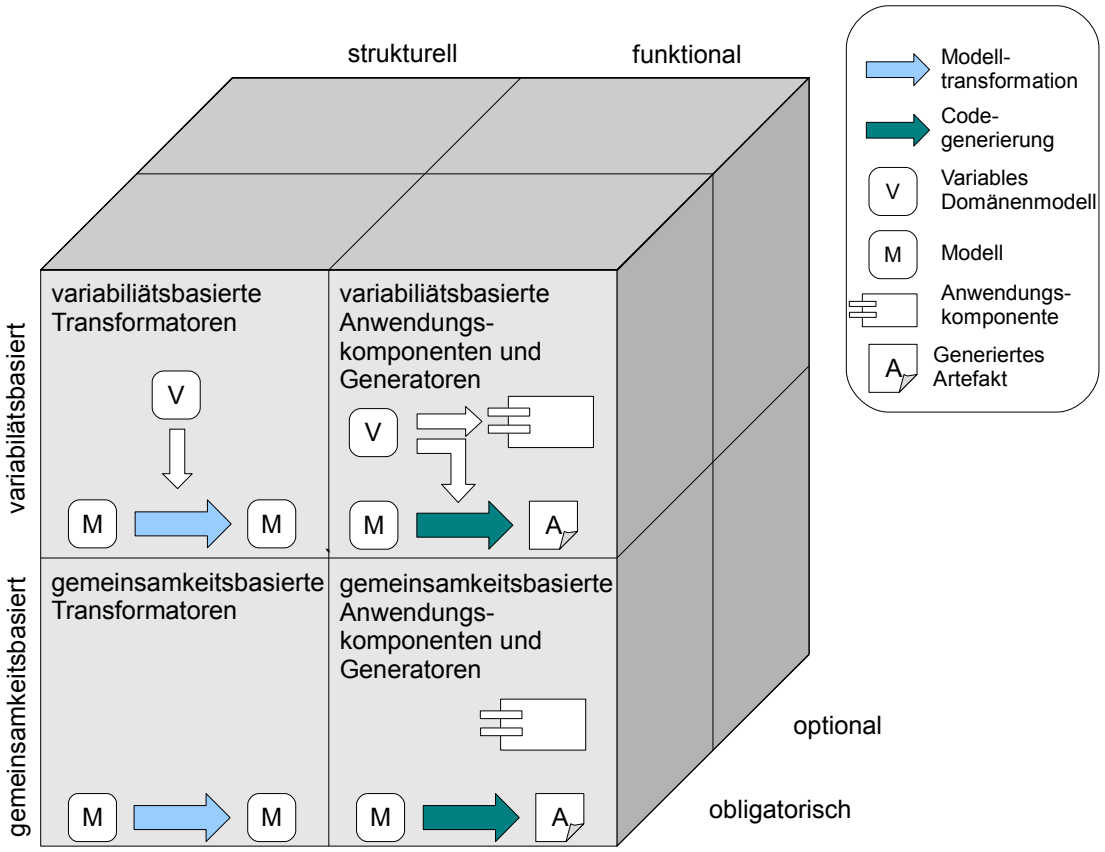


Abbildung 3.2: Dimensionen von Produktlinienkomponenten

wird. Nichtsdestoweniger ist die Einteilung aus konzeptioneller Sicht sinnvoll, um eine klare Vorstellung von den sich neu bietenden Möglichkeiten zu bekommen. Des Weiteren unterscheidet sich die Vorgehensweise in der Analyse-, Entwurfs- und Implementierungsphase, je nachdem, um welchen Produktlinienkomponententyp es sich handelt.

3.3 Überblick

Es stellt sich die Frage, unter welchen Umständen das Potential modellgetriebener Produktlinienfamilien ausgenutzt werden kann.
Prinzipiell ist es möglich, von der „grünen Wiese“ ausgehend, mehrere Produktlinien parallel zu planen, ihre Gemeinsamkeiten und Unterschiede auszuarbeiten, daraufhin die Produktlinienkomponenten zu identifizieren und schließlich die Komponenten und Produktlinien zu implementieren. Abgesehen davon, dass dieser Fall einer

parallelen Neuentwicklung *mehrerer* Produktlinien äußerst unwahrscheinlich ist, handelt es sich dabei um einen hochgradig komplexen Prozess, bei dem bereits während der Anforderungsanalyse schwierig zu fassende Abhängigkeiten auftauchen können. Um der Vielschichtigkeit des Produktlinienfamilienansatzes gerecht werden zu können, bieten sich daher iterative und inkrementelle Vorgehensweisen an. Dabei handelt es sich auch um eine der favorisierten Herangehensweisen in Forschung und Literatur. So werden in [SV05] iterative und inkrementelle Methoden bei der modellgetriebenen Softwareentwicklung bevorzugt und in [Gom05] wird die Architekturentwicklung speziell in Bezug auf modellgetriebene Software-Produktlinien ebenso als evolutionärer und hochiterativer Prozess betrachtet.
Die vorliegende Arbeit möchte daher dieses Verfahren, mit jeweils anderem Schwerpunkt, sowohl im direkt folgenden theoretisch orientierten Kapitel 4 als auch im abschließenden Implementationsteil ab Kapitel 6 anwenden. Die nachfolgenden Kapitel behandeln folgende Themen:

- Zunächst erfolgt in Kapitel 4 die **Anpassung des Referenzprozesses für Produktlinienentwicklung** in Hinblick auf die Erstellung einer Produktlinienfamilie. Dabei wird von dem konkreten Fall ausgegangen, dass *eine* Produktlinie von Grund auf neu geplant und entwickelt werden soll. Sie soll Produktlinienkomponenten unterstützen und damit die Ausgangsbasis für die Erstellung einer Produktlinienfamilie darstellen. Schwerpunktmäßig befasst sich das Kapitel mit der Domänenanalyse und dem Domänenentwurf, da dort die Weichen für eine erfolgreiche Domänenimplementierung und das „Applicaton Engineering“ gelegt werden.

- Das darauf folgende Kapitel 5 enthält die **Konzeption einer Referenzarchitektur für Produktlinienfamilien** . Dabei werden aus den bisherigen Erörterungen die zentralen Architekturtreiber extrahiert und daraus eine Architektur entwickelt, welche die besagten Anforderungen erfüllt.

- Ab Kapitel 6 erfolgt dann die **prototypische Implementierung** der Architektur auf Basis der modellgetriebenen Entwicklungsumgebung openArchitectureWare. Anhand der bereits bestehenden Produktlinie SmartHome wird in Kapitel 10 schließlich die Erstellung einer neuen Produktlinienfamilie sowie die Implementation einer Produktlinienkomponente verdeutlicht.

3.4 Zusammenfassung

Dieses Kapitel machte den Leser zunächst mit der für diese Arbeit neu entwickelten Terminologie bekannt. Die Begriffe Produktlinienfamilie und Produktlinienkomponente wurden eingeführt. Für zweitgenannte wurde eine Klassifikationsschema entwickelt, mit dessen Hilfe sie sich je nach ihren spezifischen Eigenheiten bestimmten Dimensionen zuordnen lassen. Schließlich bot das Kapitel einen Überblick über den weiteren Verlauf der Arbeit.

4 Anpassung des Referenzprozesses für Produktlinienentwicklung

Um planvolle, produktlinienübergreifende Wiederverwendung zu ermöglichen, ist es nötig, den Referenzprozess für Produktlinienentwicklung entsprechend anzupassen. Dabei wird explizit der Fall untersucht, dass eine Produktlinie von Grund auf neu zu entwickeln ist. Diese soll Produktlinienkomponenten unterstützen und damit die Ausgangsbasis für die Erstellung einer Produktlinienfamilie bilden.
Das „Application Engineering“ wird in diesem Kapitel nicht gesondert berücksichtigt, da sich für dieses keine relevanten Veränderungen ergeben. Die Domänenanalyse und der Domänenentwurf des „Domain Engineering“ hingegen spielen eine entscheidende Rolle und werden im Folgenden detailliert erörtert.

4.1 Domänenanalyse

In der Domänenanalyse lassen sich die potentiellen Kandidaten für Produktlinienkomponenten auf Basis der an die Produktlinie gerichteten Anforderungen finden. Je nachdem, ob es sich um *variable Merkmale* oder *globale Anforderungen* der Produktlinie handelt, lassen sich variabilitäts- und gemeinsamkeitsbasierte Produktlinienkomponenten identifizieren. Dieser Abschnitt untersucht insbesondere erstgenannte und ihren Einfluss auf die Erstellung des Domänenmodells.
Die klassische Domänenanalyse besteht, wie bereits in Abschnitt 2.1.1 erläutert, aus den vier Schritten:

1. Sammeln relevanter Informationen
2. Bestimmung und Abgrenzung der Domäne
3. Analyse gemeinsamer und unterschiedlicher Merkmale
4. Erstellung eines Domänenmodells

4.1.1 Sammeln relevanter Informationen

Die Informationssammlung unterscheidet sich nicht von der in der klassischen Produktlinienentwicklung, die bereits in Abschnitt 2.1.1.1 vorgestellt wurde.

4.1.2 Bestimmung und Abgrenzung der Domäne

Die Bestimmung und Abgrenzung der Produktliniendomäne kann ebenfalls analog zu Abschnitt 2.1.1.1 und damit unabhängig von der Dekompositionsabsicht geschehen.

4.1.3 Analyse der gemeinsamen und unterschiedlichen Merkmale

Hierbei ist zu analysieren, welche der *Merkmale* produktlinienübergreifendes Wiederverwendungspotential bieten. Natürlich gibt bei der Wiederverwendung vornehmlich die Codebasis den Ausschlag.[1] Am besten geeignet sind daher Merkmale, die relativ allgemeinen Charakter haben, sich also z. B. auf die technologische Platform des Endprodukts beziehen, wie Betriebssystemkomponenten oder Funktionsbibliotheken.

Aussichtsreiche Merkmale können nun Produktlinienkomponenten statt der Basisproduktlinie zugeordnet werden. Dabei sind Merkmale, die konzeptionell oder auf der Implementierungsebene zusammengehören, vorzugsweise der gleichen Komponente zuzuweisen.

Wenn die Merkmale ausreichend untersucht sind, können ihre Abhängigkeiten detailliert ausgearbeitet werden. An dieser Stelle erfolgt die Abhängigkeitsdefinition noch nicht formal. Es sei aber bereits darauf hingewiesen, dass produktlinienübergreifende Merkmalabhängigkeiten eine gewisse Komplexität einführen. Auch in Hinblick auf eine einfache Wiederverwendung in anderen Produktlinien sollten sie vorzugsweise gering ausfallen.

Nicht nur die variablen Merkmale, die nicht in jedem Produkt vorhanden sein müssen, eignen sich für Produktlinienkomponenten. Auch die funktionalen und nichtfunktionalen Anforderungen, die laut der klassischen Analysephase in *allen* Endprodukten umgesetzt werden *müssen*, können in so genannte gemeinsamkeitsbasierte Produktlinienkomponenten (siehe Abbildung 3.2) ausgelagert werden. Schließlich ist es auch möglich, eher implementierungsbezogene Artefakte wieder zu verwenden. Deren Identifikation kann erst zu einem späteren Zeitpunkt geschehen und erfolgt daher in der Domänenentwurfsphase.

4.1.4 Erstellen des Domänenmodells

Beim Domänenmodell handelt es sich um eine formale und explizite Ausarbeitung der Analysephase bestehend aus Domänenlexikon, Domänendefinition, Konzeptmodellen und Merkmaldiagramm. Da es als Basis für die Domänenentwurfsphase dient, kommt ihm eine entscheidende Bedeutung im Referenzprozess zu.

[1] Eine Ausnahme bilden hier rein strukturelle Produktlinienkomponenten (siehe Abbildung 3.2), die sich allein darauf beschränken, über Transformationen auf der Modellebene Änderungen durchzuführen.

4.1.4.1 Domänenlexikon

Das Domänenlexikon definiert alle in der Produktlinie erlaubten Fachbegriffe, was insbesondere *alle in Merkmaldiagrammen* verwendeten Begriffe mit einbezieht.
Geht man davon aus, dass die aktuelle Produktlinie sozusagen die Ausgangsbasis einer Produktlinienfamilie bilden soll, kommt dem Domänenlexikon familienweite Bedeutung zu. Auch der Domänenwortschatz später hinzukommender Produktlinien darf keine Doppeldeutigkeiten und Missinterpretationen zulassen. Es bietet sich daher an, ein *einziges* Lexikon für alle Familienmitglieder zu führen.
Falls es vorkommen sollte, dass die Domänen von zwei Produktlinien einen Begriff unterschiedlich definieren möchten, muss trotzdem sichergestellt sein, dass keine Komponente diesen missversteht. Da Abhängigkeiten zwischen Komponenten und Produktlinien auf Basis ihrer jeweiligen Merkmale bestehen können, wären Begriffsverwechslungen an dieser Stelle fatal.
Am einfachsten lässt sich dies über ein familienweites Domänenlexikon verhindern, das jede Begriffsdefinition einer bestimmten Menge von Produktlinien und Komponenten zuordnet. Begriffsdefinitionen können dann auch mehrfach vorkommen. Nur wenn sichergestellt ist, dass eine Produktlinie und eine Produktlinienkomponente keine Widersprüche im Wortschatz aufweisen, dürfen sie zusammenarbeiten.

4.1.4.2 Domänendefinition

Unter der Domänendefinition versteht man die textuelle Beschreibung des Geltungsbereichs der Domäne einer Produktlinie. Dabei besteht die Domäne der Produktlinie nun aus mehreren Teilen, nämlich aus ihrer Kerndomäne (ohne Produktlinienkomponenten) und denen der Produktlinienkomponenten.

Definition der Produktlinienkomponentendomänen. Die Domänen der Komponenten sind ausführlich textuell zu erläutern. Dazu gehören Einsatzbeispiele und Regeln, die bei der Auswahl der Komponentenmerkmale zu beachten sind.
Des Weiteren beschreibt die Komponente hier die Erwartungen, die sie an Merkmal- und Konzeptmetamodelle der Produktlinie stellt. Diese können sich zum einen auf die *Integrationsanforderungen* der Komponente in einer Produktlinie beziehen. Es wurde festgelegt, dass sich die Produktlinien prinzipiell beliebig in ihren Domänenmetamodellen unterscheiden können. Um dennoch keine unsinnigen Konfigurationen von Produktlinien und Komponenten zu erhalten, könnte eine Komponente in ihren Integrationsanforderungen beispielsweise die explizite Existenz eines bestimmten Merkmals fordern. Nur wenn dieses vorhanden ist, kann die Integration erfolgen.
Zum anderen muss eine bereits integrierte Produktlinienkomponente für die Auswahl ihrer Merkmale und Konzeptmodellelemente *Auswahlanforderungen* definieren. Sie muss also beispielsweise Bedingungen festlegen, welche Merkmale der Produktlinie gewählt sein müssen, um ein weiteres ihrer Merkmale auszuwählen.
Auch sind diverse andere, komplexere Bedingungen denkbar, beispielsweise das Vorhandensein bestimmter Modell- oder Metamodellelemente in Konzeptmodellen.

Einige der hier dargestellten Erwartungen lassen sich im späteren Teil der Domänenanalyse auch noch formalisieren, was die Definition von Merkmalabhängigkeiten angeht beispielsweise im Merkmalmodell. Bezüglich der Abhängigkeiten zu Konzeptmodellen und -metamodellen wären beispielsweise auch OCL-artige Ausdrücke vorstellbar. Die Integration einer Produktlinienkomponente jedoch allein aufgrund von formal definierten Abhängigkeiten auf Domänenebene zuzulassen oder abzulehnen scheint mir etwas zu abstrakt.
Aufgrund der Komplexität dieses Feldes sowie des Implementationsaufwands wird sich diese Arbeit im weiteren Verlauf nicht ausführlich mit formalen oder textuell formulierten Abhängigkeiten auseinandersetzen.

Definition der Domäne der Produktlinie. Diese Definition beschreibt die Gesamtdomäne der Produktlinie, indem sie ihre Kerndomäne erläutert und die verwendeten Produktlinienkomponenten spezifiziert. Die explizite Deklaration aller eingesetzten Produktlinienkomponenten wird gefordert, da es sich bei ihnen prinzipiell um beliebig invasive und umfangreiche Module handeln kann, die aktiv auf den Umfang der Domäne einwirken. Ihr Einsatz muss in einem solchen Fall wohlüberlegt sein, was nur durch eine frühe und explizite Berücksichtigung während der Domänenanalyse gewährleistet werden kann.
Insbesondere ermöglicht dieses Vorgehen auch den Einsatz von Produktlinienkomponenten, die einen integralen und obligatorischen Bestandteil der Gesamtproduktlinie ausmachen, wie z. B. das Betriebssystem (*obligatorische Produktlinienkomponenten*, siehe Abbildung 3.2). Für gewisse Komponenten, die beispielsweise nur optionale und implementierungslastige Funktionalitäten bereitstellen, mag das explizite Durchlaufen der Domänenanalyse auch entfallen. Nachfolgend wird davon jedoch vereinfachend ausgegangen und der genannte Spezialfall nicht näher berücksichtigt.

4.1.4.3 Konzeptmodelle

Eine variabilitätsbasierte Produktlinienkomponente hat zum Ziel, die tatsächliche Variabilität einer Produktlinie auf Domänenebene zu erweitern. Dafür hat sie zum einen das Mittel der Merkmalmodelle, das im Anschluss untersucht werden soll. An dieser Stelle soll es nun darum gehen, wie die Komponenten explizit Konzeptmetamodelle beeinflussen können.
Die Änderung des Metamodells wird notwendig, da dieses das *strukturelle Variabilitätspotential* beschreibt, das erweitert werden soll. Es wäre auch möglich, dass jede Komponente ein separates Metamodell zur Spezifikation ihrer strukturellen Variabilität mitbringt. Dies würde jedoch bei einer großen Anzahl Komponenten schnell unübersichtlich werden und für eine Referenzierung von Metamodellelementen der Basisproduktlinie dennoch Modellweben (zumindest im Sinne von „Model-Linking“) notwendig machen.
Bringt nun eine Produktlinienkomponente neue Konzeptmetamodellelemente mit oder möchte die bestehenden der Produktlinie erweitern, muss sie dies durch Metamodellweben oder -transformation tun. Beispielsweise ist es vorstellbar, dass eine Produktli-

nienkomponente bestimmte Metamodellelemente um ein zusätzliches Attribut erweitern möchte. Noch besser wäre eine Formalisierung solcher Erwartungen, die aber im Rahmen dieser Arbeit nicht näher untersucht werden soll.
Sie kann sich dabei aber nicht darauf verlassen, ein ganz bestimmtes Konzeptmetamodell anzutreffen, da diese je nach Produktlinie variieren können. Konkrete Erwartungen an das Metamodell sind daher zumindest textuell in die Domänendefinition der Produktlinienkomponente mit aufzunehmen.
Um die Erwartungen an das Metamodell der Basisproduktlinie möglichst gering halten zu können, ist es zudem sinnvoll, den Webevorgang parametrisieren zu können. Die Entscheidung, welche Metamodellelemente in einem konkreten Fall zu weben sind, kann dann auf den Komponentenintegrator[2] verlagert werden und muss nicht bereits während der Komponentenentwicklung getroffen werden.

4.1.4.4 Merkmalmodell

Im Abschnitt 2.1.1 wurden bereits Merkmalmodelle mit ihren vier Bestandteilen Merkmaldiagramm, Kompositionsregeln, Merkmaldefinitionen und Entscheidungsrichtlinien beschrieben. Nun erfolgt deren Anpassung an den Produktlinienfamilienansatz.

Merkmaldiagramm und Kompositionsregeln. Das Merkmaldiagramm ist als Kernstück des Merkmalmodells besonders wichtig zur Darstellung der möglichen Merkmalskombinationen einer Produktlinie. Für Produktlinienkomponente lassen sich analog ebenfalls Merkmaldiagramme erstellen. In beiden Fällen beschreiben die Merkmale die möglichen Konfigurationsoptionen. Die Struktur der Diagramme leitet sich dabei wie üblich aus der textuellen Analyse der Abhängigkeiten ab. Zur Formulierung bestimmter Bedingungen reicht jedoch die Diagrammsyntax nicht aus, weshalb auf Kompositionsregeln zurückgegriffen werden muss. Im Folgenden wird davon ausgegangen, dass Merkmaldiagramme nur eine grafisch darstellbare Untermenge von Kompositionsregeln bilden, die man verallgemeinernd als *Merkmalabhängigkeiten* betrachten kann.
Die besondere Schwierigkeit liegt in der Definition von *produktlinienübergreifenden Merkmalabhängigkeiten*, also solchen zwischen Produktlinien und Komponenten oder von Komponenten untereinander. Wie bereits im Paragraph über die Domänendefinition bemerkt, sind zwei Ebenen der Abhängigkeiten zu betrachten: die Integrationsanforderungen bezüglich der Merkmalexistenz und die fachlichen Anforderungen bezüglich der Merkmalauswahl. Ein mögliches Konzept hierfür wäre die Einführung von *Merkmaldeklarationen*.[3] Es wird nachfolgend anhand von Abhängigkeiten von Produktlinienkomponenten zu Produktlinien erläutert. Jedoch sind Merkmalabhängigkeiten genauso gut in der anderen Richtung oder zwischen Komponenten vorstellbar,

[2] Der Komponentenintegrator muss entsprechend dem betrachteten iterativen Entwicklungsprozess beim Integrieren einer Produktlinienkomponente die Domänenanalysephase durchlaufen. So ist sichergestellt, dass er genügend Domänenwissen hat, um die Integration korrekt zu parametrisieren.
[3] Unter Merkmaldeklaration verstehe ich eine Art von „Merkmalschnittstelle“, die übliche Merkmaldefinitionen „implementieren“ können.

wobei diese Fälle sich ganz analog behandeln lassen.
Angenommen, eine Komponente referenziert bei der Definition ihrer Abhängigkeiten eine Merkmaldeklaration. Nur wenn eine Produktlinie diese Deklaration dann „implementiert", kann eine *Integration* der Komponente überhaupt stattfinden. Die *Auswahl* eines bereits integrierten Produktlinienkomponentenmerkmals wiederum hängt davon ab, ob die Implementation der Deklaration, auf die es sich bezieht, ausgewählt ist oder nicht.
Die gemeinsame Darstellung solcher von unterschiedlichen Produktlinienkomponenten und Produktlinien definierten Abhängigkeiten kann relativ einfach über Kompositionsregeln geschehen. Eine einfache Konjugation aller Regelsätze reicht aus. Für die Darstellung der Merkmal*diagramme* gibt es hingegen mehrere Alternativen. Diese hängen davon ab, wie die multiplen Merkmal*modelle* und die auf ihnen basierenden Variantenmodelle der Produktlinie und ihrer Komponenten physisch miteinander in Verbindung stehen:

- Die Merkmal- und Variantenmodelle der Produktlinie und der in sie integrierten Komponenten können vollständig getrennt voneinander gehalten werden. Hier kann jedoch bei der Konfiguration vieler Produktlinienkomponenten leicht die Übersicht verloren gehen.

- Variantenmodelle könnten auch die Instanzen mehrere Merkmalmodelle, die durch Merkmaldeklarationen verknüpft sind, gleichzeitig darstellen. Obwohl die Merkmalmodelle weiterhin physisch getrennt sind, werden mit ensprechender Werkzeugunterstützung im Variantenmodell *alle* Merkmale und Merkmalabhängigkeiten berücksichtigt.[4]

- Des weiteren ist eine physische Verschmelzung der Merkmalmodelle zu einem einzigen vorstellbar. Das Variantenmodell kann dann auf dessen Basis ohne weitere Zusatzmechanismen instantiiert werden.

Im Folgenden soll insbesondere der drittgenannte Fall einer expliziten Komposition von Merkmalmodellen untersucht werden. Diese ermöglicht im Merkmaldiagramm eine übersichtlichere und kompaktere Darstellung als separate Diagramme und erlaubt es, auch in Umgebungen zu arbeiten, die keine multiple Merkmalmodellinstantiierung unterstützen.[5]
Zwei Alternativen bieten sich für die Darstellung von Merkmaldiagrammen, je nachdem, ob Merkmaldeklarationen unterstützt werden oder nicht.

- Verschmelzung der Konzeptknoten

  Unter der Annahme, das keine grafisch modellierbaren Abhängigkeiten zwischen den Merkmalen von Produktlinien und Komponenten bestehen oder keine Merkmaldeklarationen unterstützt werden, können einfach die Konzeptknoten der Diagramme zusammengelegt werden (siehe Abbildung 4.1).

[4] Mir ist kein Werkzeug bekannt, das diese Fähigkeit besitzt.
[5] Dies ist bei openArchitectureWare, auf dem der Implementierungsteil dieser Arbeit basiert, der Fall.

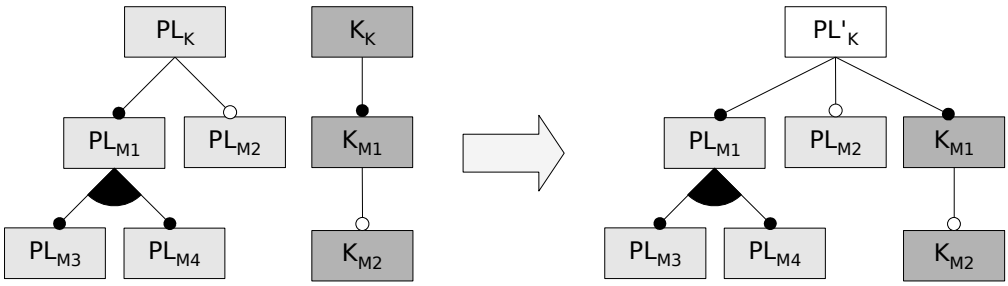


Abbildung 4.1: Verschmelzung der Konzeptknoten

- Verschmelzung über Merkmaldeklarationen

  Um produktlinienübergreifende Abhängigkeiten grafisch im Zieldiagramm darzustellen, kann die Produktlinienkomponente bei der Definition ihres Merkmaldiagramms Merkmaldeklarationen verwenden. Bei der Verschmelzung der Merkmaldiagramme können dann alle Subbäume des Komponentenmerkmaldiagramms, deren Wurzel eine Merkmaldeklaration bildet, an die Stelle im Merkmaldiagramm der Produktlinie abgebildet werden, an dem sich dessen Implementation befindet.

