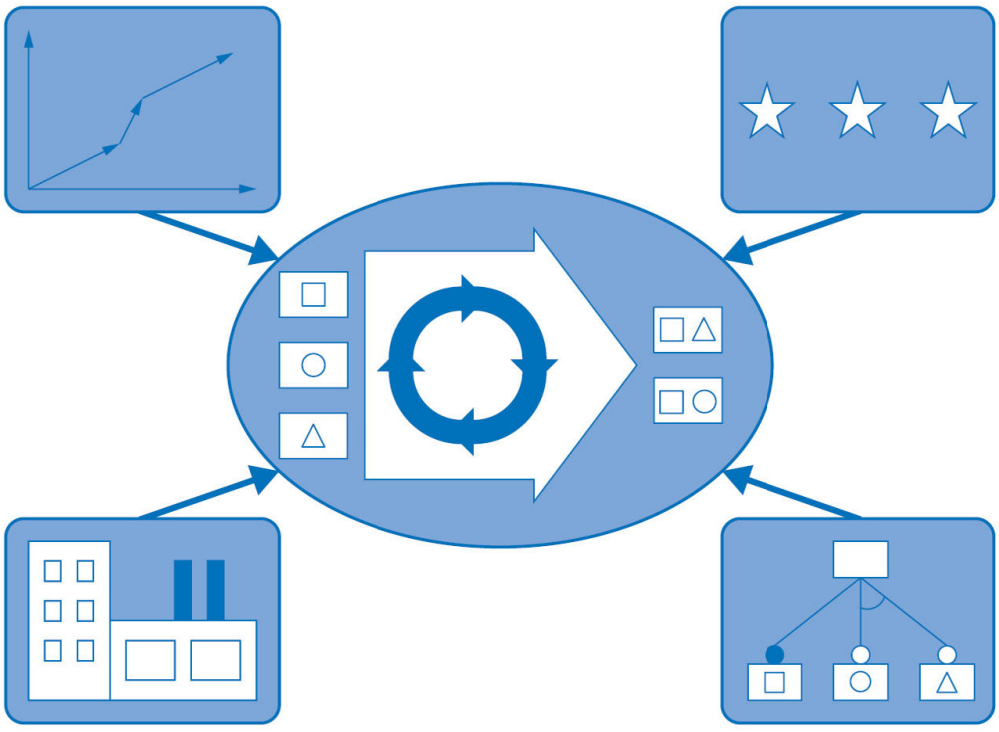
SE | RWTH AACHEN UNIVERSITY

Holger Rendel

Praktische Ansätze zur Etablierung einer Software-Produktlinie in eine bestehende Mehr-Produkt-Entwicklung



Aachener Informatik-Berichte,
Software Engineering Band 23

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Praktische Ansätze zur Etablierung einer Software-Produktlinie in eine bestehende Mehr-Produkt-Entwicklung

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Wirt.-Inf. Holger Rendel
aus Braunschweig

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Professorin Dr.-Ing. Ina Schaefer

Tag der mündlichen Prüfung: 22.03.2016

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8440-4584-0 erschienen.

[Ren16] H. Rendel:
Praktische Ansätze zur Etablierung einer Software-Produktlinie in eine bestehende Mehr-Produkt-Entwicklung.
Shaker Verlag, ISBN 978-3-8440-4584-0. Aachener Informatik-Berichte, Software Engineering, Band 23. 2016.
www.se-rwth.de/publications/

Kurzfassung

Umfang und Komplexität von Softwaresystemen steigen kontinuierlich weiter an. Gründe dafür sind zum einen die steigende Zahl funktionaler Anforderungen an die Systeme und zum anderen, dass nicht nur ein Produkt entwickelt wird, sondern eine Vielzahl verwandter Produkte. Die Komplexität steigt durch Funktionserweiterungen und die Einbindung von Variabilität. Optimierung von bestehenden Funktionalitäten und Fehlerbeseitigung erhöht den Variantenreichtum und damit die Komplexität zusätzlich. Zudem ist ein immer höherer Aufwand notwendig, um ein System über den gesamten Lebenszyklus hinweg zu begleiten. Dadurch, dass sich aus einem System viele Produktvarianten ergeben, ist für jedes Produkt ein gewisser Aufwand für die Entwicklung, den Test und die Dokumentation des Produkts notwendig.

Die negativen Effekte der steigenden Komplexität und des steigenden Aufwands sollen durch eine möglichst hohe Wiederverwendung von Entwicklungsartefakten verringert werden. Dies führt zur Nutzung von programmatischen Möglichkeiten wie Vererbung und der Entwicklung von Bibliotheken und Komponenten, die in mehreren Produkten Anwendung finden. Ein Produkt enthält in der Regel mehrere Bibliotheken bzw. Komponenten, die jeweils einen definierten Funktionsumfang bereitstellen. Bei einer Software-Produktlinie wird diese Vorgehensweise ebenfalls angewendet, verbunden mit der Möglichkeit für den Kunden, das entstehende System nach seinen Wünschen zu konfigurieren.

Wie eine Entwicklung mittels Produktlinien-Methodiken ausgestaltet sein kann, ist mittlerweile in einigen Publikationen beschrieben. In der Praxis ist es jedoch oft so, dass schon mehrere Einzelproduktentwicklungen vorliegen, die auf eine produktlinienbasierte Entwicklung umgestellt werden sollen. Die theoretischen Ansätze aus der Literatur für diesen Schritt lassen sich nicht direkt in die Praxis übertragen, sondern müssen den projektspezifischen Gegebenheiten angepasst werden. Das Ergebnis ist eine praxistaugliche Adaption der Methodiken, die für den aktuellen Kontext relevant sind. Es ist die projektspezifische Kombination aus mehreren theoretischen Vorgehen notwendig, um ein optimales Ergebnis zu erhalten.

In dieser Arbeit werden Ansätze zur Etablierung einer Software-Produktlinie im Rahmen einer Steuergeräte-Entwicklung beschrieben. Dazu werden die allgemeinen und individuellen Einflussfaktoren skizziert und bewertet. Basierend darauf wird eine Vorgehensweise definiert, die zur Einführung von Produktlinien-Methodiken in ein existierendes Projekt

beiträgt. Dabei werden zunächst im Entwicklungsprozess mehr Modelle verwendet, die den Automatisierungsgrad erhöhen. Auf Basis dieser Modelle wird Variabilität erfasst, im Entwicklungsprozess erstmals effizient modelliert und mittels Werkzeugen in weiteren Artefakten des Entwicklungsprozesses strukturiert berücksichtigt. Das beschriebene Vorgehen wurde in der Praxis angewandt und mit messbarem Erfolg umgesetzt. Somit wurde eine Optimierung des Entwicklungsprozesses erreicht, damit dieser zukünftig weiter in eine Produktlinie überführt werden kann.

Abstract

The size of software systems as well as their complexity is continuously growing. This is caused by additional functional requirements and the fact that not only one system is developed but a set of related systems. Complexity increases through improvement of functionality and the use of variability. Optimizations and error corrections increase also the number of variants. Furthermore a higher effort is required to maintain a system throughout its complete life cycle. Due to the number of product variants resulting from one system an appropriate effort is required to develop, test and document each product.

The negative effects of the complexity should be compensated with an efficient reuse of development artifacts. Using inheritance and libraries as well as components, which can be used in more than on product is a common strategy when implementing reuse. A product contains several libraries or components, which realize specific functionality. Software product lines also use this mechanism but with the enhancement for the customer to configure the resulting system for his needs.

The development with software product lines is described in several publications. In practice often multiple single products should be evolved into a product line based development. The theoretical approaches cannot be easily transferred into practice but must be adjusted to a project-specific situation. Thus, an optimal result can be gained by combining different theoretical approaches to a practical adaption of product line methodologies.

This thesis describes approaches to establish a software product line for the development of an electronic control unit. Therefore, common and individual factors are examined and evaluated to define a strategy to implement product line methodologies into an existing project. The key is to increase the usage of models to raise the level of automation. Next, variability is defined in these models and hence, efficiently described for the first time. Tools use the variability in further artifacts in the development process. This strategy was implemented in practice and empirically evaluated resulting in an improved development process, which can be further transformed into a product line.

Danksagung

Ganz besonders möchte ich mich an dieser Stelle bei Prof. Dr. Bernhard Rumpe bedanken, der meine Arbeit betreut hat. Zahlreiche hilfreiche Ratschläge und Diskussionen haben mich bei der Anfertigung der Dissertation sehr unterstützt. Insbesondere möchte ich mich bei ihm für die Möglichkeit bedanken, die Inhalte der Arbeit bei Volkswagen anzuwenden, was den Grundstein für meine weitere berufliche Karriere gelegt hat.

Des Weiteren möchte ich mich bei Prof. Dr. Ina Schaefer bedanken, die sich als weitere Gutachterin für die Dissertation zu Verfügung gestellt hat. Die Zusammenarbeit an zahlreichen Veröffentlichungen haben sehr zu dem positiven Ausgang der Promotion beigetragen. Vielen Dank geht auch an Prof. Dr. Joost-Pieter Katoen als Prüfer und Leiter der Prüfungskommission und an Prof. Dr. Stefan Decker als Prüfer.

Meine Arbeit am Lehrstuhl und diese Dissertation wurde ebenfalls durch eine Reihe von Mitarbeitern am Lehrstuhl unterstützt. Vor allem möchte ich hier Holger Krahn, Christian Berger und Tim Gülke erwähnen, mit denen ich in der Zeit sehr intensiv zusammengearbeitet habe. Auch vielen weiteren aktuellen und ehemaligen Mitarbeitern möchte ich hier danken (in alphabetischer Reihenfolge): Marita Breuer, Timo Greifenberg, Hans Grönniger, Sylvia Gunder, Arne Haber, Christoph Herrmann, Andreas Horst, Carsten Kolassa, Thomas Kurpick, Achim Lindt, Markus Look, Cem Mengi, Antonio Navarro Pérez, Claas Pinkernell, Dirk Reiß, Jan Oliver Ringert, Martin Schindler, Steven Völkel, Galina Volkova, Ingo Weisemöller, Andreas Wortmann

Ein großer Dank geht an die Entwicklung der Lenkungselektronik bei Volkswagen, wo ich ebenfalls viel Unterstützung bei der Umsetzung der Arbeit erfahren habe. Hier möchte ich insbesondere Carsten Busse, Fabian Wolf, Christian Hopp, Lars Gottwaldt und Karsten Rowold hervorheben, die seitens Volkswagen sich als wertvolle Ansprech- und Diskussionspartner bereitgestellt haben. Dank geht auch an die Doktoranden, die ich bei Volkswagen kennengelernt habe und die mich auf meinem Weg begleitet haben.

Meiner Familie und Schwiegerfamilie möchte ich danken, dass sie mich während der ganzen Zeit unterstützt und motiviert haben. Nicht zuletzt haben die Promotionsvorhaben des einen oder anderen Bruders dazu beigetragen, dass ich meine Promotion erfolgreich abschließen wollte. Die Unterstützung in der finalen Phase und am Tag der Prüfung werde ich euch nie vergessen. Vielen Dank geht auch an alle weiteren Verwandte, Freunde und Bekannte, die zum Gelingen der Promotion beigetragen haben.

Der größte Dank geht an meine Frau Franziska, die mir die notwendige Zeit für die Dissertation gegeben hat, und die mich das eine oder andere Mal wieder aufgebaut hat, wenn es mal nicht nicht so lief. Ihre Sichtweise auf die Dinge hat mir geholfen mich auf das Wesentliche zu konzentrieren und mich nicht in Details zu verlieren. Mit ihr als verlässliche Partnerin an meiner Seite konnte ich mich intensiv mit meiner Arbeit beschäftigen. Zuletzt geht mein Dank auch an meinen Sohn Moritz, dessen Lächeln und Art mich jederzeit von den Sorgen des Alltags abgelenkt haben.

Disclaimer

Die Ergebnisse, Meinungen und Schlüsse dieser Dissertation sind nicht notwendigerweise die der Volkswagen AG.

The results, opinions and conclusions expressed in this thesis are not necessarily those of Volkswagen AG.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Kontext

Neue Funktionalität heutiger elektronischer Systeme wird überwiegend durch Software realisiert. Im Automobilbereich sind neue Assistenzsysteme fast ausschließlich durch elektronische Bauteile mit entsprechender Software möglich. Eine Optimierung der Entwicklung dieser Software wird immer wichtiger, um diese effizient anzubieten.

In immer komplexer werdenden Systemen werden immer mehr Funktionen bzw. Features angeboten. Aus dem Funktionsangebot kann ein Kunde sein persönliches Paket zusammenstellen, das seinen Bedürfnissen entspricht. Dabei müssen einige Basisfunktionen wie das Antiblockiersystem (ABS) immer vorhanden sein, was durch gesetzliche Vorgaben bestimmt wird. Unter den Funktionen gibt es Abhängigkeiten, beispielsweise setzt ein Abstandstempomat (ACC) den normalen Tempomat voraus. Es gibt jedoch auch Funktionen, die sich gegenseitig ausschließen.

Auf Seiten der Hersteller bedeutet dies, dass man für die persönliche Auswahl des Kunden ein angepasstes System (Variante) ausliefert. Der Aufwand für den Hersteller, jede Variante separat zu entwickeln, zu testen und zu dokumentieren, ist sehr hoch. Aus diesem Grund wird ein möglichst großer Anteil der Systeme konfigurierbar ausgelegt und wiederverwendet. Der Anteil der Wiederverwendung in der Steuergerätesoftware steigt deshalb stetig [HKK04]. Um weiterhin die Kundenwünsche gezielt zu befriedigen, werden sogenannte Baukästen erstellt, aus denen sich die gewünschten Teile für eine Systemvariante entnehmen lassen.

Im Rahmen einer Software-Produktlinienentwicklung werden die Ziele von Kunden und Herstellern zusammengeführt. Der Kunde will ein individuell konfiguriertes System erhalten, der Hersteller möglichst wenig Aufwand bei der Umsetzung des Systems haben. Die Software-Produktlinie zeichnet sich dadurch aus, dass es eine Zuordnung von Features zu Teilen des Baukastens gibt und eine kundenindividuelle Kombination dieser Teile

möglich ist. Durch die Wiederverwendung qualitätsgesicherter Teile reduziert sich der Aufwand für Entwicklung und Wartung von Systemvarianten [KBB+02].

Der einfachste Weg eine produktliniengetriebene Entwicklung umzusetzen ist eine neustartende Entwicklung nach den Anforderungen von Produktlinien auszurichten. Dazu gehören separate Prozesse für die Entwicklung der Baukastenbestandteile und der Systemvarianten. Hingegen ist eine Migration mehrerer vorhandener Einzelsystementwicklungen zu einer Produktlinienentwicklung aufwändiger. Wie eine solche Migration erfolgen kann, lässt sich nur grob umreißen (dargestellt z.B. in [BK04]), da diese immer von den spezifischen Ausgangsbedingungen abhängig ist. Die Einbeziehung von erfahrenen Experten, die mit ihrem praktischen Wissen die Umsetzung der Produktlinien-Methodiken unterstützen, ist in diesem Fall empfehlenswert.

Die Ausgestaltung einer produktlinienbasierten Entwicklung ist von einer Reihe von Einflussfaktoren wie z.B. Ressourcen und Regularien abhängig. Deshalb kann es sein, dass im ersten Schritt noch nicht alle Maßnahmen für eine Entwicklung als Software-Produktlinie umgesetzt werden können. Die Ausweitung der generativen Softwareentwicklung [CE00] schafft jedoch die Voraussetzungen, um später vollständig einen großen Anteil der Produktlinien-Methodiken zu unterstützen (siehe Abbildung 1.1). Dabei muss beachtet werden, dass wesentliche Eigenschaften von Produktlinien, wie die Variabilität, nicht ausgeschlossen werden.

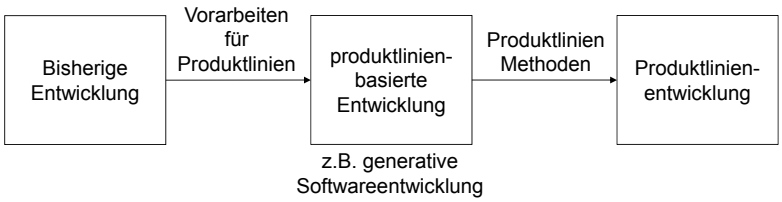


Abbildung 1.1: Produktlinienbasierte Entwicklung

Im Rahmen dieser Arbeit wurde an einem konkreten Anwendungsfall bei der Softwareentwicklung für elektromechanische Lenksysteme bei der Volkswagen AG untersucht, mit welchen Möglichkeiten sich eine Software-Produktlinie umsetzen lässt, und ein angepasstes Vorgehen entwickelt. Zu Beginn der Untersuchung gab es zwei elektromechanische Lenkungssysteme, die bereits den Serienstatus erreicht hatten und in Fahrzeuge für Endkunden eingebaut wurden. Die Entwicklung beider Lenkungssysteme erfolgte nach strengen internationalen Sicherheitsanforderungen, da ein Fehler in diesen Systemen teilweise gravierende Folgen hat. Die Lenkungen erzeugen große Lenkkräfte, um Funktionen wie das automatische Einparken zu realisieren.

Das erste System wurde zusammen mit dem VW Tiguan im Jahr 2007 eingeführt. Diese APA-Lenkung (Achs-Parallel-Antrieb) [PH11] ist für Fahrzeuge des Volkswagen Konzerns mit quer eingebautem Motor konzipiert. Mittlerweile ist diese Lenkung in weiteren Modellen wie im VW Passat und VW Touran zu finden.

Das zweite System ist für Fahrzeuge mit längs eingebautem Motor entwickelt worden, der vorwiegend in Oberklasse-Limousinen zu finden ist. Durch den eingeschränkten Bauraum kann der Motor nicht parallel zur Zahnstange realisiert werden, sondern diese wird durch den Motor geführt. Aus diesem Grund wird dieses System RCEPS (Rack Concentric Electronic Power Steering) [PH11] genannt. Zum Einsatz kommt diese Lenkung unter anderem im Audi A4 und Audi A6.

Beide Systeme besitzen je etwa 20 Lenkfunktionen, die neben der geschwindigkeitsabhängigen Lenkunterstützung zusätzliche Basis- und Komforteigenschaften umsetzen. Außerdem besteht ein großer Teil der Software aus einem aufwändigen Sicherkeitskonzept, welches ermöglicht, nicht unterstützte Rahmenbedingungen zu beherrschen und ungewollte Reaktionen des Lenksystems zu vermeiden. Die Basissoftware, die für die Ansteuerung von Hardware und Peripherie zuständig ist, wird extern entwickelt und wird nicht im Rahmen dieser Arbeit betrachtet.

Die Software des zweiten Systems ist eine angepasste Weiterentwicklung der Software des ersten Systems. Für jedes der beiden Systeme gibt es noch zusätzliche Software-Varianten durch die Anpassung an verschiedene Fahrzeuge und an verschiedene regionale Bestimmungen (z.B. Linksverkehr). Außerdem werden z.B. auf spezielle Anfragen von Kunden oder für zukünftige Entwicklungen Prototypen aufgebaut, die ebenfalls eine Softwarevariante für die Demonstration benötigen. Wie jede Software evolviert die Software für Lenkungen auch über die Zeit, so dass sich eine große Anzahl an Versionen von Software ergeben hat. Um auch zukünftig den Überblick nicht zu verlieren und gleichzeitig jede Softwarevariante in sehr guter Qualität anbieten zu können, muss die Software nach den Methoden einer Software-Produktlinienentwicklung erstellt werden.

1.2 Wissenschaftliche Fragestellung

Die Möglichkeiten zur Integration einer Produktlinien-Methodik in ein bestehendes Entwicklungsprojekt sind immer von den konkreten Gegebenheiten abhängig. Ein Vorgehen bei der Integration kann nur teilweise aus einem Standard-Vorgehen abgeleitet werden. Deshalb ist die persönliche Erfahrung der Umsetzenden wichtig, um möglichst effizient das Ziel zu erreichen. Diese Arbeit definiert ein allgemein anwendbares Vorgehen und ergänzt dieses um Erfahrungen bei der projektspezifischen Umsetzung, um so diejenigen zu unterstützen, die in anderen Kontexten eine Produktlinienentwicklung in existierende Projekte integrieren wollen. Die zentrale wissenschaftliche Fragestellung ist also:

> Wie sieht ein Vorgehen zur Integration einer Produktlinien-Methodik in eine bestehende Entwicklung für mehrere existierende und verwandte Einzelprodukte aus und wie kann eine Produktlinien-Entwicklung im praktischen Einsatz dafür ausgestaltet sein?

Bei der Beantwortung werden weitere Problemstellungen und Fragen geklärt. Die Ergebnisse können nicht nur für Produktlinien angewendet werden, sondern auch in anderen Bereichen der Softwaretechnik:

- Eine Bewertung von Einflussfaktoren ist nicht nur bei der Produktlinienentwicklung notwendig, sondern muss auch in anderen Projekten durchgeführt werden. Welche Einflussfaktoren sind für Produktlinien wichtig und wie lassen sich diese für eine optimale Umsetzung nutzen?

- Wenn eine Produktlinienmethodik in eine laufende Entwicklung eingeführt wird, ist dafür eine Reihe von Maßnahmen notwendig. Eine der ersten Maßnahmen ist die Analyse der vorliegenden Systeme, um mögliche Ansatzpunkte für Produktlinien zu identifizieren. Wie lassen sich bestehende Systeme im Hinblick auf eine Produktlinienentwicklung analysieren?

- Wenn Maßnahmen umgesetzt werden, müssen diese vom gesamten Team getragen werden. Für die Entwickler bedeutet dies eine Änderung vertrauter Prozesse. Wie werden Mitarbeiter mit dem Vorgehen vertraut gemacht und sie für die Umsetzung gewonnen?

- Die modellbasierte Entwicklung und ihre Werkzeuge sind zentrale Bestandteile der modernen Softwaretechnik. Für eine Produktlinienentwicklung muss innerhalb der Modelle Variabilität definierbar sein. Mit welchen Mitteln lassen sich Modelle für die Produktlinienentwicklung finden, mit Variabilität versehen und mittels Werkzeugen nutzen?

1.3 Wesentliche Ergebnisse

Die wesentlichen Ergebnisse der Arbeit sind:

- Ein allgemeiner Ansatz bei der Einführung einer Produktlinien-Methodik innerhalb mehrerer paralleler Entwicklungen von ähnlichen Einzelprodukten und die projektspezifische Anwendung des Vorgehens bei der Softwareentwicklung für elektromechanische Lenksysteme bei der Volkswagen AG.

- Eine Analysemöglichkeit für bestehende verwandte Systeme und Quantifizierung der Ähnlichkeit mittels Kennzahlen.

- Ein Vorgehen, um Einflussfaktoren auf eine produktliniengetriebene Entwicklung systematisch zu erfassen und zu bewerten.

- Die Ableitung einer Strategie aus den Ergebnissen der Analyse der Systeme und aus der Bewertung der Einflussfaktoren zusammen mit der Dokumentation der Strategie mittels eines virtuellen Projektraums (VPO).

- Eine Methodik, um bestehende implizit genutzte Modelle in der Entwicklung zu erkennen und festzulegen, wie Variabilität in Modellen definiert werden soll.

- Eine Beschreibung, wie auf Basis der erkannten Modelle Werkzeuge entwickelt werden können. Hierbei wird detailliert ein Adapter für Modelle aus dem weit verbreiteten Anforderungswerkzeug DOORS vorgestellt, der eine effektive Nutzung der Modelle ermöglicht.

- Eine erfolgreiche Umsetzung und Evaluation der oben beschriebenen Ergebnisse im Rahmen mehrerer Entwicklungen für eingebettete Systeme im Automotive-Umfeld.

1.4 Aufbau der Arbeit

Im Folgenden wird kurz der Aufbau der Arbeit dargelegt.

In Kapitel 2: *Einführung in Software-Produktlinien* werden die theoretischen Grundlagen und notwendigen Informationen zu Software-Produktlinien eingeführt. Basierend auf diesem Wissen wird das folgende Vorgehen grob skizziert.

Kapitel 3: *Methodik zur Einführung einer Software-Produktlinienentwicklung* stellt die Lenkungselektronikprojekte dar, für die eine Produktlinie entworfen werden soll, und zeigt welche Maßnahmen bei einer Produktlinienentwicklung umgesetzt werden können. Diese Sammlung ist von allgemeiner Natur, von der projektspezifisch nur Teile im Rahmen eines angepassten Vorgehens umgesetzt werden sollen.

Anschließend wird in Kapitel 4: *Analyse der bestehenden Software* die technische Variabilität der Lenkungsprojekte untersucht. Dazu werden mehrere Kennzahlen vorgestellt, an den Lenkungselektronikprojekten angewendet und es wird ihre Aussagekraft für die Anwendung bei der Analyse diskutiert.

Für eine tiefere Analyse werden in Kapitel 5: *Bewertung der Einflussfaktoren für eine*

Produktlinienentwicklung zu berücksichtigende Einflussfaktoren betrachtet und wie sich diese im Rahmen der Einführung von Produktlinien auswirken. Die Einflussfaktoren werden für die Lenkungselektronikprojekte bewertet.

Aus den Erkenntnissen der vorherigen Kapitel wird in Kapitel 6: *Strategie bei der Implementierung von Software-Produktlinien* das angepasste Vorgehen entworfen, das in den nachfolgenden Kapiteln detailliert und in der Lenkungselektronikentwicklung bei Volkswagen umgesetzt wird. Kern des Vorgehens ist eine Erweiterung der modellbasierten Entwicklung mit dazugehörigen Werkzeugen.

Kapitel 7: *Modellbasiertes Vorgehen* widmet sich den einzusetzenden Modellen im Rahmen der Produktlinien-Strategie. Es wird gezeigt, wie Modelle identifiziert, über Metamodelle genutzt und für den Produktlinieneinsatz angepasst werden. Für eine Produktlinienentwicklung werden mehrere Ansätze diskutiert, wie Variabilität für die Modelle spezifiziert werden kann.

In Kapitel 8: *Erstellung von Werkzeugen* wird die Werkzeugunterstützung für die Modelle behandelt. Dabei werden die wesentlichen Anforderungen der Anwender an diese Werkzeuge berücksichtigt und die Entwicklung eines Frameworks für Werkzeuge, basierend auf DOORS, beschrieben. Zusammen mit der in den Modellen vorhandenen Variabilität stellen die Werkzeuge die Basis für die Produktlinienentwicklung dar.

Das Schlusskapitel 9: *Zusammenfassung und Ausblick* fasst die Arbeit zusammen und gibt einen Ausblick auf offene Fragestellungen, die in aufbauenden Arbeiten untersucht werden können.

Kapitel 2

Einführung in Software-Produktlinien

Ihren Ursprung haben Software-Produktlinien in den in [Dij70, Par76] beschriebenen *Program Families*. Hier wurde das Szenario der Entwicklung von mehreren ähnlichen Produkten mit hohem Wiederverwendungspotential dargestellt. Konkrete Methodiken für die Verbesserung der Wiederverwendung wurden durch [Nei80] vorgestellt. Aufbauend darauf wurden Methodiken und Werkzeuge, wie die featurebasierte Domänenanalyse [KCH$^+$90], entwickelt. In [CN02, PBL05] wurde zusammenfassend der Begriff der Produktlinie in umfangreichen Beschreibungen detailliert und der Softwareentwicklungsprozess für Produktlinien beschrieben.

Für ein umfassendes Verständnis von Produktlinien ist zunächst die Definition der grundlegenden Begriffe notwendig (Abschnitt 2.1). Anschließend wird in Abschnitt 2.2 der Softwareentwicklungsprozess für Produktlinien dargestellt. Die möglichen Variabilitätsmodelle, die bei der Modellierung von Produktlinien verwendet werden können, werden in Abschnitt 2.3 betrachtet. Da ein besonderer Fokus der Arbeit auf der Einführung von Produktlinien in ein industrielles Großprojekt liegt, wird verwandte Literatur dazu in Abschnitt 2.4 diskutiert. Die mit Produktlinien bereits erreichten Erfolge in der Praxis werden in Abschnitt 2.5 behandelt.

2.1 Grundlegende Begriffe

Grundlage der Software-Produktlinienentwicklung ist die Erstellung von Softwareprodukten (Definition 1) mittels eines Softwareentwicklungsprozesses nach Definition 2.

Soll das Softwareprodukt für mehrere Kunden angeboten werden, können nicht immer die Anforderungen aller Kunden erfüllt werden. Aus diesem Grund werden vom Hersteller mehrere ähnliche Produkte angeboten, die sich nur wenig unterscheiden oder sich in festgelegten Grenzen vom Kunden konfigurieren lassen. Dieses Vorgehen nennt sich Mass Customization (Definition 3).

Definition 1 (Softwareprodukt) *Ein Softwareprodukt besteht aus Programmen, die auf Computer-Systemen ausgeführt werden, und deren Dokumentation, die für einen Kunden bestimmt sind.*

nach [IEE91]

Definition 2 (Softwareentwicklungsprozess) *Im Softwareentwicklungsprozess werden die Kundenwünsche in ein Softwareprodukt übersetzt. Er enthält die Transformation von Kundenwünschen in Anforderungen, die Übertragung der Anforderungen in ein Design, die Implementierung des Designs und den Test der Implementierung.*

nach [IEE91]

Definition 3 (Mass Customization) *Mass Customization (deutsch: kundenindividuelle Massenproduktion) ist die Produktion von Gütern und Leistungen für einen (relativ) großen Absatzmarkt, welche die unterschiedlichen Bedürfnisse jedes einzelnen Nachfragers dieser Produkte treffen, zu Kosten, die ungefähr denen einer massenhaften Fertigung vergleichbarer Standardgüter entsprechen.*

[Pil98]

Beispiel hierfür ist die Entwicklung von Fotos als Produkt. Jeder Kunde erhält abhängig von seinen digitalen Bildern dazu passende Papierabzüge. Grundbaustein von Mass Customization ist eine Plattform (Definition 4).

Definition 4 (Plattform) *Eine Plattform stellt eine Basis bereit, auf die andere Technologien und Prozesse aufsetzen.*

nach [PBL05]

Eine Plattform kann vielfältiger Natur sein. Dazu zählen beispielsweise Arbeitsmittel und Ressourcen, um ein Produkt herzustellen. Die mit der Herstellung verbundenen Prozesse sind ebenfalls Bestandteil der Plattform. Sie erleichtert die Entwicklung von Massenprodukten, die ohne Plattform gar nicht oder nur sehr viel ineffizienter durchgeführt werden kann. Für die Erstellung der Abzüge greifen mehrere Geräte, Prozesse und vor allem auch Software ineinander. Im Bereich der Softwareentwicklung lässt sich der

Begriff Plattform noch zu Definition 5 verfeinern.

Definition 5 (Software-Plattform) *Eine Software-Plattform ist ein Satz an Subsystemen und Schnittstellen, die eine gemeinsame Struktur darstellen, aus der sich abgeleitete Produkte effizient entwickeln und produzieren lassen.*

nach [PBL05]

Eine Software-Plattform stellt definierte Schnittstellen für darauf aufbauende Produkte bereit. Über diese Schnittstellen werden sowohl von als auch zu der Plattform Informationen ausgetauscht. In einer Software-Plattform ist eine Reihe von Frameworks enthalten, die von den erstellten Produkten benutzt werden. Frameworks stellen extern entwickelte Teile der Software dar, die über definierte Schnittstellen genutzt werden. Ein Beispiel aus der Automotive-Welt ist die Autosar-Plattform [Aut], auf deren Basis immer mehr Steuergerätesoftware entwickelt wird.

Lassen sich beim Mass Customization die Unterschiede zwischen den entstehenden Produkten an bestimmten Eigenschaften (Features, Definition 6) festmachen, die ein Nutzer nach seinen Wünschen konfigurieren kann, dann wird von einer Software-Produktlinie gesprochen. In Definition 7 werden Software-Produktlinien beschrieben, wie sie in dieser Arbeit verstanden werden.

Definition 6 (Feature) *Ein Feature ist eine vom Kunden wahrnehmbare Eigenschaft eines Systems.*

nach [PBL05]

Definition 7 (Software-Produktlinie) *Eine Software-Produktlinie ist eine Menge an Software-Systemen, die sich einen gemeinsamen Kern an verwalteten Features teilen, um bestimmte Marktsegmente oder Aufgaben zu bedienen, und dabei mittels eines vorgegebenen Vorgehens aus Kernbestandteilen entwickelt werden.*

nach [CN02]

Features sind für den Kunden wahrnehmbare Eigenschaften eines Systems. Diese sind beispielsweise in einem Fahrzeug-Konfigurator zu finden, in dem bestimmte Funktionen wie ein Spurhalteassistent als Feature vorhanden sind. Wie diese letztendlich im System umgesetzt werden, ist für einen Kunden nicht von Belang, sondern wird durch die Entwickler des Systems bestimmt. Diese beiden Sichten auf das System werden im Kontext

von Software-Produktlinien als Problem Space (Sicht des Kunden, Definition 8) und Solution Space (Sicht des Entwicklers, Definition 9) beschrieben. In [VG07] wird dargestellt, dass Problem und Solution Space orthogonal zum Entwicklungsprozess sind.

Definition 8 (Problem Space) *Der Problem Space ist eine domänenspezifische Abstraktion, mit welcher ein Produkt einer Software-Produktlinie spezifiziert werden kann.*

nach [Cza04]

Definition 9 (Solution Space) *Der Solution Space [. . .] besteht aus implementierungsabhängigen Abstraktionen, welche instanziiert werden können, um konkrete Implementierungen aus den Spezifikationen des Problem Space zu erstellen.*

nach [Cza04]

Im Entwicklungsprozess entstehen eine Reihe Artefakte (Definition 10).

Definition 10 (Artefakt) *Ein Artefakt ist ein physische Informationseinheit, die im Softwareentwicklungsprozess und in einem Softwareprodukt genutzt oder produziert wird. Beispiele für Artefakte sind Modelle, Quelltexte, Skripte, ausführbare Dateien, Datenbanken und Textdateien, wie eine Mail, Anforderungen oder Testreports.*

nach [OMG10c]

Variabilität in diesen Artefakten wird in Form von Variationspunkten (Definition 11) und zugehörigen Varianten (Definition 12) dargestellt. Variationspunkte stellen eine Stelle in einem Artefakt dar, an der Variation möglich ist. Varianten sind Ausprägungen dieser Variationspunkte. Zum Beispiel ist die Farbe eines Autos ein Variationspunkt mit den Varianten *grünes Auto* und *rotes Auto*. Sowohl die möglichen Auswahlmöglichkeiten (grün und rot im Beispiel) und die entstehenden Produkte (grünes und rotes Auto) werden mit dem Begriff Variante bezeichnet.