  Abbildung 4.2 zeigt die Verschmelzung eines Merkmaldiagramms mit einem Teilbaum, dessen Wurzel die Merkmaldeklaration K_{MDekl} darstellt. Das Merkmaldiagramm „implementiert“ mit dem Merkmal PL_{MImpl} die Deklaration. Der Teilbaum unterhalb der Merkmaldeklaration kann daher an dieser Stelle im Merkmaldiagramm eingefügt werden.

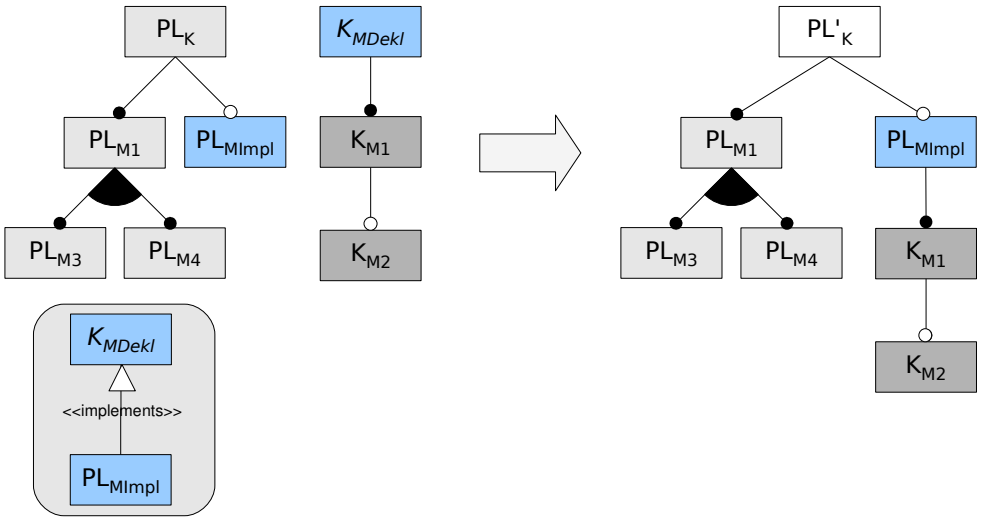


Abbildung 4.2: Verschmelzung bei Merkmaldeklarationen

Hinweis

Merkmaldeklarationen und damit die Spezifikation produktlinienübergreifender Abhängigkeiten konnten aus Zeitgründen und wegen Nutzung des proprietären Merkmalmodellierungswerkzeugs pure::variants in der Implementierung nicht umgesetzt werden. Für den Implementierungsteil wurde daher die einfachere Lösung eines Merkmalmodellwebers gewählt, der sich auf die Verschmelzung der Konzeptknoten beschränkt.

Merkmaldefinitionen. Die genauen Definitionen der Merkmale können für jedes Merkmal separat betrachtet werden, unabhängig davon, ob sich um ein Merkmal, eine Produktlinie oder eine Komponente handelt. Praktikabel erscheint es, die Definition des Merkmals mit der in der Domänendefinition konsistent zu halten oder diese sogar explizit zu referenzieren.

Entscheidungsrichtlinien zur Merkmalauswahl. Die Richtlinien, die bei der Entscheidung zur Auswahl eines Merkmals unterstützen, können entweder textuell oder durch formale Regeln angegeben werden. Während der erstgenannte Fall keine größeren Probleme darstellt, handelt es sich bei zweitgenanntem quasi um „nicht zwingende“ Kompositionsregeln und können ähnlich wie diese behandelt werden.[6]

4.2 Domänenentwurf

In der Domänenentwurfsphase entstehen zwei Erzeugnisse: zum einen eine **Anwendungsreferenzarchitektur,** auf der alle Endprodukte basieren können und zum anderen der **Anwendungserstellungsprozess,** der das „Application Engineering“ im Idealfall nahezu vollständig automatisiert. Die Herausforderung liegt nun darin, diese beiden so bedacht zu planen und zu entwickeln, dass die Bedürfnisse der Basisproduktlinie und Komponenten sowie die Bedürfnisse *zukünftiger* Produktlinien und Komponenten so weit wie möglich abgedeckt sind.
Aufbauend auf Abschnitt 2.3 werden nun vornehmlich die Grundvoraussetzungen erörtert, die für einen erfolgreichen Produktlinienfamilienentwurf nötig sind. Die eigentliche Konzeption einer Referenzarchitektur für Produktlinienfamilien erfolgt im anschließenden Kapitel 5.

4.2.1 Entwurf der Anwendungsreferenzarchitektur

Hierbei ist für die gesamte Produktlinienfamilie zunächst ein Anwendungs- und Instantiierungs*meta*modell aufzustellen oder festzulegen. Diese müssen vielseitig genug

[6] Werkzeuge wie pure::variants gehen genau so vor.

sein, um die Architektur der Basisproduktlinie, genauso wie die zukünftiger Produktlinien und Komponenten, beschreiben zu können. Anschließend ist dann die Anwendungsarchitektur, also das Anwendungsmodell selbst, der Basisproduktlinie und ihrer Komponenten auszuarbeiten und zu evaluieren.

4.2.1.1 Analysephase

Die Analysephase dient primär der Wahl des Architekturentwurfsmechanismus. Es ist dabei nicht erforderlich, dass alle Produktlinien einer Familie nach exakt dem gleichen Entwurfsmechanismus vorgehen. Dieser kann weiterhin von den Architekturtreibern der jeweiligen Produktlinien bestimmt sein. Eine der entscheidenden Voraussetzungen für den Einsatz von Produktlinienfamilien und -komponenten im modellgetriebenen Umfeld hingegen sind jedoch gemeinsame Anwendungs- und Instantiierungs*meta*modelle, da nur so die Integrationsfähigkeit sichergestellt werden kann.
Ein gemeinschaftliches **Anwendungsmetamodell** ist dabei nicht so unwahrscheinlich oder undurchführbar, wie es auf den ersten Blick erscheinen mag. Tatsächlich ist das Anwendungsdesign aufbauend auf dem UML-Metamodell weit verbreitet, und es besteht oft kein Bedarf, dessen Ausdruckskraft zu erweitern. Eher im Gegenteil kann in gewissen Fällen der Wunsch bestehen, sich bewusst nur auf Kernkonzepte beschränken zu wollen, was durch den Entwurf eines eigenen Anwendungsmetamodells geschehen kann. Dieses ist dann jedoch mit Bedacht zu wählen, da sich nachträgliche Änderungen an diesem auf alle Produktlinien und Komponenten auswirken.
Entscheidet man sich, wie in Abschnitt 2.3.2 bereits erörtert, im eigenen Metamodell für explizite Architekturelemente, z. B. für ein schichtenbasiertes Design, wirkt sich dies auf die Endprodukte aller Familienmitglieder aus. Es kann sich daher lohnen, die makroskopischen Architekturelemente nur implizit anhand von Paketnamen zu spezifizieren, was zwar eine geringere Ausdruckskraft hat, jedoch die Wiederverwendbarkeit des Metamodells erhöht.
Das **Instantiierungsmetamodell** bezieht sich auf das Anwendungsmetamodell, da es letztendlich die resultierende Konfiguration der Anwendungskomponenten zur Laufzeit beschreibt. In Abschnitt 2.3.2 wurden bereits die dabei möglichen Alternativen zur Darstellung von Instantiierungsmodellen vorgestellt: die Instantiierung der Anwendungsmodelle selbst durch Objektdiagramme oder über explizite Instantiierungsmetamodelle, die nur mit dem Anwendungsmetamodell „verlinkt“ sind.

4.2.1.2 Modellierungsphase

In der Modellierungsphase erfolgt die schrittweise Verfeinerung der Produktlinie und ihrer Komponenten entsprechend ihres Architekturentwurfsmechanismus auf Basis des Anwendungsmetamodells. Dieser Abschnitt geht vereinfachend davon aus, dass der Anwendungsentwurf komponentenbasiert erfolgt. Die Modellierungseinheit wird daher als Anwendungskomponente bezeichnet. Das Ergebnis dieser Phase ist eine auf Anwendungskomponenten basierende Anwendungsreferenzarchitektur.

Identifikation weiterer Produktlinienkomponenten. Bereits während der Domänenanalyse wurden die Merkmale der variabilitätsbasierten Produktlinienkomponenten erkannt und ausgearbeitet. Auch einige der gemeinsamkeitsbasierten Produktlinienkomponenten konnten anhand der globalen Anforderungen an die Produktlinie identifiziert werden.
Während der Modellierung der Anwendungskomponenten lassen sich zudem noch weitere Produktlinienkomponenten finden. In Abschnitt 3.2 wurden bereits einige der hierfür aussichtsreichen Kandidaten wie Betriebssysteme, Middleware oder Funktionsbibliotheken genannt. Ist ihr Wiederverwendungspotential groß genug, können sie in separate Produktlinienkomponenten ausgelagert werden. Bei den genannten Beispielen wird es sich üblicherweise um obligatorische Produktlinienkomponenten handeln, ohne die eine Produktlinie keine funktionsfähigen Endprodukte erzeugen kann.

Modellierung der Anwendungskomponenten. Die auszumodellierenden Anwendungskomponenten, die gemeinsam die Referenzarchitektur der Produktlinie bilden werden, lassen sich je nach ihrer Zugehörigkeit unterscheiden:

- Für die **Produktlinie** selbst erfolgt die Modellierung ihrer Anwendungskomponenten wie gehabt entsprechend ihres Architekturentwurfsmechanismus unter Berücksichtigung der Architekturtreiber.

- Für die **Produktlinienkomponenten** erfolgt die Ausmodellierung analog, wobei die Anwendungskomponenten insbesondere in Hinblick auf eine gute Wiederverwendbarkeit zu entwerfen sind. Im Normalfall werden gemeinsamkeitsbasierte Anwendungskomponenten auf gemeinsamkeitsbasierte Produktlinienkomponenten abgebildet, entsprechendes gilt für variabilitätsbasierte Komponenten. Dies ist aber keine zwingende Regel. Zum einen ist es möglich, dass auch variabilitätsbasierte Produktlinienkomponenten einen gewissen Anteil an gemeinsamkeitsbasierten Anwendungskomponenten mitbringen. Zum anderen ist vorstellbar, dass gemeinsamkeitsbasierte Produktlinienkomponenten, die also keine eigene Variabilität mitbringen, die Merkmale der Basisproduktlinie auswerten und so variabilitätsbasierte Anwendungskomponenten mit sich führen können. In diesem Fall muss die Produktlinienkomponente sicherstellen, dass das Merkmal auch tatsächlich bei der Produktlinie vorhanden ist. Dies kann über die textuelle Beschreibung als Integrationsvoraussetzung in der Domänendefinition geschehen oder über Merkmaldeklarationen (siehe Abschnitt 4.1.4.4).

Schnittstellen. Eine wichtige Rolle spielen die Schnittstellen, an denen die Anwendungskomponenten der Produktlinienkomponenten an die der Produktlinie andocken können. Es handelt sich hier um ein ähnliches Problem, wie es in jeder Produktlinie aufgrund von der je nach Merkmalauswahl unterschiedlichen Kombination von Anwendungskomponenten auftritt. Die möglichst lose Kopplung zwischen den variablen Anwendungskomponenten stellt für Produktlinienfamilien ein wichtiges Krite-

rium dar, wobei für die unterschiedlichsten Anwendungsfälle Entwurfsmuster bereitstehen [GHJV95], beispielsweise „Broker“ oder „Factories“.
Bei der komponentenbasierten Softwareentwicklung werden üblicherweise programmiersprachliche Schnittstellen verwendet, um von der konkreten Implementation einer Anwendungskomponente zu abstrahieren. Entsprechend benötigen diese auch ein Modellelement im Anwendungsmodell zur Repräsentation. Eine Interaktion zwischen zwei Anwendungskomponenten kann immer nur dann stattfinden, wenn sie sich auf das gleiche Schnittstellen-Modellelement beziehen. Das Modellelement kann schließlich während des Anwendungserstellungsprozesses dazu verwendet werden, programmiersprachliche Schnittstellen zu generieren.

4.2.1.3 Evaluierungsphase

Zusätzlich zur Evaluierung des Entwurfs der gesamten Produktlinie können die Komponenten nochmals separat überprüft werden. Neben den durch die Produktlinie induzierten Architekturtreibern ist dabei insbesondere ihre leichte Wiederverwendbarkeit ein zentrales Bewertungskriterium.

4.2.2 Entwurf des Anwendungserstellungsprozesses

In Abschnitt 2.3.2 wurde bereits auf die Besonderheiten beim Entwurf des Anwendungserstellungsprozesses für modellgetriebene Produktlinien eingegangen. Er beinhaltet insbesonders die Auswahl bzw. den Entwurf aller noch nicht festgelegten Metamodelle und die Planung der Transformationen zwischen diesen. Auch hier müssen sowohl die aktuelle Produktlinie und deren Komponenten wie zukünftige Familienmitglieder berücksichtigt werden.
Damit Produktlinienkomponenten auf unterschiedliche Produktlinien wirken können, ist zunächst zu entscheiden, welchen Schritte des modellgetriebenen Prozesses alle Familienmitglieder vollziehen sollen. Damit wird gleichzeitig eine Schnittstelle definiert, auf der die Produktlinienkomponenten aufsetzen können.
Im Folgenden wird dabei von der in Abbildung 2.10 dargestellten Ablaufstruktur ausgegangen. Die Anlehnung an die MDA stellt jedoch kaum eine Einschränkung dar, da sie sowohl mehrere Metamodelle und Modelle pro Ebene als auch ebeneninterne Transformationen unterstützt. Auf der anderen Seite können sich auch modellgetriebene Produktlinienfamilien, die eigentlich keinen so komplexen Prozess benötigen, durch Weglassen einiger Modelle und Ebenen an diesem Ablauf orientieren.

4.2.2.1 Metamodelle

Die Metamodelle der unterschiedlichen Ebenen spielen für die Integrationsfähigkeit innerhalb einer Produktlinienfamilie eine entscheidende Rolle, da die Transformatoren und Generatoren auf deren Basis arbeiten.

Metamodelle auf CIM-Ebene. Bereits bei der in Abschnitt 4.1.4 geschilderten Domänenmodellerstellung wurde darauf eingegangen, dass Produktlinien einer Familie prinzipiell beliebige Merkmal- und Konzeptmetamodelle haben können, mit denen die Produktlinienkomponenten zurecht kommen müssen. Auch wurden dort bereits Webeverfahren vorgeschlagen, die es den Komponenten trotz dieser Ungewissheit ermöglichen, die Domänen einer Produktlinie tatsächlich zu erweitern.

Metamodelle auf PIM-Ebene. Das Anwendungs- und das Instantiierungsmetamodell stellen, wie bereits erläutert, einen integralen Bestandteil des Produktlinienfamilienansatzes dar. Sie müssen daher für alle Produktlinien und Komponenten gleich sein.

Metamodelle auf PSM-Ebene. Es kann je nach Bedarf beliebig viele Plattformen und somit PSMs geben, sowohl innerhalb einer Produktlinienfamilie als auch innerhalb einer einzigen Produktlinie. Bei MDSD geht man häufig davon aus, dass der PIM-PSM-Transformator weitgehend automatisiert arbeitet, und dass Plattformhersteller diesen für einschlägige Kombinationen wie UML zu J2EE mitliefern. Solange eine Produktlinienkomponente nicht aktiv in die Generierung des Plattformcodes eingreifen will, ist dieser für sie transparent.[7]

4.2.2.2 Planung der Transformationen

Die zu planenden Transformationen ergeben sich zunächst aus den Übergängen zwischen den gewählten Metamodellen. Zusätzlich können auch noch innerhalb einer der MDA-Ebenen Transformationen stattfinden, um Modelle zu modifizieren oder schrittweise mit Informationen anzureichern. Hierbei gilt, dass alle Transformationen, auf die Produktlinien einwirken können sollen, explizit für die gesamte Produktlinienfamilie gelten müssen. Der Transformationsablauf stellt somit eine zu implementierende „Schnittstelle“ dar, an der die Komponenten andocken können, um ihre Wirkung auf die Produktlinien zu entfalten.

4.2.3 Konzeption einer Referenzarchitektur für Produktlinienfamilien

Die bisher auf dem Produktlinien-Referenzprozess und auf modellgetriebener Software aufbauenden Überlegungen sind alle noch allgemeinen Charakters. So definieren sie noch keine Architektur für ein System, das die Entwicklung von Produktlinienfamilien ermöglicht oder legen sich auf eine Technologie fest. Vielmehr bestimmen sie die Anforderungen, die an eine solche Architektur zu richten sind.
Um den konzeptionellen Entwurf einer Referenzarchitektur für Produktlinienfamilien von den bisherigen Grundlagen abzusetzen, erfolgt dessen Erörterung im anschließenden Kapitel.

[7] Hierbei wird davon ausgegangen, dass die per Hand zu programmierende Anwendungslogik in einer plattformunabhängigen Form erstellt werden kann. Dies ist auch bei der Produktlinie im Implementierungsteil dieser Arbeit der Fall.

4.3 Domänenimplementierung

Während der Domänenimplementierungsphase erfolgt die programmiersprachliche Erstellung der eigentlichen Anwendungslogik sowie der Transformatoren und Generatoren. Beim Produktlinienfamilienansatz müssen dabei für Produktlinien und Komponenten getrennte Artefakte erstellt werden. Die Domänenimplementierung basiert dabei auf der Referenzarchitektur für Produktlinienfamilien, die nachfolgend vorgestellt wird.

4.4 Zusammenfassung

Dieses Kapitel behandelte die nötigen Anpassungen am Referenzprozess für Produktlinienenwicklung, um produktlinienübergreifende Wiederverwendung mittels des Produktlinienfamilienansatzes zu ermöglichen.
In der Domänenanalysephase wurde zunächst beleuchtet, wie die Dekomposition einer Produktlinie entsprechend ihrer Merkmale ablaufen kann. Anschließend wurden mögliche Darstellungsweisen für Domänenmodelle von Produktlinien und Komponenten erörtert, die eine Rekomposition ermöglichen.
Im Rahmen der Domänenentwurfsphase wurden die modellgetriebenen Voraussetzungen analysiert, die für die Erstellung der Anwendungsreferenzarchitekturen mehrerer Produktlinien unter dem Dach einer Produktlinienfamilie von Belang sind. Dabei wurden mit den Voraussetzungen an die Referenzarchitekturmodellierung sowie an den Anwendungserstellungsprozess sowohl statische als auch dynamische Gesichtspunkte berücksichtigt. Auch auf die Identifikation weiterer, implementierungsnaher Produktlinienkomponenten wurde eingegangen.
Schließlich wurde die Konzeption einer Referenzarchitektur für Produktlinienfamilien gefordert, welche das nachfolgende Kapitel 5 behandeln wird.

5 Konzeption einer Referenzarchitektur für Produktlinienfamilien

Die Integration von Produktlinien und ihren Komponenten stellt eine der größten Herausforderungen des Produktlinienfamilienansatzes dar. Dabei sind ganz unterschiedliche Gesichtspunkte zu betrachten.
Zunächst werden die zentralen Architekturtreiber aus dem im vorigen Kapitel beschriebenen Referenzprozess extrahiert. Dann wird daraus eine Referenzarchitektur abgeleitet. Die Architektur wird dann hinsichtlich der dynamischen Aspekte, die ihre Bestandteile zu unterstützen haben, beleuchtet.

5.1 Architekturtreiber aus dem Referenzprozess für Produktlinienfamilien

Die zentrale Anforderung an die Referenzarchitektur ist die Unterstützung der Wiederverwendung von Modellen, Transformatoren, Generatoren und Anwendungscode.
Aus der Domänenanalysephase kommen primär funktionale Anforderungen bezüglich der Manipulation von Domänenmetamodellen, die keinen architekturrelevanten Einfluss haben. Der Domänenentwurf erkennt in den gemeinsamen Anwendungs- und Instantiierungsmetamodellen eine der entscheidenden Grundlagen der Integrationsfähigkeit. Zudem wird ein gemeinschaftlich definierter modellgetriebener Anwendungserstellungsprozess als zweiter ausschlaggebender Punkt identifiziert.
Daraus lassen sich folgende **Architekturteiber** für eine Referenzarchitektur für Produktlinienfamilien ableiten:

1. Je nach spezifischen Erfordernissen der zu erstellenden Endprodukte kann es u. U. mehrere Produktlinienfamilien geben. Die zu entwerfende Architektur sollte dies also berücksichtigen. Dabei sollten jedoch alle Basisfunktionalitäten, die für die Integration von Komponenten und Produktlinien benötigt werden, global bereitgestellt werden.

2. Des Weiteren kommt eine direkte Interaktion zwischen Produktlinien und Komponenten nicht in Frage. Komponenten sollten sich nur auf die „Schnittstelle“ des Anwendungsprozesses beziehen und nicht auf eine konkrete Implementation einer Produktlinie. Würde die Komponente hingegen aktiv Zugriff auf die Produktlinie haben, könnte diese Schnittstelle eventuell umgangen werden.

Umgekehrt sollten Produktlinien nicht explizit die Nutzung jeder einzelnen Komponente in der Definition ihres Anwendungserstellungsprozesses vorsehen müssen. Daher wird eine weitere Instanz nötig, um die Integration beider zu bewerkstelligen.

3. Schließlich sollen die Eingriffe der Komponenten in die Produktlinie nicht invasiv stattfinden. Die unangereicherte Produktlinie ist also immer klar von der der Komponenten getrennt zu halten, um die problemlose und separate Weiterentwicklung beider zu gewährleisten.

5.2 Referenzarchitektur für Produktlinienfamilien

Aufgrund der identifizierten Architekturtreiber wird die in Abbildung 5.1 gezeigte Referenzarchitektur vorgeschlagen. Die Architekturtreiber lassen sich dabei auf die Schichten eins bis drei des Diagramms abbilden.

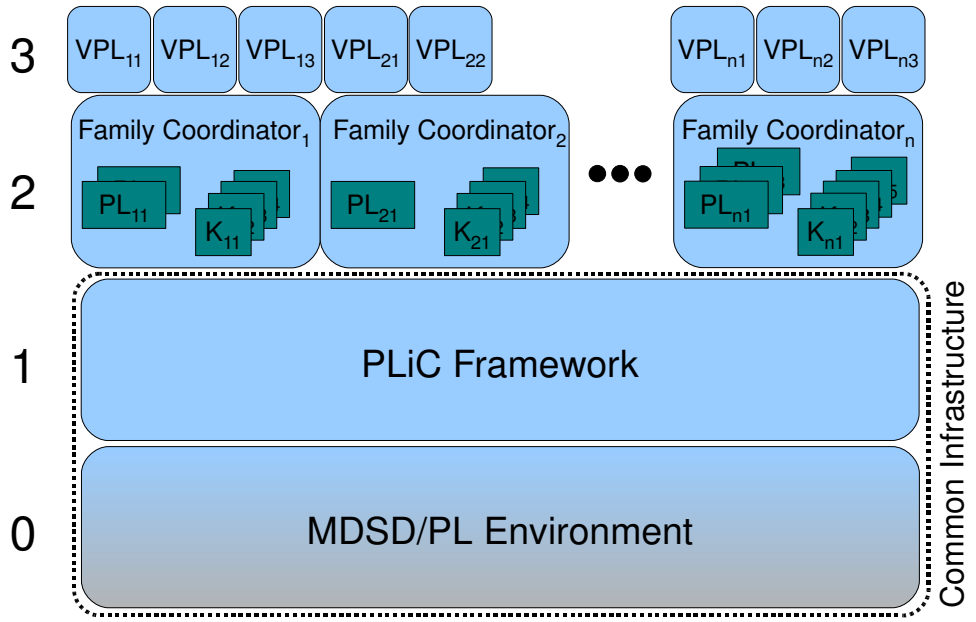


Abbildung 5.1: Die Referenzarchitektur für Produktlinienfamilien

0. Die Grundlage der Architektur stellt eine übliche *modellgetriebene Produktlinieninfrastruktur* dar. Sie bietet grundlegende Funktionen wie Transformationen und Codegenerierung, sowie die Möglichkeit, Variantenmodelle zur Konfiguration von Endprodukten einer Produktlinie auszuwerten.

1. Aufbauend auf Schicht 0 werden zentrale Basisfunktionalitäten zur Definition von Produktlinienfamilien vom *PLiC Framework* (*Product Line Components Frame-*

work) bereitgestellt. Es stellt Hilfsmittel zur Definition des Anwendungserstellungsprozesses der Produktlinienfamilien bereit und unterstützt diese bei der Koordination der Produktlinien und Komponenten. Gemeinsam mit der darunterliegenden Schicht bildet sie eine Infrastruktur (*Common Infrastructure*), die von der darüberliegenden Schicht verwendet werden kann.

2. Für jede Produktlinienfamilie ist ein separater *Familienkoordinator* zu entwerfen. Er sorgt für die Integration der Komponenten in die Produktlinie und koordiniert die Interaktion der beiden während des Anwendungserstellungsprozesses.

3. Um sicherzustellen, dass die Produktlinien und Komponenten unabhängig voneinander weiterentwickelt werden können, ermöglicht es der Familienkoordinator schließlich, *virtuelle Produktlinien* zu spezifizieren. Diese stellen eine mit Komponenten angereicherte Basisproduktlinie dar. Ihre Konfiguration erfolgt dabei über die angereicherten Domänenmodelle, wobei die ursprünglichen Produktlinienartefakte unangetastet bleiben.