Definition 11 (Variationspunkt) *Ein Variationspunkt ist eine Repräsentation einer Variabilitätsmöglichkeit mit kontextspezifischen Informationen innerhalb von Entwicklungsartefakten.*

nach [PBL05]

Definition 12 (Variante) *Eine Variante ist eine konkrete Ausgestaltung der Variabilität eines Variationspunktes.*

eigene Definition

Vom Begriff Variante muss der Begriff Version abgegrenzt werden. Eine Version ist als ein Stand im Verlauf der Entwicklung eines Produktes zu verstehen. Dagegen ist eine Variante eine Ausprägung eines Produktes, welche selber wieder mehrere Versionen durchlaufen kann. Andererseits kann eine Version auch aus mehreren Varianten bestehen. Ein anschauliches Beispiel stellen die Betriebssysteme von Microsoft dar. Windows XP, Windows Vista und Windows 7 sind Versionen, Windows XP Home und Windows XP Professional sind Varianten des erstgenannten Produktes.

Mit diesen grundlegenden Definitionen wird ein Einstieg in den Entwicklungsprozess von Software-Produktlinien möglich. Weitere Begriffe werden in späteren Abschnitten definiert, in denen sie zum Verständnis notwendig sind.

2.2 Entwicklungsprozess von Software-Produktlinien

In der Literatur sind eine Reihe von Vorgehensmodellen für Entwicklungsprozesse unter Anwendung von Software-Produktlinien zu finden. Alle haben ihren spezifischen Schwerpunkt, zum Beispiel Architektur [Bos00] oder die generative Softwareentwicklung [CE00]. Als durchgängiger Prozess sind Software-Produktlinien ausführlich in [CN02, PBL05] beschrieben. Anhand von [PBL05] wird im Folgenden dargestellt, wie eine Produktlinienentwicklung typischerweise aufgebaut ist. Die Abbildung 2.1 dient zur Erläuterung der wesentlichen Teilprozesse der Software-Produktlinienentwicklung.

Generell spaltet sich der Entwicklungsprozess in zwei Teile auf. Der obere Teil der Übersicht zeigt das Domain Engineering, in dem wiederverwendbare Artefakte entwickelt werden. Im unteren Teil ist das Application Engineering abgebildet, in dem aus den Domänenartefakten Produkte erstellt werden. Beide bilden jeweils für sich einen eigenen Softwareentwicklungsprozess mit eigenen Endprodukten ab und sind in mehrere Unterprozesse aufgeteilt. Als Vorgehensmodell wird in beiden Teilen z.B. das V-Modell [VMod] verwendet, so dass sich eine Software-Produktlinienentwicklung als doppeltes V-Modell darstellt. Es sind jedoch auch andere Entwicklungsmethoden möglich. In [PBL05] sind explizit auch der Rational Unified Process [Kru03] und das Spiralmodell [Boe88] erwähnt. Für eine Software-Produktlinienentwicklung ist nicht vorgeschrieben, dass beide Teile der Entwicklung dem gleichen Prozess folgen müssen. Es bietet sich jedoch an, da es eine Kopplung zwischen beiden Teilen gibt. Zudem können beide Teile auch ohne

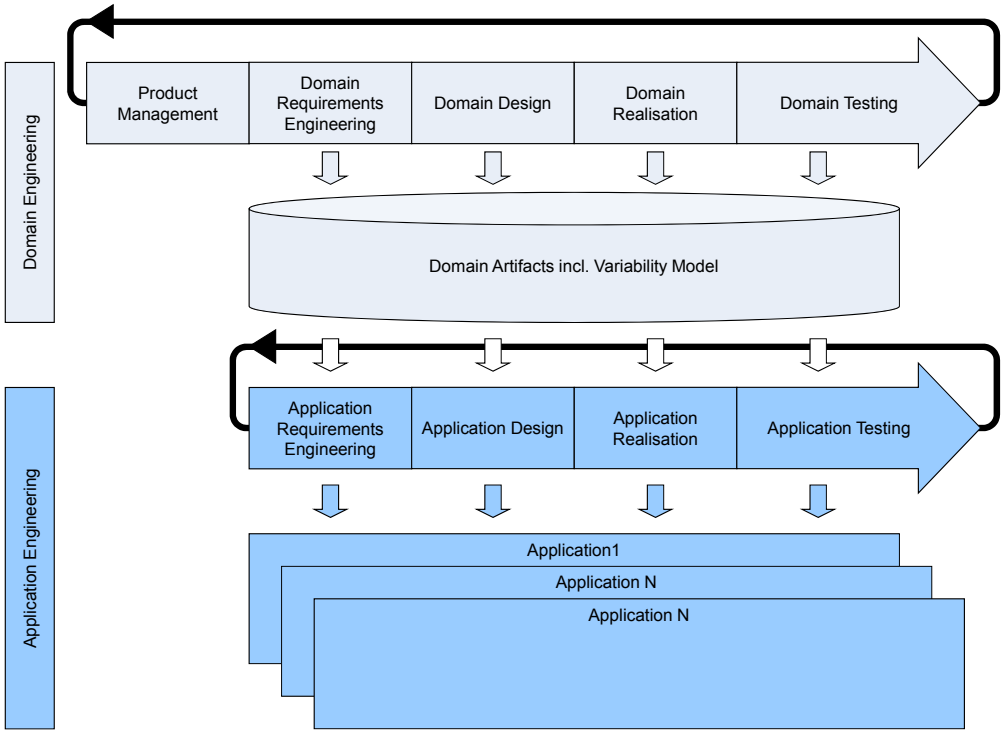


Abbildung 2.1: Software-Produktlinien nach [PBL05]

zeitliche Verzahnung durchlaufen werden.

Domain Engineering. Das Ziel des Domain Engineering ist die Entwicklung von wiederverwendbaren Artefakten. Diese sind abhängig von der Domäne, in der die Produkte der Produktlinie realisiert werden sollen. Dazu wird eine Domänenanalyse durchgeführt, um Gemeinsamkeiten und Unterschiede zu identifizieren. Prozesse und ausgetauschte Informationen zwischen den Prozessen sind in Abbildung 2.2 am Beispiel des V-Modells dargestellt. Für jede Richtung gibt es auch einen Rückkanal, der Rückmeldungen zu vorhergehenden Prozessen ermöglicht, damit dort Artefakte entsprechend neuen Informationen angepasst werden können.

Die Prozesse des Domain Engineering sind im Detail:

- **Product Management**: Hier wird die generelle Strategie und der Wirkungsbereich der Produktlinie festgelegt. Insbesondere die Auswahl, welche Produkte und Anteile von Produkten in der Produktlinie dargestellt werden sollen, wird definiert. Dieser Prozess wird als Scoping bezeichnet. Ein Überblick über vorhandene Ansätze wird in [JE09] gegeben. Basis sind z.B. Unternehmensziele oder Anforderungen von Vorgesetzten. Als Ergebnis entsteht in diesem Unterprozess eine

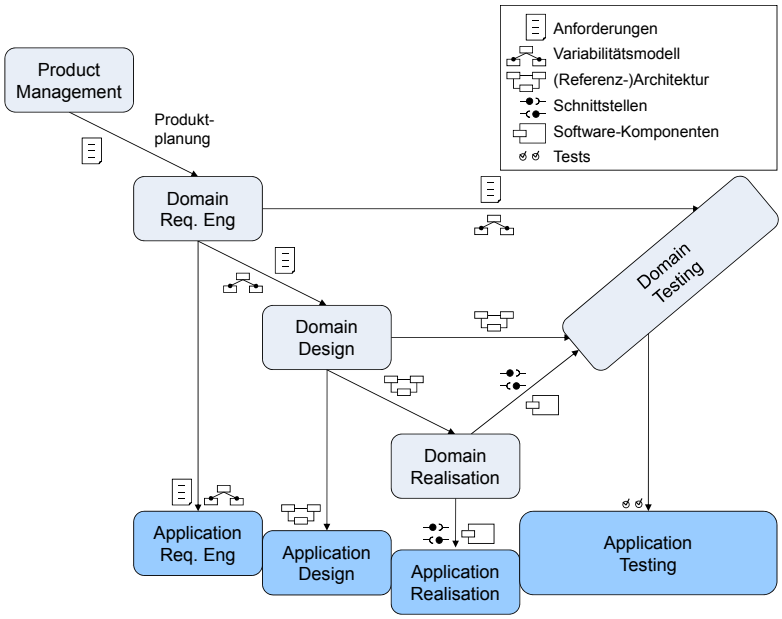


Abbildung 2.2: Prozesse und Informationen des Domain Engineering am Beispiel des V-Modells

Produktplanung, die zukünftige Produkte und deren Features beschreibt. In der Produktplanung wird zudem festgelegt, welche Features in der gemeinsamen Plattform vorhanden sind, die für alle Produkte genutzt wird.

- **Domain Requirements Engineering**: Im diesem Prozess werden die Anforderungen der Domänenartefakte festgehalten. Als Eingabe wird die Produktplanung herangezogen. Im Product Management wurde vorab schon eine grobe Domänenanalyse durchgeführt, die im Requirements Engineering verfeinert wird. Ergebnisse sind Anforderungen für die Domänenartefakte und ein Variabilitätsmodell. Da dieser Prozess für die Produktlinie sehr wichtig ist, wurde eine Reihe von Frameworks entwickelt, die mögliche Ansätze für die Domänenanalyse beschreiben. Die wichtigsten Ansätze sind FODA (Feature-Oriented Domain Analysis) [KCH+90, KCH+92] (später zu FORM (Feature-Oriented Reuse Method) [KKL+98, KLD02] weiterentwickelt), DARE (Domain Analysis and Reuse Environment) [FPDF95, FPDF98] und FAST (Family-Oriented Abstraction, Specification, and Translation) [Wei98].

- **Domain Design**: Im Domain Design wird die Referenzarchitektur als Plattform definiert. Eingabe sind die aus dem vorherigen Unterprozess erhaltenen Anforderungen mit dem dazugehörigen Variabilitätsmodell. Zusätzlich zu den Aktivitäten im Rahmen der Einzelproduktentwicklung wird in diesem Prozess definiert, welche Variabilität in welchen Architekturkomponenten umgesetzt wird.

- **Domain Realisation**: In diesem Unterprozess werden die Software-Komponenten erstellt. Die Referenzarchitektur aus dem vorherigen Prozess sowie eine Liste von zu realisierenden Anforderungen sind dafür die Basis. Aus dieser werden detaillierte Software-Komponenten entworfen und implementiert. Die Implementierung beinhaltet die Variabilität sowie Mechanismen, um diese Variabilität zu binden. Um die Software-Komponenten in späteren Prozessen nutzen zu können, wird eine Schnittstellenbeschreibung erstellt.

- **Domain Testing**: Die Validierung findet in diesem Unterprozess statt. Dazu werden die Anforderungen, das Variabilitätsmodell, die Referenzarchitektur und die Software-Komponenten mit ihren Schnittstellen zusammen überprüft. Die Tests sind so angelegt, dass sie im späteren Application Testing einfach wiederholt werden können. Die wiederverwendbaren Tests sowie die Test-Reports sind Ergebnisse dieses Prozesses.

Nach Abschluss dieser Unterprozesse sind wiederverwendbare Artefakte mit einem dazugehörigen Variabilitätsmodell entstanden, die im folgenden Application Engineering zu Produkten konfiguriert werden können. Die entstandenen Artefakte sind in der Regel nicht alleine lauffähig.

Application Engineering. Im Application Engineering werden die Artefakte des Domain Engineering zu Produkten zusammengefügt. Basierend auf Anforderungen des Kunden wird die Variabilität gebunden, die bisher in der Referenzarchitektur noch Flexibilität ermöglicht hat. Zusätzlich erfolgt die Entwicklung individueller Bestandteile der Applikation. Prozesse und ausgetauschte Informationen zwischen den Prozessen sind in Abbildung 2.3 wieder am Beispiel des V-Modells dargestellt. Auch hier existieren Rückkanäle zu vorhergehenden Prozessen des Application Engineering und auch des Domain Engineering.

Die Prozesse des Application Engineering sind im Detail folgende:

- **Application Requirements Engineering**: Hier werden die Anforderungen für das Produkt festgelegt und abgestimmt. Die Haupttätigkeit ist, Kundenanforderungen auf unterstützte Anforderungen der Produktlinie abzubilden. Dazu werden diese mit der Produktplanung sowie den Anforderungen und dem Variabilitätsmodell des Domain Requirements Engineering abgeglichen. Als Ergebnis wird daraus die Auswahl an Anforderungen erstellt, die vom konkreten Produkt erfüllt werden sollen.

- **Application Design**: In diesem Schritt wird die zukünftige Architektur des Systems entworfen. Aus der Referenzarchitektur und den Anforderungen an die Anwendung wird die Anwendungsarchitektur erstellt. Diese beinhaltet zusätzliche

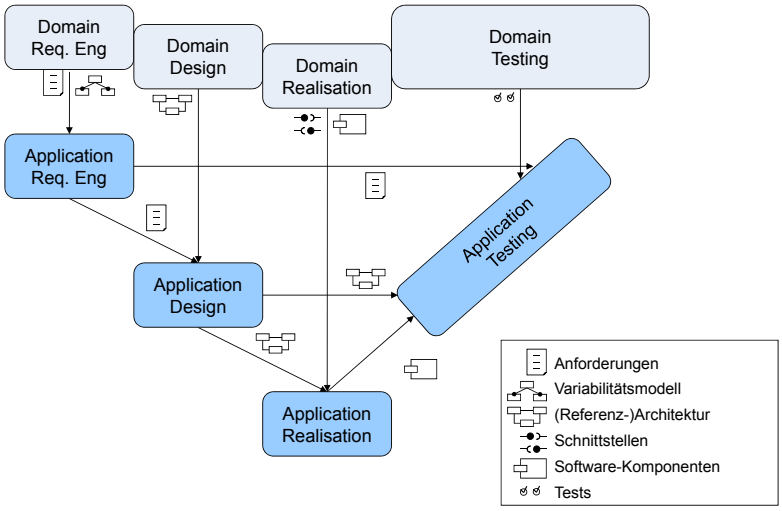


Abbildung 2.3: Prozesse und Informationen des Application Engineering am Beispiel des V-Modells

Komponenten und Architekturerweiterungen zur Referenzarchitektur.

- **Application Realisation**: In diesem Schritt wird die Anwendung erstellt. Die Anwendungsarchitektur sowie die Implementierung und Schnittstellen aus der Domain Realisation werden in diesem Prozess genutzt. Diese werden um die produktindividuellen Bestandteile ergänzt. Das Ergebnis ist eine lauffähige Anwendung.

- **Application Testing**: Aufgabe dieses Schrittes ist die Überprüfung, ob die Anforderungen, die eingangs im Application Requirements Engineering aufgestellt wurden, erfüllt sind. Alle Artefakte, die im Application Engineering erstellt wurden, werden im Application Testing überprüft. Zur Verifizierung von Anforderungen, die bereits im Domain Engineering umgesetzt wurden, werden die dazugehörigen Testfälle und Test-Reports aus dem Domain Testing verwendet. Als Ergebnis wird ein Test-Report für die Anwendung erstellt, in dem die Testergebnisse dokumentiert sind.

2.3 Variabilitätsmodelle

In Abbildung 2.4 (entnommen aus [BBM05]) sind die drei Dimensionen der Produktlinienentwicklung dargestellt. Die erste Dimension repräsentiert die Phasen der Entwicklung. Sie beginnt bei den Anforderungen des Kunden und endet bei der Implementierung im Source Code. Die zweite Dimension stellt den Abstraktionsgrad dar, der bei den Anforderungen sehr hoch ist und bei der Implementierung sehr niedrig. Beide Dimensionen

sind in der Einzelsystementwicklung ebenfalls vorhanden.

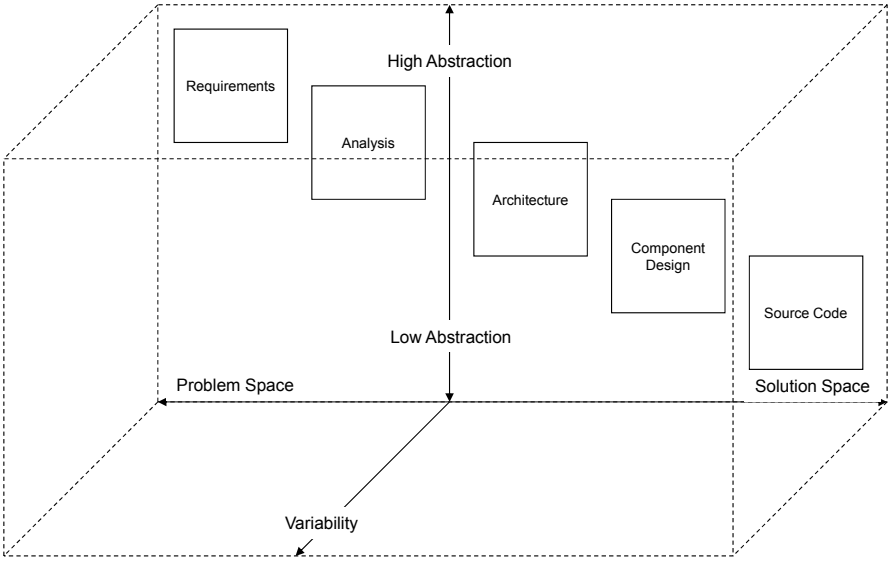


Abbildung 2.4: Die drei Dimensionen Entwicklungsphasen, Abstraktionsgrad und Variabilität in der Produktlinienentwicklung nach [BBM05]

Für die Produktlinienentwicklung kommt eine dritte Dimension dazu, nämlich die der Variabilität. Variabilitätsmodelle lassen sich anhand der ersten Dimension in zwei Gruppen aufteilen. Zum einen muss die Variabilität auf Anforderungsseite vom Kunden konfiguriert werden können (Problem Space). Auf der Implementierungsseite muss auf Basis der vom Kunden gewünschten Konfiguration die Variabilität gebunden werden (Solution Space).

Zwischen beiden Gruppen (Spaces) muss es eine Zuordnung geben, damit die Kundenwünsche in Implementierungsartefakte überführt werden. Selten ist dabei eine direkte Zuordnung möglich. In der Regel wird eine Anforderung aus dem Problem Space durch die Kombination mehrerer Elemente des Solution Space erfüllt. Andersherum kann ein Element des Solution Space mehrere Anforderungen (zumindest teilweise) erfüllen. Die Verknüpfung am Beispiel von Features in Implementierungselementen ist in [PSZ08] gezeigt. In [MBRB10] ist dies für Variabilitätsmodelle in Zusammenhang mit Funktionsnetzen gezeigt.

Innerhalb der Gruppen kann ein weiteres Mapping sinnvoll sein. In [BM07] werden Use-Case Diagramme mit Features verknüpft. Somit ist vor dem Modell, welches letztendlich mit dem Solution Space verknüpft wird, noch eine weitere Abstraktionsschicht vorhanden. Für die Verknüpfung von Problem Space und Solution Space wurden Werkzeuge entwickelt. Ein Beispiel hierfür ist das Werkzeug pure::variants [Pur], welches zahlreiche Integrationen in weitere Entwicklungswerkzeuge hat. Mögliche Darstellungen für Variabilitätsmodelle des Problem Space und Solution Space werden im Folgenden diskutiert.

2.3.1 Problem Space

Für die Darstellung von Variabilität im Problem Space werden meist Feature-Diagramme auf Basis von Features verwendet (Feature-Oriented). Eine zweite Darstellungsform, die auch oft verwendet wird, basiert auf geführten Entscheidungen (Decision-Oriented). Neben diesen beiden überwiegend genutzten Darstellungsformen gibt es weitere. Allgemein werden diese Modelle als Merkmalmodelle bezeichnet.

Feature-Oriented Feature-Diagramme wurden das erste Mal in [KCH+90] erwähnt. Im Rahmen der dort vorgestellten Methodik zur Domänenanalyse werden Feature-Diagramme zur Darstellung von Variabilität genutzt. Die Ausdrucksmächtigkeit der Feature-Diagramme wurde in [RBSP02] durch inklusive und exklusive Oder-Gruppen erweitert. Definition 13 beschreibt das Feature Model.

Definition 13 (Feature Model) *Ein Feature Model beschreibt mögliche Features eines Systems und ihre Abhängigkeiten zueinander. Features können zu einer alternativen Auswahl gruppiert werden und können andere Features benötigen oder ausschließen.*

eigene Definition

Als flexiblere Darstellung wurden in [CBUE02] die auf Kardinalitäten basierenden Feature-Diagramme eingeführt, die Variabilität ähnlich wie UML-Modelle darstellen. Eine Übersicht, wie diese Darstellungen einander entsprechen, ist in [CHE04] gegeben. Allgemein entspricht die Darstellung von Feature-Diagrammen der in Abbildung 2.5.

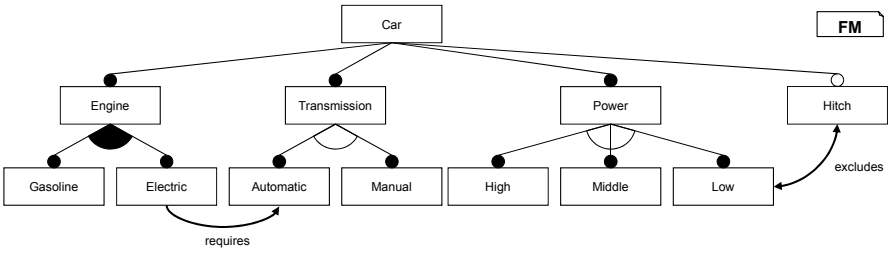


Abbildung 2.5: Feature-Diagramm, Beispiel abgewandelt aus [DK01]

Im Diagramm sind alle Features durch Kästen dargestellt. Es wird zwischen verpflichtenden und optionalen Features unterschieden. Verpflichtende Features sind am ausgefüllten Kreis und optionale Features am leeren Kreis am Kasten zu erkennen. Bei der Auswahl des Features *Car* muss zwingend auch das Feature *Engine* gewählt werden. Die Auswahl des Features *Hitch* ist optional.

Eine weitere Strukturierungsmöglichkeit sind Feature-Gruppen. Ein Feature kann sich in mehrere Subfeatures aufteilen, wie das Feature *Car* in die Subfeatures *Engine*, *Transmission*, *Power* und *Hitch*. Subfeature-Beziehungen lassen sich in exklusive Oder-Gruppen und inklusive Oder-Gruppen verfeinern. Bei ersterer muss genau ein Subfeature gewählt werden, wie bei dem Feature *Transmission*, welches die Entscheidung zwischen den Features *Automatic* oder *Manual* erfordert. Diese Entscheidungsmöglichkeit wird durch einen leeren Halbkreis dargestellt. Ein ausgefüllter Halbkreis zeigt die zweite mögliche Verfeinerung. Hier muss mindestens ein Subfeature gewählt werden. So ist bei dem Feature *Engine* sowohl die Auswahl von *Gasoline* als auch *Electric* sowie beider Features für einen Hybridantrieb möglich.

Abhängigkeiten zwischen Features können mit *requires* und *excludes*-Beziehungen dargestellt werden. Das Feature *Electric* erfordert zwingend das Feature *Automatic*. Dies wird durch eine *requires*-Beziehung ausgedrückt. Die Features *Hitch* und *Low* schließen sich gegenseitig aus, da ein kleiner Motor zu schwach für die Anhängerkupplung ist.

Das gleiche Feature-Diagramm ist mit Kardinalitäten in Abbildung 2.6 dargestellt. Zur besseren Übersicht sind die verpflichtenden Kardinalitäten (*[1..1]*) nicht abgebildet. Optionale Features werden durch die Kardinalität *[0..1]* modelliert. Exklusive Oder-Gruppen werden mit der Kardinalität *<1-1>* ausgezeichnet und inklusive Oder-Gruppen mit der Kardinalität *<1-#Subfeatures>*. Kardinalitäten erlauben die Definition von komplexeren Auswahlmöglichkeiten, z.B. *<2-3>*, um anzuzeigen, dass entweder zwei oder drei Subfeatures aus einer möglichen Menge an Features gewählt werden müssen, was ohne Kardinalitäten nicht möglich ist.

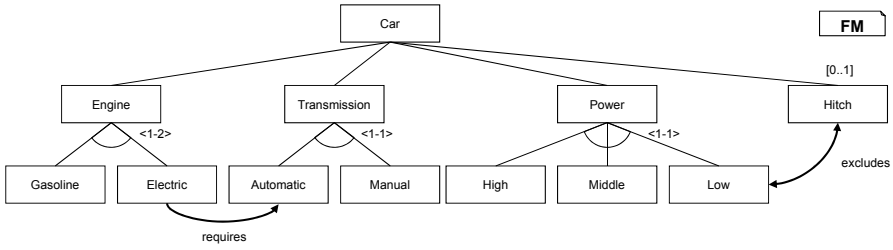


Abbildung 2.6: Feature-Diagramm mit Kardinalitäten

Es gibt mehrere Ansätze, um mit Feature-Diagrammen umzugehen, z.B. die Beschreibung mittels domänenspezifischer Sprachen [DK01, Bat05] oder die Formalisierung wie in [CHE05, CW07]. Eine Übersicht über Formalisierungsansätze ist in [THSC06, SHT06] gegeben. Die Handhabung für den Nutzer wird durch automatisierte Konfliktauflösung verbessert [BTRC05, BSTRC07, WSB+08]. Dazu werden die Aussagen des Feature-Diagramms in Ausdrücke der Aussagenlogik übersetzt und durch einen SAT-Solver mögliche Konfigurationen verifiziert. In [BSRC10] wird eine automatisierte Analyse von Feature-Modellen durch eine Reihe von Ansätzen umgesetzt. Die kompositionale Nutzung von Feature-Diagrammen ist in [ALMK08] dargestellt. Es wird formal beschrie-

ben, wie sich mehrere Diagramme zusammenfügen lassen und welche Auswirkungen dies hat.

Eine andere Logik verwendet das in [CSW08] dargestellte Konzept. Bei den dort vorgestellten *Probabilistic Feature Models* können Beziehungen durch Wahrscheinlichkeiten dargestellt werden, um Zusammenhänge zu verdeutlichen. Ein Beispiel hierfür ist, dass die Wahl des Features *Amerikanischer Markt* zu 80% zur Folge hat, dass ein automatisches Getriebe gewählt wird.

In [BBRC06] werden mögliche Optimierungen für Feature-Modelle diskturiert. Die Verbesserung der Konsistenzprüfung für Feature-Modelle ist wichtig, da solche Modelle in Zukunft immer weiter wachsen werden, um möglichst viele Features abzudecken. Nur teilweise gelöst ist die Verbindung zum Solution Space. Vor allem wie mit unterschiedlichen Abhängigkeiten im Problem und Solution Space umgegangen wird, ist Teil der zukünftigen Forschung. [CKK06] stellt dar, dass die Ausdrucksmächtigkeit von Feature-Modellen sich in der Zukunft zusätzlich vergrößern wird.

Decision-Oriented Die Idee der entscheidungsbasierten Darstellung von Variabilität wurde initial in [Bur93] im Rahmen der *Synthesis*-Methode verwendet. Frühe Wiederverwendungsstrategien nutzen als Teil des Vorgehens Entscheidungsmodelle (Decision Models, Definition 14).

Definition 14 (Entscheidungsmodell) *Ein Entscheidungsmodell ist eine Menge von Entscheidungen, mit deren Hilfe adäquat zwischen Produkten einer Produktlinie unterschieden und die Adaption auf diese Produkte geführt werden kann.*

nach [Bur93]

In [ABM00] werden Entscheidungsmodelle zur Spezifikation von Komponenten eingesetzt. Eine Komponente wird durch ihr strukturelles Modell, Verhaltensmodell und funktionales Modell charakterisiert. Das Entscheidungsmodell beschreibt zusätzlich, wie sich in verschiedenen Anwendungen eine Komponente verändert. Am Beispiel eines Aktivitätsdiagramms werden die Auswirkungen von Entscheidungen demonstriert. Ein Einblick über den Inhalt von Entscheidungsmodellen wird in [SJ04] gegeben. Diese bestehen aus Fragen mit möglichen Antworten, Beschreibungen von Interaktionen zwischen Entscheidungen sowie Relationen zwischen Entscheidungen und Variationspunkten mit deren Varianten. Letzteres ist spezifisch für das Zielartefakt, während die ersten Bestandteile unabhängig vom Zielartefakt sind.

Ein weiterer Ansatz zur Modellierung auf Basis von Entscheidungen stellt DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) dar [DRG07, DGR11].

Interaktionen zwischen Entscheidungen sind möglich, indem Wertebereiche für folgende Fragen durch getroffene Entscheidungen eingeschränkt werden. Basierend auf existierenden Ansätzen wird in [SRG11] ein allgemeines Metamodell zu Entscheidungsmodellen aufgestellt. Generell spielt bei der entscheidungsbasierten Variabilitätsmodellierung die Zuordnung zu den Artefakten des Solution Space eine größere Rolle als bei der featurebasierten Modellierung. Die Möglichkeit, den Auswahlprozess durch sinnvolle Strukturierung der Fragen zu steuern, unterstützt die Variantenauswahl gerade bei Systemen mit einem hohem Anteil an verteilter Variabilität.

Weitere Darstellungsformen In [PBL05] wird das *Ortogonal Variability Model* vorgestellt. Im Vergleich zu den vorherigen Ansätzen werden explizite Variationspunkte mit Varianten definiert. Diese Darstellung ist für den Problem Space als auch den Solution Space möglich. Das Modell dient nicht nur für den Anforderungsteil, sondern auch um verschiedene Artefakte miteinander in Verbindung zu setzen. In einer Integration mit Use-Case Diagrammen können anschließend für das resultierende System notwendige Teile abgeleitet und konfiguriert werden. UML oder spezielle UML-Profile werden ebenfalls in [PFR02, Gom04] genutzt um Variabilität auszudrücken. Dieser Ansatz lehnt sich stark an die featurebasierte Modellierung an.

2.3.2 Solution Space

Für den Solution Space existieren im Wesentlichen drei mögliche Ansätze, um Variabilität darzustellen: die negative, die positive und die transformationale Variabilitätsdarstellung. Es empfiehlt sich einen Ansatz durchgängig über alle Modelle zu verwenden, um die Komplexität nicht unnötig zu erhöhen. Bei allen Variabilitätsdarstellungen ist es wichtig, dass Inhalt und Varianten nachvollziehbar sind. Die Auswahl hängt davon ab, ob sich die Eigenschaften der Variabilitätsdarstellung mit den Anforderungen an die Entwicklung vereinbaren lassen. Generell muss das Zusammenwirken der im Problem Space getroffenen Auswahl (Features oder Entscheidungen) im Solution Space aufgelöst werden [SSS14].

Negative Variabilitätsdarstellung Die Benutzung von Modellen mit negativer Variabilität [CA05, ZHJ03] ist ein weit verbreiteter Ansatz. Bei dieser Darstellung der Variabilität werden alle Bestandteile aller Varianten im Modell festgehalten (150%-Modell) und bei der Konfiguration die Teile entfernt, die nicht für die ausgewählte Variante benötigt werden. Zur Darstellung von negativer Variabilität werden häufig Annotationen verwendet, die für ein Modellelement die Zugehörigkeit zu bestimmten Varianten zeigen [AC04, CA05]. Die negative Variabilitätsdarstellung wird im Folgenden wie in Definition 15 verstanden.

Definition 15 (Negative Variabilitätsdarstellung) *Eine Modell-Vorlage beinhaltet alle Elemente, die in validen Instanzen vorkommen können. Eine Menge an validen Instanzen ergeben die Produktfamilie.*

nach [CA05]

Auf Basis von Funktionsnetzen, die einen vergleichbaren Aufbau wie Matlab/Simulink-Modelle [Simulink] haben, wird in [GHK+07, GHK+08a] gezeigt, wie dieser Ansatz mit einem 150%-Modell einer Architektur umgesetzt wird. Funktionsnetze werden hier als Schnittstelle zwischen Anforderungen und der Realisierung einer Autosar-Methodik [Aut] für ein System (Zuordnung von Funktionalitäten auf eine technische Architektur) eingesetzt. Sie basieren auf SysML [OMG10a], welches logische Systemarchitekturen in Form von internen Blockdiagrammen (IBD) darstellt. Das 150%-Modell stellt alle möglichen Blöcke und Signale für ein System im Fahrzeug dar. Views zeigen nur Teile des Systems , in denen mittels Szenarios detaillierte Informationen bezüglich einer Variante spezifiziert werden [GHK+08b]. Diese Teile entsprechen den konsistenten Varianten des Systems. Darauf aufbauend werden in [GKPR08] Views mit einem Feature-Diagramm verknüpft, um einen Bezug zum Problem Space herzustellen.

Der Vorteil dieser Ansätze ist, dass Annotationen oder Zusatzinformationen bei vielen Modellen einfach integriert werden können. Oft erfordern jedoch neue Varianten viele Anpassungen der Annotationen, um ein konsistentes Modell zu erhalten. Ein weiterer Nachteil ist, dass ein 150%-Modell nicht mit den bestehenden Werkzeugen modellierbar ist, da diese Modelle normalerweise Kontextbedingungen verletzten. Zudem ist die Erstellung von 150%-Modellen aus existierenden Modellen ein langwieriger Prozess, der nur teilweise automatisiert unterstützt wird.

Positive Variabilitätsdarstellung Die positive Variabilitätsdarstellung (siehe Definition 16) ist eine weitere Möglichkeit, um Variabilität zu modellieren [NK08]. In [AJTK09] wird dies auch als Superimposition bezeichnet. Dieser Ansatz komponiert wieder verwendbare Teile (Komponenten) zu einem funktionierenden Gesamtsystem. Die ursprüngliche Idee, Systeme auf diese Weise zu modellieren, wurde in [Nei80] veröffentlicht. In Systemen, die komponentenbasiert aufgebaut sind, kann dieser Ansatz sehr gut implementiert werden. Wenn sich ein System nicht oder nur schlecht in komponierbare Einheiten unterteilen lässt, erhöht sich jedoch die resultierende Komplexität. In [LLE07] wird vorgestellt, wie sich ein kompositionaler Ansatz in der Entwurfs- und Entwicklungsphase kombiniert umsetzen lässt.

In [Men12] wird positive Variabilität, basierend auf Funktionsnetzen, beschrieben. Das genutzte Metamodell erlaubt die Spezifikation von (z.B. optionalen) Bestandteilen, die

Definition 16 (Positive Variabilitätsdarstellung) *Bei der positiven Variabilität werden mehrere Artefakte durch die Zusammenfügen ihrer Unterstrukturen auf Basis ihrer Gemeinsamkeiten variiert.*

eigene Definition

an definierte Variationspunkte eingehängt werden. Durch die unterschiedliche Ausprägung der Variationspunkte entstehen mehrere Varianten.

Transformationale Variabilitätsdarstellung Als dritte Möglichkeit zur Modellierung von Variabilität gibt es transformationale Ansätze, die ein gegebenes Grundprodukt durch definierte Änderungen in andere Varianten transformieren [HMPO+08, Sch10]. In Definition 17 ist festgehalten, wie die transformationale Variabilitätsdarstellung im Folgenden verstanden wird.

Definition 17 (Transformationale Variabilitätsdarstellung) *Ein Kern-Modell repräsentiert ein valides Produkt einer Produktlinie. Δ-Modelle spezifizieren Änderungen am Kernmodell durch das Hinzufügen, Ändern und Entfernen von Modell-Elementen, um neue Produkte zu beschreiben.*

nach [Sch10]

Eine Umsetzung dieses Ansatzes ist das Delta-Modeling [CHS10, SBB+10], das für Software-Architekturen in [HKR+11a, HKR+11b] umgesetzt ist. Dieser Ansatz lässt sich sowohl in Matlab/Simulink nutzen [HKM+13] als auch für jede beliebige Sprache anwenden [HHK+13] und kann auch kompositional verwendet werden [DSW14]. Anhand von Transformationen ist gut nachzuvollziehen, welche Teile sich von einer Variante zu einer anderen ändern. Die Konfiguration einer Variante entspricht einer Reihe von durchzuführenden Transformationen basierend auf einer initial festgelegten Variante. Für einen sinnvollen Einsatz eines transformationalen Ansatzes muss jedoch eine Werkzeugunterstützung existieren, die gegebenenfalls erst entwickelt werden muss. Einen umfassenden Überblick über Ansätze zu featurebasierten Modelltransformationen gibt [CH06].

2.4 Gestaltung einer Entwicklung als Software-Produktlinie

Für die Einführung von Produktlinien-Methoden wird beispielsweise wie in [CJMN06] vorgestellt vorgegangen. Der erste Schritt ist eine Analyse des Umfelds. Basierend auf der Analyse werden im zweiten Schritt die Möglichkeiten und Ansatzpunkte für eine Produktlinie definiert. Im dritten Schritt werden die identifizierten Ansatzpunkte genutzt und umgesetzt. Diese Schritte werden sowohl am Produkt (durch Variabilitätsmodellierung) und am Prozess (Werkzeuge, die Variabilität nutzen) als auch an der Organisationsstruktur (z.B. separate Teams für Domain und Application Engineering) vollzogen.

Wie man von einer Mehr-Produkt-Entwicklung zu einer Produktlinienentwicklung migrieren kann, ist in [Bos00] beschrieben. Die Migration wird in zwei Dimensionen aufgeteilt, woraus sich vier Vorgehensweisen ergeben, welche ihre spezifischen Vor- und Nachteile haben. Es wird in der ersten Dimension zwischen neuen und existierenden Produkten und in der zweiten Dimension zwischen dem evolutionären und dem revolutionären Ansatz unterschieden. In Tabelle 2.1 sind die möglichen Vorgehensweisen kurz dargestellt.

Tabelle 2.1: Vorgehensweisen bei der Einführung von Produktlinien nach [Bos00]

	Evolutionär	Revolutionär
Neue Produkte	Die Produktlinie wächst mit neuen Anforderungen für einzelne Produkte	Die Produktlinie wird anhand möglicher zukünftiger Anforderungen aufgebaut
Existierende Produkte	Evolution der Entwicklung von bestehenden Produkten und Komponenten zu einer Produktlinienentwicklung	Gestaltung einer Produktlinienentwicklung für einen Teil der existierenden Produkte und möglicher zukünftige Produkte

Ein *evolutionärer* Ansatz bei der Einführung einer Produktlinienentwicklung für *neue Produkte* sieht vor, dass die Produktlinie mit den enthaltenen Produkten im Laufe der Zeit mit neuen Produkten wächst. Somit kann von Anfang an eine strukturierte, vom Produktliniengedanken getriebene, Entwicklung aufgebaut werden. Die Nutzung und Adaption von bestehender Infrastruktur kann freigestellt werden, da es keine alten Produkte gibt, die mit der Produktlinie unterstützt werden müssen. Der Ansatz kann besonders gut genutzt werden, wenn ein Unternehmen seine Produkte auf neue Domänen ausweiten will. Die anfangs nötige Investition in die Produktlinie ist im Vergleich zu einem revolutionären Ansatz klein, es besteht jedoch die Gefahr, dass Architektur und weitere Artefakte sich zu nah an den initialen Produkten orientieren und somit unflexibel für folgende Änderungen durch neue Produkte werden.

Führt man eine Produktlinienentwicklung für *neue Produkte* auf Basis eines *revolutionären* Einsatzes ein, wird die Produktlinie auf Basis möglicher Anforderungen für zukünftige Produkte aufgebaut. Es entsteht nicht unmittelbar ein Produkt, sondern Artefakte, die Teile der später entstehenden ersten Produkte bilden. Zukünftige Trends für Produkte, die mit der Produktlinie realisiert werden sollen, müssen in den initialen Anforderungen enthalten sein. Die initiale Investition ist bei diesem Einsatz höher als beim evolutionären Ansatz, da erst später Produkte und somit Einnahmen entstehen. Vorteil ist aber auch hier die strukturierte Einführung einer Produktlinienentwicklung.

Ein *evolutionärer* Ansatz auf Basis *existierender Produkte* sieht vor, dass man bestehende Entwicklungen in eine Produktlinienentwicklung überführt. Somit müssen für alle bestehenden Produkte Gemeinsamkeiten und Unterschiede identifiziert werden. Danach kann Produkt für Produkt dann in die Produktlinie überführt werden. Das Risiko bei diesem Ansatz ist im Vergleich zum nachfolgend beschriebenen revolutionären Ansatz gering, da Investition in kleinen Schritten entsprechend des aktuellen Überführungsgrads getätigt werden und diese Investitionen sich schnell rentieren. Ein weiterer Vorteil ist, dass die Weiterentwicklung der Produkte nicht unterbrochen werden muss. Der Nachteil ist, dass Altlasten übernommen werden müssen, die einer strukturierten Produktlinienentwicklung teilweise entgegenstehen.

Der *revolutionäre* Ansatz für *existierende Produkte* ermöglicht eine Produktlinienentwicklung, die nicht nur Anforderungen der bisherigen Produkte erfüllt, sondern auch Anforderungen an zukünftige erfüllen kann. Im Unterschied zum evolutionären Ansatz können nötige Auswirkungen auf die Architektur erfordern, dass Produkte nicht nach und nach umgestellt werden können, sondern nur ein kompletter Wechsel in einem großen Schritt möglich ist. Dieser kann lange dauern und muss durch die Nutzung eines alternativen Ansatzes unterstützt werden. In der Zeit der Umstellung werden alle Weiterentwicklungsaktivitäten angehalten und sämtliche Anstrengungen auf die Entwicklung der Produktlinienarchitektur gerichtet. Die initiale Investition ist bei diesem Ansatz ebenfalls sehr hoch und risikobehaftet, da sich die Investition erst nach der vollständigen Umsetzung des Ansatzes amortisieren kann. Der Umfang der Arbeiten kann sogar soweit gehen, dass eine komplette Neuentwicklung der Produkte notwendig wird. Mit einer erfolgreichen Umsetzung des Ansatzes können nicht nur zukünftige, sondern auch bestehende Produkte abgebildet werden.

Die beiden letztgenannten Ansätze werden in [SPK06] als reaktiv bzw. proaktiv beschrieben. Das reaktive Vorgehen benutzt wiederverwendbare Artefakte an den Stellen, wo sich die Möglichkeit bietet. Dies ist besonders hilfreich bei Domänen, bei denen sich zukünftige Features schlecht voraussagen lassen. Das proaktive Vorgehen benötigt eine initiale Investition in eine Produktlinie, bei der wiederverwendbare Artefakte entwickelt werden, die erst später in ein Produkt einfließen. Einen Mittelweg stellt der extraktive Weg [Kru02] dar. Dieser nutzt leichtgewichtige Techniken und eine feste Menge an Produkten, um daraus eine Produktlinienentwicklung zu bilden. Die Gemeinsamkeiten, die

sich beim reaktiven Ansatz erst im Laufe der Entwicklung ergeben, werden von Anfang an genutzt.

[PBL05] stellt vier Möglichkeiten für die Einführung einer Produktlinienentwicklung dar, von denen sich drei in den bereits beschriebenen Ansätzen wiederfinden. Als vierter Ansatz wird zusätzlich der taktische Ansatz vorgestellt. Bei diesem Ansatz werden Probleme, die im Laufe der Entwicklung auftauchen, versucht mit Produktlinien-Methodiken zu lösen, was Potential für die Einführung einer Produktlinienentwicklung bereitstellt. Die Zuordnungen der beschriebenen Möglichkeiten zueinander sind in Tabelle 2.2 abgebildet, wobei von links nach rechts der Aufwand für die Umsetzung steigt. Gleichzeitig steigen auch die Möglichkeiten, die Produktlinienentwicklung strukturiert einzusetzen.

Tabelle 2.2: Vergleich der Ansätze für die Einführung von Produktlinien

Quelle	Ausprägung			
[Bos00]	evolutionär			revolutionär
[SPK06, Kru02]	reaktiv		extraktiv	proaktiv
[PBL05]	inkrementell	taktisch	Pilotprojekt	Big Bang

Für die Einführung einer Produktlinienentwicklung ohne bereits existierende Produkte gibt es mehrere Beispiele. Oft wird jedoch nur auf organisatorische Aspekte eingegangen oder es werden generelle Techniken für die Einführung vorgeschlagen. Wie im Detail eine bestehende Entwicklung hin zu einer Entwicklung mit Produktlinien-Methoden überführt werden kann, ist oft wenig dokumentiert.

Das in [BFK+99] vorgestellte PuLSE-Framework (**P**rod**u**ct **L**ine **S**oftware **E**ngineering) stellt detailliert dar, wie Software-Produktlinien eingeführt werden können. Die Methodologie ist in drei Dimensionen aufgeteilt: die Entwicklungsphasen zur Erstellung der Produktlinie, die technischen Komponenten, die in den Entwicklungsphasen genutzt werden sowie die Unterstützungskomponenten, wie z.B. die Organisationsstruktur. Im KobrA-Ansatz [ABM00] wird gezeigt, wie PuLSE für objektorientierte Softwareprojekte eingesetzt wird.

[Sch04] skizziert exemplarisch die notwendigen Schritte. Als erstes erfolgt eine Analyse der bisherigen Situation sowie der Möglichkeiten und Risiken der zukünftigen Produktlinie. Daraus folgt als zweiter Schritt eine Liste von Maßnahmen, die in einem dritten Schritt umgesetzt wird. Zuletzt muss die Produktlinie gelebt werden und beispielsweise Mitarbeiter für eine Produktlinie geschult werden.

Eine sehr konkrete Möglichkeit zur Umsetzung auf Quellcode-Basis wird in [YGM06] für eine Produktlinie am Beispiel einer Motorsteuerung gezeigt. Die Vorteile einer Produktlinie werden über den ROI (Return on Investment) quantifiziert. Anhand des automatisierten Auffindens von ähnlichem Quellcodebestandteilen in mehreren Produkten werden potentielle Stellen zur Wiederverwendung identifiziert. Der ähnliche Quellcode

wird in vier Typen eingeteilt, die einen unterschiedlich hohen Wiederverwendungsgrad haben. Die Produktlinie wird anschließend nach diesen gefundenen Gemeinsamkeiten und Unterschieden gestaltet.

Einen tieferen Einblick in die Umsetzung gibt [DM07] am Beispiel einer Produktlinie für Softwaresysteme zur Überwachung von elektronischen Fahrkarten. Die Umsetzung wird anhand eines UML-gestützten Produktlinien-Ansatzes beschrieben, der in [Gom04] dargestellt ist. Eine bestehende Entwicklung wird iterativ zu einer Produktlinienentwicklung transformiert. Die oben beschriebenen Vor- und Nachteile des Vorgehens sind hier ebenfalls sichtbar.

Als mögliche Bewertung für den Produktlinieneinsatz wird in [CKMM08] das „Product Line Planning Game“ vorgeschlagen. Dieses agile Vorgehen, welches für die herkömmliche Entwicklung von Produkten genutzt wird, wurde an die Rollen und Aktivitäten des Produktlinien-Ansatzes angepasst. Das dargestellte Vorgehen war in dem konkreten Anwendungsfall erfolgreich und trägt dazu bei agil mögliche Produktentwicklungen für Produktlinien zu bestimmen.

Jede Produktlinienentwicklung unterscheidet sich abhängig von dem Umfeld, in dem sie eingesetzt wird. So gibt es unternehmensspezifische Lösungen, die jeweils für sich die Schwerpunkte setzen. Das Framework in [Thi02] wurde beispielsweise für den Einsatz bei Bosch entworfen.

2.5 Beispiele für den erfolgreichen Einsatz von Software-Produktlinien

In diesem Abschnitt werden Beispiele aus der Praxis beschrieben, in denen erfolgreich eine nach Software-Produktlinienentwicklung umgesetzt wurde. Produktlinien-Methoden können über ein weites Spektrum an Systemen und Domänen angewendet werden.

Ein erfolgreiches Beispiel, welches häufig in der Literatur erwähnt wird, ist [BCO96, BCK03]. Es wurden bereits 1985 Produktlinien-Techniken eingesetzt, um Softwaresysteme unter anderem für die schwedische und dänische Marine zu entwickeln. Basierend auf der gezielten Nutzung von Gemeinsamkeiten und Unterschieden in den Aufträgen (bzw. Anforderungen) der beiden Kunden konnten viele Teile wiederverwendet werden. Die Produktlinienentwicklung erfolgte für sieben unterschiedliche Schiffstypen. Die Entwicklungszyklen für weitere Produkte sind von acht auf zwei Jahre verkürzt worden.

In [Bos99] werden Herausforderungen und Umsetzungen für Produktlinienarchitekturen bei den schwedischen Firmen Axis Communications AB und Securitas Larm AB vorgestellt. Als wesentliche Herausforderung haben sich die unterschiedlichen Sichten auf die

Architekturen, einmal von Seiten der Industrie und einmal von Seiten der akademischen Welt, dargestellt. Es existieren die drei Kernprobleme:

- Artefakte liegen in verschiedenen Versionen vor
- Abhängigkeiten zwischen den Artefakten
- der Kontext zwischen den Artefakten ändert sich

Für die Probleme wurden die Gründe analysiert und Forschungsfragestellungen abgeleitet. Insgesamt kommen die Autoren zu dem Schluss, dass Produktlinien-Architekturen erfolgreich in kleinen und mittelgroßen Unternehmen eingesetzt werden können und langfristig den Erfolg der Unternehmen ausmachen.

In der Automotive-Welt beschreibt [RGDM03] ein angestrebtes Baukastenprinzip, welches die Ideen der Produktlinie beinhaltet. Ausgehend von einem OEM (Audi) wird beschrieben, wie Verantwortlichkeiten verteilt sind und welche Anforderungen an Zulieferer gestellt werden. Das Vorgehen wird über die in der Automotive-Branche üblichen Musterphasen beschrieben. Um möglichst wenig Redundanz im Entwicklungsprozess zu haben, müssen wesentliche Schnittstellen und die Softwarearchitektur früh festgelegt werden.

In [BHJ+03] werden mehrere reale Produktlinien-Szenarien miteinander verglichen. Die unterschiedlichen Rahmenbedingungen wie Produktanzahl und Produktstabilität sind ermittelt und im Rahmen der Szenarien genauer untersucht worden. Das Ergebnis war, dass im Wesentlichen keine explizite Organisationsstruktur für Produktlinien eingeführt wurde, sondern oft kleine Teams aus der bestehenden Entwicklung gebildet wurden, die die Thematik behandelten. Insgesamt ist jedoch klar, dass die Produktlinien-Praktiken miteinander zusammenhängen und so nach und nach in der Industrie eingesetzt werden, wenn der erste Schritt konsequent gemacht wird. Dabei ist es wichtig, dass die individuellen Probleme mit dem Erfahrungsschatz über die Produktlinienentwicklung gelöst werden.

Ein solcher Erfahrungsschatz ist in [BKPS04] zu finden, wo mehrere Erfahrungsberichte zu Produktlinien vorhanden sind. Beispielsweise wird eine mögliche Umsetzung einer Architektur mit Variabilität in [HFT04] dargestellt. Ein Erfolgsfaktor im eingebetteten Bereich ist, mit der immer größer werdenden Anzahl an Sensoren umgehen zu können. Kern des Vorgehens ist eine Referenzarchitektur mit gezielt gesetzten Variationspunkten. In einer kleinen Produktlinie lassen sich mit diesem Vorgehen Erfolge erzielen, jedoch steigt mit Umfang einer Produktlinienentwicklung auch deren Komplexität. Die Autoren folgern, dass dann eine umfassende Formalisierung notwendig ist. Dies ist bei einer Einführung oft nicht gegeben, sodass eine Unterstützung für nur teilweise formalisierte Modelle existieren muss.

In [VK05] wird für das Unternehmen Market Maker Software AG eine Produktlinienentwicklung eingeführt. Das Unternehmen bietet verschiedene Webservices für den Börsenhandel an. Wesentliche Erfahrungen aus dieser Einführung waren, dass der Wartungsaufwand um 60% und die Time-to-market auf 50% reduziert wurden. Ein großes Team ist bei der Entwicklung nicht entscheidend, da kleine Teams Probleme oft besser lösen können. Die Umsetzung wurde durch eine stabile Domäne und durch eine komponentenbasierte Entwicklung erleichtert. Als nicht zu unterschätzende Gefahr sollte das Lead-Projekt gesehen werden, da bei einer ungenauen vorherigen Analyse dieses wesentliche Parameter einer Produktlinienentwicklung schon festlegen kann, die später nicht oder nur noch schwer zu ändern sind.

Wie Software-Architekten Produktlinien-Architekturen entwickeln und warten ist in [GE10] dargestellt. Hierfür wird die Entwicklung bei Scania und Volvo gegenübergestellt. Obwohl sehr unterschiedliche Gegebenheiten in den einzelnen Entwicklungen vorlagen, gab es einige ähnliche Vorgehen. Beispielsweise ließ sich der Ablauf bei Änderungen an der Architektur auf die gleichen fünf Schritte zurückführen.

Für die Mobilfunkwelt wurde in [MLF+10] mit Features versucht, die Funktionen einer Produktlinie zu modellieren. Das Ziel wurde erreicht, jedoch mit dem Ergebnis, dass das entstandene Modell sehr umfangreich ist. Die aufgetretenen Probleme der Domänenebene, der Theorie, des Projektmanagements und der Werkzeuge werden zusammengefasst und Lösungsvorschläge vorgestellt. Die Umsetzung in den Solution Space wird von den Autoren nicht besprochen.

In [DSB04, DSB05] werden für die Robert Bosch GmbH und Thales Nederland B.V. die Probleme beim Ableiten möglicher Produkte aus Produktlinien behandelt. Die wesentlichen Probleme ergeben sich dabei bei der Komplexität und durch die hohe Anzahl an impliziten Eigenschaften. Diese impliziten Eigenschaften zeigen sich durch Abhängigkeiten von unterschiedlichen Produktteilen, die die Eigenschaften des Produktes festlegen. Mit einer höheren Komplexität wird es immer schwieriger diese zu erkennen und mit ihnen umzugehen.

In [DSF07] wird die Werkzeugunterstützung für Produktlinien untersucht. Anhand eines Eigenschaftskatalogs, der Möglichkeiten (z.B. Unterstützung für Feature Modeling) eines Werkzeugs abfragt, werden vier Werkzeuge bewertet. Die Werkzeuge haben verschiedene Stärken und Schwächen, bieten jedoch für bestimmte Domänen den idealen Funktionsbedarf. Werkzeuge müssen nicht immer insgesamt ein komplettes Vorgehen nach Produktlinien unterstützen, sondern können auch nur Teile davon verständlicher machen. In diese Kategorie fällt das Werkzeug CIDE (Colored IDE) [KAK08], welches mittels Farben die gezielte Bearbeitung von Varianten anschaulicher macht und somit vereinfacht.

2.6 Zusammenfassung

Es existiert eine breite Basis an Veröffentlichungen für Produktlinienentwicklungen. Die Abgrenzung zu anderen Themen ist nicht immer scharf definierbar. Wiederverwendung und Variabilität als zentrale Bestandteile von Produktlinien sind seit Jahren in vielen Bereichen der Softwaretechnik verankert. Selbst wenn beide Aspekte gemeinsam auftreten, handelt es sich nicht zwangsläufig um eine Produktlinie.

Der Entwicklungsprozess mit Produktlinien ist gut dokumentiert. Variabilitätsdarstellungen, die alle ihre Vor- und Nachteile haben, sind ebenfalls bekannt. Werkzeuge, die spezifische Aufgaben in einer Produktlinienentwicklung lösen, sind vorhanden. Es gibt zudem eine Reihe erfolgreicher Einführungen von Produktlinienentwicklungen.

Es ist eine große Herausforderung, eine bestehende Entwicklung einzelner Produkte durch eine Produktlinienentwicklung zu ersetzen. Viele Forschungsarbeiten gehen von neu entstehenden Entwicklungen aus, wo entsprechende Prozesse und Werkzeuge leicht eingeführt werden können. Bei einer bereits bestehenden Entwicklung kann in den wenigsten Fällen direkt auf neue Prozesse und Werkzeuge umgestellt werden. Eine kontinuierliche Migration erlaubt bestehende Produkte weiter zu unterstützen und aktuelle Entwicklungen ohne viele Anpassungen weiterzuführen. Es ist eine besondere Betrachtung des Kontextes notwendig, um optimale Ergebnisse zu erzielen.

Bei den Umsetzungen zeigt sich, dass jedes Mal ähnliche Tätigkeiten anfallen, wenn eine Produktlinienentwicklung begonnen wird. Kein Ansatz lässt sich jedoch komplett ohne Anpassungen auf ein weiteres Szenario übertragen. Eine erfolgreiche Umsetzung benötigt ein individuelles Vorgehen, basierend auf dem bisherigem theoretischen Wissen und den praktischen Möglichkeiten.

Kapitel 3

Methodik zur Einführung einer Software-Produktlinienentwicklung

Der Einsatz der beschriebenen Prozesse und Methoden in eine bereits existierende Mehr-Produkt-Entwicklung hängt immer vom konkreten Kontext, hier der Softwareentwicklung für elektromechanische Lenksysteme bei der Volkswagen AG, ab. Um die Effizienz der Entwicklung zu erhöhen, war der Wunsch der Projektbeteiligten, die Möglichkeiten einer Software-Produktlinienentwicklung zu nutzen. Für diese existierende Entwicklung wird in diesem Kapitel eine Methodik zur Gestaltung eines individuellen Vorgehens, basierend auf dem revolutionären Ansatz mit existierenden Produkten nach [Bos00], vorgestellt, die in dieser Arbeit passend zum im Automobilunternehmen üblichen Projektkontext entworfen wurde.

Das Vorgehen ist das in Abbildung 3.1 dargestellt. In einer Analysephase werden zunächst das System in seinen existierenden Varianten und die Einflussfaktoren analysiert. Aus den Ergebnissen der Analysephase wird in der Planungsphase das konkrete auf den Kontext angepasste Vorgehen zur Einführung einer Produktlinien-Methodik entwickelt. In der anschließenden Realisierungsphase werden die in der Planungsphase festgelegten Maßnahmen umgesetzt.

Das allgemeine Vorgehen ist ähnlich zu dem in [Sch04] beschriebenen Transitionsprozess zur Einführung einer Produktlinienentwicklung. Darüber hinaus werden in diesen Transitionsprozessen weitere Themen, wie Geschäftprozesse, behandelt, die in dieser Arbeit nicht berücksichtigt werden sollen, da diese einen möglichst schlanken Einstieg in das Thema Software-Produktlinien beschreibt. Zudem ist es selten so, dass sofort ein komplettes Team für die Einführung einer Produktlinie zur Verfügung gestellt wird. Vielmehr ist es in der Regel so, dass einzelne Personen oder ein kleines Team damit beauftragt werden zu untersuchen, wie eine Software-Produktlinienentwicklung realisiert werden kann. Ein weiterer Vorteil des Vorgehens ist, dass es nicht nur im Hinblick auf Produktlinien angewendet werden kann, sondern als Ergebnis auch eine bestehende Entwicklung optimiert.

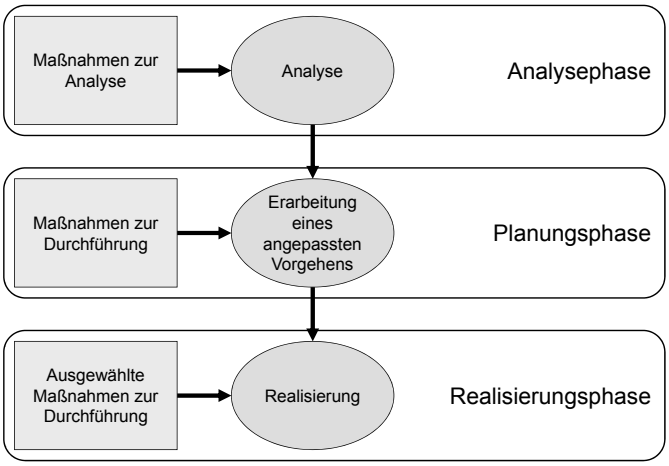


Abbildung 3.1: Allgemeines Vorgehen zur Erreichung einer Produktlinienentwicklung

Im folgenden Abschnitt 3.1 wird auf den Kontext für die konkrete Umsetzung eingegangen. Danach werden in Abschnitt 3.2 die möglichen Maßnahmen, die zur Umsetzung einer Software-Produktlinienentwicklung führen, beschrieben. Die Abhängigkeit der Maßnahmen untereinander wird in Abschnitt 3.3 untersucht.

3.1 Praktische Erprobung im Projektkontext

Zur praktischen Erprobung der entwickelten Vorgehensweise, Technik und Werkzeuge dient die Softwareentwicklung für elektromechanische Lenksysteme bei der Volkswagen AG. Elektromechanische Lenksysteme haben sich in den letzten Jahren immer mehr durchgesetzt und kommen in einer Reihe von Fahrzeugklassen zum Einsatz. Durch den geringeren Energieverbrauch können durchschnittlich 0,3 Liter Kraftstoff auf 100 Kilometern im Vergleich zu hydraulischen Systemen eingespart werden [Pan11]. Dies ist möglich, da ein elektromechanisches Lenksystem nur dann Energie aufnimmt, wenn wirklich gelenkt wird, wohingegen bei einem hydraulischen System die zugehörige Pumpe die ganze Zeit über Druck aufbauen und aufrecht erhalten muss. Ein weiterer Vorteil ist der Entfall der Hydraulikflüssigkeit bei elektromechanischen Lenkungen, was sich positiv auf die Umwelt auswirkt. Assistenzsysteme wie das automatische Einparken werden erst durch elektromechanische Lenksysteme möglich.

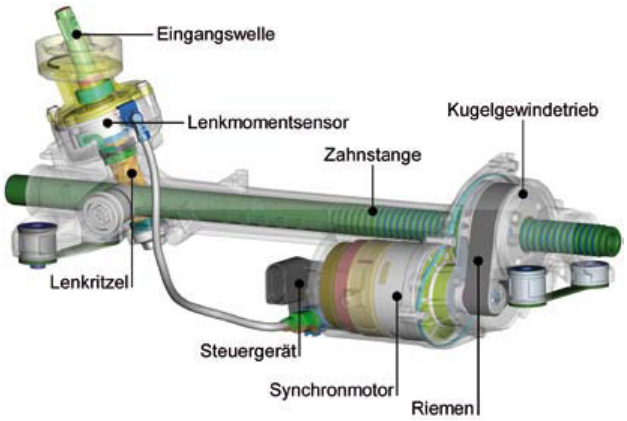


Abbildung 3.2: Lenksystem APA aus [JSB+08]

3.1.1 Produkte

Elektromechanische Lenksysteme werden in eine Reihe von Fahrzeugen im Volkswagen Konzern eingesetzt. Das erste System, welches von Volkswagen selber entwickelt wurde, war die APA-Lenkung [JSB+08] (APA = Achs-Parallel Antrieb, Abbildung 3.2). Sie wurde 2007 mit dem VW Tiguan eingeführt und sukzessiv in weiteren Modellen mit quer eingebautem Motor des VW Konzerns verbaut. Aus dem Lenkmoment wird im Steuergerät eine Unterstützungskraft errechnet, die über einen Elektromotor auf die Zahnstange wirkt. Durch den Kugelgewindetrieb ist die mechanische Funktion der Lenkung auch bei einem Ausfall des Motors gewährleistet.

Im zweiten System, der RCEPS-Lenkung [SSDJ12] (RCEPS = Rack-Concentric Electronic Power Steering, Abbildung 3.3) kommt ein anderes Antriebskonzept zum Einsatz. Dieses System ist für Fahrzeuge mit längs eingebautem Motor konzipiert, wie ihn der Audi A6 hat, in dem dieses Lenksystem zum ersten Mal verbaut wurde. Mittels der durch den Elektromotor geführten Zahnstange wird Raum eingespart, der in Fahrzeugen mit längs eingebautem Motor anders bemessen ist. In allen Fahrzeugen der Mittelklasse und höher (inklusive SUVs) wird dieses System eingesetzt, bzw. soll es eingesetzt werden.

Die zugehörige Steuergerätesoftware wird in beiden Fällen mittels Matlab/Simulink-Modellen [Simulink] und der Sprache C [C95] implementiert. Diese Steuergerätesoftware ist in eine Basissoftware und eine Applikationssoftware aufgeteilt. Die Basissoftware wird von einem externen Zulieferer bereitgestellt, der auch die Entwicklung der Hardware und des Motors für die elektromechanische Lenkung übernimmt. Vom Umfang her enthält die Basissoftware die Bestandteile, die auch von Autosar [Aut] für eine Basissoftware vorgesehen sind, also das (Echtzeit-)Betriebssystem, Treiber für die Fahrzeug-Bus-Kommunikation, das Speichermanagement sowie die Ansteuerung des Motors.

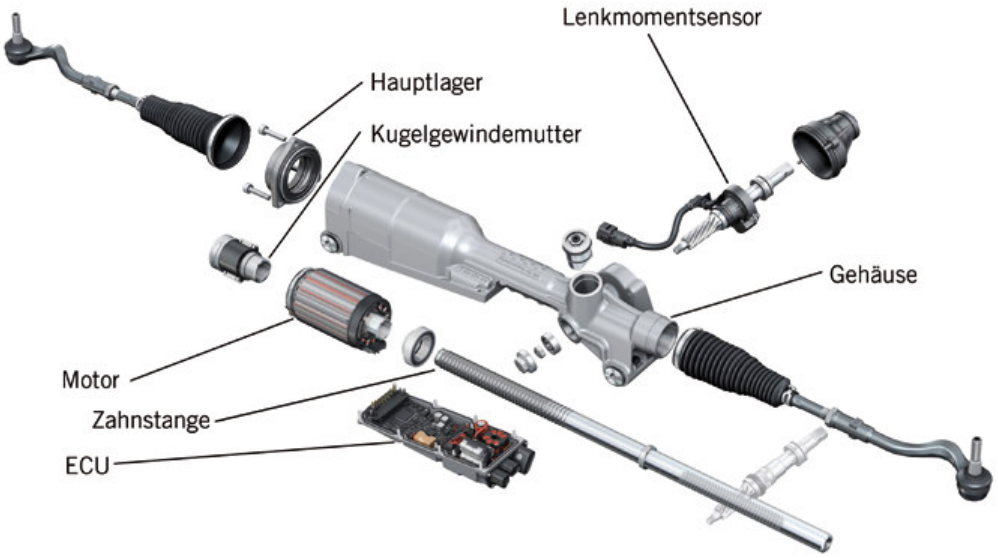


Abbildung 3.3: Lenksystem RCEPS aus [SSDJ12]

Die Applikationssoftware stellt Funktionen der Lenkung bereit und wird in der Lenkungselektronik-Entwicklung der Volkswagen AG erstellt. Abhängig von der Konfiguration des Fahrzeugs werden folgende Funktionen realisiert:

- Die *geschwindigkeitsabhängige Lenkunterstützung* sorgt dafür, dass bei langsamer Fahrt mehr unterstützt wird als bei schneller Fahrt. Bei langsamer Fahrt ist durch die Radreibung auf der Straße mehr Kraft notwendig, bei schneller Fahrt wird so ein präziseres Lenken ermöglicht.

- Bedingt durch die Trägheit des elektromechanischen Systems wird durch die *aktive Rückstellung* die Rückstellung des Lenkrads in die Mittellage (beispielsweise nach Kurven) unterstützt. Mit der *Geradeauslauffunktion* werden dazu schiefe Fahrbahnen und Seitenwindeinflüsse berücksichtigt, damit das Fahrzeug auch ohne manuelle Lenkeingriffe in der Spur bleibt.

- Für den Komfort sorgt eine aktive Dämpfungsfunktion, die verhindert, dass Störungen von der Straße am Lenkrad spürbar sind. Durch die *Stabilitäts- und Dynamikfunktion* ist jederzeit ein optimales Ansprechverhalten der Lenkung sichergestellt. Für ein präzises gleichbleibendes Lenkgefühl über den gesamten Temparaturbereich kommt eine *Reibungskompensation* zum Einsatz.

- Die Schonung der Mechanik und gleichzeitige Erhöhung des Komforts im Bereich der mechanischen Endanschläge übernimmt die *Software-Endanschlagfunktion*, indem in diesem Bereich gezielt verringert wird.

- Im Falle einer Störung des Fahrzeug-Bordnetzes ermöglicht eine *Unterspannungsfunktion* die adäquate Reaktion, so dass sich das Lenksystem maßvoll der aktuellen Situation adaptiert.

- Das Lenksystem enthält die Aktorik für mehrere Assistenzsysteme. Für das *automatische Einparken* und den *Spurhalteassistenten* werden Lenkmomente plausibilisiert und dem Lenksystem aufgeprägt. Die Plausibilisierung verhindert die fehlerhafte Ansteuerung.

- Bedingt durch das Übersetzungskonzept bei Fahrzeugen mit quer eingebautem Verbrennungsmotor wird bei leistungsstarken Fahrzeugen eine *Schiefziehkompensation* durchgeführt, um spurgetreues Verhalten bei Beschleunigungsvorgängen zu ermöglichen.