Produktlinienvirtualisierung

Unter Produktlinienvirtualisierung wird im Folgenden die Abstraktion von den konkreten Ressourcen Produktlinie und Komponente verstanden. Analog zu der Virtualisierung im Hardwarebereich kann eine Ressource von mehreren virtuellen Einheiten (virtuellen Produktlinien) gemeinsam benutzt werden, mehrere Ressourcen (eine Produktlinie und mehrere Komponenten) zu einer virtuellen Einheit zusammengefasst werden. Nach außen hin verhalten sich die Einheiten hingegen identisch zu einer üblichen Produktlinie. Sie lassen sich über ihre Domänenmodelle konfigurieren und erstellen auf deren Basis ein Endprodukt.

5.3 Integrationsprozess

Die im vorherigen Abschnitt beschriebene Referenzarchitektur muss auch dynamische Aspekte berücksichtigen. Insbesondere die strikte Separierung von Produktlinien und Komponenten (Architekturtreiber 3) durch den Familienkoordinator erfordert einen Integrationsablauf, der in folgende Phasen unterteilbar ist (siehe Abbildung 5.2):

1. Bootstrapping (Familiengründung)

   In dieser Phase wird der Familienkoordinator initialisiert und mit allen seinen Komponenten und deren Integrationskonfigurationsmöglichkeiten bekannt gemacht. Das Durchlaufen dieser Phase wird nur nötig, wenn neue Produktlinienkomponenten dazukommen oder sich bestehende in den Konfigurationsoptionen für ihre Integration geändert haben.

2. Komponentenkonfiguration

   Während der Komponentenkonfiguration wird die Beschaffenheit einer neuen virtuellen Produktlinie manuell konfiguriert. Zunächst sind die Komponenten auszuwählen, die in die virtuelle Produktlinie zu integrieren sind. Dann muss die Art und Weise ihrer Integration festgelegt werden, was davon abhängt, welche Integrationskonfigurationsmöglichkeiten die Komponenten anbieten. Schließlich muss auch die Basisproduktlinie gewählt werden, auf welcher die virtuelle Produktlinie basieren soll.

3. Komponentenintegration

   In der Integrationsphase erfolgt die Virtualisierung der Basisproduktlinie und ihre Erweiterung durch die Produktlinienkomponenten. Da die Konfiguration des Anwendungserstellungsprozesses jeder Produktlinie üblicherweise über ihre Domänenmodelle erfolgt, besteht die Integration also insbesondere in der Erweiterung der Domänen*meta*modelle der Produktlinie. Diese Phase muss nur durchlaufen werden, wenn neue Produktlinienkomponenten in eine virtuelle Produktlinie eingebunden werden sollen oder sich bestehende in wesentlichen, die Domäne betreffenden Eigenschaften geändert haben.[1]

4. Produktlinienkonfiguration

   Anhand der erweiterten Domänenmodelle kann nun die virtualisierte Produktlinie manuell konfiguriert werden.

5. Anwendungserstellung

   Schließlich erfolgt die Anwendungserstellung auf Basis der gewählten Konfiguration. Die Dynamik dieses Prozesses stellt eine besondere Herausforderung dar. Die zentrale Frage hierbei lautet, wie erreicht werden kann, dass die Komponenten mit Hilfe des Produktlinienkoordinators auf möglichst einfache Weise den Anwendungserstellungsprozess der Basisproduktlinie beeinflussen können.

Die Phasen eins, drei und fünf lassen sich automatisieren, während die Konfigurationsphasen jeweils manuell zu erfolgen haben. Nicht immer müssen alle Phasen voll ausgeprägt sein. Beispielsweise ist im Falle gemeinsamkeitsbasierter Produktlinienkomponenten eine Erweiterung der Domänenmetamodelle unnötig.
Der *genaue* Inhalt dieser Phasen fällt für jede Produktlinienfamilie unterschiedlich aus, da jede andere Modelle, Metamodelle und einen anderen Ablauf der Anwendungserstellung hat. Der Familienkoordinator, der genau die Aufgabe hat, die Phasen für eine konkrete Familie festzulegen, ist also für jede Familie unterschiedlich. Um ihn

[1] Genau genommen ist der Begriff *Komponentenintegration* etwas unklar, da es sich nicht um eine physische Einbindung der Komponenten in eine Produktlinie handelt. Es handelt sich um eine Virtualisierung, bei der Artefakte, die tatsächlich physisch geändert werden müssen, also insbesondere die Domänenmetamodelle, vorher kopiert werden. Die *Integration* erfolgt also nicht invasiv bezüglich der Basisproduktlinie.

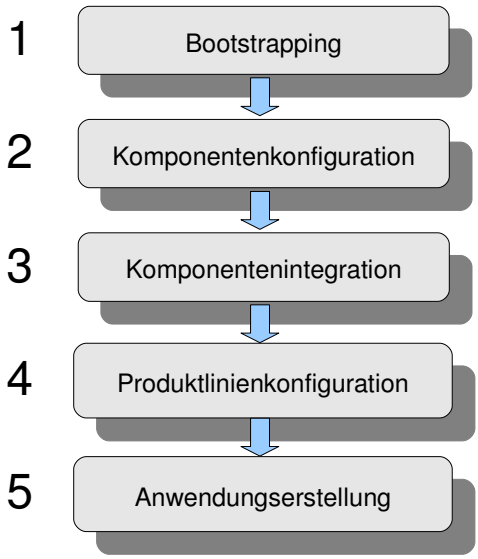


Abbildung 5.2: Der Integrationsprozess von Produktlinienfamilien

dennoch mit akzeptablem Aufwand zu implementieren muss das PLiC Framework sinnvolle Funktionen bereitstellen, welche dessen Bedürfnisse abdecken.
Da die Umsetzung der Referenzarchitektur sehr technisch geprägt ist, erfolgt deren weitere Erarbeitung nach der Vorstellung der implementationstechnischen Grundlagen in Abschnitt 7.

5.4 Zusammenfassung

Dieses Kapitel entwickelte anhand der Anforderungen des vorher erweiterten Referenzprozesses für Produktlinienentwicklung eine Referenzarchitektur für Produktlinienfamilien. Sie ist schichtenartig aufgebaut und ermöglicht die gemeinsame Nutzung der Ressourcen Produktlinie und Komponente durch mehrere so genannte *virtuelle Produktlinien*. Des Weiteren wurde ein Integrationsprozess definiert, der schrittweise die Verknüpfung der Komponenten mit der Basisproduktlinie beschreibt.

6 Implementation: Überblick

Das Ziel der Implementation ist die prototypische Umsetzung der in Kapitel 5 vorgestellten Referenzarchitektur für Produktlinienfamilien. Darauf aufbauend wird mit Hilfe einer Beispielproduktlinienfamilie dessen Funktionsfähigkeit nachgewiesen. Der Implementationsteil der Arbeit verteilt sich auf die anschließenden vier Kapitel:

- Im folgenden Kapitel 7 wird zunächst ein **Überblick über openArchitectureWare** gegeben, das als modellgetriebene Entwicklungsumgebung mit Produktlinienunterstützung die Grundlage des PLiC Frameworks bildet. Anschließend wird die **SmartHome Produktlinie** beschrieben. Sie bildet im späteren Teil der Implementierung die Basis für die Erstellung einer Produktlinienfamilie.
- Im Kapitel 8 erfolgt die **Konzeption der Implementierung.** Sie beleuchtet den in Abschnitt 5.3 vorgestellten Integrationsablauf hinsichtlich dessen Umsetzbarkeit mit openArchitectureWare. Dabei werden **Entwurfsentscheidungen für den Familienkoordinator** gefällt, der als Vermittler zwischen Produktlinien und Komponenten einen zentralen Bestandteil der Architektur ausmacht.
- Kapitel 9 schildert zunächst die systematische **Erweiterung von openArchitectureWare** auf Basis der Entwurfsentscheidungen. Anschließend wird erläutert, wie das **PLiC Framework** bei der Implementation eines Familienkoordinators zur Definition eines konkreten Integrationsprozesses einer Produktlinienfamilie anzuwenden ist.
- Kapitel 10 beschreibt die Implementation eines **Familienkoordinators** auf Basis von SmartHome, sowie die Erstellung einer **Produktlinienkomponente.** Abschließend wird anhand einer neu definierten virtuellen Produktlinie der Integrationsprozess konkret veranschaulicht.

7 Implementation: Basis

7.1 openArchitectureWare

Bei openArchitectureWare (oAW) handelt es sich um ein Framework zur Unterstützung modellgetriebener Softwareentwicklung. Es erlaubt die Arbeit mit Modellen zahlreicher Quellen (UML, XML, JavaBeans, ...), wobei die Unterstützung von Modellen des Eclipse Modeling Frameworks auf ECore–Basis am umfassendsten ist. Dabei besitzt oAW eigene Sprachen, um Modelltransformatoren (Xtend), Codegeneratoren (Xpand) und deren Abfolge (Workflow) zu definieren. Die Editoren für diese Sprachen sowie die Anbindung der Workflow Engine sind als Eclipse-Erweiterungen realisiert. Durch die Integration in die Entwicklungsumgebung können einfache Aufgaben direkt von dort aus realisiert werden, ohne sich mit dem Java-API des Frameworks auseinandersetzen zu müssen.
Eine zusätzliche Erweiterung[1] ermöglicht es oAW, ein Variantenmodell der Produktlinienmanagement-Software pure::variants einzulesen. Dieses kann dadurch in den gesamten Produkterstellungsprozess mit einbezogen werden, was die bedingte Ausführung von Workflow-, Transformator- oder Generatorbefehlen gestattet.

7.1.1 Workflow

Über die *Workflow Engine* lassen sich Abläufe definieren, um selbst erstellte Metamodelle und Modelle einzulesen, zu transformieren und schließlich Anwendungscode daraus zu generieren. Die Beschreibung der Befehlsabfolge geschieht dabei in einer an XML angelehnten Sprache. Die zentralen Sprachelemente dabei sind **Workflow-Komponenten** (*Workflow Components*) und **Workflow-Cartridges** (*Workflow Cartridges*).[2]
Bis auf die bedingte Ausführung von Sprachelementen aufgrund der Merkmalauswahl im Variantenmodell ist dabei keine Beschreibung von Kontrollfluss möglich.

7.1.1.1 Workflow-Komponenten

Bei den Workflow-Komponenten handelt es sich um in Java implementierte Klassen, die eine bestimmte Schnittstelle implementieren müssen. Einige dieser Komponenten sind bereits vordefiniert, so solche zum Einlesen und Schreiben von Modellen und Aufrufen von Transformatoren und Generatoren.

[1] Bisher ist die Erweiterung nur über CVS zu beziehen: `http://architecturware.cvs.sourceforge.net/architecturware/oaw_v4/utils/org.openarchitectureware.util.featureconfig.pv/`

[2] Da es für die openArchitectureWare-spezifischen Fachausdrücke *Workflow* und *Cartridge* keinen angemessenen deutschen Begriffe gibt, werden sie im Folgenden unübersetzt belassen.

In Abbildung 7.1 ist ein Beispiel-Workflow abgebildet. Dessen Aufgabe besteht zunächst darin, die *Property* `myUri` auf einen bestimmten Wert zu setzen. Bei *Properties* handelt es sich um eine speziellen Typ von Variablen in Workflows, welcher im Folgenden aus Gründen der Eindeutigkeit nicht in seiner deutschen Übersetzung „Eigenschaft“ gebraucht wird.

Danach startet der Workflow die Workflow-Komponente (Java-Klasse), die sich im Klassenpfad unter `oaw.emf.XmiReader` befindet und setzt ihre beiden Variablen `uri` und `modelSlot` auf die entsprechenden Inhalte. Die Property `${myUri}` wird dabei zuvor durch die einen Pfad enthaltende Zeichenfolge `/home/user/model.xmi` ersetzt.

```
<workflow>                  <!-- Dateiname: readerWorkflow.oaw -->
    <property  name       = "myUri"
               value      = "/home/user/model.xmi"
    />

    <component class      = "oaw.emf.XmiReader"
               uri        = "${myUri}"
               <modelSlot value = "mySlot" />
    />
</workflow>
```

Abbildung 7.1: Workflow mit Komponente zum Einlesen von Modellen

Es handelt sich bei `XmiReader` um eine von oAW vordefinierte Klasse. Sie liest ein Modell aus der Datei mit dem Pfad in `uri` und schreibt dessen Hauptspeicherrepräsentation in die Slot-Variable `mySlot`. Analog funktionieren auch die weiteren Workflow-Komponenten.

Dass `modelSlot` ein XML-Element ist, wohingegen `uri` ein XML-Attribut darstellt, macht übrigens keinen Unterschied, beide Schreibweisen sind gleichberechtigt. Die Elementschreibweise ist nur dann zwingend, wenn einer Workflow-Komponente mehrere Parameter gleichen Namens übergeben werden sollen. Dies ist erforderlich, da der SAX-Parser keine XML-Attribute gleichen Namens erlaubt.

Der Unterschied zwischen Slot- und Property-Variablen ist folgender: Properties bewirken einfach nur Zeichenkettenersetzungen, die der Workflow-XML-Parser beim Aufbau des Workflows vornimmt. Slots hingegen können beliebige Objekte enthalten, sind jedoch nur innerhalb von Workflow-Komponenten zugreifbar.

Neben normalen Slots gibt es auch so genannte List-Slots. Sie können eine beliebige Anzahl an Modellen enthalten und sind nützlich, um Produktlinien beliebig viele Modelle eines Typs injizieren zu können. Dies ist insbesondere für Produktlinienkomponenten interessant, da deren Anzahl und somit die Anzahl der zusätzlichen Modelle nicht von Anfang an bekannt ist.

7.1.1.2 Workflow-Cartridges

Die Workflow-Cartridges ähneln „include“-Anweisungen anderer Programmiersprachen. Ein Workflow kann damit einen anderen einbinden, was auf eine gewisse Art funktionaler Abstraktion entspricht. Mangels leistungsfähiger Kontrollflussmechanismen ist jedoch (abgesehen von der bedingten Ausführung aufgrund der Merkmalauswahl) immer fix verdrahtet, welche Cartridges wie oft ausgeführt werden.
Der in Abbildung 7.2 abgebildete Workflow ruft zunächst den in Abbildung 7.1 gezeigten über den Cartridge-Mechanismus auf, um das Modell einzulesen. Dann erfolgt eine Modelltransformation, die im nachfolgenden Abschnitt noch näher erklärt wird.

```
<workflow>
    <cartridge file = "./readerWorkflow.oaw" />

    <component
        class     = "org.oaw.xtend.XtendComponent"
        id        = "trafoID"
        invoke    = "transformNames::transformNames(mySlot)"
        modelSlot = "myTransformSlot"
    />
<workflow>
```

Abbildung 7.2: Workflow mit Cartridge und Transformation

Neben normalen kennt oAW auch noch abstrakte Workflows. Diese sind gekennzeichnet durch das zusätzliche Attribut `abstract="true"` im `<workflow>`-XML-Element. Sie können nicht direkt ausgeführt werden, wie dies sonst über Eclipse mit *„Run as ... ⇒ oAW Workflow“* möglich ist. Da sie Properties verwenden, die sie selbst nicht definieren, eignen sie sich jedoch gut als parametrisierbare Cartridges. Der aufrufende Workflow setzt die Properties dann entweder explizit über Attribute des XML-Element `<cartridge>`, oder einfach, indem er als XML-Attribut `inheritAll="true"` angibt, dass dem Subworkflow alle im aufrufenden Workflow definierten Properties vererbt werden.

7.1.2 Expression Language

Bei der Expression Language handelt es sich um eine Subsprache, die den Kern der beiden Sprachen Xtend und Xpand ausmacht. Sie wird in der Dokumentation von oAW auch als „a syntactical mixture of Java and OCL“ [oawa] umschrieben. Von OCL erbt die Sprache z. B. die Fähigkeit, flexibel mit Mengen und Hierarchien von Objekten umzugehen, von Java einige der Basisoperatoren.

7.1.2.1 Xtend

Mit Xtend lassen sich Modelltransformationen und andere Funktionen spezifizieren (siehe Abbildung 7.3).
Aufgerufen werden Transformationen, wie bereits in Abbildung 7.2 dargestellt, über die Transformatorklasse `XtendComponent`. Sie ruft die im Parameter `invoke` spezifizierte Xtend-Funktion `transformNames()` aus der Datei `transformNames.ext` auf und übergibt ihr das Modell aus dem Slot `mySlot`. Nach der Transformation speichert sie das Ergebnis im Slot `myTransformSlot`. Der Parameter `id` ermöglicht die eindeutige Identifikation der Transformation.

```
import mymetamodel;                  // Dateiname: transformNames.ext

// Transformation
MyModel transformNames(MyModel model):
    model.components.process() -> // Aufruf je Komponente
    model;                        // Gib Modell zurück

// Funktion
process(MyComponent c):
    c.setName(c.name + 'Impl');    // Konkateniere "Impl"
```

Abbildung 7.3: Transformation mit Xtend: Änderung der Komponentennamen

Xtend hat einige interessante Fähigkeiten. So kann Java-Code von Xtend aus aufgerufen werden. Auch unterstützt es rudimentäre Aspektorientierung [KLM+97, KHH+01]. So lässt sich um eine Transformation ein Around-Advice spezifizieren, der sie mit weiterer Funktionalität anreichern kann. Der Advice kann mittels Platzhaltern auf mehrere Transformationen zugleich wirken (siehe Abbildung 7.4).

```
import mymetamodel; // Dateiname: adviceFile.ext

around transformN*(MyModel model):
    let newModel = (MyModel)ctx.proceed() : (
        // Mache noch ein paar Modellmanipulationen
    newModel
    );
```

Abbildung 7.4: Around-Advice mit Xtend

Das Weben des Aspekts erfolgt durch den Aufruf der Workflow-Komponente namens `XtendAdvice` (siehe Abbildung 7.5). Diese erhält als Parameter `extensionAdvice` die Datei, in welcher der Advice spezifiziert ist und in `adviceTarget` die ID der Transformation, auf die sie wirken soll (siehe Parameter `id` in Abbildung 7.2). Streng

genommen relativiert dieser Mechanismus die *Obliviousness* [Fil01], da die Angabe der ID bei transformierenden Workflow-Komponenten optional ist, der Webevorgang aber ohne ID nicht durchgeführt werden kann.
Die Workflow-Komponente `XtendAdvice` bewirkt übrigens eine Art Load-Time-Weaving. Bereits beim Aufbau des Syntaxbaumes erfolgt die Registration eines Advice bei der transformierenden Workflow-Komponente. Diese sorgt dann selbst dafür, den Advice um die eigentliche Transformation herum auszuführen.

```
<component
    class           = "org.oaw.xtend.XtendAdvice"
    adviceTarget    = "trafoID"
    extensionAdvice = "adviceFile"
/>
```

Abbildung 7.5: Komponente zum Weben eines Xtend-Advice

7.1.2.2 Xpand

Xpand ist eine Sprache zur Erzeugung von Text aus Modellen. Sie baut ebenfalls auf der Expression Language auf, ist aber als „Template-Sprache“ darauf spezialisiert, Dateien mit dynamischem Inhalt zu generieren. Abbildung 7.6 zeigt ein Beispiel, das die Datei `componentNames.txt` mit den Namen aller im Modell definierten Anwendungskomponenten generiert.

```
«DEFINE model FOR MyModel»
    «FILE "componentNames.txt"»
        «FOREACH components AS c»
Found component «c.name».
        «ENDFOREACH»
    «ENDFILE»
«ENDDEFINE»
```

Abbildung 7.6: Texterzeugung mit Xpand

Xpand-Templates werden ganz ähnlich wie Xtend-Transformationen über Workflows angestoßen. Zudem unterstützt Xpand ebenfalls Aspektorientierung über das Schlüsselwort «`AROUND`».

7.2 SmartHome Demonstrator

Der SmartHome Demonstrator ist eine auf openArchitectureWare basierende, modellgetriebene Produktlinie zur Heimautomation. Sie wurde im Rahmen des Forschungs-

projekts AMPLE[3][VG07] entwickelt und zeigt den Stand der Technik bezüglich Variabilitätsbeherrschung mit aspektorientierter Technologie.
Die Aspektorientierung erstreckt sich auf die Transformatoren, Generatoren und den Programmcode und ermöglich so einen sehr flexiblen Umgang mit den variablen Bestandteilen der Produktlinie. Dabei werden die bereits erläuterten Around-Advice-Definitionen von oAW intensiv genutzt. Die Handhabung von Aspekten im Programmcode wird unter anderem durch explizite Modellierung von Aspekten im PIM-Modell und Generierung von Aspekt-Stubs auf Implementierungsebene erreicht. Der Aspekt selbst muss dann in diesem Falle per Hand implementiert werden.
Primär steht in dieser Arbeit jedoch nicht der aspektorientierte Teil, sondern die modellgetriebene Produktlinie im Fokus. Um sie herum soll eine Produktlinienfamilie entstehen. Daher folgt nun ein Überblick über den Aufbau des SmartHome Demonstrators, wobei insbesondere die Metamodelle, Transformatoren und Generatoren dabei im Mittelpunkt stehen.

7.2.1 Überblick

Der SmartHome Demonstrator ist eine Produktlinie zur Heimautomation. Domänenexperten, wie beispielsweise Architekten, können also mit ihm ein Gebäude mit diversen Sensoren, Aktoren und Steuergeräten modellieren. Durch wiederholte Modelltransformationen und Codegenerierung entsteht daraus schließlich ein fertiges, verteiltes Programm, das die entsprechende Logik implementiert. Die Anwendungserstellung erfolgt nach einem der MDA angelehnten Ablauf. Es werden dabei nur etwas andere Begriffe verwendet und die Metamodelle sind selbst entworfen und basieren auf ECore. Abbildung 7.7 zeigt den Ablauf, der nun nachfolgend erläutert wird.

7.2.2 CIM-Ebene

Zu Beginn kann der Domänenexperte die Produktlinie über das *Variantenmodell* eines Merkmalmodells konfigurieren, worüber sich bestimmte globale Eigenschaften und Merkmale wie Sicherheits- (*Security*) und Komfortaspekte einstellen lassen (siehe Abbildung 7.8).
Dann kann er über das *PS-Modell* (*Problem Space Model*) die strukturellen Aspekte der Anwendung definieren. Ohne technologische Details und in einer für ihn als Domänenexperten verständlichen Sprache kann er hier den Aufbau des Hauses, die dortigen Geräte und ihre Verknüpfungen untereinander modellieren.

7.2.3 PIM-Ebene

Das *CBD-Modell* (*Component Based Development*) stellt ein generisches Modell zur spezifikation von Anwendungskomponenten dar, mit dem sich sowohl die statische Anwendungsstruktur als auch die dynamischen Gesichtspunkte beschreiben lassen. Es vereint somit in sich ein Anwendungs- und ein Instantiierungsmodell.

[3] *A*spect-Oriented *M*odel-Driven Product *L*ine *E*gineering.

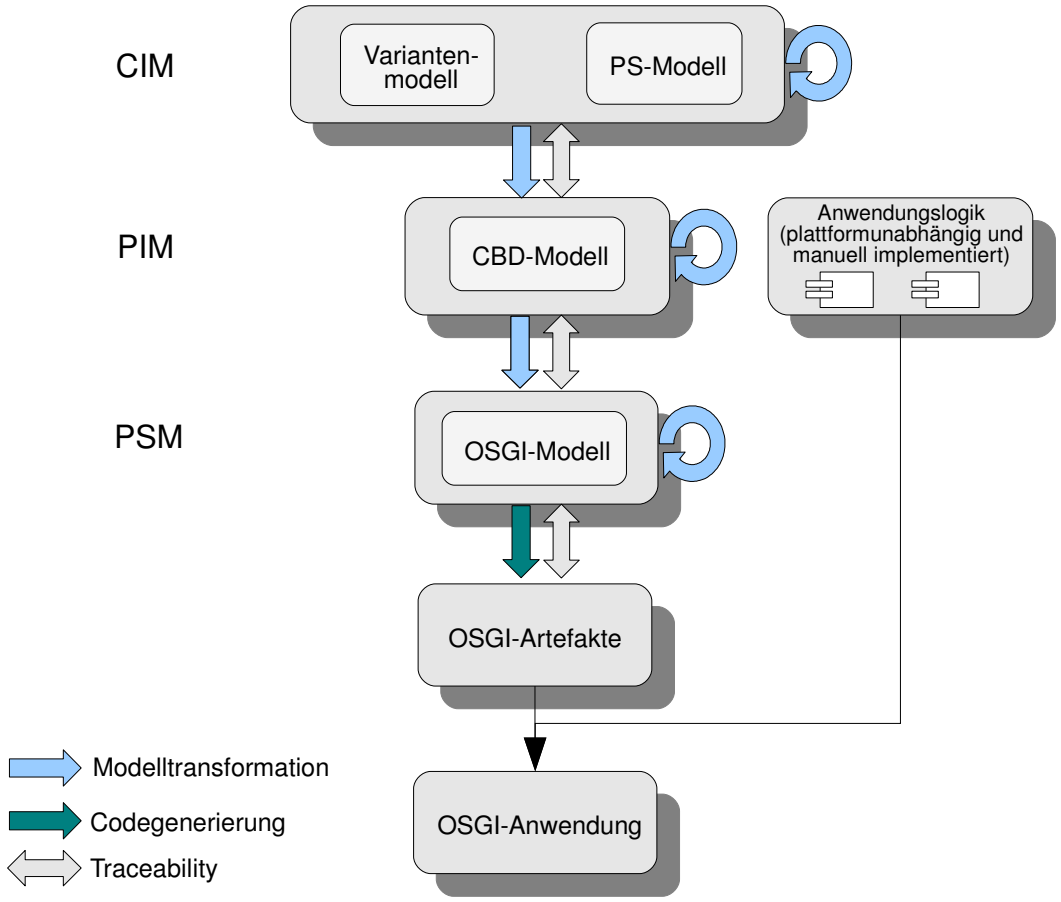


Abbildung 7.7: Ablauf der Anwendungserstellung bei SmartHome (nach [Voe07])

Die eigentliche Implementation der Anwendungskomponenten erfolgt bereits auf der plattformunabhänigigen CBD-Ebene in manuell erstelltem Java. Für die Abbildung auf die Plattform ist dann nur die Generierung von Wrappercode nötig. Dies hat den Vorteil, dass die Anwendung auf beliebigen Plattformen ohne Änderungen an der Anwendungslogik ablaufen kann, solange diese Plattformen nur Java-Unterstützung bieten. Dies mag nicht für alle modellgetriebenen Situationen der richtige Ansatz sein, kann sich aber für die schnelle Portierbarkeit von Anwendungen als durchaus sinnvoll erweisen.