- Für die Sicherheit und ein stabiles Fahrzeugverhalten werden *Lenkmomentempfehlungen bei ESP-Eingriffen* umgesetzt.

- Zur Erhöhung der Verfügbarkeit kann der als Hardware vorhandene Lenkmomentsensor bei dessen Ausfall durch einen *virtuellen Lenkmomentsensor* ersetzt werden, der aus dem aktuellen Winkel des Motors und dem Lenkwinkel des Sensors am Fahrzeuglenkrad ein Lenkmoment errechnet.

- Das individuelle Lenkgefühl kann über diverse Programme von sportlich bis komfortabel mit einer *Profilfunktion* verstellt werden.

Ebenfalls Bestandteil der Applikationssoftware sind Sicherheitsfunktionen, die die genannten Funktionen überwachen. Somit wird eine Absicherung der Applikationssoftware nach den internationalen Sicherheitsnormen erreicht.

3.1.2 Entwicklungsprozess

Der Entwicklungsprozess folgt dem V-Modell [VMod]. Da das Lenksystem eine Komponente des Gesamtfahrzeugs ist, definiert die Abteilung, die das Gesamtfahrzeug entwickelt, die entsprechenden Anforderungen (Lastenhefte). Seitens der Lenkungsentwicklung werden diese auf Anforderungen für das Lenksystem übertragen und gemäß der Architektur weiter detailliert. Dazu werden die mechanischen und elektronischen Anteile separat behandelt. Der Elektronikteil wird hinsichtlich der Hardware und der Software weiter unterteilt.

So entstehen letztendlich die Anforderungen an die Software, die sowohl funktionale Ziele als auch Sicherheitsziele definieren. Die beschriebenen Funktionen bilden Komponenten der Software, die atomar entwickelt und getestet werden. Diese werden anschließend

gemäß der oben beschriebenen Aufteilung wieder integriert und abgesichert. Jede Integrationsstufe wird durch eine Freigabe begleitet, bei der durch Reviews geprüft wird, ob neben den funktionalen Zielen auch Qualitäts- und Sicherheitskriterien eingehalten wurden.

3.2 Beschreibung der Maßnahmen

Nachfolgend sind die Maßnahmen beschrieben, die sich aus den Erkenntnissen über Software-Produktlinien aus Kapitel 2 in Arbeitspakete ableiten lassen, die Schritte zu einer produktlinienbasierten Entwicklung darstellen. Es ist für eine erfolgreiche Entwicklung nicht zwingend notwendig alle Maßnahmen umzusetzen. In der Praxis bedeutet das, dass nach der Analyse aus den verfügbaren Maßnahmen diejenigen ausgewählt werden, die unter Berücksichtigung der Analyseergebnisse einen möglichst großen Nutzen bieten.

Zunächst werden die Maßnahmen zur Analyse beschrieben, die immer durchgeführt werden müssen, um einen Einblick in die bisherige Entwicklung zu bekommen. Dafür ist die Untersuchung der Software und die Bewertung der Einflussfaktoren notwendig. Die Maßnahmen zur Durchführung tragen zur Realisierung der Produktlinienentwicklung bei. Neben der Definition der durchzuführenden Strategie gibt es eine Reihe von möglichen Maßnahmen, die zur Umsetzung der Strategie beitragen.

Untersuchung der Software (M1). Bei einer Software-Produktlinien-Strategie ist es wichtig die Software zu kennen. Mit einer Domänenanalyse werden Gemeinsamkeiten und Unterschiede der Software erfasst. Auf Architekturebene bedeutet dies, zu wissen, welche Komponenten in den Software-Varianten existieren, und die Schnittstellen zu vergleichen. Somit können Gemeinsamkeiten und Unterschiede quantifiziert und ansatzpunkte für genauere Untersuchungen identifiziert werden. Komponenten mit gleichen Schnittstellen können beispielsweise intern unterschiedliche Funktionen erfüllen. Diese Art der Softwareanalyse wird in [DMH02, HDM03] ebenfalls beschrieben.

Bewertung der Einflussfaktoren (M2). Neben der Implementierung ist es auch wichtig das Umfeld der Entwicklung zu kennen. Es müssen eine Reihe von Einflussfaktoren berücksichtigt und bewertet werden. Die Ausprägung der Einflussfaktoren beeinflusst direkt den Umfang der Möglichkeiten, die für eine Umsetzung existieren. Im Rahmen einer Produktlinienentwicklung werden in der Literatur Einflussfaktoren selten explizit betrachtet. In dieser Arbeit geht es darum, wie die Einflussfaktoren sich in der Realität auswirken, weshalb sie intensiv betrachtet werden.

Definition der Strategie (M3). Jede Entwicklung ist anders und jede Produktlinienentwicklung erfordert die Definition einer angepassten Strategie. In der Strategie ist festgehalten, welche der hier aufgeführten Maßnahmen in welchem Zeitrahmen von welchen Personen umgesetzt werden. Sie stellt das angepasste Vorgehen auf Basis der Domänenanalyse dar. Je mehr Maßnahmen umgesetzt werden, desto wichtiger ist, dass diese Strategie detailliert festgehalten und bei Bedarf während der Umsetzung aktualisiert wird. In der Literatur wird an diesem Punkt oft auf Standard-Vorgehensweisen verwiesen (z.B. [Sch04]), die hauptsächlich die Anpassung der Entwicklung und der Architektur beschreiben. Dabei ist eine Voraussetzung, dass viele Vorbedingungen schon erfüllt sind und sofort Produktlinien-Methoden verankert werden können. Dies ist in der Realität jedoch nur selten der Fall.

Kommunikation der Produktlinieninhalte an die Mitarbeiter (M4). Die in Maßnahme M3 definierte Strategie muss an alle Beteiligten kommuniziert und mit ihnen abgestimmt werden. Für die Umsetzung einer Produktlinie ist es aber auch wichtig, dass für die Mitarbeiter und vor allem für die Führungsebenen die Produktlinie aufbereitet wird. Es muss schnell erfasst werden können, welche Produkte im Rahmen der Produktlinienentwicklung berücksichtigt werden und welche beteiligten Personen welche Rollen erfüllen. In der Literatur wird auf die Kommunikation und den Kommunikationsweg zu den Mitarbeitern nicht eingegangen. Dabei spielt die Art der Vermittlung der Informationen zu den Mitarbeitern eine wichtige Rolle, damit diese auch von den Mitarbeitern verstanden und verinnerlicht werden.

Automatisierte Erfassung von Gemeinsamkeiten und Unterschieden (M5). Im Rahmen der Untersuchung mit der Maßnahme M1 wurden Gemeinsamkeiten und Unterschiede der Software erfasst. Die Software kann sich im Verlauf der Umsetzung einer Produktlinienentwicklung ändern, was insbesondere dann vorkommt, wenn Produktlinienmethoden innerhalb einer laufenden Entwicklung umgesetzt werden. Es wird im Verlauf der Umsetzung immer wieder eine Überprüfung der Gemeinsamkeiten und Unterschiede notwendig sein, damit sichergestellt ist, dass die Umsetzung auf dem richtigen Weg ist. Eine manuelle Erfassung bedeutet einen immensen Aufwand, weshalb eine Automatisierung dieses Schrittes sinnvoll ist. In der Literatur gibt es eine Reihe von Kennzahlen (z.B. [ZDW+08]), die das Potential haben automatisch ermittelt zu werden.

Identifikation und Nutzung von Modellen (M6). Kern einer erfolgreichen Produktlinienumsetzung ist die Identifikation von bisher nur implizit genutzten Modellen. Nur mit einer modellbasierten Entwicklung kann die Komplexität der heutigen Software-Produkte bewältigt werden. Beispielsweise ermöglichen die Modelle der UML die effiziente Entwicklung qualitativ hochwertiger Software [Rum11]. Dabei müssen nicht in jedem Fall neue Modelle entwickelt werden, sondern oft gibt es bereits Modelle, die sich

in weiteren Kontexten nutzen lassen. Dazu muss teilweise die Semantik dieser Modelle präziser definiert werden, wobei der Nutzen größer ist als der Aufwand, der dafür anfällt [HR04]. An Modellen lassen sich für eine Produktlinienentwicklung besser Variabilitätsmechanismen anwenden, um Varianten zu beschreiben.

Einsatz von Werkzeugen (M7). Modelle entfalten erst ihr volles Potential, wenn es Werkzeuge im Entwicklungsprozess gibt, die bisherige Aktivitäten automatisieren. Beispiele hierfür sind die Quellcode-Generierung aus Matlab/Simulink-Modellen und die Ableitung von Java-Code aus Modellen der UML/P [Rum12]. Überall, wo eine systematische Transformation von einer Sprache in eine andere notwendig ist, können Werkzeuge optimal eingesetzt werden. So ist es möglich auch Dokumentation und weitere Entwicklungsartefakte mittels Werkzeugen aus Modellen zu erzeugen.

Anbindung bestehender Datenbanken (M8). Nicht alle Modelle der Entwicklungsprojekte bei Volkswagen liegen als eine Datei auf der Festplatte vor. Sie können in unterschiedlichen Systemen in unterschiedlichen Formen liegen. Beispiele sind SQL-Datenbanken und Anforderungsmanagement-Systeme, die Informationen anwendungsspezifisch zugänglich machen. Für eine automatisierte Verarbeitung der Modelle im Rahmen der Entwicklung muss eine Schnittstelle für Werkzeuge geschaffen werden, sofern diese noch nicht existiert. Dabei unterstützt ein Metamodell bei der Festlegung, wie Informationen aus den Systemen extrahiert werden sollen.

Festlegung der Variabilitätsrealisierung (M9). Wenn Varianten realisiert werden, muss von vorne herein klar sein, wie diese definiert werden sollen. Es existieren dazu mehrere Möglichkeiten (siehe Abschnitt 2.3). Eine Verwendung von mehreren Variabilitätsrealisierungen ist hierbei sehr verwirrend. Aus diesem Grund soll für die Entwicklung bei Volkswagen untersucht werden, welche Form der Variabilitätsrealisierung am besten geeignet ist. Diese Wahl sollte sorgfältig getroffen werden, da ein späterer Wechsel eine umfangreiche Änderung der Entwicklungsprozesse und der Entwicklungsartefakte notwendig macht.

Nutzung der Variabilität in Modellen (M10). Nach der Entscheidung für eine Variabilitätsrealisierung muss diese auch in den Modellen angewandt werden. Je nach gewähltem Paradigma ist eine Erweiterung der Modelle oder die Erstellung zusätzlicher Artefakte notwendig. Am Anfang ist der Umgang mit der Variabilität noch nicht unbedingt vertraut, so dass es im Zuge der Entscheidung auch Sinn macht, die verschiedenen Variabilitätsmöglichkeiten an kleinen Modellen durchzuspielen und Erfahrungen damit zu sammeln. Die Realisierung der Variabilität ist in der Literatur eingehend beschrieben

(z.B. in [GHK+08b] für die negative Variabilitätsdarstellung). Ein Vergleich am praktischen Beispiel zeigt die Vor- und Nachteile im jeweiligen Projektkontext.

Nutzung von Variabilität in der Entwicklung (M11). Ein großer Schritt Richtung einer Produktlinienentwicklung wird durch die durchgängige Nutzung der Variabilität in den Entwicklungsartefakten erreicht. Dazu müssen die Varianten nicht nur in den Ursprungsmodellen, sondern bis hin zum Quellcode sichtbar sein. Damit einher geht, dass Artefakte, die bisher nur durch Kopien entstanden sind, durch Varianten ersetzt werden. Der Umgang mit der Variabilität ist Kern von vielen Veröffentlichungen zum Thema Software-Produktlinien und wird auch entsprechend tief betrachtet (z.B. in [PBL05]).

Werkzeugentwicklung und -einsatz für die Produktlinienentwicklung (M12). Neben der Definition der Variabilität ist es auch notwendig diese zu binden, damit konkrete Produkte entstehen. Dies wird mit passenden Werkzeugen innerhalb der Produktlinienentwicklung erreicht. Abhängig vom gewählten Variabilitätsmodell wird die Bindung der Variabilität durch Standard-Werkzeuge unterstützt. Oft müssen jedoch Erweiterungen oder neue Werkzeuge für den gewählten Kontext entwickelt werden.

Nutzung von Variabilität im Test (M13). Auch im Test müssen Varianten berücksichtigt werden, um diese gezielt zu testen und bestehende Tests für neue Varianten zu erweitern. Dies ist nur möglich, wenn auch die Tests durch ein Variabilitätsmodell beschrieben sind, damit nicht zu viel, aber auch nicht zu wenig getestet wird. Der Test von Produktlinien-Applikationen wird ebenfalls in [KPRR04] thematisiert.

Anpassung der Architektur an die Variabilität (M14). Die Variabilität muss auch in der Architektur verankert werden. Dazu müssen gemeinsam genutzte Teile und individuelle Teile voneinander getrennt werden. So entstehen neue Komponenten, die auf der Implementierungsebene erstellt werden müssen. Die Architektur ist ein wichtiger Bestandteil einer Produktlinienentwicklung und wird deshalb auch in der Literatur eingehend behandelt (z.B. in [PBL05]).

Definition eines Merkmalmodells für den Problem Space (M15). Um die Variabilität für den Kunden nutzbar zu machen, wird ein Modell benötigt, wie es in Abschnitt 2.3.1 beschrieben wird. Dazu muss zunächst entschieden werden, ob z.B. ein Feature-Diagramm oder eine entscheidungsbasierte Darstellung gewählt wird. Danach muss das Modell definiert und mit dem Solution Space verknüpft werden. Es gibt viele Veröffentlichungen, die sich mit möglichen Merkmalmodellen für den Problem Space beschäftigen (siehe Abschnitt 2.3).

Strukturierung nach Entwicklungsteams für Domain und Application Engineering (M16). Für eine Produktlinienentwicklung müssen separate Entwicklungen und eigene Entwicklungsteams für Domain und Application Engineering, wie in Abschnitt 2.2 beschrieben, eingeführt werden. Nur durch eigenständige Teams werden alte Strukturen aufgebrochen, damit nicht die alte Entwicklung unter neuem Namen weiterläuft. Diese Maßnahme wird auch in [BHJ+03] als wichtig erachtet.

Aufbau und Nutzung einer Modulbibliothek (M17). Eine Modulbibliothek stellt die vollständige Umsetzung einer Produktlinien-Strategie dar. Artefakte des Application Engineering werden in die Modulbibliothek aufgenommen und im Application Engineering verwendet. Über diese Schnittstelle werden Domain Engineering und Application Engineering wieder zu einem Prozess zusammengeführt. Der Inhalt der Modulbibliothek wird in der Literatur meist als Assets bezeichnet.

3.3 Abhängigkeiten der Maßnahmen

Der Großteil der Maßnahmen kann nicht atomar durchgeführt werden, sondern bedingt die Durchführung anderer Maßnahmen. Die inhaltlichen und zeitlichen Abhängigkeiten der Maßnahmen sind in Abbildung 3.4 dargestellt. Zur besseren Übersicht sind die Maßnahmen in Bereiche zusammengefasst.

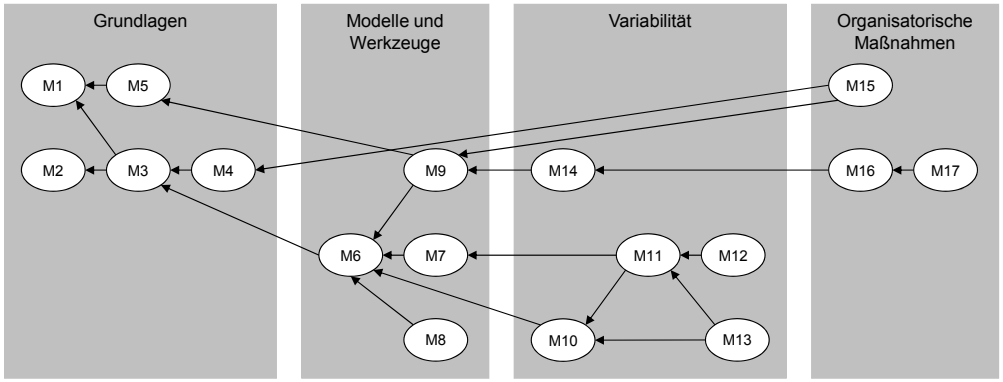


Abbildung 3.4: Abhängigkeiten der Maßnahmen zur Einführung von Produktlinien

Der erste Bereich enthält die Grundlagen für die Produktlinienentwicklung. Die Maßnahmen M1 (Untersuchung der Software) und M2 (Bewertung der Einflussfaktoren) stellen die Maßnahmen im Rahmen der Analyse dar. Es empfiehlt sich M1 gleich um die Maßnahme M5 (Automatisierte Erfassung von Gemeinsamkeiten und Unterschieden) zu ergänzen. Die Ergebnisse aus M1 und M2 werden in der Maßnahme M3 (Definition der

Strategie) verwendet. Mit der Maßnahme M4 (Kommunikation der Produktlinieninhalte an die Mitarbeiter) kann diese dann dem Team kommuniziert werden.

Der zweite Bereich ist der Bereich Modelle und Werkzeuge. Dabei steht die Maßnahme M6 (Identifikation und Nutzung von Modellen) zunächst im Vordergrund. Darauf aufbauend werden die Maßnahmen M7 (Einsatz von Werkzeugen), M8 (Anbindung bestehender Datenbanken) und M9 (Festlegung der Variabilitätsrealisierung) durchgeführt. Für letztere sind die Gemeinsamkeiten und Unterschiede aus M5 notwendig.

Im dritten Bereich wird die Variabilität genauer betrachtet. Die Variabilitätsfestlegung aus M6 und M9 wird in der Maßnahme M10 (Nutzung der Variabilität in Modellen) verwendet. Modelle mit Variabilität stellen die Basis für die Maßnahme M11 (Nutzung von Variabilität in der Entwicklung) dar. Die Variabilität muss dann weiter in den Maßnahmen M12 (Werkzeugentwicklung und -einsatz für die Produktlinienentwicklung) und M13 (Nutzung von Variabilität im Test) berücksichtigt werden. Parallel dazu muss die Maßnahme M14 (Anpassung der Architektur an die Variabilität) durchgeführt werden, die auf den Ergebnissen von M9 aufbaut.

Der vierte Bereich enthält organisatorische Maßnahmen. Mit den Ergebnissen von M4 und M9 wird die Maßnahme M15 (Definition eines Merkmalmodells für den Problem Space) durchgeführt. Aus der angepassten Architektur von M14 wird die Maßnahme M16 (Strukturierung nach Entwicklungsteams für Domain und Application Engineering) und darauffolgend M17 (Aufbau und Nutzung einer Modulbibliothek) durchgeführt.

3.4 Zusammenfassung

In diesem Kapitel wurden die möglichen Maßnahmen für die Einführung einer produktlinienbasierten Entwicklung in der Softwareentwicklung für elektromechanische Lenksysteme bei der Volkswagen AG besprochen. Dazu wurde die bisherige Entwicklung vorgestellt und auf Kapitel 2 basierende umsetzbare Arbeitspakete abgeleitet. Anschließend wurden die Abhängigkeiten der Maßnahmen untersucht.

In Tabelle 3.1 werden die Maßnahmen, die zur Einführung einer produktlinienbasierten Entwicklung führen, zusammengefasst. Die umgesetzten Maßnahmen werden in den Folgekapiteln beschrieben. Dabei wird neben der allgemeinen Darstellung auch die Realisierung im konkreten Projektumfeld besprochen. Nicht alle Maßnahmen wurden im Rahmen dieser Arbeit umgesetzt.

Tabelle 3.1: Maßnahmen für die Einführung einer Produktlinienentwicklung

Nummer	Maßnahme
Grundlagen	
M1	Untersuchung der Software
M2	Bewertung der Einflussfaktoren
M3	Definition der Strategie
M4	Kommunikation der Produktlinieninhalte an die Mitarbeiter
M5	Automatisierte Erfassung von Gemeinsamkeiten und Unterschieden
Modelle und Werkzeuge	
M6	Identifikation und Nutzung von Modellen
M7	Einsatz von Werkzeugen
M8	Anbindung bestehender Datenbanken
M9	Festlegung der Variabilitätsrealisierung
Variabilität	
M10	Nutzung der Variabilität in Modellen
M11	Nutzung von Variabilität in der Entwicklung
M12	Werkzeugentwicklung und -einsatz für die Produktlinienentwicklung
M13	Nutzung von Variabilität im Test
M14	Anpassung der Architektur an die Variabilität
Organisatorische Maßnahmen	
M15	Definition eines Merkmalmodells für den Problem Space
M16	Strukturierung nach Entwicklungsteams für Domain und Application Engineering
M17	Aufbau und Nutzung einer Modulbibliothek

Kapitel 4

Analyse der bestehenden Software

Wenn mehrere Einzelprodukte in eine Produktlinie überführt werden sollen, dann steht am Anfang eine Vermutung, dass die Produkte Ähnlichkeiten haben. Für die adäquate Beurteilung müssen die bestehenden Softwarestände systematisch auf Gemeinsamkeiten und Unterschiede untersucht werden. Dafür wird in diesem Kapitel eine Methode vorgestellt, wie die Ähnlichkeit quantifiziert werden kann. Mittels Kennzahlen kann das Wiederverwertungspotential dargestellt werden, womit die Maßnahme M1 (Untersuchung der Software) realisiert wird. Da diese Bewertung nicht nur am Anfang, sondern kontinuierlich geschehen muss, ist eine automatisierte Ermittlung dieser Kennzahlen als Implementierung der Maßnahme M5 (Automatisierte Erfassung von Gemeinsamkeiten und Unterschieden) notwendig.

Ein Ansatzpunkt für die Extraktion der bestehenden Variabilität ist die Untersuchung der Anforderungen. Da diese ganz unterschiedlich für mehrere Systeme definiert sein können, ist es schwierig Gemeinsamkeiten zu finden. Es gibt verschiedene Methoden, um ähnliche Anforderungen zu erkennen und einander zuzuordnen [ASB+08]. Daraus kann ein Variabilitätsmodell für den Problem Space aufgestellt werden. Aus diesem Variabilitätsmodell (in vielen Fällen ein Feature-Diagramm) können mit dem in [HKO+06] vorgestellten Framework dann Qualitätsattribute bewertet werden.

Für den Solution Space muss die Architektur eines Systems untersucht werden. Eine Metrik, die angewendet werden kann, wird in [DMH02, HDM03] präsentiert. Basierend auf einer bereits durchgeführten Aufteilung in gemeinsame und optionale Komponenten werden, die Schnittstellen der Komponenten analysiert um festzustellen, ob die Einteilung in gemeinsame und variable Teile sinnvoll ist.

Für die Betrachtung des Systems aus Verhaltenssicht bietet [GLS08] einen Ansatz. Die Semantik des Systems wird in Labeled Transition Systems modelliert. Mit einer solchen Verhaltensbeschreibung kann ein System nach Gemeinsamkeiten im Verhalten untersucht werden. Ein Überblick über mögliche Metriken zur Beurteilung von Software Produktlinien ist in [MA09] und [ZDW+08] zu finden.

Die im Rahmen dieser Arbeit entwickelte Methodik basiert auf einer vereinfachten Repräsentation von Blockdiagrammen wie Matlab/Simulink, in der die produktidentifizierenden Funktionen modelliert sind. Diese Repräsentation für den Solution Space wird in Abschnitt 4.1 dargestellt. Darauf basierend werden in Abschnitt 4.2 Kennzahlen vorgestellt, die Gemeinsamkeiten und Unterschiede quantifizieren. Theoretisch werden die Kennzahlen in Abschnitt 4.3 ermittelt und in Abschnitt 4.4 für die Entwicklungsprojekte bei Volkswagen eingesetzt. Darüber hinaus gehende Analysemöglichkeiten werden in Abschnitt 4.5 diskutiert.

4.1 Datenmodell zur Analyse

In der modellbasierten Entwicklung werden eingebettete Systeme häufig durch Blockdiagramme dargestellt [CFGK05]. Diese Blockdiagramme enthalten Blöcke, die eine Funktionalität (z.B. Berechnung) beinhalten, und Verbindungen zwischen diesen Blöcken, die den Datenfluss zeigen (siehe Abbildung 4.1). Um eine möglichst einfache und leichtgewichtige Repräsentation zu nutzen, wurde in dieser Arbeit das Datenmodell auf diese grundsätzlichen Elemente beschränkt. In [GL00] wird ebenfalls diese Sicht auf ein System als Grundlage für Metriken verwendet.

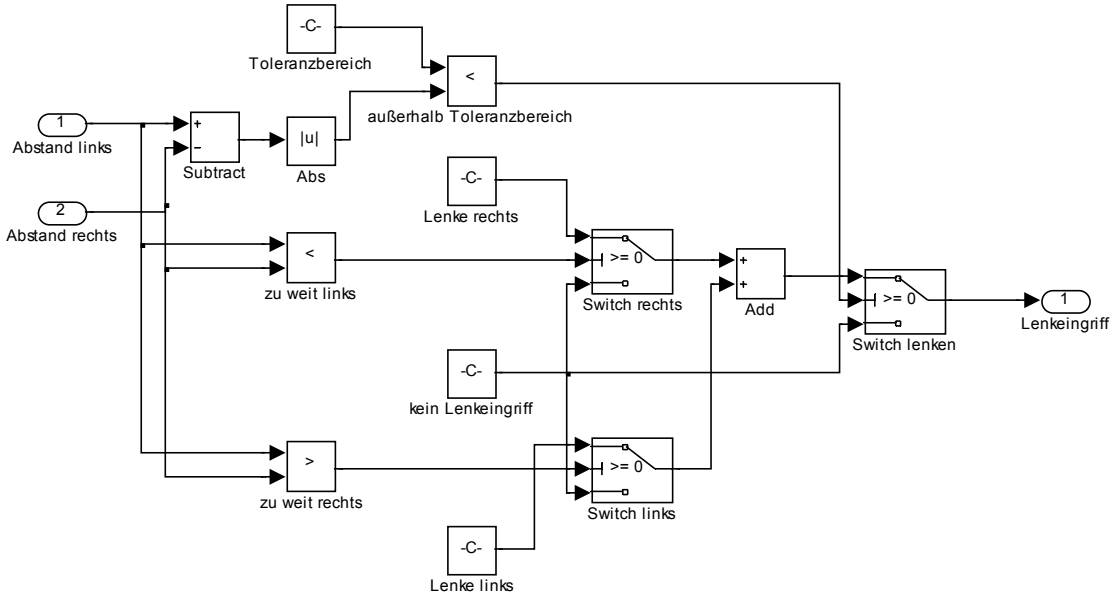


Abbildung 4.1: Matlab/Simulink als Beispiel für ein Blockdiagramm

Viele Systeme lassen sich mit wenigen Elementen wie Komponenten und Verbindungen modellieren. Matlab/Simulink und ASCET sind Beispiele für Software-Werkzeuge, die mit diesen Elementen arbeiten. Diese Werkzeuge bieten darüber hinaus weitere

Sprachelemente, die zur Modellierung eines Systems eingesetzt werden können. MontiArc [HRR12, RRW13, RRW14] ist eine Sprache zur Modellierung der Architektur von verteilten Systemen, welche ebenfalls auf Komponenten mit Verbindungen basiert. Zusätzliche Sprachelemente schaffen weitere Modellierungsmöglichkeiten, die für das im folgenden verwendete System nicht notwendig sind.

4.1.1 System

Grundsätzlich wird ein System S_i als eine Menge von Komponenten C_i und Datenflüssen D_i verstanden (siehe Definition (4.1)).

$$S_i = (C_i, D_i) \tag{4.1}$$

Eine Komponente $c \in C_i$ wird durch einen vollqualifizierten Namen gekennzeichnet. Dies ermöglicht die implizite Definition von Subsystem-Beziehungen. Komponenten haben folgende Eigenschaften:

- Vollqualifizierter Name der Komponente: Dafür wird im Folgenden $fullname(c)$ verwendet.
- Name der Komponente: Im Folgenden wird der Name mittels $name(c)$ angesprochen.
- Name des Pfades, in dem die Komponente liegt: Mittels $path(c)$ kann auf diesen zugegriffen werden. $path(c)$ und $name(c)$ ergeben zusammen $fullname(c)$.

Wenn eine Komponente den vollqualifizierten Namen „subsystem_a/komp_1“ hat, dann referenziert $fullname$ genau diesen Wert. Mittels $name$ wird auf „komp_1“ zugegriffen und mittels $path$ auf „subsystem_a“. Es gibt immer zwei Spezialkomponenten, welche die Systemgrenze darstellen. Aus der „Quelle“ entstammen alle Datenflüsse, die außerhalb der Systemgrenze ihren Ursprung haben. Datenflüsse, die ihr Ziel außerhalb der Systemgrenzen haben, münden in der Komponente „Senke“.

Die Datenflüsse $d \in D_i$ stellen atomare Datenflüsse von einer Komponente zu einer anderen dar. Datenflüsse bestehen aus folgenden Informationen.

- Startkomponente: Auf diese wird im Folgenden mittels $source(d)$ zugegriffen.
- Zielkomponente: Im Folgenden wird diese mittels $target(d)$ angesprochen.

- Inhalt des Datenflusses (Informationstyp): Für diesen wird im Folgenden $cont(d)$ verwendet.

Aus der *source* bzw. *target* Beziehung eines Datenflusses lassen sich für Komponenten Zugriffsmöglichkeiten für die Eingänge und Ausgänge ableiten. $inputs(c)$ bezeichnet alle eingehenden Datenflüsse einer Komponenten und analog dazu werden mit $outputs(c)$ alle ausgehenden Datenflüsse angesprochen (Definition (4.2).

$$\begin{aligned} inputs(c) &= \{d \in D_i \mid target(d) = c\} \\ outputs(c) &= \{d \in D_i \mid source(d) = c\} \end{aligned} \tag{4.2}$$

4.1.2 Ähnlichkeitsfunktion

Der Vergleich mehrerer Systeme erfolgt über zwei Ähnlichkeitsfunktionen, jeweils eine für Komponenten ($simfc$) und eine Datenflüsse ($simfd$). Diese Funktionen geben für zwei Komponenten bzw. Datenflüsse zurück, ob wie ähnlich diese sind (siehe Definition (4.3)). Das Ergebnis 1 bedeutet, dass die beiden Komponenten bzw. Datenflüsse ähnlich sind, eine 0, dass sie keinerlei Ähnlichkeit aufweisen.

$$\begin{aligned} simfc &: c \times c \mapsto \{0,1\} \\ simfd &: d \times d \mapsto \{0,1\} \end{aligned} \tag{4.3}$$

Ein Beispiel für eine Ähnlichkeitsfunktion ist die Funktion ne (nameequality) in Definition (4.4).

$$ne(c_1, c_2) = \begin{cases} 1 & \text{wenn } name(c_1) = name(c_2) \\ 0 & \text{sonst} \end{cases}$$

$$ne(d_1, d_2) = \begin{cases} 1 & \text{wenn } name(source(d_1)) = name(source(d_2)) \\ & \quad \wedge name(target(d_1)) = name(target(d_2)) \\ & \quad \wedge cont(d_1) = cont(d_2) \\ 0 & \text{sonst} \end{cases} \tag{4.4}$$

Diese Funktionen vergleichen Komponenten anhand ihres Namens. Das Subsystem, in dem sie liegen, wird nicht berücksichtigt. Das Gleiche gilt für den Vergleich der Datenflüsse, bei denen nur die Namen der Start- und Zielkomponente von Bedeutung sind.

Mit dieser Ähnlichkeitsfunktion werden Komponenten und Datenflüsse auch dann als ähnlich definiert, wenn Komponenten in andere Subkomponenten verschoben wurden.

Eine weitere Ähnlichkeitsfunktion beschreibt die Ähnlichkeit, basierend auf den eingehenden und ausgehenden Datenflüssen einer Komponente, um beispielsweise umbenannte Komponenten zu identifizieren. Diese Funktion *pie* (partial interface equality) ist in Definition (4.5) festgehalten.

$$pie(c_1, c_2, n) = \begin{cases} 1 & \text{wenn } \frac{|\{d_1 \in inputs(c_1) | \exists d_2 \in inputs(c_2) \colon cont(d_1) = cont(d_2)\}|}{|inputs(c_1)| + |inputs(c_2)|} \geq \frac{n}{2} \\ & \quad \wedge \frac{|\{d_1 \in outputs(c_1) | \exists d_2 \in outputs(c_2) \colon cont(d_1) = cont(d_2)\}|}{|outputs(c_1)| + |outputs(c_2)|} \geq \frac{n}{2} \\ 0 & \text{sonst} \end{cases} \tag{4.5}$$

Für die Funktion für die partielle Schnittstellen-Gleichheit kann durch den Parameter n definiert werden, wie groß die Übereinstimmung sein muss. Bei $n = 1$ muss die Schnittstelle identisch sein, für $n = 0,5$ müssen sich nur die Hälfte aller Eingangs- bzw. Ausgangssignale auch in der anderen Komponente wiederfinden. Die Schnittstellen-Gleichheit lässt sich für Datenflüsse nicht sinnvoll definieren, da jeder Datenfluss nur seine jeweilige Start- und Zielkomponente kennt. Ein Abgleich der Namen der Start- und Zielkomponente soll bei der partiellen Schnittstellen-Gleichheit gerade nicht stattfinden, um Umbenennungen zu ermöglichen.

Ähnlichkeitsfunktionen lassen sich auch kombinieren. So lässt sich die Ähnlichkeitsfunktion *npie* (name and partial interface equality), die zum einen die Namensgleichheit und zum anderen die partielle Schnittstellengleichheit fordert, durch die Definition (4.6) darstellen, die die Formeln für *ne* (Definition (4.4)) und für *pie* (Definition (4.5)) kombiniert. So können gleich benannte Komponenten, die in der Software verschoben und teilweise modifiziert wurden, relativ zuverlässig wiedererkannt werden.

$$npie(c_1, c_2, n) = ne(c_1, c_2) \times pie(c_1, c_2, n) \tag{4.6}$$

4.2 Kennzahlen zur Analyse

Mit der Systemdefinition und einer Ähnlichkeitsfunktion lassen sich Kennzahlen definieren, die die Ähnlichkeit mehrere Systeme beschreiben. Die Systeme werden anhand des Indizes $i = 1 \ldots k$ unterschieden. Dabei macht es keinen Unterschied, ob es sich um mehrere Varianten eines Systems oder mehrere Versionen eines Systems handelt.