7.2.4 PSM-Ebene

Das *OSGI-Modell* schließlich bildet die Komponenten des CBD-Modells auf die OSGI-Platform [All05] ab. Nachdem das PS-Modell bis zum OSGi-Modell transformiert wurde, erfolgt schließlich die Generierung von OSGi-Wrappern für die Anwendungs-

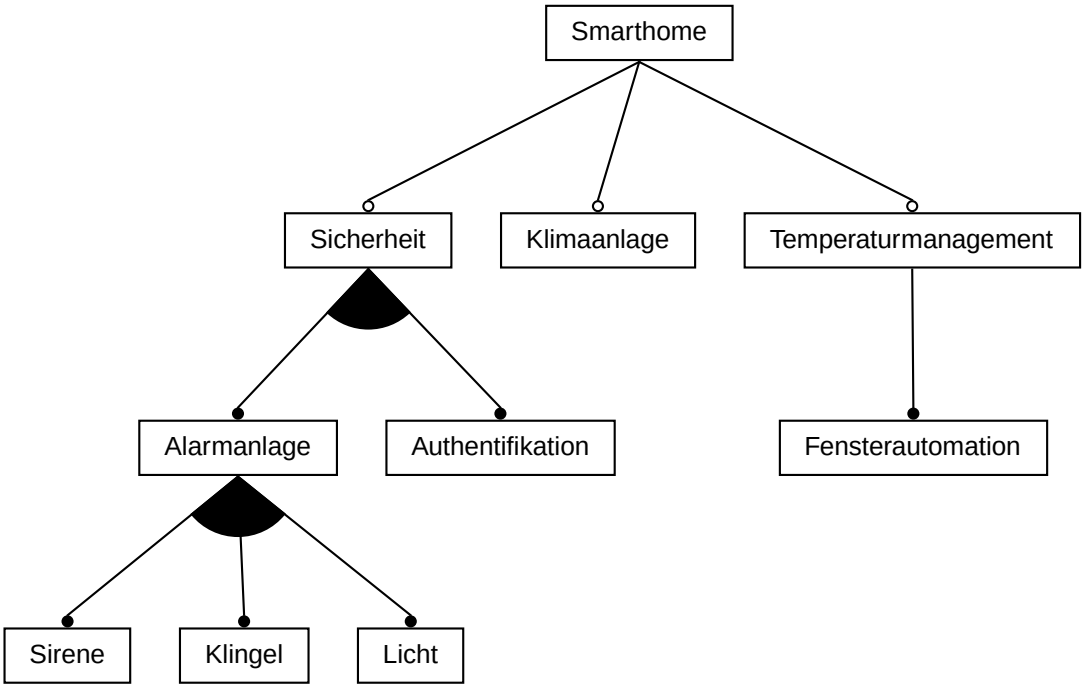


Abbildung 7.8: Ausschnitt aus dem SmartHome Merkmaldiagramm

komponenten sowie aller weiteren für die Lauffähigkeit der Anwendung benötigten Artefakte.

7.2.5 Weitere Besonderheiten

Eine Besonderheit stellt das Variantenmodell dar. Es steht nicht wie üblich für Domänenmodelle nur bei der CIM-PIM-Transformation zur Verfügung, sondern ist innerhalb des gesamten Erstellungsprozesses global von allen Transformatoren, Generatoren, Workflows und von Java aus abrufbar. Es steht somit in gewisser Weise orthogonal zum Ablauf der MDA. Des Weiteren gewährleistet SmartHome mit Unterstützung von oAW die Verfolgbarkeit von Domänenmodellelementen bis hin zum Anwendungscode (*Traceability*).

7.3 Zusammenfassung

Dieses Kapitel gab zunächst eine Einführung in openArchitectureWare (oAW), das als modellgetriebene Entwicklungsumgebung mit Produktlinienunterstützung die Grundlage des Implementierungsteils darstellt. *Workflows* dienen hierbei zur Spezifikation des modellgetriebenen Anwendungserstellungsprozesses. Mit *Xtend* und *Xpand* be-

sitzt oAW eigene Sprachen zur Definition von Modelltransformationen und zur Codegenerierung, die sogar Aspektorientierung unterstützen.
Der zweite Teil des Kapitels behandelte die SmartHome Produktlinie, auf deren Basis im Kapitel 10 eine Produktlinienfamilie sowie eine Produktlinienkomponente entstehen wird. Bei ihr handelt es sich um eine Produktlinie zur Heimautomation, die zwar deutliche Ähnlichkeiten mit dem Prozessablauf der MDA hat, deren Metamodelle jedoch auf Basis von ECore definiert wurden.

8 Implementation: Konzept

Dieses Kapitel beleuchtet, wie der Integrationsprozess für Produktlinienfamilien über Mittel von openArchitectureWare umgesetzt werden kann.

8.1 Umsetzung des Integrationsprozesses durch Definition von Workflows

Workflows stellen *das* Mittel zur Definition von modellgetriebenen Abläufen in oAW dar. Daher ist der Workflow-Mechanismus (siehe Abschnitt 7.1.1) der Angriffspunkt, um auch den Integrationsprozess einheitlich definieren zu können. Nur die Phasen eins, drei und fünf bedürfen besonderer Beachtung, da die anderen beiden manuell zu vollziehen sind. Es sind also grundsätzlich nur drei Workflow-Abläufe zu betrachten, einer für das Bootstrapping, einer für die Komponentenintegration und einer für die Anwendungserstellung.
Die Interaktion hinsichtlich der Referenzarchitektur (siehe Abbildung 5.1) für einen dieser Abläufe erfolgt von oben nach unten. Aufrufe erfolgen also ausgehend von der virtuellen Produktlinie über den Familienkoordinator zum PLiC Framework, welches wiederum auf oAW basiert.
Der Familienkoordinator ist dabei jeweils dafür zuständig, die Produktlinienkomponenten entsprechend der Konfiguration der virtuellen Produktlinie einzubinden. Weder die virtuelle Produktlinie noch die Basisproduktlinie rufen die Produktlinienkomponenten direkt auf.
Virtuelle Produktlinien bestehen zunächst aus nichts weiter als drei einfachen Workflows, jeweils einer für die Bootstrap-, Komponentenintegrations- und Anwendungserstellungsphase. Sie rufen damit jeweils parametrisiert eine Worfklow Cartridge des Familienkoordinators auf. Dieser sorgt dann mit Hilfe der unteren Schichten dafür, dass nach Ausführung der Workflows die enstprechenden Artefakte bereitstehen, um die nächste Phase des Integrationsprozesses durchlaufen zu können.

8.1.1 Bootstrapping

Während des Bootstrappings erfolgt innerhalb der virtuellen Produktlinie ein parameterloser Aufruf an den *Bootstrap-Workflow* des Familienkoordinators. Dieser erzeugt für die virtuelle Familie daraufhin ein *Integrationsmerkmalmodell* (siehe Abbildung 8.1). Mit dessen Instantiierung lässt sich in der nachfolgenden Komponentenkonfigurationsphase spezifizieren, welche der Produktlinienkomponenten integriert werden sollen und auf welche Weise dies zu erfolgen hat. Durch die Wahl eines *Merkmalmodells*

zur Integration wird betont, dass der Produktlinienfamilienansatz einen weiteren Abstraktionsgrad einführt. Durch das Integrationsmerkmalmodell lassen sich verschieden geartete Produktlinien erzeugen, welche sich wiederum über Merkmalmodelle konfigurieren lassen und erst dann ein Endprodukt erstellen.

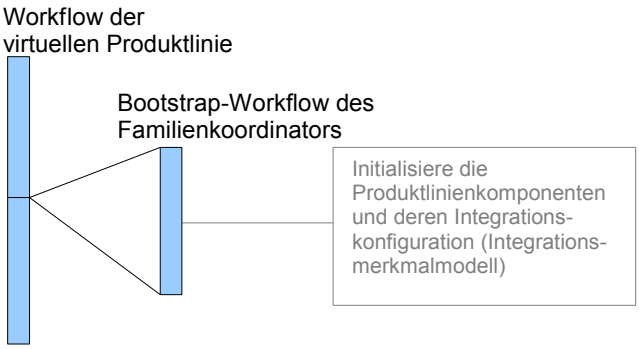


Abbildung 8.1: Konzeptionelles Verhalten des Bootstrap-Workflows

8.1.2 Komponentenintegration

Nachdem eine Konfiguration des Integrationsmerkmalmodells erstellt wurde (*Integrationsvariantenmodell*), dient dieses zur Parametrisierung des *Integrations-Workflows* des Familienkoordinators. Dessen Aufruf erfolgt ebenfalls durch einen einfachen, durch die virtuelle Produktlinie definierten Workflow (siehe Abbildung 8.2). Je nachdem, was in dem Integrationsvariantenmodell spezifiziert wurde, nimmt der Familienkoordinator entsprechende Erweiterungen an den Domänenmetamodellen der ursprünglichen Produktlinie vor. Um dies zu erreichen, müssen ihm das Merkmal- und das Konzeptmetamodell der Basisproduktlinie als weitere Parameter übergeben werden. Die dadurch entstehenden erweiterten Metamodelle stellt er dann der virtuellen Produktlinie zur Verfügung. In der vierten Phase des Integrationsprozesses, der Produktlinienkonfiguration, werden von diesen dann Instanzen gebildet (Varianten- und Konzeptmodell) die entsprechend des gewünschten Endprodukts manuell zu konfigurieren sind.

8.1.3 Anwendungserstellung

In der anschließenden Anwendungserstellungsphase übergibt ein dritter Workflow der virtuellen Produktlinie das Varianten- und das Konzeptmodell an den *Laufzeit-Workflow* des Familienkoordinators (siehe Abbildung 8.3). Dieser sorgt dann je nach Konfiguration für das Weben eines jeden oAW-Advice *um die Transformatoren und Generatoren des Basisworkflows der Produktlinie*. Dies erfordert, dass der Kontext des Basisworkflows von der Stelle aus zugreifbar ist, an der der Webeaufruf stattfindet. Aufgrund der weiteren Konzeption des Familienkoordinators lässt sich dies nur mit Hilfe

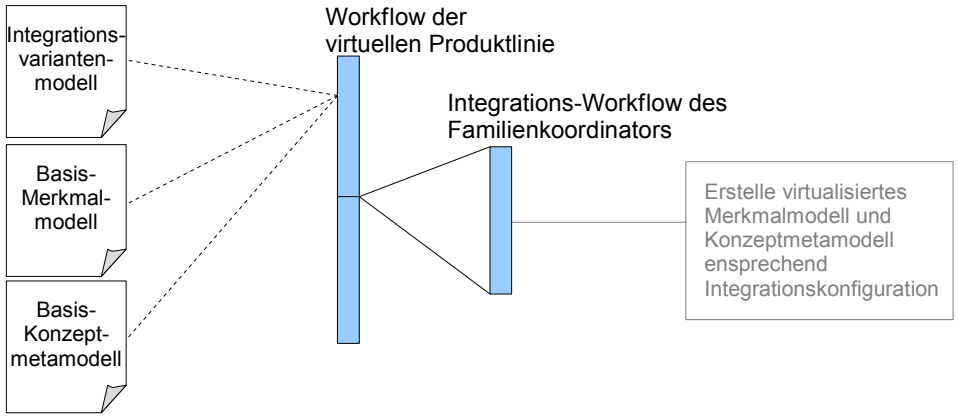


Abbildung 8.2: Konzeptionelles Verhalten des Integrations-Workflows

einer Erweiterung des Workflow-Mechanismus von oAW selbst verwirklichen, was in Abschnitt 9.1 geschehen wird. Ähnliches gilt auch für den Fall, dass Modelle in Slots eingefügt werden sollen, die vom Kontext der Basisproduktlinie heraus zugreifbar sein sollen.

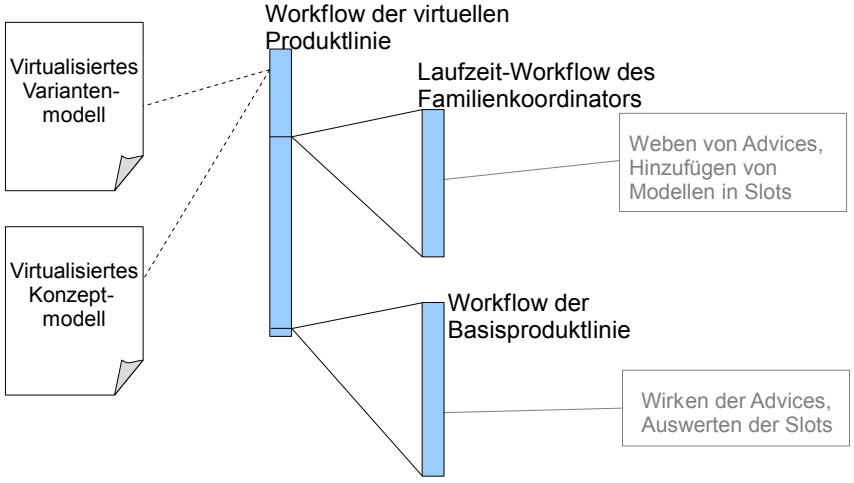


Abbildung 8.3: Konzeptionelles Verhalten des Laufzeit-Workflows

8.2 Der Familienkoordinator

Die vorangegangene Schilderung des Familienkoordinators ist sehr vereinfacht dargestellt. Die umfangreichen Tätigkeiten, die er durchführt, sollten nicht alle für jede

Familie neu implementiert werden müssen. Daher stellt ihm das PLiC Framework die Funktionen, die er üblicherweise dafür benötigt, bereit. Insbesondere gehören dazu Funktionen zum Erstellen des Integrationsmerkmalmodells, zum Erweitern von Merkmal- und Konzeptmetamodellen und zur Unterstützung des Webens von Advice–Spezifikationen.
Es wird festgelegt, dass der Familienintegrator, und mit diesem gleichsam eine Familie, allein durch die Definition von Workflows zu erstellen ist. Wie oben gezeigt ist für jede der drei nicht manuellen Phasen dafür konzeptionell die Definition eines einzigen Workflows nötig(*Boostrap-Workflow, Integrations-Workflow, Laufzeit-Workflow*).[1] Alles Übrige soll durch die Funktionen des PLiC Frameworks bereitgestellt werden. Diese Einschränkung ermöglicht eine sehr leichte Definition von neuen Produktlinienfamilien, erfordert dafür aber ein durchdachtes System der Interaktion aller Beteiligten.
Für die Erstellung von Produktlinienkomponenten lassen sich zwei Alternativen für ihren Aufbau identifizieren. Zum einen könnte sie eine minimale Schnittstelle implementieren. Dort könnten sie dann beispielsweise einen Workflow oder Java-Code hinterlegen und mit dessen Hilfe selbst *aktiv* auf die Produktlinie einwirken. Zum anderen könnten sie eine umfangreichere Schnittstelle anbieten, an denen sie die Artefakte wie Modelle, Advice-Spezifikationen für Transformatoren und Generatoren sowie Anwendungscode *passiv* anbieten, und diese dann durch den Familienkoordinator im Rahmen des Integrationsprozesses genutzt werden können.
Um eine einheitliche Struktur der Komponenten zu gewährleisten, wurde die zweitgenannte Alternative gewählt. Zudem verhindert dies Redundanz, da Komponenten u. U. häufig sehr ähnlichen Bedarf im Integrationsprozess haben werden. Schließlich ist damit die Schnittstelle, über welche die Komponenten auf die Produktlinie einwirken können, eindeutig festgelegt. Der offensichtliche Nachteil dieser Lösung ist, dass jeder Bedarf der Komponenten vorhergesehen werden muss. Wie später noch gezeigt werden wird, wird die Implementierung auch dies ermöglichen. Es soll jedoch an dieser Stelle zunächst primär davon ausgegangen werden, dass eine Komponente passiv Artefakte zu Verfügung stellt, während der Familienkoordinator für deren Integration sorgt.
Dieser Ansatz hat den weiteren Nutzen, dass der Ablauf des Integrationsprozesses in dem Familienkoordinator durch die Definition der dortigen Workflows verankert werden kann und dadurch klar festgeschrieben wird.
Schließlich ist der größte Effizienzgewinn, dass sich die Redundanz in der Spezifikation von Workflows auf ein Minimum beschränken lässt, da für jede Komponente prinzipiell der Integrationsprozess genau gleich abläuft. Der Familienintegrator muss somit für jede der Phasen des Integrationsprozesses nur *einen parametrisierbaren Workflow* wiederholt aufrufen, der dann entsprechend der Parameter für die Integration der ausgewählten Produktlinienkomponente sorgt. Abbildung 8.4 zeigt diese Überlegung.
Um dieses Konzept in die Tat umzusetzen sind jedoch einige Punkte zu klären:

[1] Tatsächlich werden aufgrund der Implementierung für die Komponentenintegration und die Anwendungserstellung zwei Workflows benötigt.

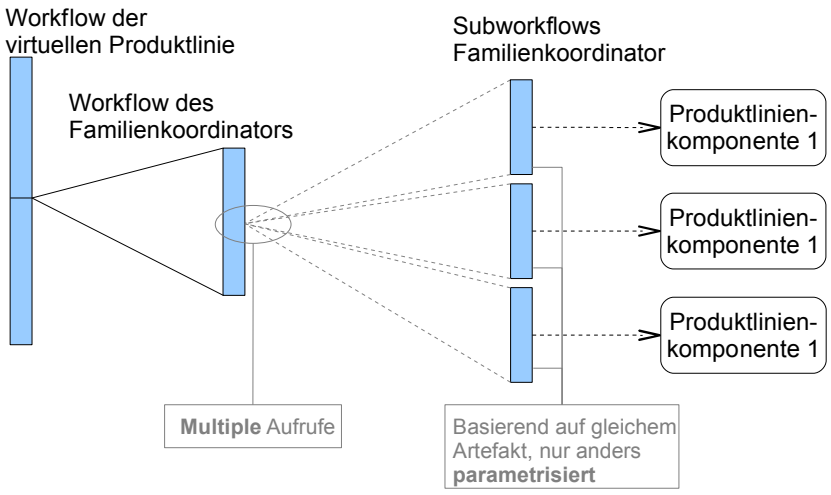


Abbildung 8.4: Parametrisierbare Subworkflows im Familienkoordinator

- **Multiplizität**

  Wie ist es zu erreichen, dass von einem Workflow aus *mehrere* Subworkflows gestartet werden können, deren Anzahl nicht im vornherein festliegt? Für diesen Fall stellt oAW bisher keinen Mechanismus bereit.

- **Parametrisierung**

  Aus welcher Quelle bezieht man die Informationen, *welche* Produktlinienkomponenten mit einzubeziehen sind und *wie* erreicht man, dass im Subworkflow die Informationen über die zu integrierenden Komponente ankommen und ausgewertet werden können?

- **Verhaltensspezifikation**

  Welche Hilfsmittel unterstützen den Entwickler des Familienintegrators bei Aufgaben wie dem Erstellen des Integrationsmerkmalmodells und zum Erweitern von Merkmal- und Konzeptmetamodellen?

Diese Fragen werden im anschließenden Kapitel beantwortet, wobei die ersten beiden eng mit der internen Funktionsweise von oAW zusammenhängen und daher im Rahmen dessen Erweiterung in Abschnitt 9.1 behandelt werden. Die letzte Frage wird durch die den Integrationsprozess unterstützenden Funktionen des PLiC Frameworks in Abschnitt 9.2 beantwortet.

8.3 Zusammenfassung

Dieses Kapitel befasste sich mit der Umsetzung des Integrationsprozesses in Bezug auf die technischen Möglichkeiten von openArchitectureWare. Der Workflow-Mecha-

nismus wurde als entscheidender Angriffspunkt für diese Aufgabe erkannt. Die Funktionen, welche die Workflows in den verschiedenen Phasen zu erfüllen haben, wurden skizziert. Dabei handelt es sich um die Erstellung des Integrationsmerkmalmodells im *Bootstrap-Workflow*, die Erweiterung von Merkmalmodell und Konzeptmetamodell im *Integrations-Workflow* und die Vorbereitung der MDSD-Umgebung mittels Registration von Advice-Spezifikationen und Slots im *Laufzeit-Workflow*.
Daraufhin erfolgte die Vorstellung eines Konzepts für den Familienintegrator. Durch die Definition der kompletten Workflow-Logik durch ihn selbst wird Redundanz in den Komponenten vermieden und die Schnittstelle zur Produktlinie und den Komponenten klar festgeschrieben.

9 Implementation: PLiC Framework

Das PLiC Framework bildet die grundlegende Plattform für den Aufbau von Produktlinienfamilien. Es soll Abstraktionen bereitstellen, mit denen Produktlinienkoordinatoren definiert werden können, die wiederum von virtuellen Produktlinien genutzt werden können, um eine Basisproduktlinie mit Komponenten anzureichern.
Der Workflow-Mechanismus von oAW ist für die Definition von komplexeren modellgetriebenen Abläufen nicht flexibel genug. Er unterstützt nahezu keine Beschreibung von Kontrollfluss und insbesondere nicht die von Schleifen. Da jedoch die dynamische Einbindung von einer prinzipiell beliebigen Anzahl von Komponenten notwendig ist, muss er entsprechend erweitert werden, was in Abschnitt 9.1 geschieht.
Erst anschließend kann das eigentliche Framework mit seinen Funktionalitäten beschrieben werden, was in Abschnitt 9.2 erfolgt.

9.1 Erweiterung des Workflow-Mechanismus

Dieser Abschnitt beschreibt die am Workflow-Mechanismus von oAW durchgeführten Erweiterungen. Insbesondere soll es möglich werden, *multiple* Aufrufe von Subworkflows *parametrisiert* durchführen zu können.

9.1.1 Analyse möglicher Erweiterungsmechanismen

Wie bereits in Abschnitt 7.1.1 erwähnt, bieten Workflows bis auf optionale Ausführung aufgrund der Merkmalauswahl keine Kontrollflüsse. Dies reicht nicht aus, um eine beliebige Anzahl an Produktlinienkomponenten in den Ablauf integrieren zu können Daher muss der Mechanismus ausgebaut werden.
Eine direkte Erweiterung der Syntax und Semantik der Workflows um explizite Kontrollflusselemente wie `<while>` oder `<for>` wäre eine durchaus naheliegende Option. Aufgrund des Aufwands, den die Erweiterung des Syntaxbaumes des Workflow-Parsers mit sich bringt und wegen des begrenzten Zeitrahmens ließ sich dies jedoch nicht verwirklichen. Zudem würde durch den zunehmenden Ausbau der Mächtigkeit der Sprachkonstrukte die Grenze zu ausgewachsenen Programmiersprachen verschwimmen. Die intendierte Einfachheit der Ablaufdefinition würde dadurch verschwinden.
Somit bleibt noch die Erweiterung über die zentralen Elemente der Workflows, Workflow-Cartridges und Workflow-Komponenten, zu untersuchen:

- Workflow-Cartridges

Diese bieten, wie erwähnt, nur eine „include“-artige Semantik. Die Bestandteile des aufgerufenen Workflows werden bisher bereits zur Parse-Zeit direkt in den Syntaxbaum des ihn einbindenden Workflows integriert. Somit kommt eine Erweiterung über diesen Mechanismus einer kompletten Überarbeitung des Syntaxbaums gleich.

- Workflow-Komponenten

  Da Workflow-Komponenten eigens dafür vorgesehen sind, Workflows mit eigener Funktionalität anzureichern, bieten sie den viel versprechendsten Ansatz, die Workflow-Semantik mit vertretbaren Mitteln zu erweitern.

9.1.2 Erweiterung über Workflow-Komponenten

Alle für den Anwendungserstellungsprozess notwendigen Funktionalitäten, wie Modelle einlesen und schreiben, transformieren, Code generieren u. Ä. sind bisher als Workflow-Komponenten definiert. Genauso wie beim Weben eines Advice (siehe Abbildung 9.1, links) wird immer nur genau eine Aktion durchgeführt. Jede dieser Komponenten könnte nun neu geschrieben werden, um z. B. statt nur eines Advice die Advice-Spezifikationen aller über die Merkmalauswahl konfigurierten Produktlinienkomponenten zu weben (siehe 9.1, rechts). Für diese Funktionalität müsste jedoch jede Workflow-Komponente erneut implementiert oder erweitert werden.

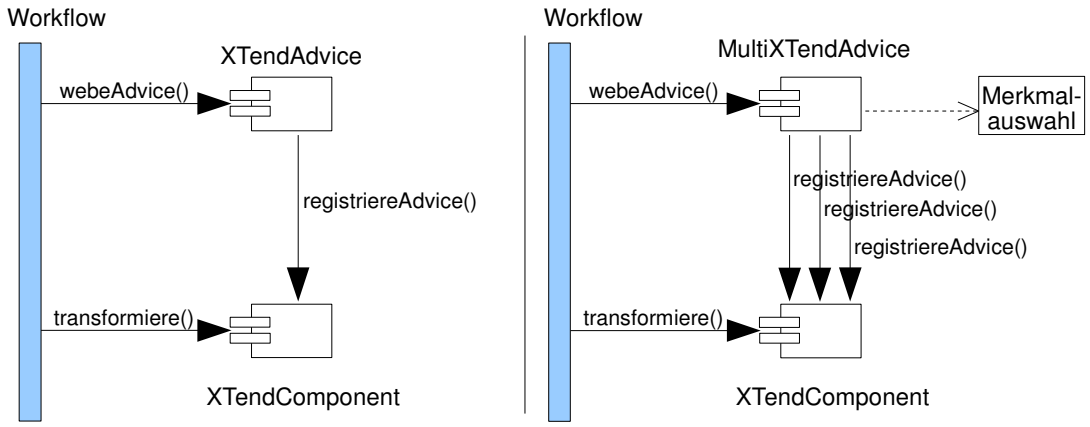


Abbildung 9.1: Mögliche Erweiterung einer XTendAdvice-Worfkflow-Komponente

Eine flexiblere Möglichkeit bietet sich, wenn man eine Workflow-Komponente mit eigenen, erweiterten Cartridge-Fähigkeiten implementiert. Während der bisherige Cartridge-Mechanismus eine rein inkludierende Semantik hat (siehe Abbildung 9.2, links), kann eine Workflow-Komponente , Dank ihrer Implementierung in Java, eine beliebige Anzahl neuer *Sub*workflows starten (siehe Abbildung 9.2, rechts).

In den Subworkflows können die Workflow-Komponenten von oAW dann ohne weitere Veränderung verwendet werden. Die beschriebene Komponente wird im Folgenden `MultiCartridge` genannt.

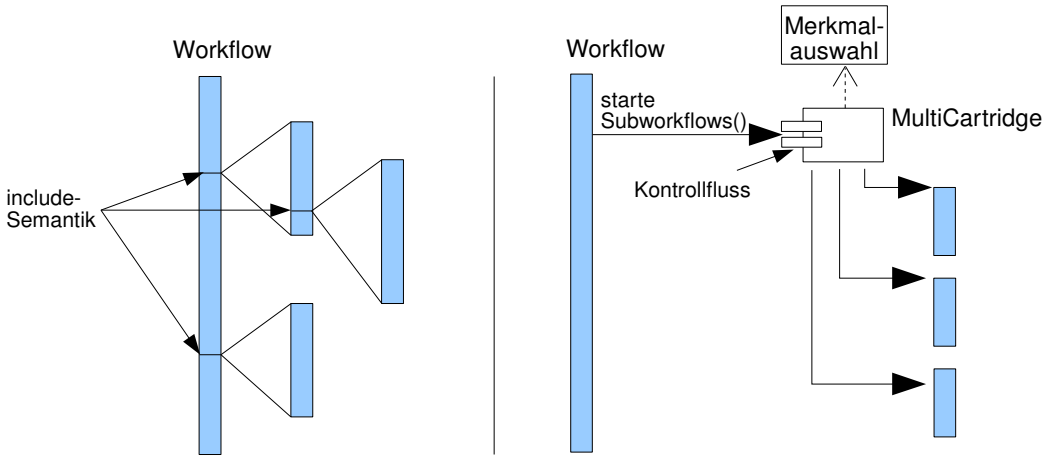


Abbildung 9.2: Der Cartridge-Mechanismus von oAW und die MultiCartridge

9.1.3 MultiCartridge für Produktlinienkomponenten

Die `MultiCartridge` wurde eigentlich als Teil des PLiC Frameworks implementiert. Da sie jedoch einige Veränderungen in der Implementierung von oAW nötig macht, wird sie bereits an dieser Stelle entwickelt.

9.1.3.1 Auswahl des Kontrollflussmechanismus

Durch die als Workflow-Komponente implementierte `MultiCartridge` kann nun die komplette Mächtigkeit von Java verwendet werden. Über die dafür vorgesehene Klasse `WorkflowRunner` des oAW-Frameworks kann sie beliebig viele neue Subworkflows starten. Die Anforderungen sind aufgrund des Produktlinienkomponenten-Kontextes jedoch recht spezifisch, es müssen daher keine beliebigen Kontrollflussmechanismen unterstützt werden. Konkret ist es nur nötig, dass die `MultiCartridge` über die ausgewählten Produktlinienkomponenten informiert ist und entsprechend agiert. Zudem ist es ausreichend, für jede Produktlinienkomponente genau einen Subworkflow zu starten, da in diesem prinzipiell beliebige Logik enthalten sein kann. Für die `MultiCartridge` bieten sich zwei Implementationsalternativen:

- Je Produktlinienkomponente wird jeweils ein separater, in einer anderen Datei implementierter Subworkflow gestartet (Abbildung 9.3, links).

- Es wird immer die gleiche Datei aufgerufen um die Subworkflows zu starten, jedoch unterschiedlich *parametrisiert*(Abbildung 9.3, rechts).

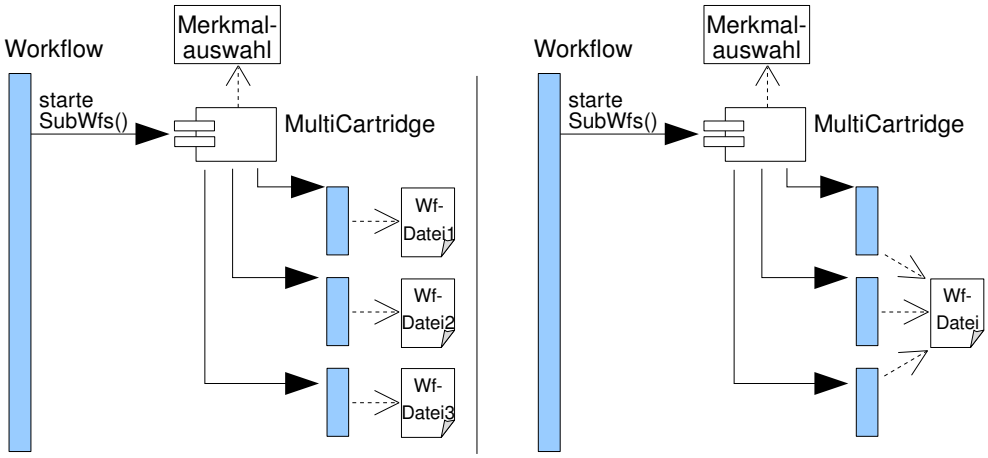


Abbildung 9.3: Alternative Entwürfe für die MultiCartridge

Hierzu ist folgende Überlegung anzustellen: Der Erstellungsprozess der Produktlinien innerhalb einer Produktlinienfamilie soll in so weit standardisiert sein, dass sich der Komponente eine einheitliche Schnittstelle bietet. Daher ist damit zu rechnen, dass die Subworkflows der Produktlinienkomponenten häufig sehr ähnliche Bedürfnisse haben (Modelle und Anwendungslogik hinzügen, Advice-Spezifikationen registrieren, ...). Um dies zu unterstützen, wäre eine Parametrisierung eines Subworkflows aus Gründen der Redundanzvermeidung und zum einfacheren Entwurf von Produktlinienkomponenten der von separaten Subworkflows vorzuziehen. Daher werde ich im Folgenden auf erstgenannte Alternative näher eingehen.

9.1.3.2 Auswahl des Parametrisierungsmethode

Bei der Parametrisierung von Subworkflows handelt es sich um eine Art funktionaler Abstraktion: Der Subworkflow selbst stellt eine Funktion dar, die mit Hilfe übergebener Parameter, im konkreten Fall oAW-Properties (siehe auch Abschnitt 7.1.1.1), erfolgt. Genau genommen müssen sich die übergebenen Werte je Subworkflow-Aufruf nur in exakt einer Property unterscheiden. Diese wird im Folgenden entsprechend der Schreibweise für Properties in Workflows als `${plicName}` bezeichnet.
Die `MultiCartridge` wird speziell auf den Bedarf von Produktlinienfamilien ausgelegt. Daher benötigt sie Informationen über die aktuell ausgewählten Produktlinienkomponenten, um die Property `${plicName}` entsprechend zu setzen. In oAW ist für Konfigurationsdaten das globale „Singleton-Objekt“ `GlobalConfiguration` vorgesehen, in welches jeweils Variantenmodelle eingelesen werden können.

Je nachdem, ob es sich um die Phase der Komponentenintegration oder der Anwendungserstellung handelt, enthält `GlobalConfiguration` ein anderes Variantenmodell. In der erstgenannten Phase handelt es sich um ein Integrationsvariantenmodell, in der zweitgenannten um das angereicherte Variantenmodell der virtuellen Produktlinie zur Laufzeit.
Um nun die Informationen, für *welche* Produktlinienkomponenten ein Subworkflow aufgerufen werden soll, in beiden Fällen aus diesen Variantenmodellen extrahieren zu können, wird zunächst verlangt, das *jede* Komponente sowohl ein Integrations- als auch ein Laufzeitmerkmalmodell mit sich führen muss. In beiden Fällen müssen diese jeweils genau ein Merkmal besitzen, das als Namen den eindeutigen Bezeichner der Komponente trägt.
Im Integrationsmerkmalmodell der Komponente muss es sich hierbei um ein optionales Merkmal handeln. In der Bootstrap-Phase werden die einzelnen Integrationsmerkmalmodelle der Komponenten dann zu einem verwebt, was durch Verschmelzung der Konzeptknoten geschieht. Es ergibt sich dadurch ein Merkmalmodell, in dem für jede Komponente der Familie in der Komponentenkonfigurationsphase einzeln bestimmt werden kann, ob ihre Integration stattfinden soll oder nicht.
Im Laufzeitmerkmalmodell einer Komponente muss die Auswahl des Merkmals mit dem eindeutigen Bezeichner obligatorisch sein. Nach dem Weben der Laufzeitmerkmalmodelle der Komponenten in das Merkmalmodell der Basisproduktlinie in der Komponentenintegrationsphase enthält dieses somit ein zusätzliches Merkmal. Dieses selbst hat zwar keinen Einfluss auf die Variabilität, jedoch ist daraus ablesbar, *dass* die Komponente mit diesem Namen integriert wurde, und *dass* sie in der Anwendungserstellungsphase von der `MultiCartridge` zu berücksichtigen ist.
Um die anderen Merkmale von diesen speziellen Komponentenmerkmalen unterscheiden zu können, wird gefordert, dass jedem dieser Komponentenmerkmale das Attribut `plicComponent=true` gesetzt wird.
Abbildung 9.4 verdeutlicht dies anhand eines Beispielmerkmalmodells eines bereits gewobenen Laufzeitmerkmalmodells einer virtuellen Produktlinie.
Die angereicherte Produklinie enthält die beiden Produktlinienkomponenten mit Namen `Authentifizierung` und `Datenbankverbindung`. Eine `MultiCartridge` würde die ihr übergebene Subworkflow-Datei also genau zweimal aufrufen, wobei die diskriminierende Property `${plicName}` jeweils auf den aktuellen Komponentennamen gesetzt wird.

9.1.3.3 Vererbung von Properties

Die übliche Workflow-Cartridge von oAW unterstützt es, mittels des `inheritAll`-Attributs alle Properties des aufrufenden Workflows in den aufgerufenen zu übernehmen. Diese praktische Fähigkeit ist jedoch eng mit dem nativen Cartridge-Mechanismus von oAW verknüpft und kann daher nicht für Workflow-Komponenten wie die `MultiCartridge` verwendet werden.
Daher wurde die Klasse `WorkflowParser` von oAW um die Fähigkeit erweitert, beliebigen Workflow-Komponenten, die ein `inheritAll`-XML-Attribut mit dem Wert

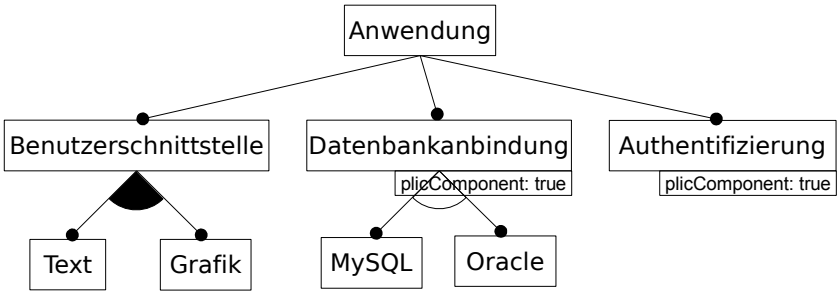


Abbildung 9.4: Verschmolzenes Merkmaldiagramm mit plicComponent-Attributen

`true` besitzen, alle Properties des übergeordneten Workflows als Parameter zu übergeben. Die `MultiCartridge` reicht die erhaltenen Parameter dann wiederum an alle Subworkflows als Properties weiter.

9.1.3.4 Erweiterung des Property-Parsers von Workflows

Da nun ein Subworkflow mehrere Male mit jeweils unterschiedlich gesetzter Property `${plicName}` aufgerufen wird, stellt sich die Frage, wie sich dieser Umstand ausnutzen lässt, um dynamisch einen anderen Advice zu registrieren oder anderen Code zu generieren.
Tatsächlich erweist sich der bisherige Mechanismus zur Auflösung von Properties als nicht sehr mächtig. Er besteht nämlich allein darin, ein einziges Mal alle Zeichenfolgen nach Ausdrücken wie `${myProperty}` abzusuchen und diese durch den Wert der Property, also z. B. `myPropertyValue` zu ersetzen.
Wie in Abschnitt 8.2 festgesetzt, stellen Produktlinienkomponenten passiv Artefakte wie Modelle oder Transformatoren an bestimmten Stellen bereit, die durch Subworkflows des Produktlinienkoordinators entsprechend verarbeitet werden. Um allein durch den unterschiedlichen Wert der Property `${plicName}` die unterschiedlichen Artefakte auffinden zu können, dürfen sich die Pfadnamen der Komponenten also immer nur um diesen einen Teil unterscheiden. Die Angabe des Attributs `modelUri="${plicName}/arte.fakt"` würde dann je nach Wert der Property entweder die Zeichenfolge `"plicA/arte.fakt"` oder `"plicB/arte.fakt"` erzeugen.
Der Property-Parser von oAW befindet sich in der Klasse `VisitorInitializer` und wurde um zwei Fähigkeiten erweitert, um nicht diesen Einschränkungen unterliegen zu müssen:

- Rekursive Ersetzung von Properties

  Properties können nun in andere geschachtelt sein. Hat die Property mit Namen `${lowerCaseO}` den Wert `o`, sind so sinnfreie wie gültige Ersetzungsfolgen ermöglicht wie:

```
"${MyPr${lowerCaseO}perty}" ⇒
"${MyProperty}" ⇒
"MyPropertyValue"
```

Nützlich wird das wiederholte Ersetzen der Properties erst durch die nachfolgend beschriebene Erweiterung.

- Ermöglichen von Funktionsaufrufen über den Property-Mechanismus

  Anstatt die Ersetzungen von Property-Namen zu Property-Werten aus einer Datenstruktur des Syntaxbaumes zu beziehen, kann der Mechanismus mit etwas Erweiterung auch zum Aufruf von Funktionen verwendet werden. Dabei entsteht folgende Ersetzungskette:

  ```
  "${mypack.MyClass.myMethod(${var1}, ${var2})}" ⇒
  "${mypack.MyClass.myMethod(value1, value2)}" ⇒
  "myMethodResult"
  ```

  Damit dieser Aufruf funktioniert, müssen die Properties `var1` und `var2` im aktuellen Workflow definiert sein, und im Klassenpfad muss eine statische Methode mit Namen `myMethod` in der Klasse `mypack.MyClass` existieren. Sie muss zwei String-Parameter annehmen und ihr Rückgabewert muss ebenfalls ein String sein. Der Methoden-Parser ist dabei über die Reflektionsmechanismen von Java implementiert.

Somit lässt sich nun die vollständige Mächtigkeit von Java direkt aus Workflows heraus nutzen. Die Property `${plicName}` kann nun an statische Methoden übergeben werden, die daraus beliebig komplexe Zeichenketten erzeugen können.
Für viele Fälle reicht jedoch immer noch der ursprüngliche Property-Ersetzungsmechanismus aus, solange man sich nur darauf beschränkt, dass alle Produktlinienkomponenten sich in einem Verzeichnis befinden, das genau den gleichen Namen trägt wie ihr eindeutiger Bezeichner, und ihre sonstige Verzeichnisstruktur identisch ist.

9.1.4 Workflow-Hierarchien zur Registration von Advice-Spezifikationen und Slots

Wie bereits in Abschnitt 8.1.3 erwähnt, besteht eine der Hauptaufgaben von Subworkflows während der Anwendungserstellungsphase des Integrationsprozesses darin, Advice-Spezifikationen um Transformationen oder Generierungen zu registrieren. Der dazu nötige Mechanismus basiert, wie in Abschnitt 7.1.2.1 erläutert, auf eindeutigen Identifikatoren (IDs) der zugehörigen Workflow-Komponenten, auf die man sich bei der Advice-Spezifikation mittels des Attributs `adviceTarget` beziehen kann.
Ein Workflow wird bereits vor Beginn des des Anwendungserstellungsprozesses durch oAW vollständig geparst. Ein Advice sucht hierbei schon vor Ausführung der ersten Workflow-Komponente nach einer Transformator- bzw. Generator-Komponente mit der entsprechenden ID und registrieren sich bei ihr. Sobald die Komponente dann an

die Reihe kommt wird, sorgt sie selbst für die Ausführung jedes registrierten Around-Advice in der Reihenfolge wie sie registriert wurden.
Das Problem beim Aufruf von Subworkflows über die `MultiCartridge` ist, dass die Subworkflows eigentlich einen komplett vom aufrufenden Workflow getrennten Suchraum darstellen. Die IDs des Workflows sind also im Subworkflow nicht vorhanden, so dass die Registration fehlschlagen muss.
Um dennoch die Registration von Advice-Spezifikationen aus diesen Subworkflows heraus zu ermöglichen, muss eine Verbindung zwischen diesen beiden hergestellt werden. Die `MultiCartridge` stellt hierbei das integrierende Element dar, da sie noch im Kontext des aufrufenden Workflow läuft. Durch die Erweiterungen der oAW-Klasse `WorkflowRunner`, die in der `MultiCartridge` für den Start eines Subworkflows verwendet wird, erhält diese nun im Konstruktur eine Referenz auf den aufrufenden Workflow.
Durch Erweiterung der Klasse `CompositeComponent` wird nun jedes Mal, wenn die Suche einer ID im Namensraum eines Workflows fehlschlägt, überprüft, ob eine Referenz auf einen hierarchisch übergeordnenten Workflow existiert. Ist dies der Fall, wird anhand dieser Referenz in diesem weitergesucht, bis entweder eine Workflow-Komponente mit entsprechender ID gefunden wird oder kein weiterer übergeordneter Workflow mehr existiert.
Ein ähnlicher Mechanismus wurde zudem implementiert, um es zu ermöglichen, in Slots (siehe Abschnitt 7.1.1.1) des übergeordneten Workflows Modelle einzufügen. Diese können dann bei Ablauf des Anwendungserstellungsprozesses der Basisproduktlinie von einem Transformator- oder Generator-Advice wieder ausgelesen und weiter verarbeitet werden. Der beschriebene Mechanismus lässt sich beispielsweise dazu nutzen, neue Modellelemente über einen Transformator-Advice in ein bestehendes Modell hineinzuweben.

9.2 Überblick über das PLiC Framework

Das PLiC Framework ist eine Sammlung von Hilfsmitteln, die den Anwendungsentwickler bei der Implementation des Familienkoordinators unterstützen. Es erleichtert hierbei die Umsetzung aller in Abschnitt 5.3 bestimmten Integrationsphasen, die nicht manuell durch den Benutzer durchzuführen sind, also der Bootstrap–, Komponenteninteg rations- und Anwendungserstellungsphase. Es ist als Ecplise-Plugin im Projekt `plic.util` implementiert.
Im Folgenden sollen nun die wesentlichen Bestandteile des Frameworks vorgestellt werden. Ihre konkrete Anwendung zur Definition einer Produktlinienfamilie erfolgt dann im anschließenden Kapitel.
Da ein Familienkoordinator allein über Workflows implementiert ist, stellt die Verwendung von Workflow-Komponenten einen wesentlichen Bestandteil seiner Entwicklung dar.
Die `MultiCartridge`, die im vorigen Abschnitt bereits erarbeitet wurde, ist im PLiC Framework angesiedelt. Durch sie können gestartete Subworkflows eines Familienko-

ordinators zur Einbidung der Produktlinienkomponenten viele der Workflow-Komponenten verwenden, die bereits Bestandteil von oAW sind. So können sie insbesondere während der Anwendungserstellung Advice-Spezifikationen um diverse Transformatoren weben. Auch ist es während der Komponentenintegrationsphase möglich, das Konzeptmetamodell mit den Basismitteln von oAW zu transformieren, so dass daraus ein erweitertes Metamodell für die virtuelle Produktlinie entstehen kann.
Jedoch sind für einige Aufgaben des Integrationsprozesses noch weitere Hilfsmittel nötig, die im Folgenden erläutert werden.

9.2.1 Bootstrapping

Die Workflow-Komponente `PlicBootstrap` erledigt das Bootstrapping (siehe Abbildung 9.5). Sie erzeugt hierbei ein familienweites Integrationsmerkmalmodell aus den Integrationsmerkmalmodellen, die jede Produktlinienkomponente mit sich führen muss. Das daraus generierte Modell beschreibt dann die Integrationsoptionen aller Produktlinienkomponenten einer Familie. Das Weben der Merkmalmodelle erfolgt hierbei über die Verschmelzung der Konzeptknoten.

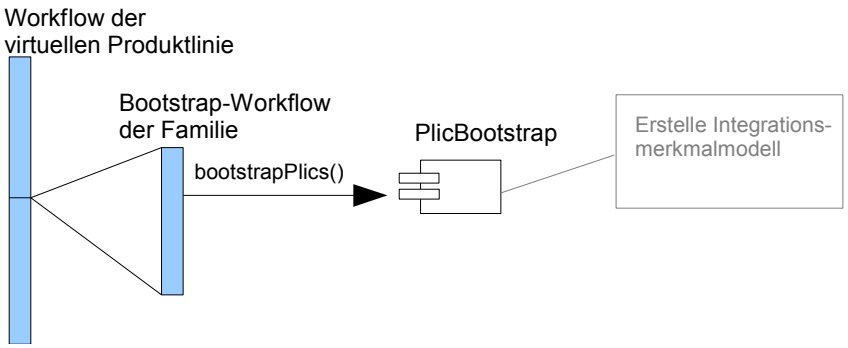


Abbildung 9.5: Bootstrapping mit dem PLiC Framework

9.2.2 Komponentenintegration

In der Komponentenintegrationsphase sind primär die Domänenmetamodelle zu ändern, wozu der Familienintegrator den `MultiCartridge`-Mechanismus verwenden kann. Für die im Integrationsvariantenmodell ausgewählten Komponenten ruft er Subworkflows parametrisiert auf, um die entsprechenden Metamodellerweiterungen vorzunehmen (siehe Abbildung 9.6).
Zur Erweiterung des Konzeptmetamodells kann er hierbei die `XtendComponent` nutzen, die oAW für Modelltransformationen vorsieht. Für das Weben von Merkmalmodellen kann er hingegen die im PLiC Framework implementierte Workflow-Komponente `FeatureModelWeaver` verwenden.

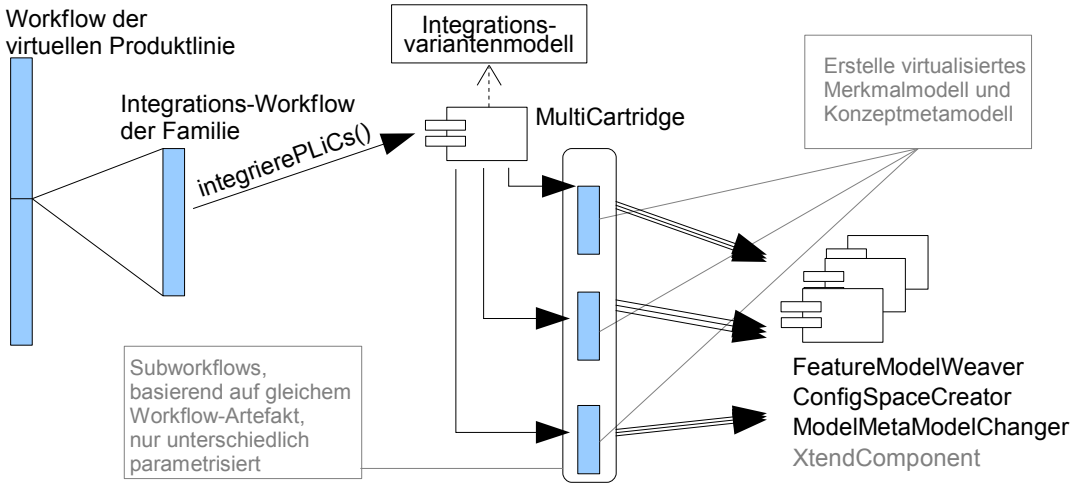


Abbildung 9.6: Komponentenintegration mit dem PLiC Framework

Es können sogar die *Instanzen* dieser Metamodelle, also Konzeptmodell und Variantenmodell, aus der ursprünglichen Produktlinie übernommen werden. Dies ermöglichen der `ConfigSpaceCreator`, der für das zu übernehmende Variantenmodell einen neuen Konfigurationsbereich erstellt (pure::variants-spezifisch), und der so genannte `ModelMetaModelChanger`, der die Referenz auf das Metamodell eines Modells ändert.
Mit den beiden letztgenannten Funktionen ist jedoch mit Bedacht umzugehen, da sie je nach Ausmaß der Veränderungen in den Metamodellen fehlschlagen können. Im Zweifel ist es daher besser, komplett neue Modelle anzulegen, auch wenn dadurch der Konfigurationsaufwand, der bisher in die Basisproduktlinie gesteckt wurde, verloren geht.

9.2.3 Anwendungserstellung

Das Ziel des Familienkoordinators in der Anwendungserstellungphase besteht darin, den Workflow der Basisproduktlinie mit Funktionalität und Modellen anzureichern. Dies geschieht, indem Advice-Spezifikationen um die Transformatoren und Generatoren der Basisproduktlinie gewebt und deren List-Slots mit weiteren Modellen bestückt werden. Dank der bereits beschriebenen Erweiterungen von oAW können die Workflow-Komponenten von openArchitectureWare, die für diese Aufgabe zuständig sind (`XtendAdvice` und `GeneratorAdvice`) ohne weitere Anpassungen dafür verwendet werden. Zum Einfügen von Modellen in List-Slots, die dann innerhalb der Transformatoren und Generatoren wieder ausgelesen werden können, bietet das Framework noch zusätzlich den `RootSlotListAdder`. Schließlich können Produktlinienkomponenten auch noch manuell implementierte Anwendungslogik mitbringen, deren Integration der Familienkoordinator ebenfalls erledigt.