Für die erste Kennzahl ist die Menge der ähnlichen Komponenten *Common Components* (CC) maßgebend. Diese ist in Definition 4.7 festgehalten. Die Menge CC enthält die Komponenten des ersten Systems, die über die Ähnlichkeitsfunktion eine ähnliche Komponente in jedem anderen untersuchten System haben. Generell werden bei allen Kennzahlen, die Komponenten berücksichtigen, die Spezialkomponenten „Quelle“ und „Ziel“ nicht mit in die Berechnung einbezogen. Diese beiden Spezialkomponenten stellen Teile dar, die außerhalb des Systems sind. Beispiele in der Praxis sind Sensoren oder Aktoren.

$$CC = \{c_1 \in C_1 \mid \forall i = 2 \ldots k \ \exists c_i \in C_i : simfc(c_1, c_i) = 1\} \tag{4.7}$$

Daraus lässt sich die Kennzahl *Size of Common Components* ($SoCC$) ableiten. Diese ist durch Definition (4.8) beschrieben. Mit dieser Kennzahl lässt sich ausdrücken, ob überhaupt Ähnlichkeiten vorhanden sind. Diese Kennzahl basiert auf der oben beschriebenen Annahme, dass sich Ähnlichkeiten aus gleichnamigen Komponenten herleiten lassen. Gibt es keine Komponente, die in mehreren Systemen vorhanden ist, dann ist die Umsetzung einer Produktlinie nicht sinnvoll.

$$SoCC = |CC| \tag{4.8}$$

Für den Einsatz einer Produktlinie ist jedoch nicht die absolute Anzahl an gemeinsam genutzten Komponenten interessant, sondern der gemeinsame Anteil je System (Definition (4.9)). Diese Kennzahl zeigt an, welcher Anteil bei mehreren Systemen theoretisch in allen Systemen wiederverwendet werden kann, und setzt dazu die Anzahl der gleichen Komponenten zu der Gesamtanzahl aller Komponenten eines Systems in Beziehung.

$$RSoCC_i = \frac{SoCC}{|C_i|} \tag{4.9}$$

$SoCC$ zeigt die Übereinstimmung nur aufgrund des Namens einer Komponente an. Viel aussagekräftiger für die Ähnlichkeit ist jedoch ein Vergleich der Signalflüsse in den Systemen. Ein Signalfluss entspricht genau einer Kante in der oben beschriebenen Darstellung. Mit dieser Annahme lässt sich analog zu der Definition (4.7) die Menge von gemeinsamen Datenflüssen über die Systeme *Common Dataflows* (CD) nach Definition (4.10) bestimmen.

$$CD = \{d_1 \in D_1 \mid \forall i = 2 \ldots k \ \exists d_i \in D_i : simfd(d_1, d_i) = 1\} \tag{4.10}$$

Aus der Definition (4.11) ergibt sich dementsprechend die *Size of Common Dataflows* ($SoCD$). Der relative Wert für jedes einzelne System im Rahmen einer Produktlinien-Betrachtung lässt sich durch Definition (4.12) ermitteln.

$$SoCD = |CD| \tag{4.11}$$

$$RSoCD_i = \frac{SoCD}{|D_i|} \tag{4.12}$$

Neben den Gemeinsamkeiten sind die Unterschiede von Interesse. Die Komponenten, die in keinem anderen System genutzt werden, werden durch Menge *Individual Components* (IC, Definition (4.13)) je System beschrieben.

$$IC_i = \{c_i \in C_i \mid \forall j = 1 \ldots k, j \neq i \nexists c_j \in C_j : simfc(c_i, c_j) = 1\} \tag{4.13}$$

Die Kennzahl *Size of Individual Components* ($SoIC$) in Definition (4.14) gibt an, bei wie vielen Komponenten in den jeweiligen Systemen keine Wiederverwendung möglich ist. Den nicht wiederverwendbaren Anteil am System gibt die Kennzahl *Relative Size of Individual Components* ($RSoIC$, Definition (4.15)) an.

$$SoIC_i = |IC_i| \tag{4.14}$$

$$RSoIC_i = \frac{SoIC_i}{|C_i|} \tag{4.15}$$

Ähnliche Kennzahlen können auch für die Datenflüsse definiert werden. Die Menge *Individual Dataflows* (ID) ergibt sich aus Definition (4.16).

$$ID_i = \{d_i \in D_i \mid \forall j = 1 \ldots k, j \neq i \nexists d_j \in D_j : simfd(d_i, d_j) = 1\} \tag{4.16}$$

Somit ergibt sich die *Size of Individual Dataflows* ($SoID$) in Definition (4.17) und die Kennzahl *Relative Size of Individual Dataflows* ($RSoID$) in Definition (4.18).

$$SoID_i = |ID_i| \tag{4.17}$$

$$RSoID_i = \frac{SoID_i}{|D_i|} \tag{4.18}$$

4.3 Exemplarische Durchführung der Analyse

In diesem Abschnitt werden die Kennzahlen an einem Beispiel angewendet, welches auch in [BRR10a] genutzt wird. Dabei werden drei Varianten eines Systems verglichen. Die erste Variante (S_1, Abbildung 4.2) stellt ein Türsteuergerät dar, welches ab einer festgelegten Fahrzeuggeschwindigkeit die Türen verriegelt. Bei der zweiten Variante (S_2, Abbildung 4.3) handelt es sich ein Türsteuergerät, welches elektrische Fensterheber ansteuert. Für Cabrios wird die dritte Variante (S_3, Abbildung 4.4) eingesetzt.

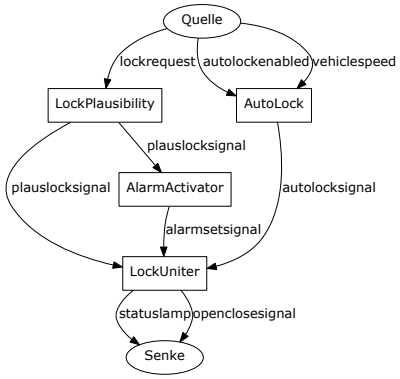


Abbildung 4.2: Beispiel 1: Türsteuergerät mit Auto-Lock

Mittels der Definition (4.4) werden Komponenten anhand ihres Namens verglichen. Auf dieser Basis wird jetzt die Kennzahl CC (Common Components, Definition (4.7)) über alle drei Systeme ermittelt. In CC befinden sich die drei Komponenten mit den Namen AlarmActivator, LockPlausibility bzw. LockUniter.

Die Betrachtung der Datenflüsse CD (Common Dataflows, Definition (4.10)) führt zu sechs Datenflüssen, die in allen Systemen vorhanden sind. Es handelt sich um die Datenflüsse der Tabelle 4.1.

In der Tabelle 4.2 werden diese Kennzahlen am Beispiel der oben eingeführten Systeme dargestellt. Anhand der Kennzahl $SoCC$ lässt sich erkennen, dass potentielle Ähnlichkeiten vorhanden sind. Dass diese Kennzahl größer als Null ist, ist die Voraussetzung für die

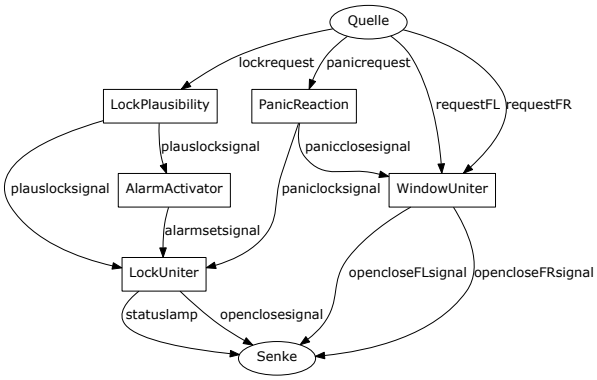


Abbildung 4.3: Beispiel 2: Türsteuergerät mit elektrischen Fensterhebern und Panik-Knopf

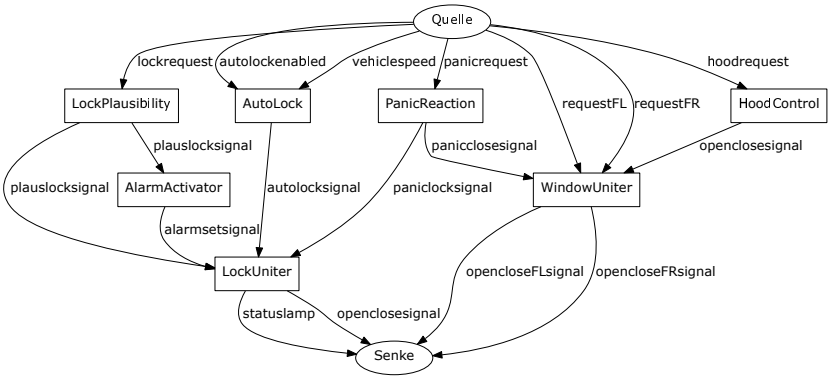


Abbildung 4.4: Beispiel 3: Türsteuergerät für Cabrios

weitere Analyse. Für von Grund auf unterschiedliche Systeme macht eine Produktlinie keinen Sinn.

Die Elemente der Menge *CC* sind die Komponenten, die für den Kern einer Produktlinie geeignet sind. Da diese Komponenten in allen Systemen vorkommen, ist der Wiederverwendungsgrad sehr hoch. Zur Bestimmung eines Lead-Systems für eine Produktlinie ist das System geeignet, welches zum größten Anteil aus dem Kern besteht. Dies ist genau das System, welches das höchste *RSoCC* hat, also in diesem Fall das System 1.

Die Werte, die analog die Datenflüsse betrachten, ergeben die gleiche Aussage, wobei

Tabelle 4.1: gemeinsame Datenflüsse

Name der Startkomponenten	Name der Zielkomponenten	Inhalt
Quelle	LockPlausibility	lockrequest
LockPlausibility	LockUniter	plauslocksignal
LockPlausibility	AlarmActivator	plauslocksignal
AlarmActivator	LockUniter	alarmsetsignal
LockUniter	Senke	statuslamp
LockUniter	Senke	openclosesignal

Tabelle 4.2: Kennzahlen des Beispiels

Kennzahl	alle Systeme	System1	System2	System3
$\|C\|$		4	5	7
$SoCC$	3			
$RSoCC$		75,00%	60,00%	42,86%
$SoIC$		0	0	1
$RSoIC$		0,00%	0,00%	14,29%
$\|D\|$		9	13	18
$SoCD$	6			
$RSoCD$		66,67%	46,15%	33,33%
$SoID$		0	0	2
$RSoID$		0,00%	0,00%	11,11%

diese genauer sind. Gemeinsame Komponenten beschreiben nur eine Ähnlichkeit, wohingegen gemeinsame Datenflüsse auf eine funktionale Gleichheit hindeuten. Somit ist $SoCD$ mit den Datenflüssen CD besser zur Ermittlung des Lead-Systems geeignet.

Bei der Untersuchung der individuellen Anteile stellt sich heraus, dass nur das System 3 eine Komponente hat, welche in keinem anderen System genutzt wird. Somit wird jede Komponente, die für das zweite System entwickelt wird, in weiteren Systemen entwickelt. Für das dritte System muss danach nur eine zusätzlich Komponente entwickelt werden, um alle Systeme in der Produktlinie darstellen zu können.

4.4 Anwendung der Kennzahlen bei Volkswagen

Für eine praktische Anwendung der Kennzahlen bei Volkswagen wurden drei verschiedene Softwarestände der Lenkung untersucht, die jeweils eine der drei bis dahin existen-

ten Generationen von Lenkungensystemen repräsentierten [BRR+10b]. Dabei handelt es sich teilweise um Entwicklungsstände, die noch nicht den vollen Funktionalitätsumfang der Seriensoftware abbilden. Ziel der Anwendung war zum einen zu überprüfen, ob die Kennzahlen die Erfahrung aus der manuellen Untersuchung der Systeme widerspiegeln, und zum anderen zu evaluieren, wie sich die Ähnlichkeitsfunktion für partielle Schnittstellen-Gleichheit (*npie*, Definition (4.6)) mit unterschiedlichen Grad der Ähnlichkeit (Parameter n) an einem realen Beispiel verhält.

Zur Untersuchung wurden alle Matlab/Simulink-Modelle der jeweiligen Software in ihre Subsysteme zerlegt, welche jeweils eine Komponente des Systems darstellen. So ergab sich über alle drei Softwarestände hinweg eine niedrige vierstellige Anzahl an Komponenten. Der Parameter der Funktion *npie* wurde in 10%-Schritten von $0,1$ bis $0,9$ variiert. Die Werte 0 und 1 wurden bewusst herausgelassen, da sich im ersten Fall eine komplette Ähnlichkeit und im zweiten Fall gar keine Ähnlichkeit der Systeme ergeben würde. Zusätzlich wurden die Kennzahlen noch mit der Ähnlichkeitsfunktion für Namensähnlichkeit *ne* berechnet um die Werte mit *npie* vergleichen zu können.

Auf Basis der so erhaltenen Systeme wurden alle Kennzahlen berechnet. Aus Gründen der Geheimhaltung werden hier nur die Kennzahlen für *RSoCC* (Relativer Anteil an ähnlichen Komponenten je System) und *RSoIC* (Relativer Anteil an individuellen Komponenten je System) dargestellt, von denen sich jedoch auch die wesentlichen Aussagen ableiten lassen. Die Ergebnisse sind in Tabelle 4.3 bzw. in Abbildung 4.5 dargestellt.

Tabelle 4.3: Ähnlichkeit bei unterschiedlicher Ähnlichkeitsfunktion

n	$RSoCC_1$	$RSoCC_2$	$RSoCC_3$	$RSoIC_1$	$RSoIC_2$	$RSoIC_3$
0,1	33,33%	25,45%	18,54%	24,50%	36,73%	72,95%
0,2	32,79%	25,03%	18,24%	24,50%	36,86%	73,35%
0,3	32,07%	24,48%	17,84%	25,23%	37,28%	73,65%
0,4	31,17%	23,80%	17,33%	26,31%	37,96%	74,05%
0,5	31,17%	23,80%	17,33%	26,49%	38,51%	74,35%
0,6	29,73%	22,70%	16,53%	29,55%	40,85%	75,15%
0,7	28,47%	21,73%	15,83%	31,89%	42,92%	76,05%
0,8	28,47%	21,73%	15,83%	32,97%	43,88%	76,15%
0,9	28,11%	21,46%	15,63%	33,69%	44,43%	76,35%
ne	45,05%	34,39%	25,05%	15,86%	29,44%	64,73%

Die manuelle Untersuchung der Systeme hat ergeben, dass der Funktionsumfang von System zu System zugenommen hat. Dies spiegelt sich in den abnehmenden Anteilen für gemeinsame Komponenten (RSoCC) und den zunehmenden Anteilen individueller Komponenten (RSoIC) von einem System zum nächsten wider. Der Parameter n der Ähnlichkeitsfunktion $npie(c_1, c_2, n)$ hat auf diese Aussage keine Auswirkung, da sie für alle ermittelten Werte gilt. Besonders hervorzuheben ist, dass die relative Anzahl an

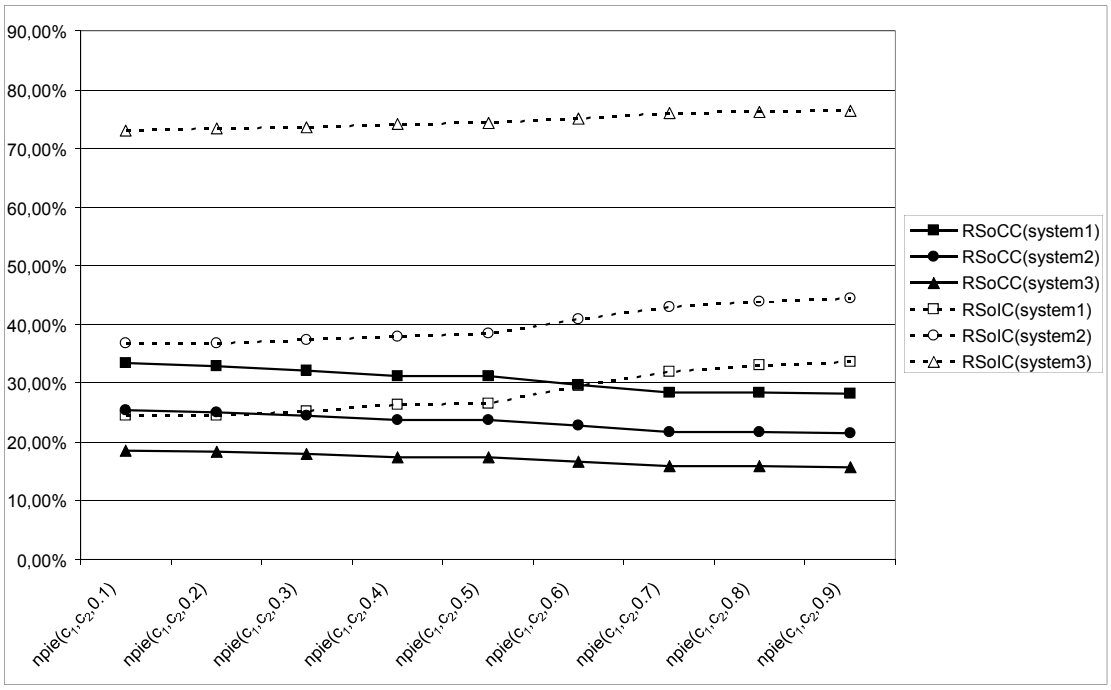


Abbildung 4.5: Ähnlichkeit bei unterschiedlicher Ähnlichkeitsfunktion

individuellen Komponenten für das dritte System besonders hoch ist, was für eine Produktlinie heißt, dass hier der Anteil, der in einer Produktlinie wiederverwendet werden könnte, besonders gering ist. Dies wurde bei der manuellen Analyse nicht in dem Maße festgestellt.

Der Einfluss des Parameters n ist marginal. Die Relation der Kennzahlen der Systeme ändert sich durch den Parameter nicht. Der Schwankungsbereich der Kennzahlen liegt unter 10%. Somit macht es für diese Systeme keinen großen Unterschied, welcher Wert für den Parameter n gewählt wird. Im Vergleich mit der Funktion der Namensähnlichkeit *ne* sind in Tabelle 4.3 größere Abweichungen zu sehen. Die Werte für *RSoCC* liegen nochmal ca. 10% über dem Maximalwert von *npie* bzw. für *RSoIC* ca. 10% unter dem Minimalwert von *npie*. Als Erkenntnis lässt sich aus den Kennzahlen ableiten, dass, auch wenn die Subsysteme gleiche Namen haben, die Funktionalität der Subsysteme sich ändert.

4.5 Weitere Analysemöglichkeiten

In [ZC03] sind eine Reihe möglicher Metriken beschrieben. Neben der Kalkulation der produktspezifischen Kosten werden Produktkomplexität und der ROI (Return on Invest-

ment) als bewertbare Größen genutzt, die für einzelne Systeme ebenso errechnet werden können. Für Produktlinien sind die Entwicklungszeit (Time To Market) und Feature-Abdeckung aussagekräftigere Größen. Meist werden alle quantitativen Größen auf die Kosten zurückgeführt. Das COCOMO II-Modell [BAB+00] ermöglicht eine Abschätzung, jedoch muss dieses um Produktlinien-Methodiken ergänzt werden [NA10].

Die qualitative Betrachtung der Produktlinie kann ebenfalls genutzt werden. [Flo08] stellt beispielsweise vor, wie eine Software damit untersucht werden kann. Dies kann anhand der Qualitätsattribute der ISO 9126 [IEC91] geschehen, die Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Übertragbarkeit beinhalten. In [BCK03] werden für die qualitative Bewertung der Architektur die Qualitätsattribute Verfügbarkeit, Modifizierbarkeit, Performanz, Sicherheit, Testbarkeit und Benutzbarkeit beschrieben. Alle diese Metriken sagen jedoch wenig über die Qualität der Produktlinie an sich aus. Vielmehr ist die möglichst gute Erfüllung dieser Qualitätskriterien eine Basis für eine Produktlinie.

Als Qualitätsattribute für Produktlinien werden in [EM05] Attribute mit Bezug zu Variabilität dargestellt wie Modifizierbarkeit (Änderungen über die Zeit) oder Konfigurierbarkeit (Variation eines Produkt). Hier sind einige aus der Einzelsystementwicklung bekannten Qualitätsattribute vorhanden. Für die Modifizierbarkeit sind die Attribute Erweiterbarkeit, Übertragbarkeit und Skalierbarkeit beispielsweise von Belang. Die Konfigurierbarkeit lässt sich an den Attributen Wiederverwendbarkeit, Kompositionalität und Interoperabilität festmachen.

4.6 Zusammenfassung

Für die Analyse mehrerer Systeme wurde eine Methode gezeigt, wie sich diese mit einfachen Mitteln vergleichen lassen. Systeme werden als Komponenten mit Datenflüssen modelliert und mittels eine Ähnlichkeitsfunktion verglichen. Die Gemeinsamkeiten und Unterschiede, die zwischen den Systemen vorhanden sind, wurden mit Kennzahlen beschrieben. Diese Kennzahlen wurden sowohl für ein fiktives kleines Beispiel als auch im Produktlinien-Einführungsprojekt bei Volkswagen angewendet.

Durch die unterschiedlichen Ähnlichkeitsfunktionen lassen sich bestimmte Spezifika der Systeme herausstellen. Beispielsweise können mit der Funktion *npie* besonders gut verschobene und teilweise bearbeitete Blöcke abgedeckt werden. Blöcke, die umbenannt wurden, lassen sich mit der Funktion *pie* erkennen. Mit einer Kombination von mehreren Ähnlichkeitsfunktionen lassen sich weitere Spezifika berücksichtigen.

Bei der praktischen Anwendung der Kennzahlen lassen sich aus den individuellen Bestandteilen in den Systemen präzisere Aussagen treffen als aus den gemeinsamen Anteil-

len. Hohe individuelle Anteile in Systemen lassen Ausreißer besser erkennen als deren gemeinsame Anteile, die auf einem ähnlichen Niveau liegen. Zusammenfassend stützen die Kennzahlen die Erkenntnisse einer manuellen Analyse.

Damit ist die Maßnahme M1 (Untersuchung der Software) umgesetzt. Die erhaltenen Aussagen können durch weitere Analysen ergänzt werden, die in Abschnitt 4.5 beschrieben sind. Im Rahmen der bei Volkswagen durchgeführten Analyse wurde ein Werkzeug entwickelt, welches die Berechnung der Kennzahlen aus den Matlab/Simulink-Modellen automatisiert. So kann bei Evolution der Softwarestände oder bei neuen Systemen die Berechnung einfach wiederholt werden. Dieses ist eine gute Grundlage bei der Umsetzung der Maßnahme M5 (Automatisierte Erfassung von Gemeinsamkeiten und Unterschieden).

Kapitel 5

Bewertung der Einflussfaktoren für eine Produktlinienentwicklung

Bei der Implementierung einer Produktlinienentwicklung existiert eine Vielzahl von Einflussfaktoren. Die Bedeutung und die Auswirkungen unterscheiden sich je nach Kontext. In großen Industrieprojekten stehen andere Aspekte im Vordergrund als in kleineren Projekten oder Projekten im universitären Umfeld. Aus diesen unterschiedlichen Schwerpunkten ergeben sich unterschiedliche Ansatzpunkte für die Umsetzung einer Produktlinienentwicklung. Im Folgenden werden die Einflussfaktoren beschrieben und anhand der vorgefundenen Projektlandschaft analysiert. Damit wird die Implementierung der Maßnahme M2 (Bewertung der Einflussfaktoren) durchgeführt.

Die hier aufgeführten Einflussfaktoren wurden anhand der im Rahmen bei Volkswagen durchgeführten Arbeiten zusammengestellt. Sie sind in Diskussionen mit den Mitarbeitern bei Volkswagen und mit den Erfahrungen bei der Umsetzung des Produktlinien-Einführungsprojekts zusammengestellt worden. Für ähnliche Erhebungen in anderen Industrieprojekten lässt sich diese Aufstellung als Basis nutzen. Da in anderen Projekten andere Einflussfaktoren relevant sein können, wird jeder Einflussfaktor sowohl allgemein (in den Kästen) als auch im Rahmen des Produktlinien-Einführungsprojekts (Fließtext) beschrieben. Wichtig ist, am Anfang alle Einflussfaktoren zu kennen und auch zu evaluieren, um später nicht bei der Umsetzung in eine Sackgasse zu laufen.

Die Einflussfaktoren sind in mehrere Kategorien eingeteilt. In Abbildung 5.1 ist die Kategorisierung dargestellt. Zunächst werden quantitative Einflussfaktoren behandelt (Abschnitt 5.1). Anschließend werden in Abschnitt 5.2 qualitative Einflussfaktoren dargestellt. Wie sich durch Unternehmensziele weitere Faktoren ergeben, wird darauffolgend beschrieben (Abschnitt 5.3). Zuletzt spielt die Variabilität als Einflussfaktor eine wichtige Rolle (Abschnitt 5.4).

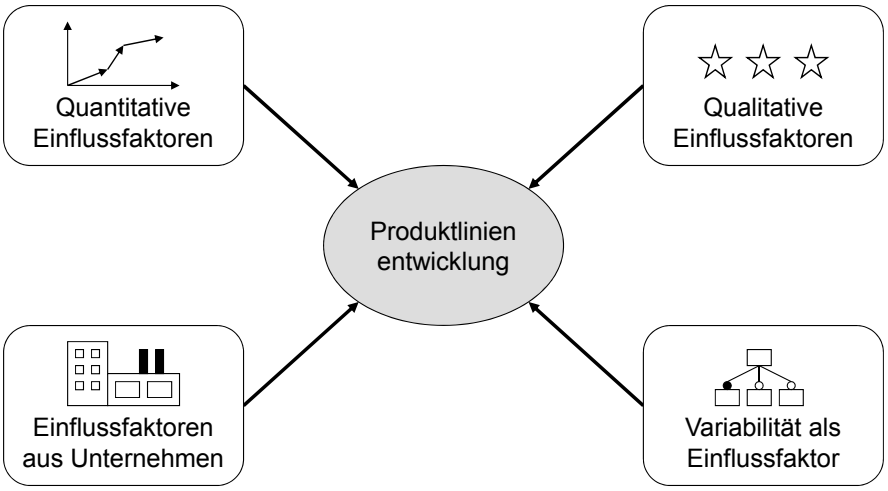


Abbildung 5.1: Einflussfaktoren bei Produktlinien

5.1 Quantitative Einflussfaktoren

Die quantitativen Einflussfaktoren sind in Abbildung 5.2 dargestellt. Zu diesen zählen Personen und Material, Kosten, Zeit, Projektgruppengröße und Komplexität.

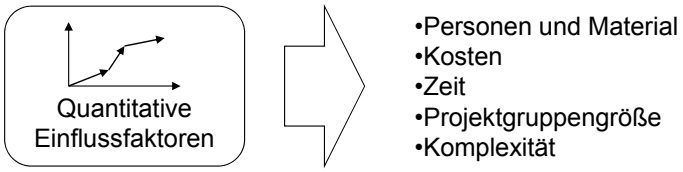


Abbildung 5.2: Quantitative Einflussfaktoren

5.1.1 Personen und Material

Die Mitarbeiter und die Materialressourcen spielen eine wichtige Rolle. Beide Ressourcen stehen dem Projekt nicht unbegrenzt und oft auch nicht exklusiv zur Verfügung. Platz für Mitarbeiter, genügend Besprechungsräume, selbst die Versorgung mit grundlegenden Dingen wie Toiletten beeinflussen direkt oder indirekt die verfügbare Anzahl und Qualität der Ressourcen. Personen und Material können nur selten kurzfristig bereitgestellt werden. Gerade bei neuen Mitarbeitern ist immer eine gewisse Einarbeitungszeit notwendig, damit sie ein Projekt in vollem Umfang unterstützen können.

Im betrachteten Kontext bei Volkswagen ist das Projekt in zwei Abteilungen organisiert, wobei eine für den mechanischen Anteil der Lenkung verantwortlich ist und eine

für den Elektronikteil. Die betrachtete Elektronikentwicklung ist in drei Unterabteilungen organisiert. Hardware wird in der ersten Unterabteilung entwickelt. In der zweiten Unterabteilung werden die Funktionalitäten des Lenksystems modelliert. Die dritte Unterabteilung erstellt die Software, die die Funktionalitäten des Lenksystems vereint, und diese um zusätzliche Sicherheitsfunktionen erweitert. Neben den Angestellten gibt es Dienstleister, die die Angestellten unterstützen.

Alle Mitarbeiter, Angestellte sowie Dienstleister sind mit den notwendigen Arbeitsmitteln ausgestattet. Durch das stetige Wachstum der Abteilung verringert sich der zur Verfügung stehende Platz stetig, so dass durch mehrere Maßnahmen neuer Platz geschaffen werden musste. Langfristig wird der Bedarf an Personen und Material steigen, da neue Entwicklungsprojekte aufgesetzt werden und die bisherigen Projekte im Rahmen einer Serienbetreuung weiterlaufen.

5.1.2 Kosten

Ein nicht unwesentlicher Faktor für ein Projekt sind die Kosten. Die Kosten der Entwicklung sollen durch den Verkauf der entstehenden Produkte abgedeckt werden. Zusätzlich muss es einen Gewinn geben, damit die investierte Arbeit rentabel ist.

Das zu erreichende Renditeziel wird pro Produkteinheit, in diesem Fall ein Lenksystem, vorgegeben. Anschließend wird mit dem Vertrieb abgestimmt, welche Laufzeit ein Produkt hat und welcher Verkaufspreis angestrebt wird. Daraus ergeben sich die erlaubten Entwicklungs- und Produktionskosten. Eine Produktlinienstrategie muss deshalb berücksichtigen, dass die erlaubten Entwicklungskosten nicht überschritten werden dürfen. Maßnahmen müssen deshalb über mehrere Projekte realisiert werden oder durch den Produktlinieneinsatz Kosten an anderer Stelle eingespart werden.

Die Notwendigkeit von Investitionen zur Optimierung des Softwareentwicklungsprozesses wurde von der Abteilung erkannt. Kurzfristige Optimierungen lassen sich durch die direkte Kostenreduzierung im Projekt problemlos umsetzen. Änderungen, die nicht unmittelbar zu einer Kostenreduzierung führen, können nur Schritt für Schritt umgesetzt werden, wenn eine ausreichende positive Kostenprognose vorhanden ist.

5.1.3 Zeit

Zeit ist ein nicht zu unterschätzender Faktor bei der Entwicklung. Häufig gibt es eng geplante Terminpläne. Puffer für zusätzliche Arbeiten gibt es meistens wenig, so dass der zusätzliche Zeitbedarf eines oder mehrerer Arbeitspakete eine Verzögerung im Gesamtterminplan hervorruft.

Diese Terminpläne sind bei einem neuen Fahrzeug schon lange im Voraus festgelegt. Zusätzliche Arbeitspakete, die durch die Einführung von Produktlinientechniken resultieren, lassen sich in einem bestehenden Terminplan selten ohne eine Verschiebung von wichtigen Meilensteinen realisieren.

Da es laufende Projekte gibt und die Einführung einer Software-Produktlinienentwicklung im Rahmen der Forschung geschieht, ist der Zeitfaktor in diesem Fall nicht so entscheidend. Für den operativen Einsatz von Forschungsergebnissen müssen diese jedoch zu bestimmten Meilensteinen vorliegen oder umgesetzt sein, damit diese in die laufende Entwicklung integriert werden können.

5.1.4 Projektgruppengröße

Um ein Projekt mit einer gewissen Qualität in einem gewissen Zeitrahmen zum Erfolg zu führen, ist eine gewisse Anzahl an Projektmitarbeitern notwendig. Je mehr Mitarbeiter an einem Projekt beteiligt sind, desto größer wird die Anzahl der benötigten weiteren Ressourcen. Das Wissen ist auf alle Mitarbeiter verteilt. Eine kleine Projektgröße bringt den Vorteil, dass für einen Produktlinien-Ansatz weniger Leute überzeugt werden müssen. Die Projektgröße wirkt sich auch darauf aus, welche Entwicklungsmethodik (z.B. das V-Modell [VMod]) idealerweise angewendet wird oder angebracht ist.

In den Lenkungsprojekten waren am Anfang der Untersuchung auf Elektronik-Seite circa 80 Entwickler tätig, deren Zahl sich im Verlauf der Zeit stetig erhöht hat. Als Entwicklungsmethodik kam das V-Modell zum Einsatz, welches für solche Projektgrößen optimalerweise eingesetzt wird. Durch die Aufteilung in Fachfraktionen waren die Informationen gut verteilt.

Durch die Erhöhung der Mitarbeiterzahl war es notwendig die Informationen besser zu dokumentieren. Dies betraf vor allem Auffälligkeiten im Entwicklungsprozess, wofür Offene-Punkte-Listen (OPLs) geführt wurden. Ein weiterer Kompensationsmechanismus für die steigende Mitarbeiterzahl war die Schaffung neuer Abteilungen und Unterabteilungen zur Konsolidierung wichtiger Informationen.

5.1.5 Komplexität

Die zu entwickelnden Produkte sollen immer mehr Anforderungen gerecht werden und neue Funktionen realisieren. Dadurch bestehen diese aus immer mehr Artefakten und die Komplexität steigt stetig an. Komplexität kann für den Entwickler verringert werden, indem Teile vereinfacht werden oder die Komplexität durch Entwicklungs-

werkzeuge kompensiert wird.

In [STB+04] wird die Entwicklung der Komplexität von Motorsteuergeräten innerhalb des Zeitraums von 1995 bis 2005 gezeigt. Das größte Wachstum ist bei der Lines of Code (LOC) der Software zu betrachten, die um das 15-fache gestiegen ist. Um das System den gegebenen Einflüssen anzupassen, hat sich die Anzahl der Parameter in der gleichen Zeit vervierfacht.

Da heutige Steuergeräte komplexe Funktionen realisieren, schlägt sich diese Komplexität in der Entwicklung der Lenkungssteuergerätesoftware nieder. Höchste Anforderungen an das haptische Verhalten sowie an die Sicherheit des Gesamtsystems resultieren in zusätzlichen Artefakten. Zur Anpassung von einer Vielzahl von Einstellungsmöglichkeiten kommen Parameter zum Einsatz, die den Quellcode handhabbarer machen.

Die Vereinfachung von Funktionalität ist selten möglich, da die bestehenden Artefakte schon sehr effizient umgesetzt werden. Es werden Standard-Werkzeuge in der Entwicklung genutzt, die bespielsweise die Komplexität eines CAN-Netzwerkes kapseln (z.B. mit CANoe [CANoe]). Positiv wirkt sich auf die Komplexität der Entwicklung aus, dass die Entwicklung auf einen Standort konzentriert ist.

5.2 Qualitative Einflussfaktoren

Zu den qualitativen Einflussfaktoren zählen die in Abbildung 5.3 dargestellten Faktoren. Es handelt sich um die Kundenbezogenheit, den Fortschritt im Produktlebenszyklus, die Innovationskultur, die einzuhaltenden Regularien, den bisherigen Einsatz von Modellen und die technische Infrastruktur.

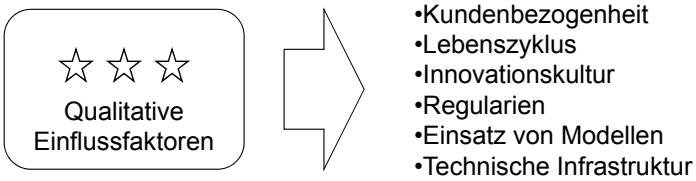


Abbildung 5.3: Qualitative Einflussfaktoren

5.2.1 Kundenbezogenheit

Wird ein Produkt für einen großen Kundenkreis entwickelt, werden nur Anforderungen erfüllt, die von einem möglichst großen Kundenkreis gefordert werden. Der gegensätz-

liche Fall ist Erfüllung der Anforderungen von einem bestimmten Kunden (Auftragsfertigung), bei dem jedes Produkt auf spezielle Kundenwünsche zugeschnitten ist. Für Software-Produktlinien müssen auch nicht-funktionale Anforderungen berücksichtigt werden [Sdd15].

Es gibt für das vorliegende Projekt nur wenige Kunden. Diese sind die Marken der Volkswagen AG, die eine Lenkung in ein Fahrzeug einbauen und zudem eine hohe Stückzahl des jeweiligen Produktes anfragen. Es handelt sich somit bei den Produkten um eine Auftragsfertigung. Folglich stehen für die Entwicklung die Kundenanforderungen im Vordergrund. Eigene Optimierungen, die nicht einem Kundenwunsch entsprechen, werden nur umgesetzt, wenn sie zur Umsetzung von Anforderungen notwendig sind.

5.2.2 Lebenszyklus

Im Produktlebenszyklus wird ein Produkt zunächst entwickelt, anschließend vertrieben und zuletzt vom Markt genommen. Je später in dem Zyklus eine Produktlinie eingeführt wird, desto unrentabler wird dies. Eine Produktlinie lohnt sich nur in der Entwicklungsphase eines Produktes. In der Vertriebsphase wird ein Produkt nur noch weiterentwickelt und Umstrukturierungen für Produktlinien lohnen sich nicht.

Die Projekte der Lenkungsentwicklung haben lange Entwicklungs- und Vertriebsphasen. Auch während ein Produkt schon auf dem Markt ist, gibt es stetige Optimierungen, um Kosten zu sparen oder auf geänderte Umweltbedingungen zu reagieren. Insofern gibt es immer wieder Weiterentwicklungen.

Da für die Weiterentwicklungen wenige Ressourcen zur Verfügung stehen, ist es nicht möglich dabei noch einen Umstieg zu einer Produktlinie abzudecken. Auch die weiteren Projekte nehmen immer die bisherigen Projekte als Basis. Insofern ist eine Produktlinienentwicklung vor allem umsetzbar, wenn eine Lenkung von Grund auf oder ein neues Produkt entwickelt wird.

5.2.3 Innovationskultur

Innovationen entstehen vor allem dadurch, dass durch Mitarbeiter Verbesserungen für die Realisierung und Fertigung eines Produkts identifiziert werden. Mitarbeiter mit einem hohen Interesse an Innovationen sind auch gewillt diese umzusetzen. Mit einer guten Innovationskultur kann sich ein Unternehmen stetig weiterentwickeln und die Implementierung von neuen Techniken wie die der Software-Produktlinien gelingt einfacher.

Die Innovationen in der Lenkungsentwicklung liegen vor allem im Produkt, jedoch weniger im Entwicklungsprozess. Dennoch ist bei den Mitarbeitern der Wunsch, den Entwicklungsprozess zu optimieren. Dazu gibt es an vielen Stellen auch Vorschläge. Die Umsetzung der Innovationen kann sich jedoch verzögern, da neue Technologien erst bewertet und die Risiken abgeschätzt werden müssen, was gerade im Bereich von sicherheitskritischen Systemen notwendig ist.

Eine anschließende Realisierung der Innovation benötigt ebenfalls Zeit. Detailverbesserungen, die mit geringem Aufwand umgesetzt werden können und nur geringe oder keine Auswirkungen auf Anforderungen haben, werden im Entwicklungsprozess relativ einfach aufgenommen. Bei größeren Änderungen, die an mehreren Stellen greifen, ist das damit verbundene Risiko schwerer abschätzbar und die Umsetzungswahrscheinlichkeit sinkt.

5.2.4 Regularien

In der Industrie spielen einige Normen und Richtlinien eine Rolle. Diese werden von unabhängigen Verbänden und Gremien aufgestellt, aber auch durch gesetzliche Vorgaben und durch interne Regelungen ergänzt. Im Entwicklungprozess müssen diese auch bei der Einführung von Produktlinien berücksichtigt werden.

Die bei in der Lenkungsentwicklung bei Volkswagen relevanten Regularien sind in Tabelle 5.1 (interne Regularien) und Tabelle 5.2 (externe Regularien) festgehalten. Eine wesentliche Norm, die für alle elektronischen Systeme relevant ist, ist die DIN EN 61508 [IEC10]. Diese legt Eigenschaften und Anforderungen von an elektronische Komponenten bezüglich ihrer funktionalen Sicherheit fest. Sie soll in Zukunft von einer für das Automotive-Umfeld zugeschnittene Norm ISO 26262 [ISO09] abgelöst werden. Für die entstehende Software müssen weitere Anforderungen, wie die Misra-Konformität [Mot04] von C-Code oder die Modellierungsrichtlinien in Matlab/Simulink-Modellen, eingehalten werden.

Tabelle 5.1: Übersicht über die erfüllten internen Richtlinien

Richtlinie	Beschreibung
interne Entwicklungsrichtlinien	Im Volkswagen Konzern wird eine allgemeine Richtlinie zur Verfügung gestellt, die Vorgaben hinsichtlich der Entwicklungsmethodik und der zu verwendenden Werkzeuge macht.
interne Modellierungsrichtlinien	Um ein allgemeines Verständnis innerhalb des Volkswagen Konzern zu schaffen, gibt es Vorgaben, wie Modelle in Matlab/Simulink modelliert werden.

Tabelle 5.2: Übersicht über die erfüllten Normen

Norm	Beschreibung
DIN EN 61508	*Funktionale Sicherheit sicherheitsbezogener elektrischer, elektronischer und programmierbarer elektronischer Systeme*: Die Anwendung dieser Norm ist nicht nur auf Automotive-Systeme beschränkt, sondern gilt auch für alle elektronischen Systeme, die beispielsweise auch in Atomkraftwerken zum Einsatz kommen. Die Sicherheitsanforderungen in den Projekten sind sehr hoch, da die Lenkung ein Bauteil mit sehr hoher Kritikalität ist.
ISO 26262	*Road vehicles – Functional safety*: Nachfolgenorm zur 61508, die besonders auf die Bedürfnisse der Automotive-Welt zugeschnitten ist. Viele Anforderungen aus der DIN EN 61508 wurden hier für Automotive-Systeme konkretisiert.
IEC 15504	*Automotive SPICE (Software Process Improvement and Capability Determination)*: Die domänenspezifische Anpassung an den Automotive-Sektor der allgemeinen SPICE-Norm beschreibt ein Reifegradmodell für die Prozesse, die bei der Entwicklung für Automotive-Steuergeräte zum Einsatz kommen.
Misra-C	*Motor Industry Software Reliability Association*: Internationale Richtlinien für Kodierungsrichtlinien von C-Code. Nicht alle in der Programmiersprache C möglichen Konstrukte dürfen im Automotive-Sektor eingesetzt werden, um den Quelltext verständlich und robuster gegenüber Fehlern zu machen. Die Regeln tragen auch dazu bei, dass Fehler früh erkannt und behoben werden können.

Die Entwicklungsprozesse sind nach dem Automotive SPICE-Reifegradmodell [IEC06] Level 3 assessiert. Das SPICE-Reifegradmodell beschreibt mit dem Reuse Program Management (REU.2) einen Prozess zur Wiederverwendung. Dieser liegt nicht im HIS-Scope und wird deshalb eher selten explizit umgesetzt und assessiert. Der Prozess beschreibt grundsätzliche Aktivitäten bei der Wiederverwendung, enthält jedoch keine Methoden oder konkreten Vorgehensweisen. Die vorliegende Entwicklung berücksichtigt neben den internationalen Normen auch unternehmensinterne Vorgaben.

5.2.5 Einsatz von Modellen

Modelle spielen heutzutage in der Softwareentwicklung eine Schlüsselrolle. Ein optimaler Entwicklungsprozess ist ohne Modelle kaum mehr möglich. Um Software-

Produktlinien umzusetzen, ist es notwendig, dass es einen möglichst großen bestehenden Modellanteil gibt und dieser für die Software-Produktlinienentwicklung vorbereitet ist.

Bei der Entwicklung von eingebetteten Systemen ist eine Reihe von Spezialisten beteiligt, die für ihr eigenes Gebiet die für sie bekannten Modelle nutzen. Das fängt bei Modellen für das Leiterplattenlayout an, führt weiter über VHDL zur Modellierung von Logik-Bausteinen und endet schließlich bei Matlab/Simulink-Modellen für eine Applikation, die auf einem Mikroprozessor läuft. Aus diesen Modellen werden die jeweils passenden Folgeartefakte generiert. Für eine Automatisierung ist es wichtig, dass nachfolgende Prozesse auf Basis dieser Modelle arbeiten können und insbesondere keine Transformation der Informationen von Hand stattfinden muss.

Für Produktlinien gibt es viele theoretische Ansätze, wie Modelle in einer Entwicklung eingesetzt werden können. Die Umsetzung in der Praxis ist meist nicht ohne Modifikation möglich. Dies liegt zum Beispiel daran, dass Mitarbeiter nicht für eine modellbasierte Entwicklung oder eine automatisierte Nutzung von Modellen sensibilisiert sind.

Im Rahmen eines Produktlinienansatzes wurden insbesondere die Modelle für die Applikation, also die Lenkungssteuergerätesoftware, untersucht. Für die Entwicklung der einzelnen Funktionen werden Matlab/Simulink-Modelle eingesetzt. Auf Basis dieser Modelle wird dann Quellcode generiert, der dann als Modul in die Applikation eingebunden wird. Somit besteht eine Basis für die modellbasierte Softwareentwicklung. Das Einbinden erfolgt jedoch über manuelle Programmierung und auf strukturierte Weise, so dass es einen potentiellen Ansatzpunkt für weitere Modelle gibt.

Zur Dokumentation werden UML-Modelle verwendet, die für die Beschreibung der Architektur genutzt werden. Aus diesen werden jedoch nicht weitere Teile der Architektur generiert und so sind sie nur grafische Darstellungen, die händisch in andere Artefakte übersetzt werden. Es werden an vielen Stellen schon explizit oder implizit Modelle verwendet, ohne alle Möglichkeiten von ihnen zu nutzen, was bei einer Software-Produktlinienentwicklung immens wichtig ist.

5.2.6 Technische Infrastruktur.

Die technische Infrastruktur für eine Produktlinie trägt maßgeblich zu ihrem Erfolg bei. Für eine Produktlinienentwicklung reicht eine einfache Versionsverwaltung nicht aus, sondern es muss ein umfangreiches Konfigurationsmanagement eingesetzt werden. Wenn mit Partnern zusammengearbeitet wird, müssen die Systeme auf beiden Seiten gekoppelt werden. Häufig sind verschiedene Systeme im Einsatz, was Erstellung und Anpassung von Systemschnittstellen und Prozessen erfordert [MO07].

Die technische Infrastruktur ist vorgegeben, soweit es das Anforderungsmanagement und das Konfigurationsmanagement. Es besteht somit die Herausforderung, über geeignete Techniken Variabilität abzubilden. Die Nutzung von werkzeugspezifischen Formaten für die Speicherung von Informationen macht es schwer, bestehende Daten in andere Werkzeuge mit besserer Produktlinienunterstützung zu migrieren oder weiterzuverwenden. Für eine mögliche Automatisierung ist ein offenes oder zumindest gut dokumentiertes und breit unterstütztes Format besser geeignet.

Die Werkzeuge, die Variabilität unterstützen müssen, sind im konkreten Beispiel Rational DOORS [Dor] zur Verwaltung von Anforderungen und Rational Synergy CM und Rational Change [Syn] für das Konfigurations- und Änderungsmanagement. Die technische Infrastruktur ist grob in Abbildung 5.4 skizziert. DOORS unterstützt für Dokumente und Anforderungen nur einen linearen Versionsstrang, bei dem Verzweigungen nicht konsistent umgesetzt werden können. Variabilität ist somit nur durch Sichten oder Kopien von Dokumenten ohne Verknüpfung zum Original möglich. Die Möglichkeiten von Synergy sind vielfältig, benötigen jedoch Einarbeitungszeit und praktische Erfahrung, um alle Funktionen in vollem Umfang einsetzen zu können.

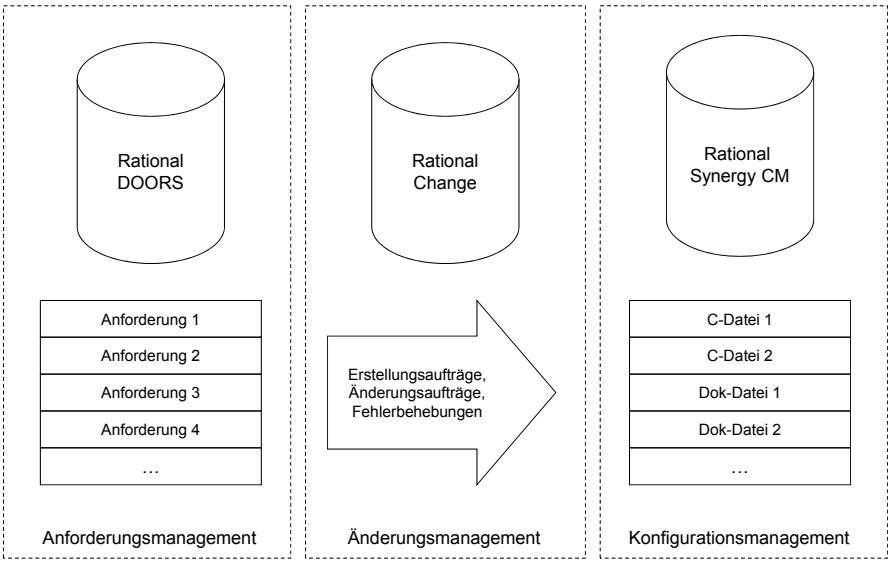


Abbildung 5.4: Technische Infrastruktur in den untersuchten Projekten

5.3 Unternehmerische Einflussfaktoren

Die Einflüsse, die durch das Unternehmen bestimmt werden, sind in Abbildung 5.5 dargestellt. Zu diesen zählen die Unternehmensziele, die Wettbewerbspositionierung und das Betriebsklima.

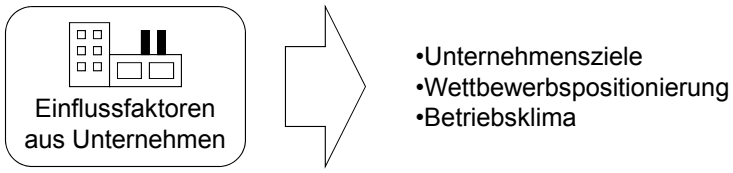


Abbildung 5.5: Unternehmerische Einflussfaktoren

5.3.1 Unternehmensziele

Jedes Unternehmen hat bestimmte Unternehmensziele. Solche Ziele können Auswirkungen auf die Implementierung einer Produktlinie haben. Ein Ziel kann sein, möglichst innovative und neue Wege zu begehen („Think different“ von Apple) oder eine gewisse Eigenschaft („Schmilzt im Mund, nicht in der Hand“ von M&Ms) zu gewährleisten. In der Regel werden mehrere Ziele gleichzeitig verfolgt. Nach [BG14] ist eine Unterstützung einer Produktlinie durch die Unternehmensführung für die erfolgreiche Nutzung notwendig.

Es gibt explizite und implizite Unternehmensziele. Erstere werden beispielsweise in Form eines Slogans nach außen kommunizieren um sich von Mitbewerbern abzugrenzen. Implizite Unternehmensziele sind meist nicht dokumentiert, sondern stellen eher eine Übereinkunft von führenden Personen eines Unternehmens dar. Durch die nicht vorhandene Dokumentation können solche Ziele schlecht in eine Anforderung oder später in eine Architektur gefasst werden und werden daher meistens nicht berücksichtigt [CMB10].

Die Unternehmensziele für Volkswagen sind in der Mach18-Strategie festgehalten [Vol08]. Ein Ziel ist durch verbesserte Prozesse entlang der gesamten Wertschöpfungskette eine höhere Produktivität zu erreichen. Dafür werden durch mehrere Maßnahmen Investitionen bereit gestellt. Die Optimierung des Entwicklungsprozesses unter Produktlinien-Gesichtspunkten unterstützt das Gesamtziel.

5.3.2 Wettbewerbspositionierung

Ein Unternehmen hat immer eine gewissen Marktposition. [MS10] stellt mögliche Wettbewerbsstrategien für Produktlinien dar. Diese werden auf zwei Dimensionen abgebildet: Wettbewerbsvorteil und Wettbewerbsbereich. Die vier möglichen Ausrichtungen haben auf den möglichen Einsatz einer Produktlinie unterschiedliche Auswirkungen.

In Abbildung 5.6 werden die Ausrichtungen dargestellt. Bei der Kostenführerschaft werden im Wesentlichen Kosten reduziert. Da die eigenen Kosten möglichst gering gehalten

werden sollen, werden bestehende Prozesse effizienter gemacht oder das Outsourcing optimiert. Die Anzahl der möglichen Produkte wird reduziert, so dass ein Produktlinien-Vorgehen sich nicht rechnet.

Bei einer Kostenfokus- oder einer Abgrenzungsfokus-Strategie ist die Menge an möglichen Produkten zu gering, als dass sich eine Umsetzung von Produktlinien lohnen würde. Der Aufwand, der für eine Produktlinie notwendig ist, wird nur durch wenige Kunden getragen, die entweder auf geringe Kosten fixiert sind oder auf einen möglichst großen individuellen Anteil, was eine mögliche Wiederverwendung minimiert. In diesem Fall sind andere Vorgehensweisen besser geeignet, um die Unternehmensziele zu erreichen.

Die bestmöglichen Voraussetzungen für eine Produktlinienstrategie ergeben sich bei einer Abgrenzungs-Strategie für den Massenmarkt. Viele Kunden helfen die initialen Kosten einer Produktlinie besser abzufedern. Eine hohe Produktvielfalt und viele Kunden machen die Wiederverwendung erstrebenswert.

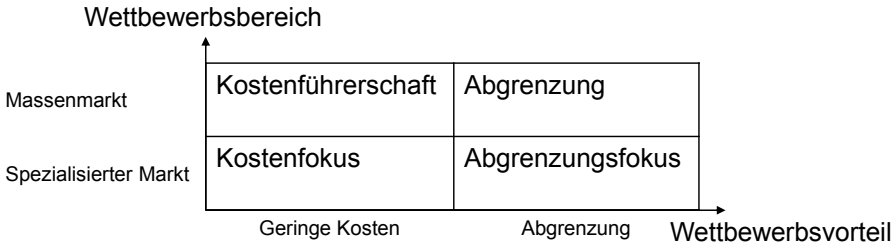


Abbildung 5.6: Mögliche Wettbewerbsstrategien

Eine starre Einordnung einer Unternehmung in die vier skizzierten Fälle ist nicht immer sinnvoll. Es gibt Zwischenstufen, die abhängig von weiteren Nebenbedingungen sind, die Perspektiven für eine Produktlinienstrategie verbessern oder verschlechtern. In [MO07] zeigt ein Erfahrungsbericht, dass die richtige Balance zwischen Gemeinsamkeiten und Spezialisierungen gegeben sein muss. Generell lässt sich sagen, dass je größer der Kundenkreis ist und je größer die Abgrenzung (also das Produktportfolio) ist, desto besser Produktlinien geeignet sind.

Eine Position, die weit von dieser optimalen Zusammensetzung entfernt liegt, schließt den effizienten Einsatz einzelner Produktlinien-Techniken nicht aus. Wenn eine Optimierung des Entwicklungsprozesses notwendig oder gefordert ist, können bei der Optimierung Variabilität und Features für zukünftige Entwicklungen berücksichtigt werden. Außerdem können sich Wettbewerbspositionierungen verschieben und so existierende Produkte eine neue Einordnung erfahren. Dann können durch die bereits erfolgte Investition in die Zukunft Einsparungen erreicht werden.

Die elektromechanische Lenkung hat einen Kostenfokus auf einem sehr spezialisierten Markt. Wenige Projekte, die kostenoptimierte Lenkungen entwickeln, stellen kei-

ne optimalen Voraussetzungen für den Einsatz einer Produktlinie dar. Bei Betrachtung der Software lassen sich jedoch Potentiale für eine Softwareproduktlinie identifizieren. Das Teilprodukt Lenkungssteuergerätesoftware kann sehr wohl im Rahmen einer Kostenführerschaft-Strategie genutzt werden. Wenn die Software hinreichend vom mechanischen und elektronischen System abstrahiert ist, könnte diese theoretisch in anderen Lenkungen eingesetzt werden, die nicht bei Volkswagen entwickelt werden.

5.3.3 Betriebsklima

Eine Produktlinienentwicklung muss von den Mitarbeitern getragen werden. Das Verhältnis zwischen Führungskräften und Mitarbeitern spielt für den Erfolg einer Produktlinie eine besondere Rolle. Mitarbeiter verhalten sich kooperativer, wenn sie in Entscheidungen involviert werden, anstatt dass diese von oben diktiert werden. Ein schlechtes Betriebsklima verringert die Chancen, dass grundsätzliche Entscheidungen weitergetragen werden.

Die Lenkungsentwicklung findet in einem guten Betriebsklima statt. Der Umgang der Mitarbeiter untereinander ist freundlich und kurze Wege beschleunigen kritische Entscheidungen. Wie in jeder größeren Gruppe kann es sein, dass nicht jeder mit jedem gut zusammenarbeiten kann. Letztendlich entsteht ein großer Zusammenhalt dadurch, dass innovative Ziele verfolgt werden und jeder dazu seinen Beitrag leisten will.

5.4 Variabilität als Einflussfaktor

Für die Untersuchung der Variabilität müssen mehrere Einflussfaktoren berücksichtigt werden (Abbildung 5.7). Variabilität verteilt sich über mehrere Ebenen, in denen diese zu unterschiedlichen Zeitpunkten gebunden wird. Die Realisierung der Variabilität ist nicht auf den Ort beschränkt, wo sie auftritt, sondern sie muss auch an anderen Stellen umgesetzt werden. Zuletzt spielen Änderungen und Evolution eine große Rolle bei der Variabilitätsbetrachtung.

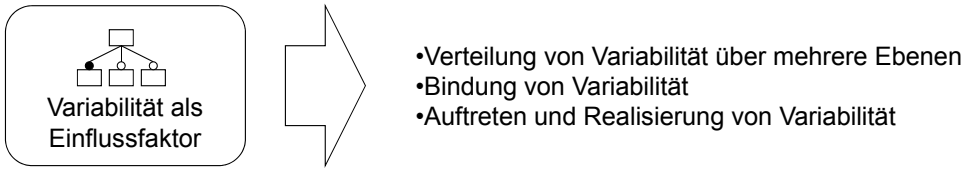


Abbildung 5.7: Variabilität als Einflussfaktor

5.4.1 Verteilung von Variabilität über mehrere Ebenen

Es gibt mehrere Möglichkeiten, wie Variabilität in einer Produktfamilie verteilt sein kann. In einem Gesamtsystem kann nicht nur die Software variabel sein, sondern auch die von ihr gesteuerte Mechanik. Durch Kombination der Variabilität entstehen viele mögliche Varianten.

Ein Beispiel dafür ist auszugsweise in Abbildung 5.8 dargestellt. Das Beispiel orientiert sich an den Gegebenheiten, die in dem untersuchten Projektumfeld vorgefunden wurden. Je nach Systemkomplexität können mehr oder weniger Ebenen existieren, die über unterschiedliche Mechaniken Variabilität aufweisen.

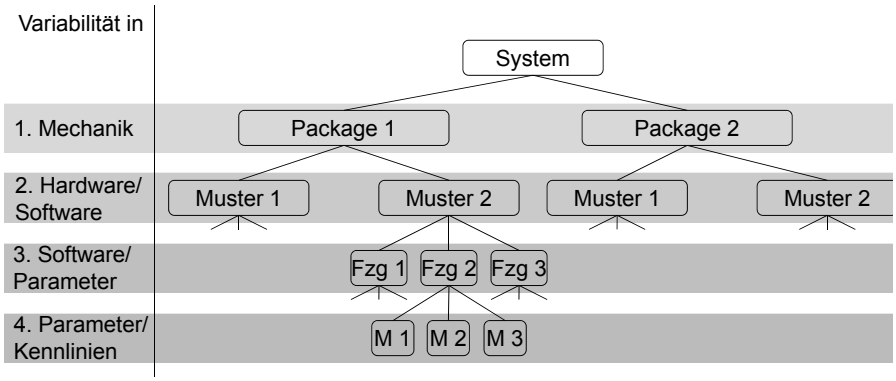


Abbildung 5.8: Verteilung von Variabilität auf mehrere Ebenen

Das System teilt sich auf oberster Ebene anhand der Mechanik-Variabilität auf. Diese kann sich beispielsweise in unterschiedlichen mechanischen Konstruktionen (Packages) für ein System ausprägen, abhängig von dem zur Verfügung stehenden Bauraum in unterschiedlichen Fahrzeugklassen und der benötigten Unterstützungsleistung. Es gibt bei den meisten Automobilherstellern Baukastensysteme, die diesen Bauraum über verschiedene Fahrzeuge hinweg festlegen. Im Falle von Volkswagen gibt es als Beispiel den Modularen Querbaukasten (MQB) [Han12].

Auf einer weiteren Ebene werden bei der Entwicklung eines Packages in der Automobilindustrie mehrere Musterphasen durchlaufen, die jeweils den Entwicklungsstand wiedergeben. Die Änderung des Musters hat Auswirkungen auf Hardware sowie Software eines Systems. Neue Sensoren als Hardware liefern zusätzliche Informationen, die in der Software berücksichtigt werden müssen, damit die neuen Sensoren sinnvoll eingesetzt werden.

In der dritten Ebene bedient jedes Muster bestimmte Fahrzeuge. Abhängig vom Fahrzeug muss der Umfang der Software angepasst werden, da einige Fahrzeuge bestimmte Funktionen nicht unterstützen oder zusätzliche Funktionen beinhalten. Grundsätzliche Fahr-

zeugdaten haben Auswirkungen auf Berechnungen im Steuergerät, wenn beispielsweise Querbeschleunigungen bei einem bestimmten Lenkeinschlag errechnet werden. Diese Variabilität wird durch Parameter erreicht, die dann fahrzeugspezifisch appliziert werden.

Innerhalb eines fahrzeugspezifischen Softwarestandes gibt es noch weitere Variabilität auf einer vierten Ebene. So wirken sich das Gewicht und der Schwerpunkt eines Fahrzeugs auf die Berechnung aus. Das Gewicht hängt von der gewählten Ausstattung ab, wobei der Motor einen wesentlichen Teil beiträgt. Damit sich das Fahrzeug bei unterschiedlichen Gewichten identisch verhält, werden jeweils Kennlinien verwendet, die haptisch ein gleiches Lenkgefühl unabhängig vom gewählten Motor hervorrufen. Zusätzlich gibt es eine Reihe von Parametern und Kennlinien, die zur Abstimmung des Systems dienen. So kann durch gezielte Änderungen ein optimales Gesamtsystem entwickelt werden, ohne für jede kleinste Änderung ein neues Kompilat zu erstellen.

Diese vier Ebenen der Variabilität erlauben viel Variation im Gesamtsystem. Für eine Software-Produktlinienentwicklung ist es wichtig die Ebenen zu kennen, da gerade durch das Zusammenwirken der Ebenen sich Abhängigkeiten ergeben.

5.4.2 Bindung von Variabilität

Aus den vorgestellten Variabilitätsebenen resultieren unterschiedliche Bindezeiten. Während der mögliche Bauraum schon sehr früh im Entwicklungsprozess definiert ist, steht der Umfang der Software erst später fest. Zur Kompilierzeit der Software ist die Auswahl der enthaltenen Funktionen festgelegt, jedoch nicht die Parameter für die einzelnen Funktionen. Diese können später zur Laufzeit gesetzt werden, um das Verhalten des Systems gezielt zu ändern.

Bei früher Bindung der Variabilität wird die Anzahl der möglichen Systeme in der Produktlinie eingeschränkt. Dies hat den Vorteil, dass weniger Systeme kundenspezifisch angepasst und getestet werden müssen. Bei einer späten Bindung der Variabilität können viele Endprodukte abgedeckt werden. Jedoch ist dieses mit einem höheren Aufwand verbunden. In der Praxis wird in der Regel eine Mischung aus beiden möglichen Bindungszeiten genutzt. Dies wird am Beispiel der Architektur des Mozilla-Browser in [GBS01] dargestellt.

Im Rahmen einer Software-Produktlinienentwicklung sind die Bindungszeiten der Variabilität in der Software von Interesse. Eine Übersicht über mögliche Bindungszeiten ist in [Beu03] gegeben:

- *Konfigurationszeit*: Schon bevor die Implementierung des Quelltextes für ein System beginnt, wird festgelegt, welche Funktionen in ihm enthalten sind. Somit entsteht für jedes Produkt ein eigener Quelltext. Die Variabilität, die so gebunden

wird, hat einen allgemeinen Einfluss auf das System und enthält im Wesentlichen gemeinsame Anteile innerhalb der Produktfamilie.

- *Kompilierzeit*: In der Implementierung werden durch Möglichkeiten der eingesetzten Programmiersprache oder Frameworks die variablen Anteile gekennzeichnet und bei der Kompilierung nur die Anteile genutzt, die für eine Variante von Belang sind. Für die Programmiersprache C werden sogenannte *ifdef*-Konstrukte eingesetzt, die durch den C-Präprozessor ausgewertet werden. Diese Variabilität kann durch entsprechende Werkzeuge unterstützt werden [KAK08].

- *Ladezeit*: Erst zur Ausführung des Programms steht fest, welche Variante eines Systems genutzt wird. Kompilierte Artefakte werden so kombiniert, dass sie ein Produkt ergeben. Hierfür können gemeinsame Bibliotheken eingesetzt werden. Die Eclipse-Plattform [Ecl] zum Beispiel weist abhängig von den eingebundenen Plugins unterschiedliche Funktionen auf.

- *Laufzeit*: Das Verhalten des Systems kann variabel zur Laufzeit gesteuert werden. Beispielsweise können Funktionen an- oder abgeschaltet werden. Eine weitere mögliche Realisierung stellt die Parametrisierung von Funktionen dar. Diese Variabilität wird nicht unbedingt vom Ersteller eines Systems gebunden, sondern kann auch vom Kunden realisiert werden, indem er im Multimediasystem einen bestimmten Radiosender auswählt. Es gibt keine eindeutige Unterscheidung zwischen Laufzeitvariabilität und benutzerdefinierter Konfiguration [HMA10].

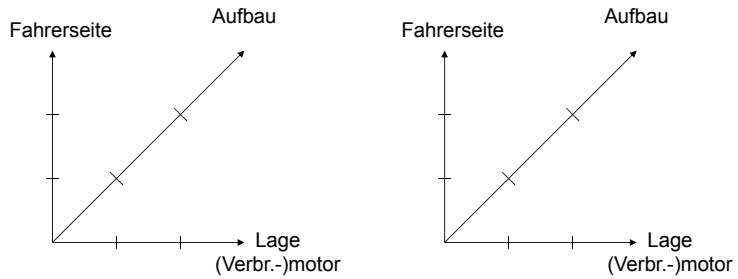
In der Lenkungssteuergerätesoftware sind die Bindungszeitpunkte der vorhandenen Variabilität unterschiedlich gestaltet. Ein Großteil der Funktionen lässt sich parametrieren und so deren Verhalten zur Laufzeit beeinflussen. Das ist auch notwendig, da letztendlich diese Parameter im praktischen Test mit dem Fahrzeug erst endgültig festgelegt werden. Die Ressourcenbegrenzungen auf dem Steuergerät und der daraus resultierende Testumfang lassen dies nur in begrenztem Umfang zu, so dass diese Informationen zur Ladezeit des Systems definiert sein müssen. Bestandteile, die zur Kompilierzeit feststehen, beinhalten im Wesentlichen die Möglichkeit eine Testsoftware zu erstellen, in der zusätzliche Testroutinen integriert sind. Zum Konfigurationszeitpunkt wird definiert, welche Funktionen in einer Lenkung vorhanden sind und wie die entsprechende Software-Architektur des Lenksystems aussieht. Diese Bindung der Variabilität wird durch das Konfigurationsmanagement unterstützt (siehe unter Abschnitt 5.2, Technische Infrastruktur).

5.4.3 Auftreten und Realisierung von Variabilität

Variabilität des Problem Space lässt sich meist nicht direkt auf Variabilität des Solution Space abbilden. Hinzu kommt, dass eine Entscheidung im Problem Space auf

mehrere Ebenen des Solution Space Auswirkungen hat. Änderungen von oberen Ebenen erfordern Anpassungen in unteren Ebenen. Ein anderer Bauraum benötigt ein anderes Package, in dem andere Hardware verbaut ist und die dadurch Anpassungen in der Software hervorruft.

In Abbildung 5.9 ist ein Ausschnitt der mechanischen Variabilität für das Lenkungsumfeld abgebildet. Die mechanische Variabilität hängt im Wesentlichen von der Umgebung ab, in der das System eingesetzt wird. Abbildung 5.9a zeigt beispielhaft, welche mechanischen Anforderungen es im Problem Space gibt. In einem Fahrzeug ist die Einbaulage des Verbrennungsmotors maßgeblich, da er einen großen Teil des Motorraums einnimmt. Die Fahrerseite hat ebenfalls Einfluss auf die Mechanik, da Bedienelemente angepasst werden müssen und sich der verfügbare Bauraum durch die Lenkstange ändert. Der Aufbau des Teils an sich bestimmt, wie und ob es eingebaut werden kann. In vielen Fällen gibt es mechanische Anbauteile zur Erfüllung von Zusatzfunktionen. Die mechanische Variabilität im Problem Space (Abbildung 5.9a) stimmt mit der Variabilität des Solution Space (Abbildung 5.9b) überein.