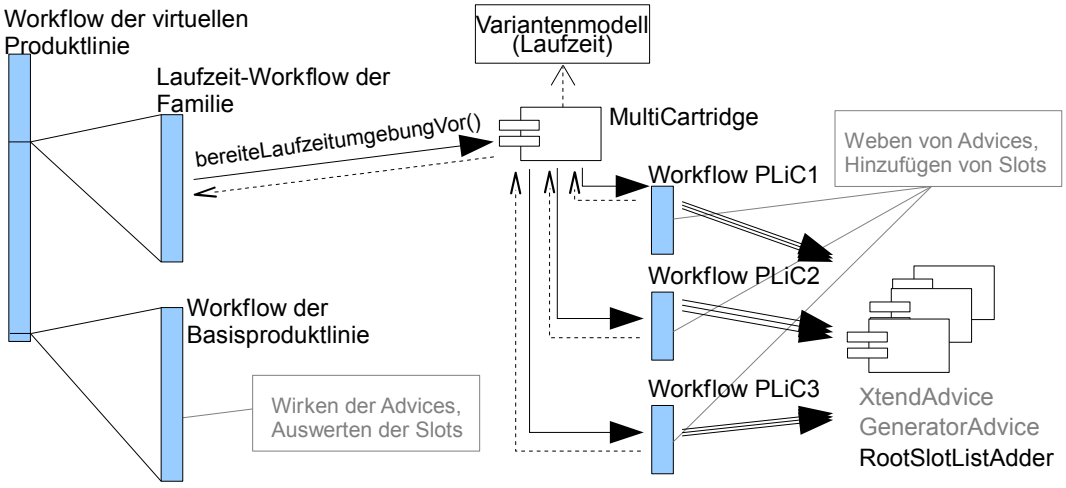


Abbildung 9.7: Anwendungserstellung mit dem PLiC Framework

9.2.4 Implementationsdetails

Die Workflow-Komponenten des Framworks sind abgeleitet von der abstrakten Klasse `AbstractWorkflowComponent2` (siehe Abbildung 9.8).

Somit müssen alle Komponenten die Methoden `checkConfigurationInternal()` und `invokeInternal()` implementieren. Erstgenannte überprüft, ob alle nötigen Parameter vorhanden sind, während die andere die Anwendungslogik enthält. Der Aufruf der Methoden wird durch die „Workflow-Engine“ von oAW erledigt, sobald die Komponenten im Workflow-Ablauf an die Reihe kommen. Sie sorgt auch im Vorfeld für die Parameterübergabe an die Workflow-Komponenten.

Die Übergabe von Parametern erfolgt jeweils über *Dependency Injection*. Dabei wird für jedes XML-Attribut oder -Element, das eine Workflow-Komponente in der XML-artigen Workflow-Datei gesetzt bekommt (siehe Abbildung 7.2), eine entsprechend benannte Funktion der Workflow-Komponente aufgerufen.

Bekommt also eine Komponente das Attribut oder Element `myParam` im Workflow gesetzt, versucht oAW, eine der drei Funktionen `setMyParam(String value)` oder `addMyParam(String value)` oder `put(String name, String value)` in deren Java-Implementation über Reflektionsmechanismen (*Reflection*) aufzurufen. Gelingt dies nicht, bricht der Workflow mit einem Fehler ab. Für die Übergabe mehrerer Parameter gleichen Namens müssen diese jeweils als XML-Element im Workflow spezifiziert sein.

Wird im Folgenden also von der Übergabe von Parametern an eine Komponente gesprochen, ist eigentlich dieser Mechanismus gemeint.

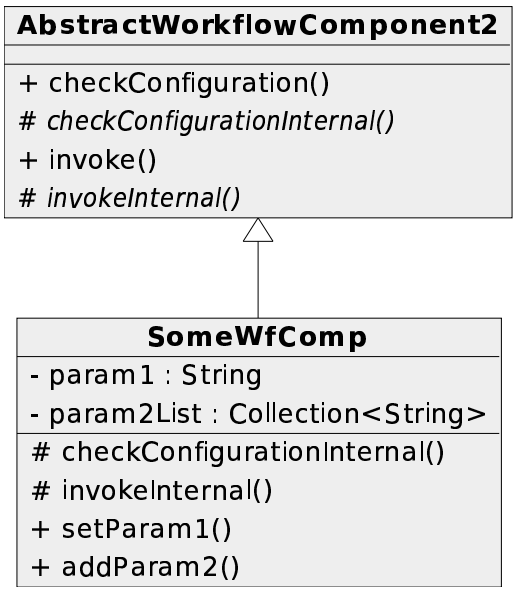


Abbildung 9.8: Die Komponente AbstractWorkflowComponent2

9.2.4.1 PlicBootstrap

Die Workflow-Komponente `PlicBootstrap` erledigt das Bootstrapping durch Erzeugung eines familienweiten Integrationsmerkmalmodells aus den Integrationsmerkmalmodellen, die jede Produktlinienkomponente mitführen muss. Die Modelle werden dabei ausgehend von dem leeren Merkmalmodell `templateFM.sfm` über wiederholte Anwendung des `FeatureModelWeaver` (s. u.) zusammengefügt.
`PlicBootstrap` erwartet folgende Parameter:

- `modelPart`

  Pfad zum Submerkmalmodell einer Produktlinienkomponente. Es können beliebig viele `modelPart` Parameter übergeben werden.

- `integrationFeatureModel`

  Pfad, an dem das familienweite Integrationsmerkmalmodell erstellt werden soll.

Verwendete Ressourcen:

- `plic/util/resources/templateFM.sfm`

  Ein leeres Merkmalmodell, in das weitere Merkmale eingewebt werden können.

9.2.4.2 FeatureModelWeaver

Der `FeatureModelWeaver` webt pure::variants Merkmalmodelle auf XML-Ebene zusammen. Er erreicht dies durch Verschmelzen der Konzeptknoten (siehe Abschnitt 4.1.4.4).

`FeatureModelWeaver` erwartet folgende Parameter:

- `inFile`

  Pfad zum Basismerkmalmodell, in das weitere Merkmale eingewebt werden sollen.

- `weaveFile`

  Pfad zum Merkmalmodell, das hineingewebt werden soll.

- `outFile`

  Pfad des gewebten Merkmalmodells.

9.2.4.3 ConfigSpaceCreator

Der `ConfigSpaceCreator` erstellt einen neuen pure::variants Konfigurationsbereich (*ConfigSpace*), basierend auf einem konkreten Merkmalmodell. In dem Konfigurationsbereich können dann Variantenmodelle basierend auf diesem Merkmalmodell angelegt werden. Seine Erstellung erfolgt durch entsprechende Anpassung der „Template-Datei“ `configspace.xml` auf XML-Basis.
`ConfigSpaceCreator` erwartet folgende Parameter:

- `destFile`

  Pfad, an dem die neue Configspace-Datei entstehen soll.

- `featureModel`

  Pfad zum Merkmalmodell, auf dem der Configspace basieren soll.

Verwendete Ressourcen:

- `plic/util/resources/configspace.xml`

  Eine Template-Datei zur Erstellung des Configspaces.

9.2.4.4 ModelMetaModelChanger

Der `ModelMetaModelChanger` ändert das Metamodell, auf das ein auf Ecore basierendes Modell verweist. Die Änderung der Referenz auf das Metamodell erfolgt auf XML-Basis.
`ModelMetaModelChanger` erwartet folgende Parameter:

- `inFile`

  Pfad, an dem sich das ursprüngliche Ecore-Modell befindet.

- `outFile`

  Pfad, an dem das neue Ecore-Modell erstellt werden soll.

- `mmFile`

  Pfad des Metamodells, welches das neu erstellte Modell in `outFile` erhalten soll.

9.2.4.5 MultiCartridge

Die `MultiCartridge` ermöglicht das wiederholte, parametrisierte Aufrufen eines Subworkflows und basiert auf den im Rahmen dieser Arbeit erweiterten Workflow-Mechanismen von oAW. Sie nimmt prinzipiell beliebige Argumente entgegen, die sie jeweils an den Subworkflow weiterreicht.
Wird sie aufgerufen, wertet sie die aktuelle Konfiguration aus (das Variantenmodell). Für jedes ausgewählte Merkmal, welches das Attribut `plicComponent="true"` trägt, wird der Subworkflow erneut aufgerufen. Dieser erhält dann die zusätzliche Property `plicName`, die dann den Namen des eben erwähnten attributierten Merkmals enthält.
`MultiCartridge` erwartet folgende Parameter:

- `cartridgeFile`

  Pfad, an dem sich die Workflow-Datei des zu startenden Subworkflows befindet.

- `inheritAll` (optional)

  Dieser Parameter wird nicht von der Komponente direkt, sondern vom erweiterten oAW-Framework verarbeitet. Er bewirkt die Vererbung aller Properties des aufrufenden Workflows an die Subworkflows.

9.2.4.6 RootSlotListAdder

Der `RootSlotListAdder` fügt dem Root-Workflow ein Modell in einen List-Slot ein. Dabei nutzt er den erweiterten Workflow-Mechanismus, um auf die Slots der übergeordneten Workflows zugreifen zu können.
`RootSlotListAdder` erwartet folgende Parameter:

- `listSlotName`

  Name des List-Slots, zu dem ein Modell hinzugefügt werden soll.

- `modelUri`

  Pfad der Datei, die das Modell enthält.

- `plicName`

  Name der Produktlinienkomponente, die das Modell zum Slot hinzufügen möchte (wird benötigt, um dem Modell eine eindeutige ID im List-Slot zuweisen zu können).

9.3 Zusammenfassung

Dieses Kapitel behandelte zunächst die vorgenommenen Erweiterungen an openArchitectureWare, die für den Produktlinienfamilienansatz nötig wurden. Sie ermöglichen der Workflow-Komponente `MultiCartridge`, auf Basis der Merkmalauswahl eines Variantenmodells mehrfach und parametrisiert den gleichen *Subworkflow* zu starten. Über zusätzliche Erweiterungen wird der Aufruf von Funktionen innerhalb von

Workflows ermöglicht, sowie die Registration eines Advice oder Slots, auch wenn sich diese in einem übergeordneten Workflow befinden.
Daraufhin erfolgte die Vorstellung des PLiC Frameworks, zunächst auf Basis dessen beabsichtigter Verwendung innerhalb des Integrationsprozesses. Zum Schluss wurden die wichtigsten Funktionen des Frameworks nochmals detailliert erläutert.

10 Implementation: Erstellung einer Produktlinienfamilie

Dieses Kapitel beschreibt die Implementation eines Familienkoordinators auf Basis von SmartHome, sowie die Erstellung einer Produktlinienkomponente für Betriebssicherheit. Abschließend wird anhand einer neu definierten virtuellen Produktlinie der Integrationsprozess konkret veranschaulicht.
Es wird im Folgenden davon ausgegangen, dass SmartHome bereits entsprechend dafür vorbereitet ist, Produktlinienkomponenten zu empfangen und in eine Produktlinienfamilie integriert zu werden. Tatsächlich hat sich im Laufe der Implementation herausgestellt, dass sich der Produktlinienfamilienansatz nahezu ohne Eingriffe in die Basisproduktlinie bewerkstelligen lässt. An den Stellen, wo dennoch Änderungen nötig waren, wird nachfolgend entsprechend darauf eingegangen.

10.1 Domänenanalyse

Die Domänenanalyse beschäftigt sich dieser Iteration nur mit den Erweiterungen, die durch die Implementation der neuen Produktlinienkomponenten verursacht werden. Auf den Familienkoordinator als technische Umsetzung einer Familie wird hingegen erst in der Entwurfs- und Implementierungsphase näher eingegangen.
Die Produktlinienfamilie soll um eine Komponente für Betriebssicherheit (*Safety*) erweitert werden. Dabei ist die Funktionalität bewusst einfach gehalten, um nicht vom wesentlichen Ziel, der Erstellung einer Produktlinienfamilie, abzuschweifen.
Ohne nun detailliert auf die gesamte Domänenalysephase einzugehen, ist in Abbildung 10.1 bereits das Merkmaldiagramm der Safety-Produktlinienkomponente abgebildet, um das es die Produktlinie SmartHome erweitern soll. Das Merkmal `safety`, hat dabei nur ein Submerkmal namens `errorinjection`, welches optional ist. Es soll in der Endanwendung die Injektion von Fehlern in Anwendungskomponenten über eine grafische Schnittstelle ermöglichen, um die Betriebssicherheitsmechanismen zu testen. Zu sehen ist weiterhin das Merkmalattribut `plicComponent`, das der `MultiComponent` in der Anwendungserstellungsphase anzeigt, dass es sich bei `safety` um den eindeutigen Bezeichner einer Produktlinienkomponente handelt.
Die eigentliche Betriebssicherheit wird durch dreifach redundante Auslegung (*Triple Modular Redundancy*, *TMR*) der Steuergeräte auf Domänenebene inklusive einer Entscheidungseinheit (*Voter*) erreicht. Um die Redundanz an ausgewählten Geräten des PS-Modells überhaupt aktivieren zu *können*, müssen die *Meta*modellelemente der entsprechenden Geräte erweitert werden. Abbildung 10.2 zeigt die Erweiterung des Metamodellelements `MovementSensor` um das Attribut `isRedundant`.

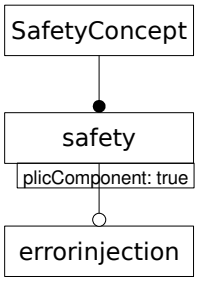


Abbildung 10.1: Merkmaldiagramm der Produktlinienkomponente für Betriebssicherheit

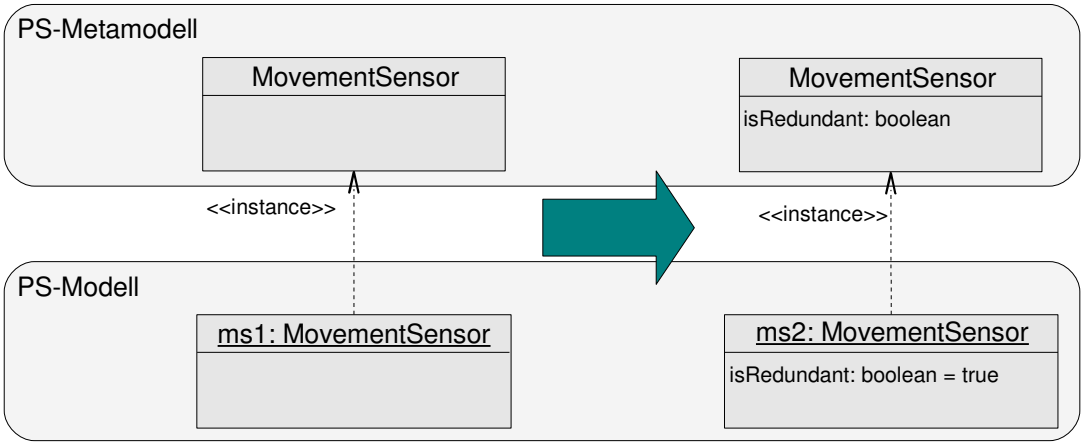


Abbildung 10.2: Erweiterung des Metamodellelements MovementSensor

10.2 Domänenentwurf

10.2.1 Entwurf der Anwendungsreferenzarchitektur

Das Anwendungs- und Instantiierungs*meta*modell sind durch das CBD-Metamodell von SmartHome bereits vorgegeben und gelten somit für alle Mitglieder der Familie als verpflichtend. Es müssen nun noch die Anwendungsmodellelemente für die Safety-Produktlinienkomponente entwickelt werden, eines für die Fehlerinjektion und ein anderes für den Voter.

Die Voter-Anwendungskomponente stellt einen Sonderfall dar. Sie soll transparent zwischen eine Server- und ein Client-Anwendungskomponente geschaltet werden. In diesem Fall muss sie genau die gleiche Schnittstelle wie der Client anbieten, dessen internen Aufbau sie aber erst in der Anwendungserstellungsphase erfährt. Daher muss sowohl die *Implementierung* der Voter-Anwendungskomponente selbst als auch

ihr zugehöriges *Modellelement* dynamisch während des Anwendungserstellungsprozesses erzeugt werden.
Die Fehlerinjektion geschieht mit Hilfe eines programmiersprachlichen Aspekts, dessen komplette Implementierung, sowohl die Handlungsanweisung als auch sein Wirkungsbereich, in AspectJ erfolgt. Das zu dem Aspekt gehörende Modellelement namens `ErrorInjectionAspect` ist eine Instanz von `CompleteCodeAspect`, dem CBD-Metamodellelement zur Modellierung von Aspekten. Da das Modellelement nur entscheidet, *ob* der Aspekt überhaupt angewendet werden soll, enthält es außer dem Namen des Aspekts selbst keine weiteren Attribute oder Relationen (siehe 10.3).

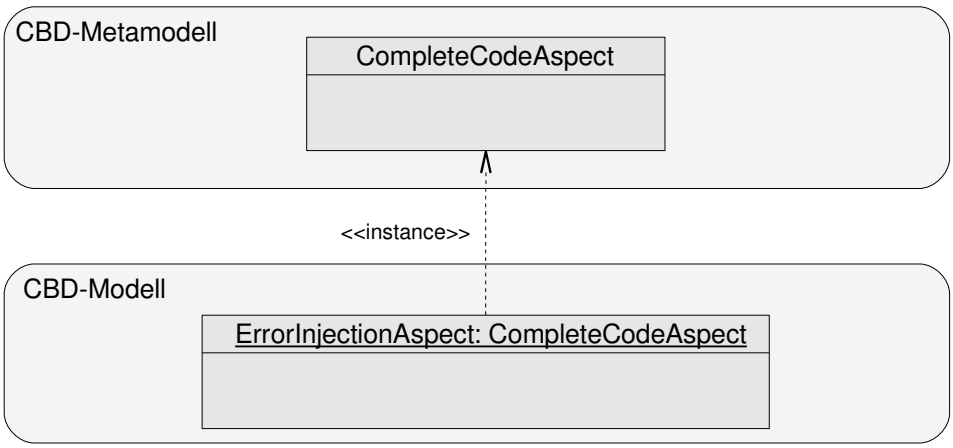


Abbildung 10.3: Modellelement des ErrorInjectionAspect

10.2.2 Definition des Anwendungserstellungsprozesses

Auch der Anwendungserstellungsprozess wird an dem von SmartHome ausgerichtet, der bereits in Abschnitt 7.2 vorgestellt wurde. Es ist nun noch festzulegen, welche „Schnittstellen“ auch nach außen für die Produktlinienkomponenten zur Verfügung gestellt werden sollen.
Abbildung 10.4 zeigt den Anwendungserstellungsprozess für die ganze Familie. Im Gegensatz zur ursprünglichen Anwendungserstellung von SmartHome ermöglicht er es, dynamisch Anwendungslogik aus dem CBD-Modell zu generieren (siehe grünen Pfeil auf PIM-Ebene). Dies wurde aufgrund der dynamischen Generierung der Voter-Anwendungskomponente nötig.
Die schwarz umrandeten Elemente in Abbildung 10.4 zeigen die festgelegten Eingriffsmöglichkeiten der Produktlinienkomponenten an. Es sind nur die Möglichkeiten markiert, die im Folgenden auch tatsächlich benutzt werden.
Die gestrichelten Umrandungen von Variantenmodell und PS-Modell zeigen an, dass in der Komponentenintegrationsphase Einfluss auf ihre Metamodelle (Merkmalmodell und PS-Metamodell) ausgeübt werden kann. Durch die übrigen markierten Ele-

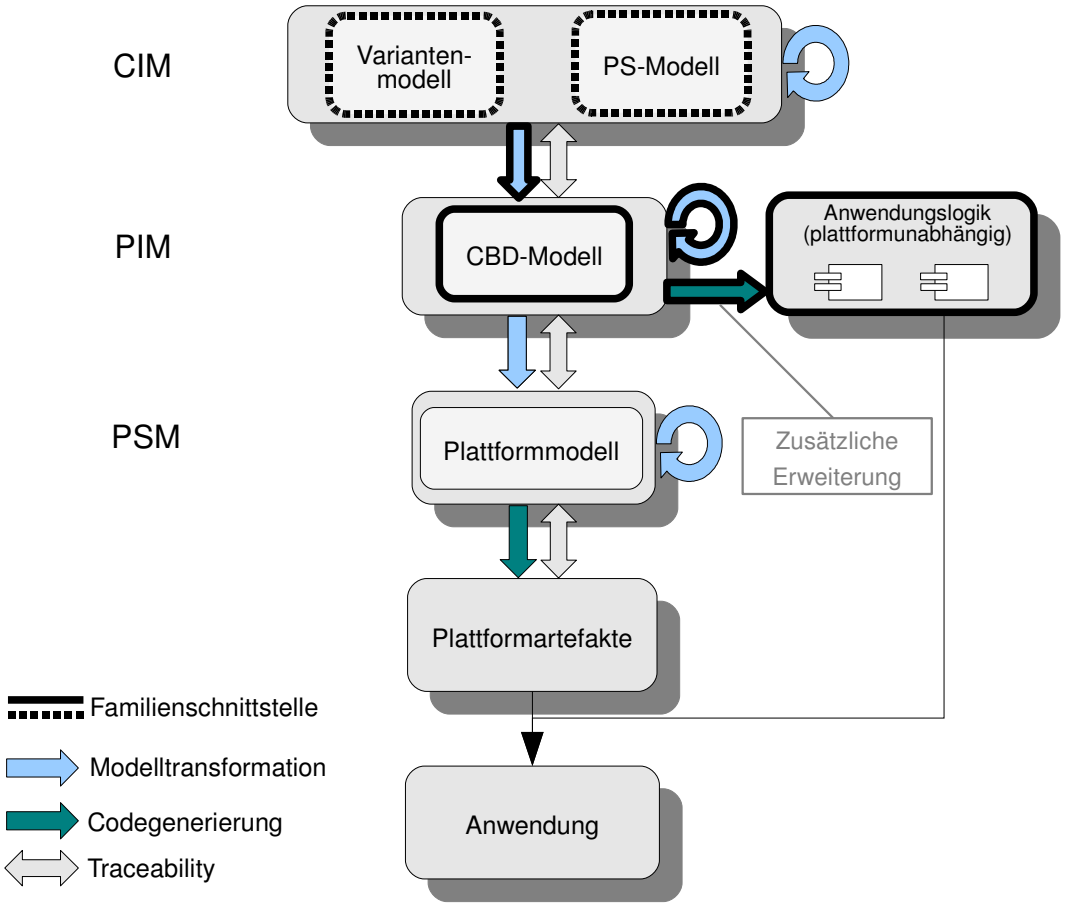


Abbildung 10.4: Anwendungserstellungsprozess der Produktlinienfamilie

mente werden folgende Einflussmöglichkeiten der Produktlinienkomponenten dargestellt:

- CIM-PIM Transformation:

  Die Transformation von den Domänenmodellen in das Anwendungsmodell kann beeinflusst werden.

- PIM-Modell

  Neue Modellelemente können dem Anwendungsmodell hinzugefügt werden.

- PIM-PIM Transformation

  Die Modellmodifikation innerhalb des Anwendungsmodells kann beeinflusst werden.

- PIM-Codegenerierung

Es kann dynamisch plattformunabhängiger Anwendungscode hinzugeneriert werden.

- Anwendungslogik

  Manuell programmierte, plattformunabhängige Anwendungslogik lässt sich ebenfalls zur enstehendenden Anwendung hinzufügen.

Somit sind alle Möglichkeiten der Einflussnahme abgedeckt: Es können Modellelemente hinzugefügtwerden, Transformatoren und Generatoren erweitert und manuell erstellte Anwendungslogik eingebracht werden. Zusätzlich können auch die Metamodelle der Domänenebene beeinflusst werden.
Die Domänenimplementierung basiert auf dem in den vorigen Kapiteln erarbeiteten Produktlinienfamilienansatz. Zunächst erfolgt die Implementation des Familienkoordinators mit Hilfe des PLiC Frameworks und oAW. Anschließend wird die Produktlinienkomponente für Betriebssicherheit entwickelt.

10.2.3 Implementation des Familienkoordinators

Da das PLiC Framework selbst nur Basisfunktionen bereitstellt, muss für jede Familie separat ein Familienkoordinator implementiert werden. Dieser definiert den Anwendungserstellungsprozess aller Familienmitglieder und er implementiert die Bootstrap-, Komponentenintegrations- und Anwendungserstellungsphase des Integrationsprozesses aus Abschnitt 5.3.
Der Familienkoordinator begründet die neue Produktlinienfamilie *SmartFamily.* Er besteht dank des PLiC Frameworks nur aus einer Sammlung von hauptsächlich abstrakten oAW-Workflows und dem familienweiten Integrationsmerkmalmodell, das die zu ihm gehörenden Produktlinienkomponenten und deren Integrationsparameter beschreibt.
Ein abstrakter Workflow geht, wie bereits in Abschnitt 7.1.1.2 erwähnt, davon aus, dass er bestimmte Properties von außen gesetzt bekommt. Dies geschieht üblicherweise über dessen Einbettung in einen übergeordneten Workflow mittels des Cartridge-Mechanismus von oAW. Die Workflows werden jeweils von einer virtuellen Produktlinie aus aufgerufen, welche die Properties entsprechend ihren Bedürfnissen setzt.
Die Implementation der Workflows ist aus rein konzeptionellen Gründen auf mehrere Projekte verteilt.[1] Nachfolgend wird davon abstrahiert und der Koordinator als eine Einheit aufgefasst, was jedoch keinerlei Schwierigkeiten bereitet, da alle Dateinamen eindeutig sind.
Folgende Artefakte implementieren die Phasen eins, drei und fünf der Produktlinienfamilienintegration:

- Bootstrapping: `wf-bootstrap.oaw`
- Komponentenintegration: `wf-integratePlics.oaw`

[1] `smartfamily.config`, `smartfamily.integrate`, `smartfamily.run`, `smartfamily.custom`

- Anwendungserstellung: `wf-prepareProductLine.oaw`

Durch den ausdrücklichen Aufruf so genannter „Custom-Workflows“ von ihren Anwendungserstellungs-Workflows aus können die Produktlinien explizit den Eingriff von Produktlinienkomponenten an prinzipiell beliebigen, jedoch im voraus zu definierenden Stellen zulassen. Wie im Folgenden noch gezeigt wird, ist ein solcher Aufruf an der Stelle nötig, kurz bevor die Basisklassen der Anwendungskomponenten generiert werden. Daher ist noch folgener Workflow nötig:

- `wf-customAPIGen.oaw`

Damit sich die Komponenten selbst um möglichst wenig aktiv kümmern müssen, wird für diese eine fixe Schnittstelle vorgegeben. Diese besteht darin, Artefakte an bestimmten Stellen bereitzuhalten (Modelle, Advice-Spezifikationen, Anwendungslogik, ...), die dann durch den Familienkoordinator integriert werden.
Nachfolgend werden nun die vier erwähnten Workflow-Dateien des Familienkoordinators näher beschrieben.

10.2.3.1 Bootstrapping

Funktion. Der Bootstrap-Vorgang dient dazu, die Komponenten und den Koordinator initial bekannt zu machen und wird durch die nicht abstrakte Workflow-Datei `wf-bootstrap.oaw` implementiert. Diese muss nur dann neu aufgerufen werden, wenn neue Komponenten hinzukommen oder sich deren Integrationsoptionen ändern.
Die Zusammenführung der Komponenten erfolgt über Erstellung eines familienweiten Merkmaldiagramms, des Integrationsmerkmalmodells, das in der pure::variants-Datei `integrationConfig.xfm` abgelegt wird. Das Merkmaldiagramm enthält alle Produktlinienkomponenten als Merkmale gemeinsam mit ihren Konfigurationsmöglichkeiten bezüglich ihrer Integration in eine Produktlinie.

Parameter. Da es sich um einen nicht abstrakten Workflow handelt, sind keine Parameter erforderlich.

Interna. Intern erfolgt das Bootstrapping durch einen einfachen Aufruf an die Workflow-Komponente `PlicBootstrap` (siehe Abschnitt 9.2.4.1).

10.2.3.2 Komponentenintegration

Funktion. Die Komponentenintegration erledigt ein Aufruf des abstrakten Workflows `wf-integratePlics.oaw`. Er erstellt eine neue, in den Domänenmodellen angereicherte virtuelle Produktlinie auf Basis der Artefakte der ursprünglichen Produktlinie bereit.

Parameter. Der Workflow erwartet folgende essentiellen Parameter:

- integrationVM

  Dabei handelt es sich um ein Variantenmodell des Integrationsmerkmalmodells `integrationConfig.xfm`. Es beschreibt, welche Produktlinienkomponenten auf welche Weise in die neue Produktlinie integriert werden sollen.

- runFM

  Pfad zum Merkmalmodell der Basisproduktlinie.

- metaModelFile

  Pfad zum PS-Metamodell der Basisproduktlinie.

- modelOutDir

  Name des Verzeichnisses, in das die Artefakte der neuen, angereicherten Produktlinie geschrieben werden sollen.

Des Weiteren kann der Workflow auch noch das bereits bestehende Variantenmodell (`runVM`) und die Modellinstanz des PS-Metamodells (`modelFile`) der Basisproduktlinie in die neue Umgebung übertragen, was aber voraussetzt, dass die Änderungen am Merkmal- und am Metamodell nur additiv und nicht invasiv vorgenommen wurden.

Interna. Zunächst erfolgt die Kopie der übergebenen Artefakte in das `modelOutDir`[2] und dessen Umwandlung in einen pure::variants-Konfigurationsbereich durch den `ConfigSpaceCreator`. Dann wird über den `MultiCartridge`-Mechanismus für jede der gewählten Produktlinienkomponenten entsprechend der Konfiguration des `integrationVM` die Anreicherung des Merkmalmodells und des PS-Metamodells vorgenommen. Dabei wird der `FeatureModelWeaver` zum Anreichern des Merkmalmodells und die normale `XtendComponent` von oAW zum Erweitern des PS-Metamodells angewandt.

10.2.3.3 Anwendungserstellung

Funktion. Der abstrakte Workflow `wf-prepareProductLine.oaw` unterstützt die Anwendungserstellungsphase. Direkt nach dessen Aufruf muss der Workflow der virtuellen Produktlinie den Workflow der Basisproduktlinie starten (wie bereits in Abbildung 8.1.3 gezeigt). `wf-prepareProductLine.oaw` bereitet dabei die Umgebung durch Weben der Advice-Spezifikationen und durch Füllen von List-Slots mit Modellen vor. Die Anwendungserstellung der Basisproduktlinie wird dadurch angereichert, ohne dass sie davon Kenntnis nehmen muss.

[2] Das PLiC Framework enthält eine Workflow-Komponente zum Kopieren von Dateien, auf deren Beschreibung im vorigen Kapitel verzichtet wurde.

Parameter. Der Workflow erwartet folgende Parameter:

- runVM

  Pfad zum mit Produktlinienmerkmalen angereicherten Variantenmodell der Produktlinie. Die Komponenten können dies auswerten, um je nach Konfiguration ihrer Merkmale andere Vorbereitungen zu treffen.

- projectRoot

  Pfad des Projekts der virtuellen Produktlinie. Dort sind die Artefakte der Anwendungslogik abzulegen, die Produktlinienkomponenten mitbringen können.

Interna. Die Vorbereitung der Umgebung hängt eng mit dem definierten Anwendungserstellungsprozess der Familie zusammen. Folgende Aktionen werden durchgeführt:

- Weben von Advice-Spezifikationen für den CIM-PIM- und die PIM-PIM-Transformator durch die Workflow-Komponente `XtendAdvice`.

- Weben von Advice-Spezifikationen für die PIM-Codegenerierung durch die Workflow-Komponente `GeneratorAdvice`.

- Integration zusätzlicher PIM-Modellelemente der Produktlinienkomponenten in den Produktlinienablauf über den `RootSlotListAdder`.

- Integration manuell programmierter Anwendungskomponenten in die Produktlinie durch deren Kopie in das `projectRoot`-Verzeichnis der virtuellen Produktlinie.

Dabei müssen nicht alle Produktlinienkomponenten jede dieser Aktionsmöglichkeiten tatsächlich nutzen. Um jedoch keine aufwändige Konfiguration notwendig zu machen, und einen einheitlichen „Komponentenstandard“ zu gewährleisten, wurde für die SmartFamily aber festgelegt, dass alle Komponenten für die oben genannten Aktionen Artefakte an klar definierten Stellen ihrer Verzeichnisstruktur vorhalten müssen. Dafür wurde eine *Template-Komponente* entworfen, die bereits die entsprechende Struktur besitzt, jedoch keinen funktionalitättragenden Advice oder weitere Modellelemente enthält. Aus Basis dieser Komponente können dann ohne großen Aufwand eigene Produktlinienkomponenten entwickelt werden.
Hier besteht durchaus noch Optimierungspotential. Aus Zeitgründen konnte jedoch kein Konfigurationsmechanismus mehr dafür entworfen werden. Eine flexiblere Lösung wäre hier beispielsweise, dass Komponenten allein eine Art Manifestdatei im Sinne einer Konfigurationsdatei an einer fixen Stelle vorhalten müssen. Dort können sie deklarativ ihre Bedürfnisse spezifizieren und den genauen Ort der einzubindenden Artefakte angeben.

10.2.3.4 Expliziter Aufruf der Komponenten durch die Produktlinie

Funktion. Der abstrakte Workflow `wf-customAPIGen.oaw` dient als Beispiel, wie ein expliziter Aufruf der Produktlinie an die Komponenten durchgeführt werden kann. Dabei dient der genannte Workflow des Familienkoordinators als Vermittler. Er leitet den Aufruf dann an den „Custom-Workflow“ jeder Komponente weiter. Auch dieser muss sich an einer eindeutigen Stelle in der Verzeichnisstruktur der Komponente befinden. In ihm kann die Komponente dann beliebige andere Workflow-Komponenten aufrufen.

Der Name `wf-customAPIGen.oaw` rührt daher, dass dieser Aufruf in der Produktlinie direkt vor die Generierung der Basisklassen aller im Endprodukt verwendeten Anwendungskomponenten zu geschehen hat. Dieser Workflow wird später benötigt, um die Fehlerinjektion implementieren zu können.

Durch den direkten Aufruf diese Workflows macht sich die Basisproduktlinie vom Familienkoordinator abhängig. Sie kann somit nicht mehr ablaufen, ohne dass sich ein so benannter Workflow in ihrem Klassenpfad befindet. Der invasive Eingriff muss daher sorgsam abgewogen werden. Da er jedoch nur minimal ist, lässt sich die Produktlinie mit geringem Aufwand wieder dazu bewegen, auch ohne Familienkoordinator ihren Dienst zu verrichten.

Parameter. Dem Workflow können beliebige Parameter übergeben werden. Er gibt diese dann transparent an den von jeder Produktlinienkomponente bereitgestellten „Custom-Workflow“ weiter. Insbesondere interessant ist es jedoch, den Workflow mit dem Attribut `inheritAll="true"` aufzurufen. Dadurch bekommen die „Custom-Workflows“ vollen Zugriff auf die Properties des Produktlinien-Workflows.

Interna. Der Workflow erledigt den Aufruf des jeweiligen „Custom-Workflows“ einer Komponente über die `MultiCartridge`. Diese ist jedoch nur für die mehrfache Ausführung einer jeweils immer *gleichen* Workflow-Datei gedacht, nur unterschiedlich *parametrisiert*. Um dennoch die unterschiedlichen Workflow-Dateien der Produktlinienkomponenten aufrufen zu können, wird der Parametrisierungsmechanismus der `MultiCartridge` ausgenutzt. Zunächst erfolgt der Aufruf des zum Familienkoordinator gehörenden Workflows `wf-customAPIGenIter.oaw` (siehe Abbildung 10.5). Dieser bindet dann je nach Parameterübergabe den Workflow einer *anderen* Produktlinienkomponente über den üblichen Cartidge-Mechanismus von oAW ein. Dies ist möglich, da die Workflow-Datei, die über den Mechanismus eingebunden wird, ebenfalls als normale Zeichenfolge übergeben wird, die durch die diskriminierende Property `${plicName}` parametrisiert werden kann.

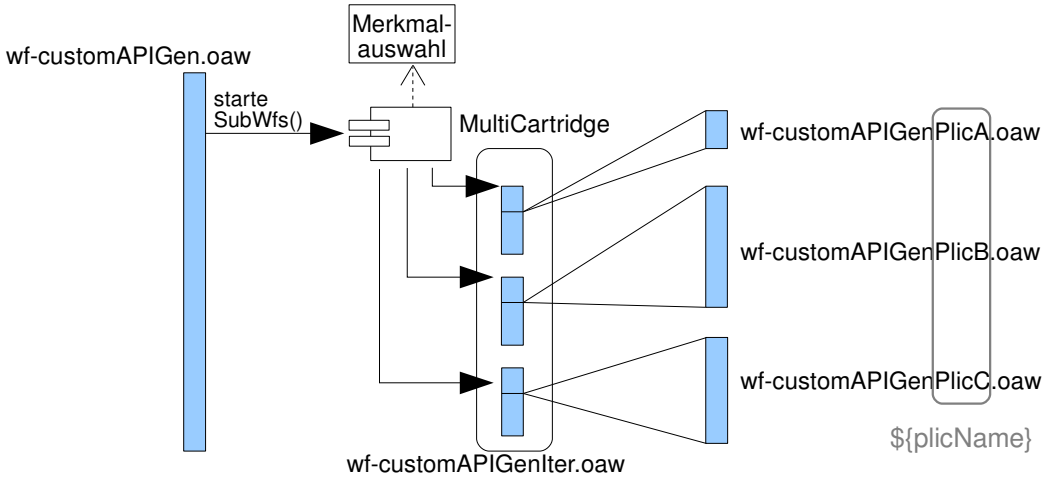


Abbildung 10.5: Aufruf von unterschiedlichen Workflows durch die MultiCartridge

10.2.4 Implementation der Fehlerinjektion

Auch die Implementation der Produktlinienkomponente erfolgte über mehrere Projekte verteilt.[3] Im Folgenden wird aus Übersichtlichkeitsgründen wiederum nur auf die Dateinamen selbst eingegangen. Dieser Abschnitt beschäftigt sich zunächst nur mit der Implementation der Fehlerinjektion, im Folgenden wird dann die Voter-Komponente betrachtet.
Wie während der Domänenanalyse spezifiziert, soll die Fehlerinjektion global im Merkmalmodell an- und abgewählt werden können und über ein grafische Benutzerschnittstelle die Manipulation der Rückgabewerte der Anwendungskomponenten ermöglichen.
Die Einbindung der Produktlinienkomponente besorgt, wie bereits erwähnt, der Familienkoordinator, während diese nur eine „Schnittstelle“ implementiert, die in der Bereitstellung von Artefakten an entsprechenden Stellen im Dateibaum besteht.
Die Reihenfolge der im Folgenden beschriebenen Artefakte entspricht der einer möglichen Abfolge ihrer Implementierung. Sie geht daher nicht streng nach dem Ablauf des Anwendungserstellungsprozesses vor.

10.2.4.1 Erstellung des Merkmalmodells

Damit die Fehlerinjektion auch in der virtuellen Produktlinie wahlweise an- und abgewählt werden kann, soll sie im Merkmalmodell der angereicherten Produktlinie erscheinen. Das in Abbildung 10.1 gezeigte Merkmalmodell der Komponente befindet sich in der pure::variants-Datei `runFMPart.xfm`. Es wird vom Familienkoordinator

[3] `smartfamily.safety.config`, `smartfamily.safety.custom`, `smartfamily.safety.lib`, `smartfamily.safety.platform`, `smartfamily.safety.trafo`

während der Integrationsphase in das bestehende Merkmalmodell der Produktlinie eingewebt. Dabei dürfen keine Namenskonflikte auftreten, was jedoch durch ein familienweites Domänenlexikon ohnehin gewährleistet sein sollte.

10.2.4.2 Erstellung des Anwendungsmodells

Das Anwendungsmodell der Produktlinienkomponente besteht, wie in Abbildung 10.3 gezeigt, nur aus dem Modellelement `ErrorInjectionAspect`. Es befindet sich in Form eines EMF-Modells in der Datei `model/lib.xmi`. Der Familienintegrator sorgt dann mit Hilfe des `RootSlotListAdder` in der Anwendungserstellungsphase dafür, dass das Modellelement in der angereicherten Produktlinie zur Verfügung steht.

10.2.4.3 Erstellung der CBD-CBD-Modelltransformation

Um das Weben entsprechend der Merkmalauswahl veranlassen, wird der CBD-CBD-Transformator mit einem Advice in der Datei `run/cbd2cbd/transform.ext` erweitert.[4] Dieser fügt dem Instantiierungsmodell sozusagen eine „Aspektinstanz“ des `ErrorInjectionAspect` hinzu, falls das Merkmal `errorinjection` konfiguriert ist. Bei der späteren Auswertung des Modells wird so darüber schließlich das Weben des programmiersprachlichen Aspekts veranlasst.

10.2.4.4 Erweiterung der Anwendungskomponentenzustände

Der SmartHome Demonstrator definiert auf der CBD-Ebene (also plattformunabhängig) ein umfassendes Komponentenmodell. Unter anderem ist es möglich, Zustandsvariablen für jede Anwendungskomponente im CBD-Modell zu definieren. Für jede der Anwendungskomponenten wird dann eine Basisklasse generiert, welche die Verwaltung dieser Variablen übernimmt.[5] In der laufenden, auf Knopflerfish[kno] basierenden OSGi-Anwendung schließlich können die Zustandsvariablen dann über eine grafische Schnittstelle eingesehen und verändert werden.
Der beschriebene Mechanismus wird nun genutzt, um jeder Anwendungskomponente für jeden ihrer angebotenen Dienste weitere Zustandsvariablen zu definieren, die ihr Verhalten im Fehlerfall festlegen sollen. Der `ErrorInjectionAspect` kann diese dann auswerten und entsprechende Manipulationen vornehmen. Folgende Variablen werden pro Dienst einer Anwendungskomponente hinzugefügt:

- Verzögerung und Verzögerungswahrscheinlichkeit

[4] Genau genommen befinden sich alle im Folgenden angegebenen Xtend- und Xpand-Around-Advice-Spezifikationen der Safety-Produktlinienkomponente nochmals in einem Verzeichnis namens `/src/smartfamily/safety/`, um Namenskonflikte zu vermeiden. Dies ist notwendig, da oAW alle Transformator- und Generator-Dateien über den Java-Classpath-Mechanismus lokalisiert und sich unterscheidende Eclipse-Projektnamen dadurch unterschlagen werden.

[5] Die Implementation der eigentlichen Anwendungslogik erfolgt jedoch weiterhin manuell.

- Rückgabewert und Wahrscheinlichkeit, dass dessen Rückgabe statt des ursprünglichen Wertes erfolgt
- Rückgabe einer Exception und Wahrscheinlichkeit, dass deren Rückgabe anstatt des ursprünglichen Wertes erfolgt
- Abweichung vom ursprünglichen Wert (normalverteilt)

Diese Zustände müssen nun, falls das Merkmal `errorinjection` ausgewählt ist, zu den Modellelementen jeder einzelnen Anwendungskomponente hinzugefügt werden, und zwar kurz bevor die Generierung der eigentlichen Basisklassen aus dem Modell erfolgt. Da an dieser Stelle in SmartHome keine Modelltransformation vorgesehen ist, wurde dort, wie bereits erwähnt, ein Aufruf an den Familienintegrator-Workflow `wf-customAPIGen.oaw` platziert. Diese leitet den Aufruf weiter an den von der Produktlinienkomponente definierten Subworkflow `wf-customAPIGensafety.oaw`.
In diesem Workflow könnten nun prinzipiell beliebige Operationen von der Produktlinienkomponente durchgeführt werden, da sie von dort aus selbstbestimmt eigene Workflow-Komponenten starten kann. Im hier vorliegenden Fall ist es ausreichend, einen Standard-Transformator von openArchitectureWare aufzurufen, der den Modellelementen der Anwendungskomponenten die besagten Zustandsvariablen hinzufügt, falls das Merkmal `errorinjection` ausgewählt ist. Der Transformator in der Datei `addErrStates.ext` implementiert.

Hinweis Um Zugriff auf das Modell zu haben, das die Anwendungskomponenten enthält, reicht es nicht aus den Workflow `wf-customAPIGen.oaw` ohne irgendwelche Parameter (Properties) aufzurufen. Zusätzlich muss noch eine Property übergeben werden, *wo*, das heißt in *welchem Slot* sich das zu manipulierende Modell befindet. Bei der Spezifikation der Schnittstelle eines „Custom-Workflows“ muss also nicht immer nur jeweils festgelegt werden, *von welcher Stelle* im Anwendungserstellungsprozess aus ein Aufruf an eine Produktlinienkomponente erfolgt, sondern auch, *welche Properties* dabei übergeben werden.

10.2.4.5 Implementation des Aspekts

Die programmiersprachliche Implementation des Aspekts enthält schließlich die Datei `ErrorInjectionAspect.aj`.[6] Sein Wirkungsbereich erstreckt sich hierbei auf alle Diensteaufrufe aller Anwendungskomponenten.
Der zugehörige Around-Advice wertet zunächst die Zustandsvariablen aus und führt die entsprechenden Manipulationen an der Diensterbringung durch, so weit es möglich ist (eine normalverteilte Abweichung vom Sollwert macht z. B. nur für numerische Werte Sinn).

[6] Sie befindet sich im Verzeichnis `src/smartfamily/safety/aspects/errorInjectionAspect/`.

10.2.5 Implementation der Voter-Anwendungskomponente

Die Voter-Funktionalität soll über ein Attribut namens `isRedundant` des PS-Domänenmodells konfigurierbar sein (siehe Abbildung 10.2). Ein auf `true` gesetztes Attribut soll dabei bedeuten, dass die zu einem PS-Metamodellelement zugehörige Anwendungskomponente dreifach redundant mit einer zusätzlichen abstimmenden Komponente, einem Voter, ausgelegt werden soll. Für einen solchen Eingriff muss sichergestellt sein, dass sich durch die Dreifachauslegung keine ungewollten Nebeneffekte ergeben. Es handelt sich also um einen invasiven Eingriff, der vorher in Analyse- und Entwurfsphase genau überlegt sein will. Es ist davon auszugehen, dass nur eine gewisse, wohldefinierte Anzahl sich als TMR-Kandidaten eignen.

10.2.5.1 Erstellung des Integrationsmerkmalmodells

Die Art und Weise, *wie* die Integration in die Produktlinie stattfinden kann, bestimmt das Integrationsmerkmalmodell der Produktlinienkomponente. Im konkreten Fall erhält es neben des obligatorischen Attributs `plicComponent` ein zusätzliches vom Typ `Collection<String> TMRCandidates` (siehe Abbilung 10.1). Das Merkmalmodell befindet sich zunächst in der Produktlinienkomponente in der pure::variants-Datei `integrationFMPart.xfm`. In der Bootstrap-Phase webt es dann der Familienkoordinator in das familienweite Integrationsmerkmalmodell `integrationFM.xfm` ein.

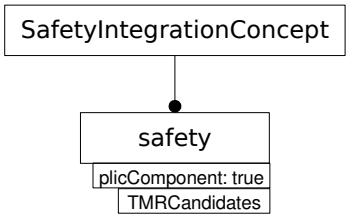


Abbildung 10.6: Integrationsmerkmaldiagramm der Produktlinienkomponente für Betriebssicherheit

Während der zweiten Phase des Integrationsablaufs, der Komponentenkonfiguration, erfolgt dann die manuelle Auswahl einer konkreten Integrationskonfiguration über ein Variantenmodell des besagten Merkmalmodells. Dabei erhält das Attribut als Werte die Namen der PS-Metamodellelemente, die sich entsprechend einer vorherigen Analyse der Produktlinie als dafür geeignet herausgestellt haben, redundant ausgelegt zu werden.

10.2.5.2 Erstellung der PSM-PSM-Modelltransformation

Die Transformation des PS-Metamodells (PSM) geschieht während der dritten Phase des Integrationsablaufs, der Komponentenintegration, und erfolgt durch die Datei

`enable/psm2psm/transform.ext`. Das Integrationsvariantenmodell wird hierbei ausgelesen, und die Metamodellelemente, deren Namen im `TMRCandidates`-Attribut genannt werden, um das boolsche Attribut `isRedundant` erweitert. Um Metamodellelemente nur aufgrund ihres Namen finden und manipulieren zu können, ist es nötig, auf dem Metametamodell basierend zu arbeiten. Es handelt sich somit um eine Art *Reflection* auf Metamodellebene.
Damit sind alle nötigen Änderungen im PS-Metamodell eingebracht. Nun kann die Produktlinie mit Hilfe eines PS-Modells und Variantenmodells manuell konfiguriert werden, um dann schließlich den Start des Anwendungserstellungsprozesses veranlassen zu können.

10.2.5.3 Erstellung der PS-CBD-Modelltransformation

Die Modelltransformation vom Domänen- zum Anwendungsmodell findet in der der Around-Advice-Datei `run/ps2cbd/transform.ext` statt. Konzeptionell handelt es sich um einen *After-Advice*. Das heißt, die Transformation findet statt, *nachdem* die Basisproduktlinie bereits die Abbildung des PS-Modells auf das CBD-Modell durchgeführt hat.
Von der Transformation sind nun zwei Aufgaben zu vollbringen. Zum einen sind die Modellelemente für die Voter-Anwendungskomponenten zu erstellen, welche die gleiche Schnittstelle wie die Basiskomponente, die redundant auszulegen ist, implementieren müssen. Zum anderen ist das Instantiierungsmodell entsprechend anzupassen, so dass die Anwendungskomponenteninstanz verdreifacht und eine entsprechende Voterinstanz hinzugefügt wird (siehe Abbildung 10.7).

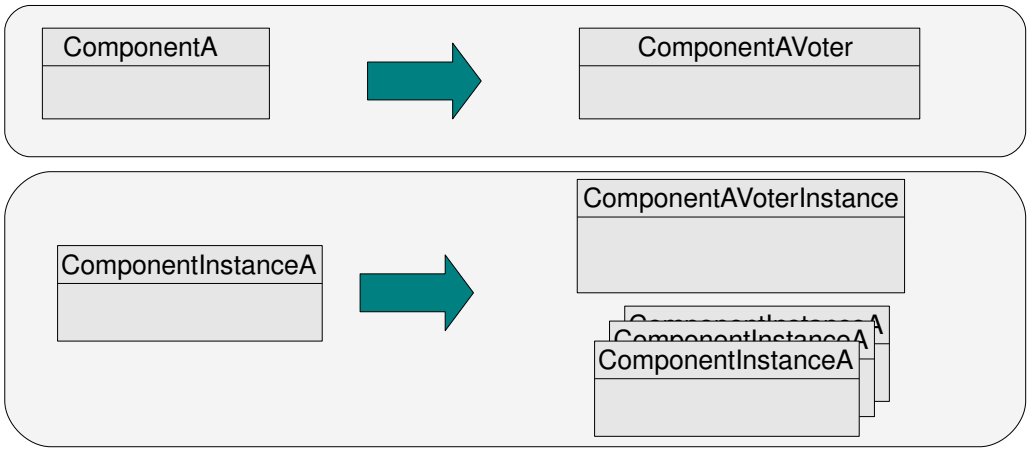


Abbildung 10.7: Zu generierende Modellelemente für eine Voter-Komponente

Schließlich sind auch die Referenzen der Instanzen aufeinander entsprechend anzupassen, so dass ein TMR-System ensteht. Dies ist schematisch in Abbildung 10.8 angedeutet.