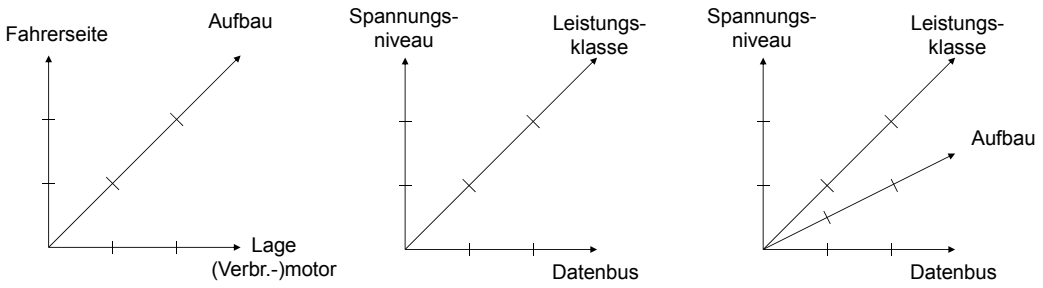


(a) Problem Space (b) Solution Space

Abbildung 5.9: Mechanische Variabilität (Ausschnitt)

Des Weiteren gibt es Anforderungen an die Hardware, die die elektrische Variabilität darstellen. Dies ist in Ausschnitten in Abbildung 5.10 dargestellt. Hierzu zählt der verwendete Datenbus (z.B. CAN, Flex-Ray, MOST, LIN), der eigene elektronische Bauteile benötigt. Bei der Auslegung ist auf das Spannungsniveau im Fahrzeug zu achten, z.B. 12 Volt bei PKW und 24 Volt bei LKW. Die (elektrische) Leistung, die ein System erreichen kann, beeinflusst auch das Hardware-Design. Zum Beispiel hat der Motor für die elektrischen Fensterheber bei größeren Scheiben eine höhere Leistungsaufnahme. Als weitere Anforderung muss jedoch der Aufbau des Gesamtsystems auf der mechanischen Ebene berücksichtigt werden, so dass die Variabilität im Solution Space (Abbildung 5.10b) die Variabilität des Problem Space aus Mechanik und Hardware (Abbildung 5.10a) abdeckt.

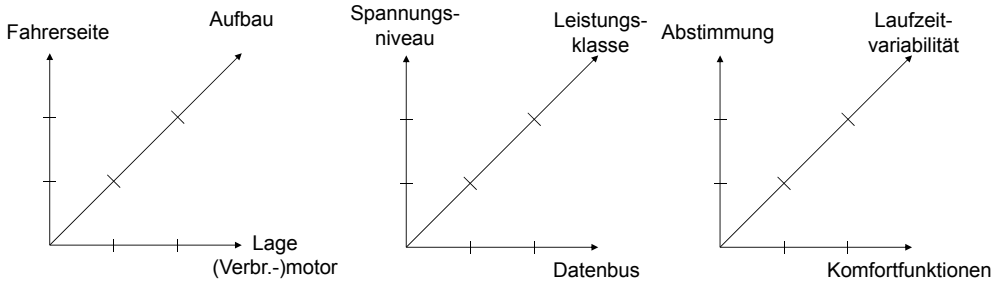
Zuletzt gibt es noch Anforderungen an die Software, die auszugsweise in Abbildung 5.11a dargestellt sind. Die Anforderungen an Komfortfunktionen werden in der Software reali-



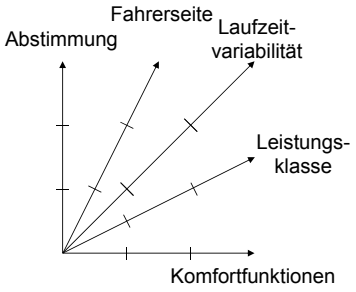
(a) Problem Space (b) Solution Space

Abbildung 5.10: Hardware Variabilität (Ausschnitt)

siert. Zudem betreffen eine Reihe von Anforderungen die Abstimmung des Systems, um die Software später dem Kundenwunsch anzupassen. Dies kann soweit gehen, dass ein bestimmtes Verhalten zur Laufzeit verändert werden kann. Zusätzlich haben noch mechanische und elektrische Anforderungen einen Einfluss auf die Software. Abbildung 5.11b zeigt diese Variabilität im Solution Space auszugsweise.



(a) Problem Space



(b) Solution Space

Abbildung 5.11: Software Variabilität (Ausschnitt)

Zusätzliche, nicht von vornherein geplante Variabilität erhält die Software durch Kompensation von mechanischem und elektrischem Fehlverhalten oder durch Optimierung des Gesamtsystems. Oft werden diese Änderungen erst in einer späten Phase umgesetzt, wenn das System integriert wird und die mechanische und elektrische Entwicklung weit-

gehend abgeschlossen ist. Als kostengünstige Lösung werden diese Punkte in der Software durch zusätzlichen Implementierungsaufwand abgestellt.

5.5 Zusammenfassung

Das Ergebnis der Maßnahme M2 (Bewertung der Einflussfaktoren) ist im Folgenden zusammenfassend dargestellt. Für die konkrete Situation (die Entwicklung der Lenkungssteuergerätesoftware) ist jeweils kurz festgehalten, welche Bedeutung welcher Einflussfaktor hat. In anderen Projekten können diese Einflüsse anders ausgeprägt sein und dort eine andere Bedeutung haben. Dieses Profil ist Basis für die individuellen Maßnahmen, mit denen eine Produktlinie instanziiert werden soll.

Im Bereich der quantitativen Einflussfaktoren (Tabelle 5.3) stehen besonders die Kosten im Vordergrund. Diese lassen sich einfach quantifizieren und bewerten. Das Ziel einer Unternehmung ist in der Regel, einen möglichst großen Gewinn zu erwirtschaften. Personen, Material, Zeit und Projektgruppengröße haben ebenfalls einen großen Einfluss auf die Umsetzung. Beim Thema Komplexität muss bei einer Produktlinie darauf geachtet werden, dass trotz der Änderungen durch die Berücksichtigung von Variabilität die Gesamtkomplexität noch beherrscht werden kann.

Tabelle 5.3: Einfluss von quantitativen Faktoren auf Produktlinien

Nr.	Einflussfaktor	Bedeutung
E1.1	*Personen und Material*	Mehr Personen und Material beschleunigen die Umsetzung einer Produktlinienentwicklung, führen jedoch auch zu einem erhöhten Verwaltungsaufwand.
E1.2	*Kosten*	Höhere Kosten müssen immer gegen Rentabilität der Produktlinie abgewogen werden.
E1.3	*Zeit*	Je mehr Zeit vergeht desto später kommen die Effekte der Produktlinie zu tragen.
E1.4	*Projektgruppengröße*	In kleinen Projekten können Innovationen wie eine Produktlinie besser umgesetzt werden.
E1.5	*Komplexität*	Höhere Komplexität muss beherrscht werden.

Im Bereich der qualitativen Faktoren (Tabelle 5.4) sind vor allem die Kundenbezogenheit, der Lebenszyklus und die Änderungsmöglichkeiten wichtig. Anhand der Kundenbezogenheit bzw. des Lebenszyklus wird oft schon entschieden, ob der Einsatz einer Produktlinienentwicklung sinnvoll ist. Ohne Änderungsmöglichkeiten lässt sich eine Entwicklung nicht anpassen, weshalb dieser Einflussfaktor eine sehr große Bedeutung hat. Für einen optimalen Einsatz von Produktlinientechniken ist eine modellbasierte Entwicklung eine

wichtige Voraussetzung, da Variabilität sich besser in Modellen abbilden lässt als im Quellcode. Eine gut ausgeprägte Innovationskultur unterstützt zusammen mit einer gut aufgestellten technischen Infrastruktur die Umsetzung, da sie Verständnis für Neuheiten schafft und so die Ideen eines Produktlinienansatzes besser verankert werden können. Ein neutraler Faktor sind Regularien, da diese in jedem Fall berücksichtigt werden müssen, auch wenn keine Produktlinienentwicklung angewandt wird.

Tabelle 5.4: Einfluss von qualitativen Faktoren auf Produktlinien

Nr.	Einflussfaktor	Bedeutung
E2.1	*Kundenbezogenheit*	Je spezifischer die Anforderungen von einzelnen Kunden sind, desto schwieriger wird es gemeinsame Teile zur Wiederverwendung zu finden.
E2.2	*Lebenszyklus*	Produktlinien erzielen bei der Entwicklung des Produktes am meisten Benefit, in der Vertriebsphase lohnt sich eine Produktlinie nicht mehr.
E2.3	*Innovationskultur*	Je besser die Innovationskultur ausgeprägt ist, desto besser kann eine Umsetzung von Produktlinien erfolgen.
E2.4	*Regularien*	Mehr Regularien können eine Produktlinie einschränken, diese sind jedoch auch bei der Einzelproduktentwicklung vorhanden.
E2.5	*Einsatz von Modellen*	Ein bestehender Einsatz von Modellen erleichtert Änderungen für eine Produktlinie.
E2.6	*technische Infrastruktur*	Je besser die bestehende technische Infrastruktur Produktlinien unterstützt, desto einfacher ist die Einführung einer Produktlinie.

Im Bereich der unternehmerischen Faktoren (Tabelle 5.5) ist bei der Produktlinienentwicklung das Betriebsklima ein sehr wichtiger Faktor. Wird der Migrationsprozess nicht oder nur in Teilen von den Mitarbeitern getragen, kann das die ganze Produktlinienstrategie gefährden. Die Wettbewerbspositionierung hat einen entscheidenden Anteil, wenn es darum geht Produktlinien einzusetzen, da nicht alle Wettbewerbssituationen gleich gut geeignet sind. Die Unternehmensziele können Produktlinien unterstützen, sind jedoch nicht Grundlage für deren Einsatz. Gibt es jedoch auf höherer Entscheidungsebene Widerstand gegen eine Produktlinienentwicklung, so ist diese gefährdet.

Die vorhandene und zu erfassende Variabilität ist der größte Einflussfaktor bei der Einführung von Produktlinien (Tabelle 5.6). Diese vollständig zu identifizieren und deren Auswirkungen abzusehen ist ein wesentlicher Teil der ersten Schritte. Dazu gehört auch, dass bekannt sein muss, in welchen Formen die Variabilität auftritt, wie sie konkret realisiert wird und zu welchem Zeitpunkt im Entwicklungsprozess sie gebunden wird. Die

Tabelle 5.5: Einfluss von unternehmerischen Faktoren auf Produktlinien

Nr.	Einflussfaktor	Bedeutung
E3.1	*Unternehmensziele*	Je mehr Produktlinien vom Unternehmen getragen werden, desto weniger wird die Umsetzung behindert.
E3.2	*Wettbewerbs-positionierung*	Viele abgegrenzte Produkte für viele Kunden bieten optimale Voraussetzung für den Einsatz einer Produktlinie.
E3.3	*Betriebsklima*	Je besser die Mitarbeiter bei der Umsetzung integriert und um so motivierter sie sind, desto größer sind die Erfolgschancen.

Verteilung der Variabilität auf mehrere Ebenen in einem Produkt mit Einflüssen über Ebenen hinweg macht eine vollständige Erfassung sämtlicher Variabilität schwierig. Nicht jede Änderung bedeutet gleich, dass eine neue Variante eines Produktes entsteht. Eine sorgfältige Trennung zwischen Evolution und Variantenbildung wird idealerweise schon durch ein Konfigurations- bzw. Versionsmanagementwerkzeug ermöglicht.

Tabelle 5.6: Einfluss von Variabilität auf Produktlinien

Nr.	Einflussfaktor	Bedeutung
E4.1	*Verteilung von Variabilität über mehrere Ebenen*	Variabilität muss in allen Ebenen erkannt werden.
E4.2	*Bindung von Variabilität*	Eindeutige Identifikation notwendig, wann welche Variabilität gebunden wird.
E4.3	*Auftreten und Realisierung von Variabilität*	Die Beeinflussung vom Solution Space aus mehreren Problem Spaces muss sauber abgebildet werden.

Kapitel 6

Strategie bei der Implementierung von Software-Produktlinien

Nach Abschluss der Analyse wird die Maßnahme M3 (Definition der Strategie) und darauf aufbauend die Maßnahme M4 (Kommunikation der Produktlinieninhalte an die Mitarbeiter) umgesetzt. Dazu werden die Ergebnisse der Analyse durch die Maßnahmen M1 (Untersuchung der Software) aus Kapitel 4 und M2 (Bewertung der Einflussfaktoren) aus Kapitel 5 zusammengeführt und daraus konkrete Handlungsmöglichkeiten für die Implementierung einer Software-Produktlinie abgeleitet. Diese Handlungsmöglichkeiten sind spezifisch für den Projektkontext. In dieser Arbeit werden die Handlungsmöglichkeiten spezifisch für die Softwareentwicklung der elektromechanische Lenksysteme bei der Volkswagen AG abgeleitet. Auf ähnliche Weise lassen sich jedoch auch andere Kontexte bearbeiten.

Die Ableitung der Strategie erfolgt zunächst durch die Zuordnung der Maßnahmen zu den Einflussfaktoren (Abschnitt 6.1). Generell haben die Einflussfaktoren Auswirkungen auf die Umsetzung der Maßnahmen, da sie dort berücksichtigt werden müssen und so die Realisierung einschränken können. Andersherum wirkt sich die Umsetzung einer Maßnahme auf die Einflussfaktoren aus. Das Ziel aller Maßnahmen ist, dass die Einflussfaktoren positiv verändert werden. Die Auswahl der Maßnahmen zusammen mit einer Reihenfolge, in der sie umgesetzt werden (Abschnitt 6.2), stellt die Strategie bei der Implementierung der Produktlinie dar. Wie diese adäquat den Projektbeteiligten kommuniziert werden kann, ist in Abschnitt 6.3 dargestellt.

6.1 Zuordnung von Maßnahmen zu Einflussfaktoren

Die Auswirkungen je Maßnahme werden den in Kapitel 5 vorgestellten Kategorien der Einflussfaktoren zugeordnet. Dazu werden die Maßnahmen der Reihe nach durchgegangen und die individuellen Auswirkungen auf die Einflussfaktoren diskutiert. Basis für die Einschätzung der Auswirkungen waren Gespräche mit den Projektbeteiligten. Anhand

der Auswirkungen lässt sich feststellen, ob die Implementierung einer Maßnahme sinnvoll ist und deshalb zur Umsetzung ausgewählt wird. Die grundlegenden Maßnahmen werden nicht besprochen, da sie schon durchgeführt wurden und für diese nicht entschieden werden muss, ob sie durchgeführt werden sollen oder nicht.

Identifikation und Nutzung von Modellen (M6). Für die Quantität hat die Identifikation und Nutzung von Modellen positive Auswirkungen auf die Kosten (E1.2), die Zeit (E1.3) und die Komplexität (E1.5) in der Entwicklung. Modelle sind in der Regel einfacher zu erstellen als z.B. Quelltext und der Umfang eines Modells ist bei gleicher Aussagekraft deutlich geringer, da Modelle sich auf die wesentlichen Elemente konzentrieren. Damit sinken der Aufwand und die Komplexität und damit auch die Kosten in der Entwicklung. Im Rahmen der Qualität sorgen Modelle für eine bessere Innovationskultur (E2.3), da die Mitarbeiter an neue Sichtweisen in der Entwicklung herangeführt werden. Der Einsatz von Modellen begünstigt den entsprechenden Einflussfaktor (E2.5) und erweitert die Basis der technischen Infrastruktur (E2.7). Auf unternehmerische Einflussfaktoren sowie auf die Variabilität hat diese Maßnahme keinen Einfluss.

Einsatz von Werkzeugen (M7). Diese Maßnahme bildet das Komplement zu der vorherigen Maßnahme und hat dementsprechend auch die gleichen Auswirkungen auf die Einflussfaktoren. Kosten (E1.2), Zeit (E1.3) und Komplexität (E1.5) verringern sich. Die Qualität steigt durch bessere Innovationskultur (E2.3), Einsatz von Modellen (E2.5) sowie durch die Nutzung weiterer technischer Infrastruktur (E2.7). Auf unternehmerische Einflussfaktoren sowie auf die Variabilität hat diese Maßnahme ebenfalls keinen Einfluss.

Anbindung bestehender Datenbanken (M8). Die Anbindung bestehender Datenbanken bringt eine großen positiven Einfluss auf die quantitativen Einflussfaktoren. Da die Informationen bisher manuell von Personen aus anderen Systemen extrahiert und aufbereitet werden müssen, lassen sich durch diese Maßnahme Personen (E1.1) und Zeit (E1.3) und damit wieder Kosten (E1.2) sparen. Zusätzlich wird die Komplexität (E1.5) für die Mitarbeiter verringert, da diese in der Anbindung gekapselt werden kann. Im Bereich der Qualität wird die technische Infrastruktur (E2.7) erweitert. Eine Auswirkung auf unternehmerische Einflussfaktoren oder die Variabilität liegt nicht vor.

Festlegung der Variabilitätsrealisierung (M9). Durch die Festlegung der Variabilität wird die Komplexität (E1.5), die durch die Varianten eines Systems entsteht, verringert. Ein positiver Effekt im Bereich der Qualität wird durch die Notwendigkeit des Modelleinsatzes (E2.5) sowie durch neue Änderungsmöglichkeiten innerhalb der Modelle (E2.6)

erreicht. Es gibt keinen Einfluss auf unternehmerische Einflussfaktoren. Bei der Variabilität ist durch die Maßnahme die Bindung (E4.2) und die Beschreibung des Auftretens und der Realisierung der Variabilität (E4.3) möglich.

Nutzung der Variabilität in Modellen (M10). Die Definition der Variabilität in Modellen verringert die Komplexität der Modelle (E1.5), indem Konstrukte und Redundanzen, die bisher in den Modellen mehrere Varianten gekennzeichnet haben, durch einen Variabilitätsmechanismus beschrieben werden. Das modellbasierte (E2.5) Vorgehen wird dadurch begünstigt. Die positiven Auswirkungen auf die Änderungsmöglichkeiten (E2.6) ermöglichen von nun an die Unterscheidung zwischen Weiterentwicklung und variablen Anteilen. Ebenso kann durch die Nutzung der Variabilität diese zu einem bestimmten Zeitpunkt gebunden (E4.2) und zusammengefasst werden (E4.3).

Nutzung von Variabilität in der Entwicklung (M11). Bei der Nutzung der Variabilität in der weiteren Entwicklung, also vom Modell bis hin zum Quellcode, werden die gleichen Einflussfaktoren wie bei der Nutzung der Variabilität in Modellen begünstigt. Die Komplexität (E1.5) verringert sich und die modellbasierte Entwicklung gefördert (E2.5). Die Änderungsmöglichkeiten (E2.6) werden erweitert sowie der Umgang mit der Variabilität (E4.2, E4.3) intensiviert.

Werkzeugentwicklung und -einsatz für die Produktlinienentwicklung (M12). Mit angepassten Werkzeugen werden die Kosten (E1.2) und die Zeit (E1.3) für die Entwicklung gesenkt. Die inhärente Komplexität (E1.5) durch mehrere Varianten wird durch die Werkzeuge gekapselt und so für die Anwender leichter handhabbar gemacht. Außerdem werden die beiden vorherigen Maßnahmen unterstützt und insbesondere die modellbasierte Entwicklung (E2.5) verbessert. Die neuen Werkzeuge stellen eine Erweiterung der technischen Infrastruktur (E2.7) dar.

Nutzung von Variabilität im Test (M13). Mit der Variabilität im Test wird entlang des kompletten V-Modells eine Nutzung der Variabilität möglich, was Zeit (E1.3) spart und Komplexität (E1.5) verringert. Die Möglichkeit, Variabilität zu binden (E4.2) und sinnvoll zusammenzufassen (E4.3), wird durch diese Maßnahme ebenfalls erweitert.

Anpassung der Architektur an die Variabilität (M14). Die Anpassung der Architektur erweitert die Änderungsmöglichkeiten (E2.6) im Rahmen der Entwicklung. Durch die Einbeziehung von Variabilität kann diese auf Architekturebene beschrieben und gebunden werden (E4.2).

Definition eines Merkmalmodells für den Problem Space (M15). Die Definition des Merkmalmodells kommt vor allem der Variabilität zugute. Über das Modell kann zwischen Auftreten im Problem Space und Bindung im Solution Space (E4.2, E4.3) unterschieden werden.

Strukturierung nach Entwicklungsteams für Domain und Application Engineering (M16). Die Trennung in zwei Entwicklungsteams hat einen geringen Einfluss auf die Personenanzahl (E1.1). Dennoch ist eine solche Aufteilung positiv zu beurteilen, da jede Gruppe sich auf ihren Teil konzentrieren kann und so insgesamt eine potentiell höhere Qualität möglich ist. Das wirkt sich auch positiv auf das Betriebsklima (E3.3) aus, da Verantwortlichkeiten geklärt sind.

Aufbau und Nutzung einer Modulbibliothek (M17). Mit der Modulbibliothek wird die Komplexität (E1.5) des Solution Space der Produktlinie für den Anwender verringert, da diese in der Modulbibliothek gekapselt ist. Dazu trägt die Modulbibliothek zur Erweiterung der technischen Infrastruktur (E2.7) bei. Die gesamte Werkzeuglandschaft wird so konsolidiert.

Im linken Teil der Tabelle 6.1 ist zusammenfassend dargestellt, wie sich die Maßnahmen positiv auf die einzelnen Einflussfaktoren auswirken. Die Maßnahmen M6 bis M13 haben die meisten positiven Effekte, während die Maßnahmen M14 bis M17 weniger Einfluss haben. Trotzdem sind diese Maßnahmen wichtig. Deshalb werden im nächsten Schritt Aufwand und Nutzen der Maßnahmen gegenübergestellt, um letztendlich die realisierbaren und wirksamsten Maßnahmen auszuwählen.

6.2 Auswahl der Maßnahmen für die Realisierung

Für die zur Auswahl stehenden Maßnahmen wurde jeweils zusammen mit den am Projekt beteiligten Personen abgeschätzt, wie hoch der Aufwand für die Umsetzung dieser Maßnahme jeweils ist. Tabelle 6.1 zeigt in der letzten Spalte die Abschätzung für die Implementierung der Maßnahmen in der Lenkungssoftwareentwicklung bei Volkswagen. Diese Abschätzung wird den positiv beeinflussten Faktoren gegenüber gestellt, die ebenfalls zusammen mit den Projektbeteiligten ermittelt wurden. Da es sich bei den Aufwänden um sensible Daten handelt, sind diese hier jeweils in die Kategorien gering, mittel und hoch eingeordnet.

Für die Auswahl werden diese Daten in ein Koordinatensystem übertragen (siehe Abbildung 6.1). Die Maßnahmen, die sich am besten für die Umsetzung eignen, sind in der unteren rechten Ecke zu sehen. Je höher ein Punkt liegt, desto höher ist auch der

Tabelle 6.1: Auswirkungen von Maßnahmen auf die Einflussfaktoren

	Einflussfaktoren					abgeschätzter
	Quantität	Qualität	Unternehmen	Variabilität	Summe	Aufwand
M6	+++	+++	o	o	6	gering
M7	+++	+++	o	o	6	mittel
M8	++++	+	o	o	5	mittel
M9	+	++	o	++	5	mittel
M10	+	++	o	++	5	hoch
M11	+	++	o	++	5	hoch
M12	+++	++	o	o	5	mittel
M13	++	o	o	++	4	hoch
M14	o	+	o	+	2	hoch
M15	o	o	o	++	2	gering
M16	+	o	+	o	2	mittel
M17	+	+	o	o	2	hoch

Maßnahmen: M6–Identifikation und Nutzung von Modellen, M7–Einsatz von Werkzeugen, M8–Anbindung bestehender Datenbanken, M9–Festlegung der Variabilitätsrealisierung, M10–Nutzung der Variabilität in Modellen, M11–Nutzung von Variabilität in der Entwicklung, M12–Werkzeugentwicklung und -einsatz für die Produktlinienentwicklung, M13–Nutzung von Variabilität im Test, M14–Anpassung der Architektur an die Variabilität, M15–Definition eines Merkmalmodells für den Problem Space, M16–Strukturierung nach Entwicklungsteams für Domain und Application Engineering, M17–Aufbau und Nutzung einer Modulbibliothek

Aufwand und je weiter rechts er liegt, desto höher ist der Nutzen der Maßnahme. Das Optimum befindet sich als in der rechten unteren Ecke.

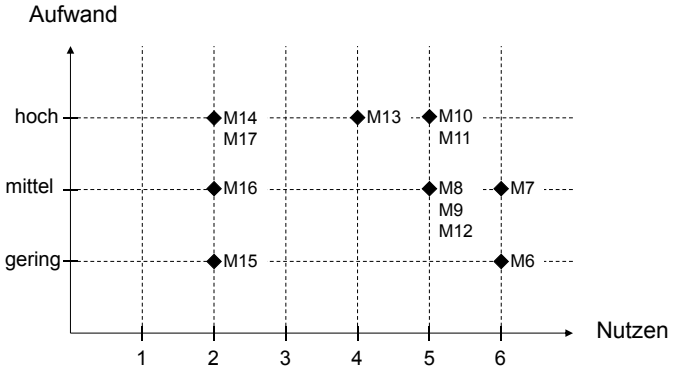


Abbildung 6.1: Aufwand und Nutzen für die Umsetzung der Maßnahmen

Da nur wenig personelle Kapazität vorhanden ist, scheiden die Maßnahmen mit hohem Aufwand für dieses Projektumfeld aus, auch wenn sie die wichtigsten Maßnahmen für eine Produktlinienentwicklung sind. Die Maßnahmen M15 und M16 bringen relativ

wenig Nutzen im Vergleich zum investierten Aufwand. Von den verbleibenden Maßnahmen M6 bis M9 und M12 benötigt letztere als Voraussetzung die Maßnahmen M10 und M11, weshalb diese nicht realisiert werden kann. Es wurden deshalb die Maßnahmen M6 (Identifikation und Nutzung von Modellen), M7 (Einsatz von Werkzeugen), M8 (Anbindung bestehender Datenbanken) und M9 (Festlegung der Variabilitätsrealisierung) umgesetzt.

Nachdem die Maßnahmen festgelegt wurden, musste die Abarbeitungsreihenfolge der Realisierungen geplant werden. Da nur eine Person dafür zur Verfügung stand, wurde die Reihenfolge aus den Abhängigkeiten der Maßnahmen (Abbildung 3.4) übernommen. Es wird zunächst als Abschluss der grundlegenden Maßnahmen im folgendem Abschnitt eine Möglichkeit beschrieben, das Ergebnis für die Entwicklung darzustellen (Maßnahme M4, Kommunikation der Produktlinieninhalte an die Mitarbeiter). Anschließend werden in Kapitel 7 die Maßnahmen M6 (Identifikation und Nutzung von Modellen) und M9 (Festlegung der Variabilitätsrealisierung) und anschließend in Kapitel 8 die Maßnahmen M7 (Einsatz von Werkzeugen) und M8 (Anbindung bestehender Datenbanken) realisiert.

6.3 Erstellung einer Übersicht über die vorhandenen Projekte

Für die Mitarbeiter, die an der Umsetzung der Produktlinie beteiligt sind, ist es wichtig eine grundsätzliche und detaillierte Übersicht über die Projekte zu haben. Gerade wenn viele Personen an einer Entwicklung beteiligt sind, sind nicht immer alle wichtigen Informationen an den richtigen Stellen verfügbar. Aus Sicht einer Produktlinienentwicklung wird dieses Problem noch verschärft, da dies dann für mehrere Entwicklungsprojekte der Fall ist und viele Informationsquellen aggregiert werden müssen. Außerdem gibt es in solch einer großen Gruppe oft Fluktuation, so dass wichtige Informationsträger die Entwicklung verlassen und neue Personen sich mit den aktuellen Informationen versorgen müssen.

In der Praxis wird durch die Maßnahme M4 (Kommunikation der Produktlinieninhalte an die Mitarbeiter) eine bessere Informationsvermittlung im Rahmen der Produktlinie erreicht. Dazu wird ein Projektraum genutzt, der alle wichtigen aktuellen Information zu allen Projekten der (zukünftigen) Produktlinie beinhaltet. Dieser Projektraum dient nicht nur zur Information des Teams, sondern gleichzeitig als Wissensbasis für eine mögliche Produktlinie. Durch diese doppelte Verwendung können schon früh die Produktlinien-Aktivitäten zu einer Verbesserung der Arbeit führen. Dadurch, dass Mitarbeiter projektbezogen eingesetzt werden, fehlt ihnen selber oft der Überblick über andere Projekte. Dieser Überblick ist aber eine Grundvoraussetzung für die Schaffung von Synergie-Effekten, wie die Nutzung gemeinsamer Features oder die Identifikation

und Behebung von Fehlern, die bei mehreren Systemen auftreten.

Zunächst muss festgelegt werden, welche Informationen im Projektraum zur Verfügung stehen sollen. Im vorliegenden Projektumfeld waren folgende Informationen zu erfassen:

- Basisinformationen (Projektname, Projektnummer, zu entwickelnde Produkte)
- Projektleiter und Teilprojektleiter (Ansprechpartner bei Problemen)
- Ansprechpartner bei Zulieferern und Kunden
- aktueller Meilensteinplan
- aktueller Status
- Projektteam
- aktueller Status der Änderungen

Wenn diese Informationen für mehrere Projekte in einem physisch vorhandenen Raum übersichtlich darstellt werden, dann wird dafür sehr viel Platz benötigt. Ein solcher Raum steht in der Realität selten zur Verfügung, da es keinen Raum in der entsprechenden Größe gibt oder sämtlicher verfügbarer Platz mit Personal besetzt ist. Letzteres ist insbesondere dann der Fall, wenn ein Projekt schnell wächst, da sich Personal schneller beschaffen lässt, als neue Gebäude zu erbauen. Die Existenz eines physischen Raumes ist gar nicht notwendig, da die Informationen überwiegend in digitaler Form vorliegen und so ein virtueller Projektraum (Virtual Projekt Office – VPO) als Realisierungsform gewählt werden kann und im vorliegenden Fall auch gewählt wurde.

Ein VPO hat den Vorteil, dass es keinen Platzbedarf für ihn gibt. Er ist auch unabhängig vom jeweiligen Aufenthaltsort verfügbar, sogar außerhalb des Arbeitsplatzes, sofern etablierte Dienste wie ein VPN genutzt werden. Damit ist er auch schnell in Besprechungen verfügbar. Durch die Einbindung in vorhandene IT-Infrastruktur müssen Dinge wie Backup der Informationen nicht extra geklärt werden.

6.3.1 Anforderungen an den Projektraum

Neben den inhaltlichen Anforderungen, die bereits dargestellt wurden, gibt es noch weitere Anforderungen. Diese zielen vor allem auf die Handhabbarkeit und Verfügbarkeit der Inhalte ab.

Alle aktuellen Informationen müssen schnell zur Verfügung stehen. Dabei sollen wichtige Informationen direkt und nicht über weitere Verlinkungen, denen noch gefolgt muss, zugänglich sein. So sind Informationen, die im Ernstfall schnell abgerufen werden müssen, einfach verfügbar. Weniger kritische Informationen, die sich weniger häufig ändern, wie

Sitzpläne, können über Verlinkungen verknüpft sein. Diese Informationen werden in der Regel gezielt und ohne Zeitdruck gesucht, so dass im Anwendungsfall die Bereitschaft Links zu folgen hoch ist. Mit Verknüpfung weiterer Informationen, wie einer Prozessbeschreibung, dient der Projektraum als ein zentraler Anlaufpunkt für übergeordnete Informationen zu allen Projekten.

In den meisten Fällen sind nicht alle Informationen des VPOs für einen Mitarbeiter relevant. Projektleiter interessieren sich hauptsächlich für die Details ihres Projekts. Mit einer Druckfunktion besteht die Möglichkeit, sich den aktuellen Stand als Vorbereitung für Besprechungen auszudrucken. Gerade bei Führungskräften, die einen vollen Terminkalender haben, ist die Zeit, die für die Sammlung der Daten aus unterschiedlichen Quellen verfügbar ist, sehr knapp bemessen. Die Zusammenstellung der benötigten Informationen muss deshalb leicht anpassbar sein.

Der Inhalt des Projektraums muss ständig aktuell gehalten werden, da sonst dieser nicht genutzt wird, wenn nur veraltete oder falsche Informationen dort vorhanden sind. Die Pflege muss zentral durch einen beschränkten Personenkreis erfolgen, so dass Informationen nicht durch jeden geändert werden können und ein konsistenter und verlässlicher Informationsstand existiert. Je größer der Kreis der bearbeitenden Personen ist, desto eher kann es geschehen, dass überflüssige oder falsche Informationen im VPO landen. Da die Aktualisierung ein regelmäßiger Vorgang ist, muss die Aktualisierung möglichst einfach sein. Die Informationen, die im VPO vorhanden sind, werden überwiegend durch Projektcontroller bereitgestellt. Anstatt die Informationen über einen Email-Verteiler zu verteilen, kann mit ähnlichem Aufwand eine zentrale Ablage der Informationen genutzt werden. Gleichzeitig wäre so sichergestellt, dass jeder sofort die aktuellen Informationen zur Verfügung hat und nicht erst in seinen Emails suchen muss.

Zusammenfassend sind die Anforderungen an den VPO im Folgenden dargestellt:

1. Inhalt

    a) Die aktuellen Informationen im VPO sollen direkt und nicht über Links verfügbar sein.

    b) Statische Informationen (Einführung in die Entwicklung, Prozesse, regelmäßig stattfindende Termine, Sitzplan) können verlinkt sein.

    c) Der VPO soll einen Informationsbereich für neue Mitarbeiter enthalten, an dem alle ersten Schritte am neuen Arbeitsplatz beschrieben sind.

2. Benutzer

    a) Der VPO soll schnell die aktuellen Informationen bereit stellen können.

    b) Es soll eine Druckfunktion existieren, um schnell einen definierten Stand aus-

zudrucken (Status eines bestimmten Projekts, Status aller Projekte).

c) Es sollen Zusammenstellungen von Projekten aus unterschiedlichen Organisationsteilen möglich sein.

3. Aktualisierung

a) Die Aktualisierung des VPOs soll nur bestimmten Personen möglich sein.

b) Die Aktualisierung des VPOs soll mit möglichst wenig Aufwand möglich sein.