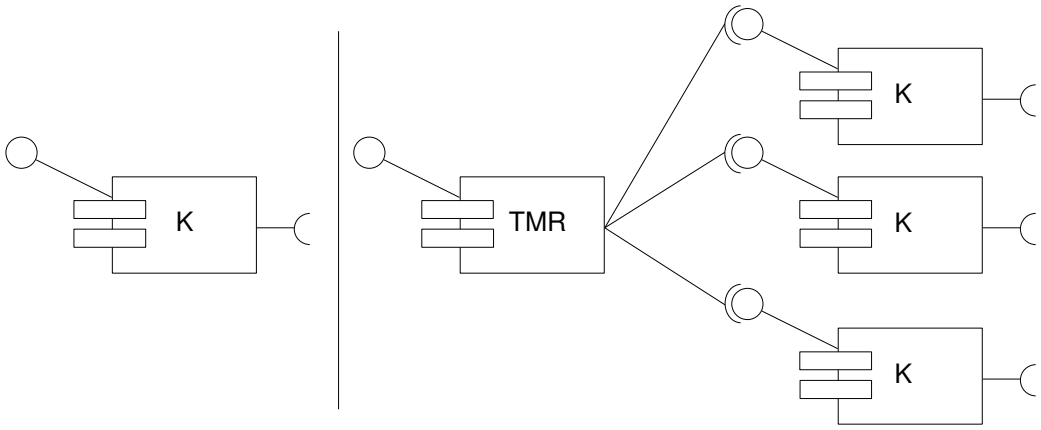


Abbildung 10.8: Redundante Auslegung einer Anwendungskomponente

Zunächst muss also die Voter-Anwendungskomponente erstellt werden. Dafür wird für jede einzelne Anwendungskomponente des CBD-Modells über den *Tracing-Mechanismus* von oAW das PS-Modellelement identifiziert, das für ihre Entstehung verantwortlich ist. Hat das PS-Modellelement nun das Attribut `isRedundant` und ist dieses auf `true` gesetzt, wird ein neues CBD-Modellelement für eine entsprechende Voter-Anwendungskomponente erstellt. Diese wird so generiert, dass sie genau die gleiche Schnittstelle wie die Basis-Anwendungskomponente bereitstellt (*Provided Interface*) und drei dieser Schnittstellen anfordert (*Required Interfaces*).

Nun müssen noch alle Anwendungskomponenten*instanzen* identifiziert werden, die redundant ausgelegt werden müssen, da es pro Anwendungskomponente beliebig viele davon geben kann. Für jede Anwendungskomponenteninstanz sind zunächst zwei weitere, gleichartige Instanzen zu erstellen. Dann ist die Voter-Anwendungskomponente zu instantiieren und die Verbindungen zwischen den Instanzen (*Konnektoren*) entsprechend zu setzten, so dass in Zukunft der Voter statt der ursprünglichen Instanz von außen aufgerufen wird und dieser die Aufrufe entsprechend an die redundanten Einheiten verteilt.

Hinweis Prinzipiell lassen sich so beliebige Anwendungskomponenten redundant auslegen. Jedoch stellen mögliche *Required Interfaces* der Basisanwendungskomponente ein Problem dar. Angenommen die drei redundanten Instanzen rufen nun über eine solche Schnittstelle eine weitere Instanz auf, kann im Allgemeinen nicht sichergestellt werden, dass diese korrekt mit den nun drei Aufrufen zurechtkommt. Dies ist aber eine Frage, die bereits während der Komponentenintegration geklärt werden sollte, da nur solche PS-Metamodellelemente das Attribut `isRedundant` erhalten sollten, die auch wirklich ohne ungewollte Seiteneffekte redundant ausgelegt werden können.

10.2.5.4 Generierung des Voter-Programmcodes

Üblicherweise erfolgt die Implementierung der eigentlichen Anwendungslogik der Anwendungskomponenten bei SmartHome, wie bereits in Abschnitt 7.2, geschildert manuell und plattformunabhängig auf CBD-Ebene. Dies ist aber für die Voter-Anwendungskomponente keine gangbare Lösung, da sie genau das Interface implementieren muss wie die Anwendungskomponente, für deren Kapselung sie sorgen soll.
Daher wird der Anwendungscode der Voter-Komponente dynamisch generiert. Der dazu nötige Xpand-Generator ist in der Datei `run/cbdgen/root.xpt` implementiert und der Familienkoordinator sorgt dafür, dass er an der entsprechenden Stelle, also nach Generierung der CBD-Modellelemente des Voters, aufgerufen wird.
Der Generator erzeugt für jeden Dienst, den der Voter bereitstellen soll, eine Methode. In dieser erfolgt dann der Aufruf der redundant ausgelegten Anwendungskomponenten über die Erzeugung neuer Java-Threads. Nach der Beendigung der Threads werden deren Ergebnisse eingesammelt, eine Voterentscheidung getroffen und diese der aufrufenden Komponente zurückgegeben.

Bemerkung Die Betriebssicherheit der hier vorgestellten Implementierung soll keiner allzu strengen Prüfung unterzogen werden. Die Verwendbarkeit von Java in sicherheitskritischen, potentiell verteilten Systemen steht erst noch am Anfang [STRS07], und die Verwendung synchroner Methodenaufrufe sowie die Erzeugung weiterer Threads können einer ernsthaften Überprüfung ebenfalls nicht standhalten. Es soll in dieser Implementierung also primär um die Präsentation der Fähigkeiten des Produktlinienfamilienansatzes gehen und nicht um die Entwicklung hochverlässlicher Sicherheitssysteme.

10.2.6 Erstellung einer virtuellen Produktlinie

Nachdem nun sowohl der Familienkoordinator als auch die Produktlinienkomponente implementiert sind, können die fünf Phasen des Integrationsablaufs nun einmal am Beispiel einer neuen virtuellen Produktlinie nachvollzogen werden.[7] Es handelt sich hierbei in gewisser Weise um *Application Engineering* im doppelten Sinne. Zunächst erfolgt die Erstellung der virtuellen Produktlinie im ersten Schritt, welche das erste Endprodukt darstellt. Sie veranlasst dann die eigentliche Anwendungserstellung, dessen Endprodukt dann die fertige Anwendung darstellt.
Im Folgenden wird der Integrationsprozess aus der Sicht einer virtuellen Produktlinie beschrieben:

- Bootstrapping

  Zunächst wird das Integrationsmerkmalmodell der Safety-Produktlinienkomponente in den Workflow `wf-bootstrap.oaw` des Familienkoordinators ein-

[7] Von einigen Details von oAW und SmartHome, die nicht dem nähren Verständnis des Ablaufs dienen, wird der Übersichtlichkeit halber abstrahiert.

getragen. Dieser Workflow ist dann einmalig parameterlos aufzurufen und sorgt dafür, das familienweite Integrationsmerkmalmodell zu erstellen.

- Komponentenkonfiguration

  Daraufhin ist ein neues oAW-Projekt anzulegen, im Folgenden virtuelle Produktlinie (VPL) genannt. In diesem sind ein neuer pure::variants-Konfigurationsbereich und ein Variantenmodell des Integrationsmerkmalmodells zu erzeugen. Das Variantenmodell kann nun entsprechend der Bedürfnisse der Komponentenintegration konfiguriert werden. Im konkreten Beispiel erfolgt dies, indem das dort existierende `safety`-Merkmal ausgewählt wird und dessen Attribut `TMRCandidates` auf die Zeichenfolge `ConcentrationSensor` [8] gesetzt wird.

- Integration

  Nun ist in dem VPL-Projekt ein neuer Workflow zu erstellen. Dieser ruft den Workflow `wf-integratePlics.oaw` des Familienkoordinators mit den oben bereits näher beschriebenen Parametern, also insbesondere dem Integrationsvariantenmodell sowie dem Merkmal- und dem PS-Metamodell der Basisproduktlinie auf. Durch den Aufruf werden die genannten Domänenmetamodelle entsprechend der gewählten Integrationskonfiguration erweitert und im VPL-Projekt gespeichert.

- Produktlinienkonfiguration

  Nun kann die manuelle Konfiguration der virtuellen Produktlinie erfolgen. Dies geschieht, indem man im VPL-Projekt ein Variantenmodell bzw. PS-Modell zu den entsprechenden Domänenmetamodellen erstellt. Im konkreten Beispiel sind ein PS-Modell mit einem `ConcentrationSensor` zu erstellen, dessen Attribut `isRedundant` auf *true* zu setzen ist. Im Variantenmodell ist `errorinjection` auszuwählen.

- Anwendungserstellung

  Die Erstellung des eigentlichen Endprodukts erfolgt durch das Anlegen einer weiteren Workflow-Datei im VPL-Projekt. Diese ruft zunächst den Workflow `wf-prepareProductLine.oaw` des Familienkoordinators mit den entsprechenden Argumenten auf, der dann die MDSD-Umgebung so vorbereitet, dass der im Anschluss aufgerufene Produktlinien-Workflow von SmartHome durch die Safety-Produktlinienkomponente angereichert wird.

Die entstehende, auf OSGi-basierende Anwendung lässt sich dann starten, die Fehler injizieren und das Verhalten des TMR-Systems beobachten. Abbildung 10.9 vermittelt hiervon einen Eindruck.

[8] Ein Konzentrationssensor wird als ein sicherheitskritisches Messgerät definiert, das den Anteil einer Chemikalie in der Luft misst.

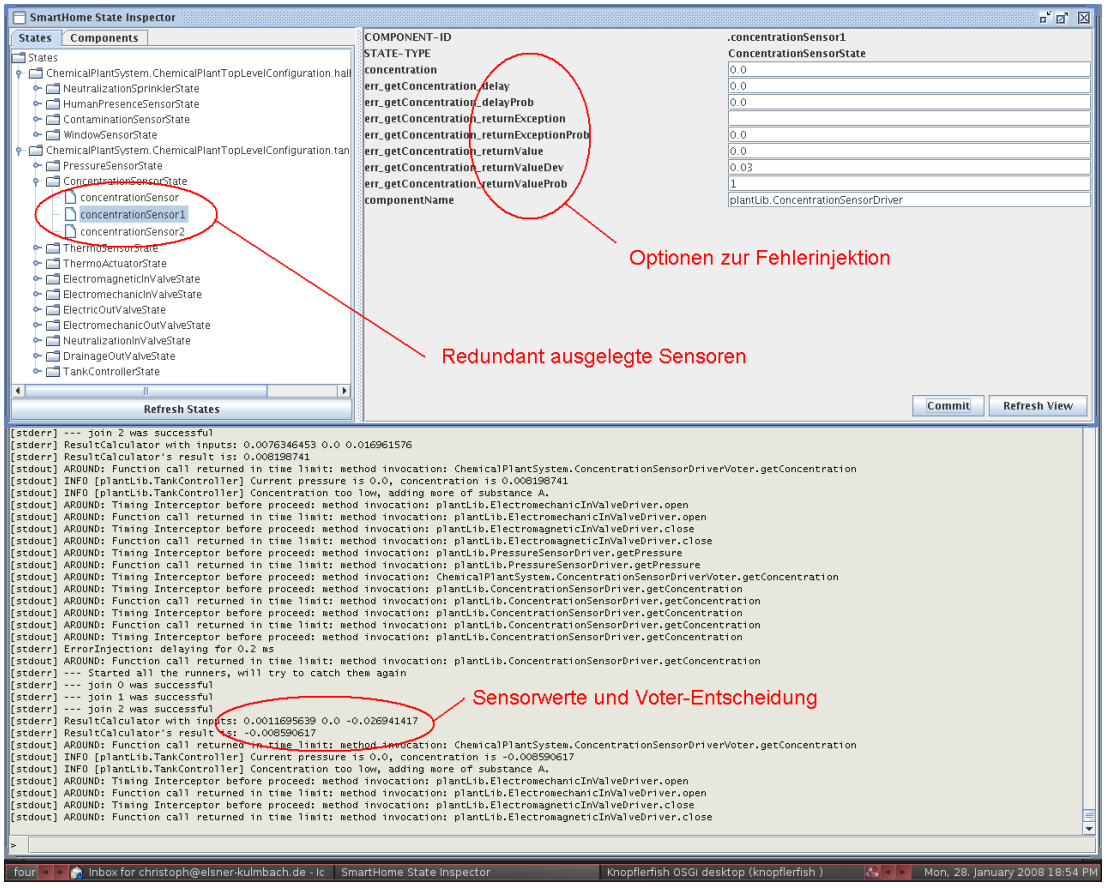


Abbildung 10.9: Die Demonstratoranwendung von SmartHome mit integrierter Produktlinienkomponente

10.3 Zusammenfassung

In diesem Kapitel wurde das PLiC Framework verwendet, um für die bestehende Produktlinie SmartHome einen Familienkoordinator zu definieren. Auf dessen Basis wurde eine Produktlinienkomponente für Betriebssicherheit implementiert. Sie zeigt beispielhaft die Möglichkeiten von Produktlinienkomponenten auf, die es ermöglichen, Produktlinien mit Modellen, Modelltransformationen, Codegeneratoren und manuell implementiertem Anwendungscode anzureichern. Schließlich wurde auch noch eine neue virtuelle Produktlinie definiert, die SmartHome und die neu entwickelte Produktlinienkomponente vereint. Sie stellt die oberste Schicht der Referenzarchitektur für Produktlinien dar und zeigt am konkreten Beispiel den Ablauf des Integrationsprozesses bis hin zur Anwendungserstellung.

11 Zusammenfassung und Ausblick

Dieses Kapitel bietet zunächst eine Zusammenfassung über den Inhalt der Arbeit. Danach wird auf themenverwandte Forschung eingegangen, um schließlich die Ergebnisse der Arbeit zu bewerten und einen Ausblick auf ihre mögliche Weiterführung zu gewähren.

11.1 Zusammenfassung

Diese Arbeit beschäftigte sich mit der produktlinienübergreifenden Wiederverwendung zwischen modellgetriebenen Produktlinien, wobei der *Produktlinienfamilienansatz* vorgestellt wurde. Er geht davon aus, dass es zwischen bestimmten Produktlinien sowohl im Lösungsraum als auch im Problemraum Wiederverwendungspotential gibt, das es zu nutzen gilt. So genannte *Produktlinienkomponenten* dienen als Kapselungseinheit für die wieder zu verwendenden Bestandteile.
Dies erforderte zunächst die Anpassung des Referenzprozesses für Produktlinien, da bereits beim Entwurf einer Produktlinie eine systematische Vorgehensweise notwendig ist, um von dem Ansatz profitieren zu können. Für die De- und Rekomposition des Problemraums wurden Konzepte erarbeitet und für den Lösungsraum die Wiederverwendungsvoraussetzungen bezüglich des modellgetriebenen Anwendungserstellungsprozesses erörtert.
Die daraufhin entwickelte *Referenzarchitektur für Produktlinienfamilien* setzt die Anforderungen aus dem angepassten Referenzprozess für Produktlinien um. Sie ermöglicht die Spezifikation einer beliebigen Anzahl von Produktlinienfamilien über so genannte *Familienkoordinatoren*. Basierend auf einer gemeinsamen Infrastruktur, dem *PLiC Framework*, sorgen sie für die Komposition von Produktlinien und Produktlinienkomponenten, die dem Anwendungsingenieur dann als *virtuelle Produktlinien* zur Verfügung gestellt werden können.
Für die Referenzarchitektur wurde eine prototypische Implementierung erstellt. Basierend auf einer bestehenden modellgetriebenen Produktlinienumgebung wurde zunächst das PLiC Framework implementiert, um dann eine Produktlinienfamilie bestehend aus der vorhandenen Produktlinie SmartHome, einer Produktlinienkomponente, einem Familienkoordinator sowie einer virtuellen Produktlinie zu entwickeln.

11.2 Themenverwandte Forschung

Modellgetriebene Software-Produktlinien stellen ein viel versprechendes Forschungsfeld dar. Insbesondere die möglichst vollständige Trennung der Belange von den An-

forderungen bis im Idealfall hin zu den Artefakten wird angestrebt. Dadurch wird versucht, eine möglichst umfangreiche Wiederverwendung von Artefakten innerhalb einer Produktlinie zu ermöglichen. Tatsächlich bestehen hier noch viele offene Fragen, wobei nicht selten aspektorientierte Methoden im Mittelpunkt der Untersuchungen stehen. Umfangreiche Forschung auf diesem Gebiet erfolgt im schon erwähnten AMPLE Projekt [VG07, GV07b], bei dem Variablitätsmanagement über den ganzen Softwareentwicklungsprozess hinweg im Vordergrund steht. Bei CALM/CADENA [CGJ+06] hingegen handelt es sich um eine modellgetriebene Umgebung, die spezielle Mechanismen zur Architekturmodellierung von Produktlinien bereitstellt. Wiederverwendung von Artefakten *zwischen* verschiedenen Produktlinien stellt jedoch für keinen der Ansätze ein Thema dar.
Die Wiederverwendung zwischen modellgetriebenen Einzelprodukten (*Model-Driven Reuse*) ist jedoch bereits im Fokus der Forschung. In [LFA+05] wird die MDA als viel versprechender Kandidat ausgemacht, den modellgetriebenen Wiederverwendungsprozess zu formalisieren. [Lar06] beschäftigt sich insbesondere damit, wie Modelle zu organisieren sind, damit ihre Bestandteile sich leicht zur Wiederverwendung eignen.
Die Verwendung von Artefakten über Produktliniengrenzen hinweg schließlich, die sich gelegentlich unter den Schlagworten *Inter-Product-Line-Reuse* oder *Multi-Product-Lines* finden lassen, stellt im Softwarebereich bisher einen nahezu weißen Fleck dar. Einzig in [Mat00] wird die Evolution und Komposition objektorientierter Frameworks untersucht und als eine mögliche Anwendung die Wiederverwendung zwischen Produktlinien genannt.
Fächerübergreifend betrachtet stellen mehrere Produktlinien im betriebswirtschaftlichen Bereich ein Forschungsthema dar. Hierbei stehen jedoch Wissenstransfer im Entwurfsbereich [NC95, NC97] oder produktlinienübergreifendes Management und Repräsentation nach außen [ZW01] im Fokus.

11.3 Bewertung

Es konnte gezeigt werden, dass heutige modellgetriebene Umgebungen mit vertretbaren Erweiterungen produktlinienübergreifende Wiederverwendung unterstützen können. Die Definition einer Produktlinienfamilie und einer Komponente für den SmartHome Demonstrator zeigt, dass der Ansatz auch auf umfangreichere modellgetriebene Produktlinien angewendet werden kann. Der prototypischen Implementierung fehlt es hingegen noch an der Definition einer zweiten Produktlinie für die entworfene Produktlinienfamilie, was aus Zeitgründen nicht mehr verwirklicht werden konnte.
Bisher ist die Spezifikation der „Schnittstellen“ zwischen Produktlinien und ihren Komponenten hingegen noch relativ komplex. Der Familienkoordinator kann diese nur durch seine tatsächliche Implementation implizit vorgeben, eine formale Deklarationsmöglichkeit fehlt bisher.
Auch stellt sich die Frage, wie sehr sich der modellgetriebene Ablauf mehrerer Produktlinien überhaupt vereinheitlichen lässt. Bereits bestehende Produktlinien verfol-

gen häufig unterschiedliche Ansätze, woraus sich schließen lässt, dass bisher noch keine einheitliche Architektur alle Bedürfnisse befriedigen konnte. Die MDA ist hierfür ein vielversprechender Kandidat, doch trotz ihrer Flexibilität und breiten Herstellerunterstützung durch die UML konnte sie sich noch nicht durchsetzen.
Schließlich berücksichtigt der Produktlinienfamilienansatz keine organisch gewachsenen und wachsenden Produktlinien. Zwar unterstützt er durchaus eine iterative Vorgehensweise, jedoch wird vorausgesetzt, dass die Schnittstelle bestehend aus Anwendungsmetamodell und Anwendungserstellungsprozess bereits von Beginn an festgelegt wird. Eine Evolution bezüglich der Metamodelle und des Erstellungablaufs scheint mit der bisherigen Werkzeugunterstützung kaum verwirklichbar.

11.4 Ausblick

Es handelt sich beim Produktlinienfamilienansatz um ein neu entwickeltes Konzept, wobei noch viele Bereiche einer näheren Untersuchung bedürfen.
Die Erweiterung des Referenzprozesses aus Abschnitt 4 kann bisher nur als Vorschlag gesehen werden, da noch umfangreiche praktische Erfahrungen fehlen. Hier wäre es angebracht, neue Produktlinien von Grund auf nach diesem Ansatz zu entwickeln und den Mehraufwand zu bewerten.
Das entwickelte Framework müsste zudem genutzt werden, um eine bestehende oder neu zu entwerfende Produktlinie konsequent zu dekomponieren, um das richtige Maß an Granularität herauszufinden, bis zu der die Zerlegung handhabbar bleibt und eine effiziente Wiederverwendung möglich ist.
Interessant wäre insbesondere der *komplette* Aufbau einer modellgetriebenen Produktlinie aus Komponenten, so dass als Basis nur noch ein leerer Rumpf bleibt, in den sich Komponenten integrieren können. Damit wäre die Produktlinienkomponente die einzige funktionalitättragende Einheit, und die bisher immer nötige Betrachtung der Beziehung und die Unterscheidung zwischen Produktlinie und Komponente würde entfallen.

12 Literaturverzeichnis

[Agr03] A. Agrawal. Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 364–368, 2003.

[All05] O.S.G. Alliance. OSGi Service Platform Service Compendium-Release 4, 2005.

[Atk01] Colin Atkinson. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.

[Beu03] Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag GmbH, Heidelberg, 2004.

[bor] Borland Homepage. `http://www.borland.com`.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[CGJ+06] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. CALM and Cadena: metamodeling for component-based product-line development. *IEEE Computer Society*, 39(2):42–50, 2006.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[CK05] K. Czarnecki and C.H.P. Kim. Cardinality-based feature modeling and constraints: A progress report. *International Workshop on Software Factories*, 2005.

[dsp] dSpace Homepage. http://www.dspace.de/.

[ea00a] D.C. Fallside et al. XML Schema Part 0: Primer. *W3C Candidate Recommendation CR-xmlschema-0-20001024, World Wide Web Consortium (W3C), Oct*, 2000.

[ea00b] R. Soley et al. Model Driven Architecture. *OMG white paper*, 2000.

[emf] Eclipse Modeling Framework Homepage. `www.eclipse.org/emf/`.

[FGDTS06] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2. 0: promises and pitfalls. *Computer*, 39(2):59–66, 2006.

[Fil01] Robert E. Filman. What is aspect-oriented programming, revisited. Technical Report 01.14, Research Institute for Advanced Computer Science, Mountain View, CA, USA, May 2001. `http://www.riacs.edu/research/technical_reports/TR_pdf/TR_01.14.pdf`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gom05] Hassan Gomaa. Architecture-centric evolution in software product lines. In *Proceedings of the ECOOP '05 Workshop on Architecture-Centric Evolution (ECOOP-ACE '05)*, Glasgow, UK, July 2005.

[GV07a] Iris Groher and Markus Voelter. Expressing Feature-Based Variability in Structural Models. *11th Software Product Line Conference (SPLC '07). Proceedings of the Workshop on Managing Variability for Software Product Lines*, 2007.

[GV07b] Iris Groher and Markus Voelter. XWeave: models and aspects in concert. *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 35–40, 2007.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and

S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[kno] Knopflerfish OSGi Homepage. `http://www.knopflerfish.org/`.

[Lar06] G. Larsen. Model-driven development: Assets and reuse. *IBM Systems Journal*, 45(3):541–553, 2006.

[LFA+05] D. Lucredio, M. Fortes, A. Alvaro, S. Almeida, and L. Meira. Towards a Model-Driven Reuse Process. In *31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Work in Progress Session*, 2005.

[LMBea01] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai et al. The Generic Modeling Environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary, May*, 17, 2001.

[mat] The MathWorks homepage. http://www.mathworks.com/.

[Mat00] Michael Mattson. *Evolution and Composition of Object-Oriented Frameworks*. Phd thesis, University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, 2000.

[NC95] K. Nobeoka and M. A. Cusumano. Multiproject strategy, design transfer, and project performance: asurvey of automobile development projects in the US and Japan. *Engineering Management, IEEE Transactions on*, 42(4):397–409, 1995.

[NC97] K. Nobeoka and M. A. Cusumano. Multi-Project Strategy and Market-Share Growth: The Benefits of Rapid Design Transfer in New Product Development. *Strategic Management Journal*, 18(3):169–186, 1997.

[NC01] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[oawa] OpenArchitectureWare User Guide Homepage. `http://www.eclipse.org/gmt/oaw/doc/4.2/html/contents/reference.html`.

[oawb] OpenArchitectureWare Homepage. `http://www.openarchitectureware.org`.

[Obj01] Object Management Group (OMG). Action Semantics Specification. ad/2001-03-01, March 2001.

[Obj02] Object Management Group (OMG). Meta Object Facility (MOF) Specification. formal/2002-04-03, April 2002.

[Obj03] Object Management Group (OMG). 2.0 OCL Specification. ptc/2003-10-14, October 2003.

[Obj07a] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. ptc/07-07-07, July 2007.

[Obj07b] Object Management Group (OMG). Unified Modeling Language (UML) 2.1.2 Infrastructure Specification. formal/2007-11-04, November 2007.

[Obj07c] Object Management Group (OMG). Unified Modeling Language (UML) 2.1.2 Superstructure Specification. formal/2007-11-02, November 2007.

[Par76] D. L. Parnas. Some hypothesis about the uses hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik, 1976.

[sma] France Telecom R&D mit SmartQVT Homepage. `http://smartqvt.elibel.tm.fr/`.

[spl] Software Product Lines homepage. `http://www.softwareproductlines.com/successes/successes.html`.

[STRS07] M. Schoeberl, B. Thomsen, A.P. Ravn, and H. Sondergaard. A Profile for Safety Critical Java. In *Proceedings of the 10th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '07)*, pages 94–101, 2007.

[SV05] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung*. dpunkt-Verlag, 2005.

[VG07] Markus Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *11th International Software Product Line Conference*, pages 233–242, 2007.

[Voe07] Markus Voelter. Writing Adaptable Software: Mechanisms for Implementing Variabilities in Code and Models. *22th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07). Tutorial.*, 2007.

[ZW01] J. Zyl and A.J. Walker. Product Line Balancing. pages 1–6, 2001.

Index