6.3.2 Realisierungsmöglichkeiten

Als Realisierungsform für diese Anforderungen stehen verschiedene Möglichkeiten zur Auswahl. Da es ein leichtgewichtiges System zur Unterstützung mit viel projektbezogenem Freiraum werden sollte, kamen umfangreiche und kostenintensive Lösungen nicht in Frage. Letztendlich reduzierten sich die Möglichkeiten darauf ein Wiki aufzusetzen oder die Informationen in einer Bildschirm-Präsentation abzulegen.

Das Wiki bietet leichte Änderungsmöglichkeiten, so dass Informationen prinzipiell durch jeden dort abgelegt werden können. Dokumente (zum Beispiel PDFs) werden in Wikis problemlos hinzugefügt oder verlinkt. Dadurch, dass ein großer Personenkreis neue Informationen hinzufügen, ändern und löschen kann, sinkt die Verlässlichkeit der Informationen. So ist eine gezielte gewollte oder ungewollte Fehlinformation durch jede einzelne Person möglich. Die Beschränkung der Schreibrechte auf einzelne Personen erzeugt zusätzlichen Aufwand für die Benutzerverwaltung. Im Wiki gibt es eine feste Struktur für die Informationen, womit die Darstellungsform eher unflexibel ist. Komplizierte Zusammenhänge lassen sich in einer freieren Form einfacher darstellen.

Die zweite Möglichkeit ist eine Bildschirm-Präsentation, in der wichtige Inhalte auf wenigen Folien zusammengefasst werden. So werden unnötige Programmwechsel und Medienbrüche vermieden und Informationen aus anderen Präsentationen können einfach durch Kopieren und Einfügen eingebettet werden. Eine solche Präsentation lässt sich durch ein Passwort schützen, so dass der Kreis der Personen, die die Präsentation ändern können, sinnvoll beschränkt werden kann. Mit den vielen Freiheiten einer Präsentation, mit denen Informationen dargestellt und verknüpft werden können, lassen sich auch komplizierte Sachverhalte darstellen. Weitere Informationen zu den einzelnen Bestandteilen lassen sich verlinken und auf anderen Folien der Präsentation unterbringen. Durch die vorhandenen Möglichkeiten einer Präsentation lassen sich eigene Navigationskonzepte für die Daten realisieren.

Für die Umsetzung von Produktlinien sind diese Informationen sehr wertvoll, weshalb es sinnvoll ist, dass die Personen, die die Produktlinie einführen, auch die Pflege des VPOs

übernehmen. Da so alle relevanten Informationen und Stati der Projekte zusammenfließen, können aktuelle Entwicklungen schnell berücksichtigt werden. Der Aufwand für die Projektcontroller bleibt gleich, da es keinen zusätzlichen Aufwand macht bei Änderungen eine Person mehr zu informieren. Oft gibt es schon definierte Email-Verteiler, die dann einmalig erweitert werden müssen.

Für eine langfristige Realisierung eines virtuellen Projektraums ist eine Lösung geeignet, die möglichst wenig manuellen Eingriff erfordert. Mit der Zeit werden Schnittstellen zu anderen Systemen (wie dem Änderungsmanagement) klarer und die ausgetauschten Informationen präzisiert. Dann ist eine Präsentation nicht unbedingt die geeignete Form, um einen VPO zu realisieren. Denkbar wäre ein eigens angepasstes System zu entwickeln, welches mit anderen Werkzeugen eines Vorgehens nach Produktlinien gekoppelt ist. Dies hängt davon ab, wie wichtig der VPO wird und wie hoch die Bereitschaft ist ihn weiterzuentwickeln.

6.3.3 Umsetzung mittels einer Präsentation

Im konkreten Projekt wurde der virtuelle Projektraum mit einer Bildschirm-Präsentation umgesetzt. Die Startseite dazu ist in Abbildung 6.2 dargestellt. Im oberen Teil sind die Projekte spaltenweise angeordnet und sind jeweils den einzelnen Produktlinien zugeordnet. Jede Produktlinie erhält eine prägnante Farbe, die auch in der weiteren Präsentation aufgegriffen wird. Ein Logo und Produktbilder sorgen für eine einfache Übersicht und Wiedererkennbarkeit der Projekte. Im unteren Teil sind die Informationen verankert, die über alle Projekte hinweg wichtig sind. So sind diese schnell erreichbar.

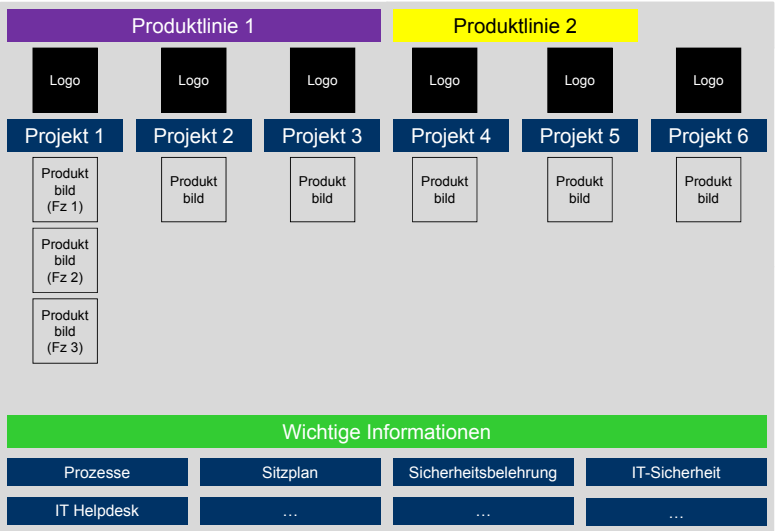


Abbildung 6.2: Startseite des VPOs

Durch Auswahl eines Projekts werden die jeweiligen Projekt-Steckbriefe (siehe Abbildung 6.3) angezeigt. Diese enthalten alle wichtigen Informationen zusammengefasst auf einem Blatt. Links oben sind wesentliche Projektparameter wie Projektname und Nummer sowie entwickelte Produkte angegeben. Rechts oben sind die wichtigsten Projektbeteiligten inklusive Telefonnummer angegeben. Damit diese Personen im Bedarfsfall schnell kontaktiert werden können, ist die Angabe einer Mobilnummer sinnvoll. Links unten sind die nächsten Meilensteine abgebildet und rechts unten die aktuellen Informationen.

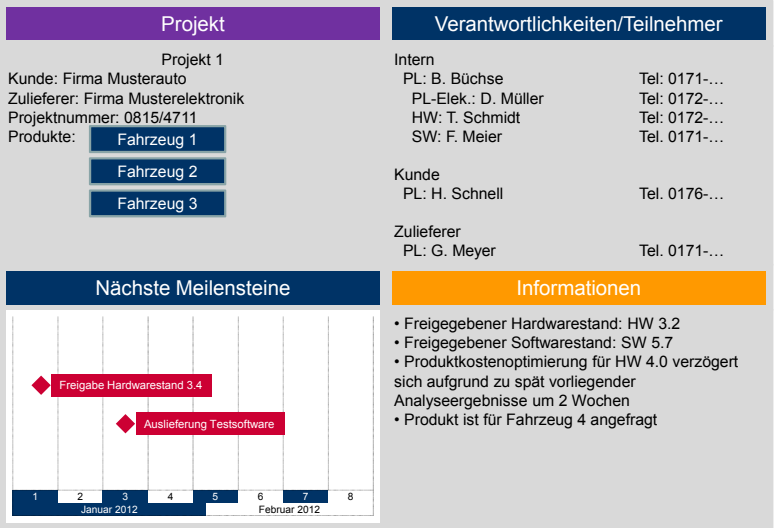


Abbildung 6.3: Projektsteckbrief bei einem Beispielprojekt

Zur Übersicht über alle Informationen im VPO eignet sich eine Sitemap (Abbildung 6.4). Diese Darstellung hilft vor allem neu hinzugekommenen Personen in der Abteilung sich durch die Fülle an Informationen zu navigieren.

Im Produktlinien-Einführungsprojekt hat sich der VPO im Laufe der Zeit vor allem zu einem Werkzeug entwickelt, welches für neue Mitarbeiter als Einstieg dient. Die Möglichkeit, neuen Mitarbeitern eine Bildschirm-Präsentation in die Hand zu geben, wurde sehr geschätzt. Somit war auch das Interesse der Projektleitung groß, dass die Informationen aktuell und gültig waren. Dadurch, dass die Wartung im Rahmen des Produktlinienvorgehens umgesetzt wurde, standen die aktuellen Informationen für den VPO auch gleichzeitig dem Produktlinien-Team zur Verfügung, womit der Informationsfluss optimiert wurde.

Der Vorteil, dass viele Informationen im VPO vorhanden waren, hatte jedoch auch den Nachteil, dass entsprechend Zeit in die Wartung des VPOs investiert werden musste. Da mit der Zeit jedoch der VPO als Informationsportal verwendet wurde (statt als aktueller Statusbericht), konnten die Anforderungen anders umgesetzt werden. Meilensteinpläne

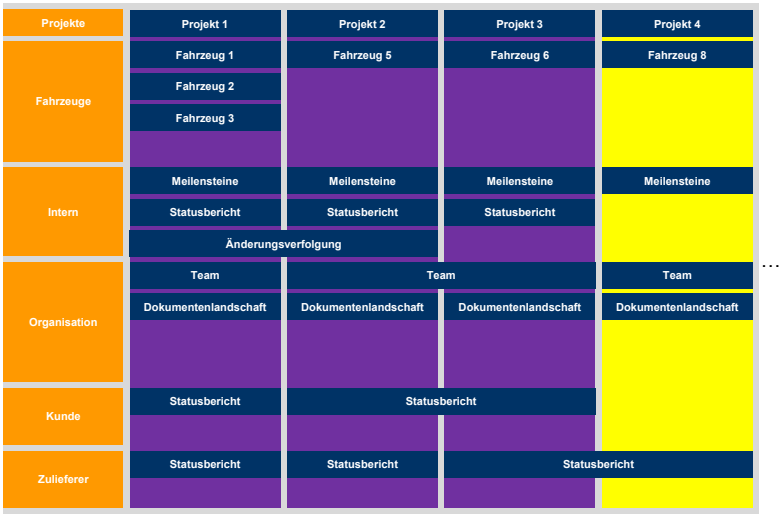


Abbildung 6.4: Sitemap für den VPO

konnten z.B. verlinkt werden, anstatt direkt eingebunden zu sein. Dies traf auch für weitere Informationen zu, so dass sich der Pflegeaufwand verringern ließ.

Mit diesem Stand des VPOs wurden nicht nur Produktlinien-Ziele erreicht, sondern auch generell die Prozess-konforme Entwicklung unterstützt. Im Rahmen eines unabhängigen Spice-Assessments wurde von den Assessoren der VPO als zentrale Informationsstelle zu allen weiteren wichtigen Informationen positiv beurteilt. Somit dient der VPO nicht nur für das eigene Team zu Information, sondern kann auch für die Außendarstellung genutzt werden.

6.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie sich die Ergebnisse der Produktlinien-Analyse zu einer Strategie zusammenführen lassen (Maßnahme M3, Definition der Strategie). In einer Wirtschaftlichkeitsbetrachtung wird der Nutzen der Maßnahmen bewertet und dem jeweiligen Aufwand gegenübergestellt. Zusätzlich muss immer betrachtet werden, welche Maßnahmen überhaupt abhängig von den gegebenen Ressourcen umgesetzt werden können.

Wenn die Strategie definiert ist, ist der nächste Schritt die Vermittlung der enthaltenen Informationen an das Projekt, was die Realisierung der Maßnahme M4 (Kommunikation der Produktlinieninhalte an die Mitarbeiter) bedeutet. Über ein Informationsportal werden nicht nur produktlinienspezifische Informationen verteilt, sondern auch allgemeine

Stati kommuniziert. Dies stellt auch unabhängig von Produktlinien eine Verbesserung im Entwicklungsprozess dar.

Kapitel 7

Modellbasiertes Vorgehen

Bei den Maßnahmen zur Einführung einer Produktlinienentwicklung spielen Modelle eine Schlüsselrolle (siehe Maßnahmen M6 – M15). Variabilität und Evolution sind anhand von Modellen besser beschreibbar als an Quelltext oder textuellen Anforderungen. Die Untersuchungen des Kapitels 5 bestätigen diesen Eindruck.

Um die zusätzliche Komplexität, bedingt durch die Variabilität, von Produktlinien gut erfassen zu können, ist ein konsequenter Einsatz einer modellbasierten Entwicklung notwendig. Im Folgenden wird dargestellt, wie man bisher nicht genutzte aber schon vorhandene Modelle sinnvoll in den Entwicklungsprozess integrieren kann (Maßnahme M6, Identifikation und Nutzung von Modellen) und wie entsprechende Modellierungssprachen abgeleitet werden können. Für Produktlinien wird zudem die Variabilitätsmodellierung in Modellen besprochen (Maßnahme M9, Festlegung der Variabilitätsrealisierung).

Dieses Kapitel beschreibt zunächst grundsätzlich die Möglichkeiten der modellbasierten Softwareentwicklung (Abschnitt 7.1). Anschließend wird in Abschnitt 7.2 darauf eingegangen, wie Modelle im Entwicklungsprozess theoretisch identifiziert werden können, und danach die Anwendung bei der Volkswagen AG gezeigt (Abschnitt 7.3). Für Produktlinien wird die Definition der Variabilität in Abschnitt 7.4 diskutiert.

7.1 Überblick über die modellbasierte Softwareentwicklung

Ein Modell hat allgemein die in Definition 18 festgelegten Merkmale. Diese Merkmale treffen für alle Modelle weitläufig benutzter Modellierungssprachen zu. Beispiele sind die UML [OMG10b, OMG10c, Rum11, Rum12] zur Beschreibung von Software-Komponenten, SysML [OMG10a] zur Beschreibung von komplexen Systemen, BPMN [OMG11] zur Beschreibung von Vorgängen und Prozessen sowie ERM [Che76] zur Beschreibung von relationalen Datenbanken. Ein weit verbreitetes Werkzeug für die modellbasierte Entwicklung ist Matlab/Simulink, welches Quellcode generieren kann und umfangreiche

Analysen und Simulationen von Modellen ermöglicht.

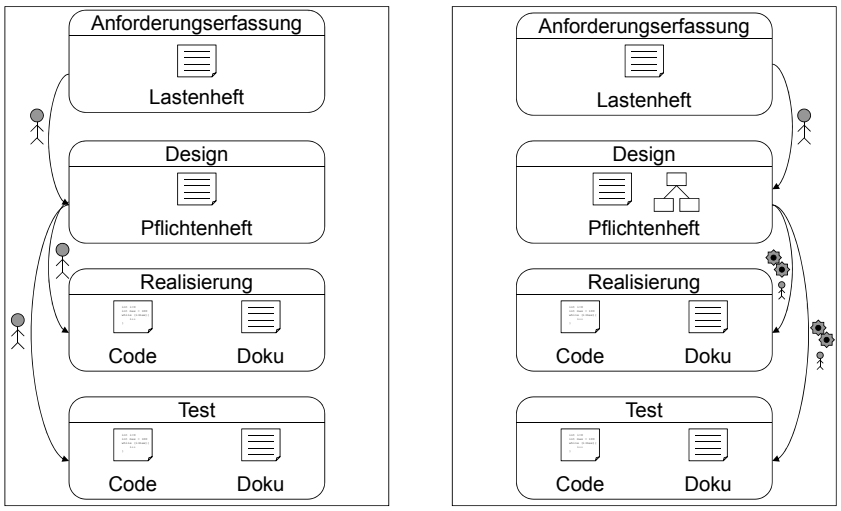
Definition 18 (Modell) *Die drei Hauptmerkmale des allgemeinen Modellbegriffs sind:*

- *Abbildungsmerkmal: Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.*
- *Verkürzungsmerkmal: Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant scheinen.*
- *Pragmatisches Merkmal: Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion*
  - *a) für bestimmte – erkennende und/oder handelnde, modellbenutzende – Subjekte,*
  - *b) innerhalb bestimmter Zeitintervalle und*
  - *c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen.*

[Sta73]

Modelle werden in der Softwareentwicklung in vielen Unternehmen schon eingesetzt [AP10, CFGK05]. Für Funktionen, vor allem in eingebetteten Systemen, werden vielerorts Matlab/Simulink-Modelle verwendet. Der Einsatz von Architektursprachen ist in der Industrie weniger verbreitet [VAC+09]. Wenn sie zum Einsatz kommen, dann werden sie in einer für das Unternehmen angepassten Form verwendet [MLM+13]. Architektursprachen können in der Designphase genutzt werden, um die Architektur eines Teilsystems oder des Gesamtsystems modellieren. Eine Reihe von Artefakten, von Zeichnungen bis hin zu Quellcode, werden als Modelle eingesetzt.

Wie der modellbasierte Ansatz sich auf die Entwicklung auswirkt, ist in Abbildung 7.1 gezeigt. Auf der linken Seite ist ein Entwicklungsprozess mit den vier Aktivitäten Anforderungserfassung, Design, Realisierung und Test abgebildet, die sich im Wesentlichen in jedem Vorgehensmodell wiederfinden. Auf der rechten Seite wird eine modellbasierte Vorgehensweise gezeigt, die vor allem beim Design und darauf folgenden Aktivitäten Einfluss hat. Das Design wird nicht in Form von natürlichsprachlichen Beschreibungen, sondern in Form von Modellen festgehalten. Aus diesen Modellen lässt sich dann sowohl Code für die Implementierung als auch für den Test automatisiert generieren. Der manuelle Aufwand ist deutlich kleiner und weniger fehleranfällig als bei der Entwicklung ohne Modelle.



(a) traditionell (b) modellbasiert

Abbildung 7.1: Modellbasierter Entwicklungsprozess

In einer stark kosten- und zeitplangetriebenen Entwicklung wird zunächst oft auf Modelle verzichtet, weil die Entwicklung entsprechender Generatoren zunächst schlecht auf diese beiden Faktoren wirkt. Dieses Vorgehen erweist sich aber als nicht optimal, wenn ein Produkt weiterentwickelt werden soll. Da nicht alle Entwurfsentscheidungen auf Code-Ebene nachzuvollziehen sind, muss zusätzlich Zeit investiert werden, um diese zu rekonstruieren. Code-Generatoren können das dazugehörige Wissen enthalten, so dass bei einer Überarbeitung diese Details dem jeweiligen Bearbeiter nicht bekannt sein müssen. Die Qualität einer Software hängt von der Modellqualität ab. Aus diesem Grund sollte diese mit entsprechenden Techniken gemessen und optimiert werden, damit schon früh im Entwicklungsprozess die Qualität der entstehenden Software sichergestellt werden kann [FHR08].

Ein weiterer Vorteil von Modellen ist, dass sie eine formale Beschreibung des Systems darstellen. Diese formale Beschreibung hat eine eindeutige Semantik, über die dann eine formale Verifikation erfolgen kann [JH05]. Eine solche Verifikation kann variable Eigenschaften der Semantik mitberücksichtigen [Grö10]. Auf Basis der formalen Beschreibung und der damit verbundenen Semantik können Modelle für eine Reihe von Aufgaben genutzt werden und es ergeben sich Möglichkeiten, die ohne Modelle nicht genutzt werden können. Beispielsweise kann mit einem Modell in einem frühen Stadium der Entwicklung das fertige System simuliert werden. Wenn diese Modelle in einem optimierten Entwicklungsprozess eingesetzt werden, können durch Konsistenzchecks schon früh eventuelle Fehler des zukünftigen Systems identifiziert und behoben werden. Je später diese Fehler erkannt werden, desto teurer wird deren Behebung [LL10]. Dieser Zusammenhang ist in Abbildung 7.2 dargestellt.

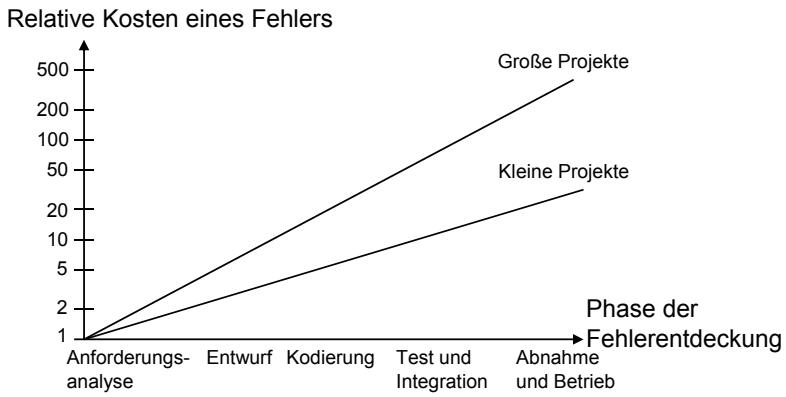


Abbildung 7.2: Relative Fehlerkosten im Verlauf des Entwicklungsprozesses

Mit domänenspezifischen Sprachen (DSLs) [MHS05] lassen sich gezielt Informationen einer bestimmten Domäne beschreiben. Zum Beispiel lässt sich mit einer DSL für ein Auto kein Haus modellieren. Den Gegensatz zu DSL stellen die GPLs (General Purpose Language) dar. Dies sind in allen Domänen bzw. Kontexten nutzbare Sprachen. Ein Beispiel hierfür stellt die Programmiersprache C dar, mit der sich sowohl Software für ein Auto als auch eine Gebäudeleittechnik programmieren lässt.

Als Sprache muss nicht nur Text verstanden werden, sondern es kann auch ein gezielt formatiertes Dokument (z.B. in DOORS) sein. Somit ergeben sich eine Reihe möglicher Modelle, die in einem Software-Entwicklungsprozess vorhanden sein können. Deren Potential wird aber nicht immer in vollem Umfang genutzt. Oft dienen sie nur zur Dokumentation des Systems aber nicht zur Entwicklung. Deshalb ist es für ein Produktlinien-Einführungsprojekt wichtig diese Modelle zu identifizieren und deren möglichen Nutzen zu evaluieren. Sind die Modelle identifiziert, kann durch die Definition von Variabilität in den Modellen ein Grundstein für eine Produktlinie geschaffen werden.

Die Bedeutung von Modellen in einer Produktlinie, vor allem einer modellgetriebenen Architektur, wird in [MA02] dargestellt. Viele Produktlinien-Ansätze lassen sich ohne Modelle nicht sinnvoll realisieren [Bra07]. In [KN06] wird darüber hinaus gezeigt, wie eine formale Verifikation ganzer Produktlinien erfolgen kann.

Sollen mehr Modelle im Entwicklungsprozess eingesetzt werden, ist es wichtig das Entwicklerteam mit der neuen Sicht vertraut zu machen und es für die Nutzung der neuen Möglichkeiten zu gewinnen. Ein erfolgversprechender Weg ist es, den bestehenden Entwicklungsprozess nur sanft zu modifizieren und um neue optionale Teile zu erweitern. Das kann beispielsweise so aussehen, dass für eine bisher händische Codierung ein Generator angeboten wird. So können sich Entwickler in der laufenden Entwicklung ein Bild von den neuen Möglichkeiten machen und akzeptieren diese eher, als wenn sie sofort auf die neue Technologie umsteigen müssen. Dieses Vorgehen wird durch die Betrachtungen

in [TH02] bestätigt.

Nach [GKRS06] müssen für die erfolgreiche Integration von Modellen in den Entwicklungsprozess drei Bedingungen erfüllt sein:

- Die Schnittstellen und die Semantik der Modelle müssen dem Entwickler bekannt sein.
- Entwicklungswerkzeuge müssen einen ähnlichen oder idealerweise besseren Komfort bieten als die bisher eingesetzten Werkzeuge.
- Es muss eine Produktivitätssteigerung erfolgen.

Unter diesen Aspekten wird im Folgenden die Nutzung der Modelle erweitert.

7.2 Methodik zur Identifikation von Modellen in einer Softwareproduktlinie

Auch wenn eine möglichst modellgetriebene Entwicklung bei der Einführung von Produktlinien hilfreich ist, ist die Entwicklung und Einführung von neuen Modellen nicht an allen Stellen sinnvoll. Falls durch Modellnutzung nur wenig Wissen abstrahiert wird und es für die Entwickler einfacher ist direkt den C-Code zu modifizieren, werden diese wenig Motivation haben, den bisher gelebten Prozess durch Einführung von Modellen zu ändern. Außerdem bedeutet die Entwicklung und Einführung neuer Modelle immer einen Aufwand, der sich in der Entwicklung auch rentieren muss. Die in Definition 18 aufgezählten Merkmale treffen auf viele Artefakte in einer Entwicklung zu. Für den in dieser Arbeit verwendeten Ansatz ist es deshalb zielführender, bereits implizit verwendete Modelle (z.B. als Dokumentation) zu identifizieren und für eine modellbasierte Entwicklung zu nutzen, als neue Modellierungssprachen einzuführen.

Im Rahmen des Projekts bei Volkswagen haben sich zur Identifikation der für den Entwicklungsprozess verwendbaren Modelle die folgenden Schritte herauskristallisiert, die in den folgenden Abschnitten näher beschrieben werden. Diese können auf alle Artefakte angewendet werden, die bisher im Entwicklungsprozess entstehen:

1. Metamodell dokumentieren
2. Artefakte identifizieren und Typ der Artefakte bestimmen
3. Modelleigenschaften der Artefakte untersuchen

4. optional: Nutzbarkeit der Modellartefakte auswerten

5. optional: Artefakte definieren

7.2.1 Metamodell dokumentieren

Um auf die vorhandenen Daten in Systemen zugreifen zu können, müssen diese sinnvoll abstrahiert und ein einfacher Zugriff realisiert werden. Dafür bietet es sich an, die Struktur des Quellsystems zu untersuchen und die möglichen Elemente zu dokumentieren. Die Ergebnisse werden in Metamodellen festgehalten, über die auf die Daten anschließend zugegriffen werden kann. Ist dieser Schritt nicht sinnvoll möglich, kann das betrachtete Artefakt nicht als Modell verwendet werden und es erübrigen sich die restlichen Schritte.

Metamodelle sind in der Softwaretechnik ein weit verbreitetes Mittel, um Modellierungssprachen genauer zu spezifizieren. Ein Metamodell zu einer Modellierungssprache beschreibt, welche Elemente in den Modellen vorkommen können. Typische Abstraktionen im Metamodell sind [SRVK10]:

- *Klassen* sind Entitätenklassen eines Systems oder einer Domäne. Entitäten sind instanziierte Klassen. Modelle und Modellelemente der Domäne sind Entitäten und können andere Entitäten enthalten. Die Klassen können Attribute definieren, die in den Entitäten einen Wert haben.

- *Assoziationen* stellen Verbindungen zwischen den Klassen dar.

- Die *Spezialisierung* ist eine Assoziation, die anzeigt, dass zwei Klassen verwandt sind („IS-A relation“).

- Durch eine *Hierarchie* können Klassen kompositional aufgebaut sein.

- Mit *Nebenbedingungen* können komplexere Eigenschaften festgelegt werden, die valide Modelle erfüllen müssen.

7.2.2 Artefakte identifizieren und Typ der Artefakte bestimmen

In der Lenkungsentwicklung bei Volkswagen hat sich das Anforderungsmanagementsystem als Quelle für mögliche vorhandene Modelle bei der Einführung einer Produktlinie herausgestellt. Textuelle Anforderungen eines Lastenheftes werden in Pflichtenheften und daraus abgeleiteten Dokumenten immer weiter verfeinert. Je weiter verfeinert wird,

desto mehr teilen sich die Dokumente in der Regel auf mehrere Dokumente auf. Letztendlich wird eine Ebene erreicht, in der ein abgeschlossener Sachverhalt beschrieben wird, der atomar für einen Entwickler erfassbar ist.

Wie diese Ebene oder das jeweilige Artefakt ausgestaltet ist, stellt sich unterschiedlich dar. Die Detaillierung kann weiterhin natürlichsprachliche Anforderungen enthalten, sehr formal einem bestimmten Aufbau folgen (z.B. durch Tabellen), aber auch eine beliebige Mischung aus beidem sein. Für die weitere Behandlung ist es sinnvoll die Artefakte, die einen gleichen Aufbau haben, einem selbstgewählten Typ zuzuordnen, der mehrere Artefakte mit ähnlichen Eigenschaften zusammenfasst und die im Entwicklungsprozess Input für den gleichen Prozess/das gleiche Werkzeug darstellt.

7.2.3 Modelleigenschaften der Artefakte untersuchen

Für jeden Typ wird anschließend untersucht, inwiefern die zugeordneten Artefakte einen Modellcharakter haben. Zu Modellen zählen zum Beispiel Anforderungsdokumente oder Teile von Anforderungsdokumenten, wenn diese die in Definition 18 aufgezählten Eigenschaften erfüllen. Eine als Anforderung definierte Kommunikationsmatrix beschreibt die Architektur eines Systems (Abbildungsmerkmal nach Definition 18). Da nur die Schnittstelle der enthaltenen Komponenten dargestellt wird, ist das Verkürzungsmerkmal (nach Definition 18) auch gegeben. Mit dieser Beschreibung einer Architektur kann diese einfacher erfasst werden und beispielsweise mit einem geeigneten Generator Quellcode automatisch erstellt werden (Pragmatisches Merkmal nach Definition 18).

In der Regel sind in einem Anforderungsmanagementsystem eine Vielzahl von Dokumenten enthalten. Deshalb sollte sich bei der Analyse vor allem auf die Dokumente konzentriert werden, die wenig natürlichsprachliche Anforderungen enthalten und viele atomare Eigenschaften für Anforderungsobjekte bereit halten. Artefakte mit viel natürlichsprachlichem Inhalt sind weniger als Modelle geeignet.

7.2.4 Nutzbarkeit der Modellartefakte auswerten

Ist ein Artefakt als verwendbares Modell identifiziert worden, wird dessen Nutzbarkeit evaluiert. Die Verwendung eines Modells mit einem oder mehreren dazu passenden Werkzeugen stellt das Ziel bei dem Modelleinsatz dar. Zum einen muss dazu die Qualität des Modells stimmen, zum anderen ein entsprechendes Werkzeug entwickelt oder angepasst werden. Dabei ist zu untersuchen, ob die Informationen des Modells schon verkürzt und aufs Wesentliche reduziert aufgeführt sind und falls nicht, ob dies sinnvoll möglich ist. Für die Erhöhung der Qualität und die Einführung eines Werkzeugs ist Aufwand erforderlich, der später in der Entwicklung idealerweise mehr als kompensiert wird. Ansonsten

lohnt sich ein Einsatz des Modells im Rahmen einer modellbasierten Entwicklung aus betriebswirtschaftlichen Gründen nicht.

Für die Einführung einer Produktlinie ist es besonders interessant, wenn in dem jeweiligen Modell Variabilität dargestellt ist. Dies unterstützt die Rentabilität der modellbasierten Entwicklung, da durch ein geeignetes Werkzeug die Möglichkeit besteht, die zusätzliche Komplexität besser beherrschbar zu machen. Generell muss bei der Abschätzung auch in Erwägung gezogen werden, welche Möglichkeiten in Zukunft auch mit entsprechenden Modellen bestehen, die den Entwicklungsprozess optimieren.

Für Werkzeuge gilt, dass Spezialwissen, welches nur projekt- oder unternehmensintern existiert, nicht in allgemein zugänglichen Werkzeugen vorhanden ist. Zudem müssen diese Werkzeuge auf die genutzten Modelle angepasst werden, sofern keine allgemein verfügbaren Formate genutzt werden. Für eine nachhaltige und zukunftsorientierte Nutzung von Modellen sollte es möglich sein, einfach neue Werkzeuge zu entwickeln. Deshalb wird in Kapitel 8 detaillierter auf Werkzeuge eingegangen.

7.2.5 Artefakte definieren

Erweist sich ein Artefakt als mögliches Modell, muss dieses über eine konkrete Syntax definiert werden. Diese ist die Repräsentation der Instanzen des im ersten Schritt aufgestellten Metamodells. Dadurch ergibt sich die Möglichkeit, die Modelle auf syntaktische Korrektheit zu überprüfen, um Fehler in Werkzeugen zu vermeiden oder ein Werkzeug einzuführen, was genau eine solche Prüfung ermöglicht.

Eine weitere Stufe stellt die Überprüfung der semantischen Korrektheit der Modelle dar. Hierbei sind Konventionen und Kontextbedingungen notwendig, die für die Modelle überprüft werden. Die Prüfung selber kann ebenfalls durch Werkzeuge erfolgen, die für diesen Zweck entwickelt wurden. Diese können dann automatisiert beispielsweise die Einhaltung von Kodierungsrichtlinien oder Verknüpfungen von Dokumenten prüfen.

Die Überprüfung der Korrektheit von Modellen sollte jederzeit möglich sein, ohne dass dabei Code generiert wird. Nur so wird die Konsistenzprüfung als leichtgewichtiger und hilfreicher Prozess verstanden. Jede Abweichung muss vom Entwickler selber behoben werden. Nur er hat die Kompetenz zu entscheiden, ob etwas wirklich falsch ist oder ob die Konsistenzprüfung fehlerhaft ist. Für einfache und automatisiert korrigierbare Fehler kann eine Hilfestellung gegeben werden, die letztendliche Entscheidung muss jedoch immer beim Entwickler liegen.

Für die Code-Generierung hat eine vorgeschaltete Überprüfung den Vorteil, dass die Quellartefakte schon einmal geprüft werden und potentielle Unklarheiten bei der Generierung nicht mehr auftreten können. Das macht auch die Entwicklung von Generatoren

einfacher, da mögliche Abweichungen nicht in jedem Generator separat behandelt werden müssen. Zudem sind Abweichungen, die erst im Rahmen der Codegenerierung erkannt werden, aufwendiger zu beheben, da Änderungen am Modell auch Auswirkungen auf weitere Teile des Entwicklungsprozesses haben können.

7.3 Anwendung der Methodik bei Volkswagen

In weiten Teilen der Automobilindustrie wird, wie auch in der Lenkungsentwicklung bei Volkswagen, das Anforderungsmanagementsystem DOORS eingesetzt. Im Folgenden wird beschrieben, wie mit der vorgestellten Methodik bei Volkswagen die in DOORS enthaltenen Modelle identifiziert wurden. Die dargestellten Konzepte lassen sich auf andere Systeme übertragen. Die Methodik ist nicht nur auf das Anforderungsmanagementsystem beschränkt, sondern kann auf alle Teile bzw. Artefakte im Entwicklungsprozess angewendet werden.

7.3.1 Metamodell dokumentieren

Das Werkzeug DOORS wird in den untersuchten Projekten des Produktlinien-Einführungsprojekts zum Management von Anforderungen eingesetzt. Für dieses Werkzeug wird im Folgenden ein Metamodell gezeigt, mit dem auf die enthaltenen Daten zugegriffen werden kann. Das dargestellte Metamodell beschränkt sich auf die für das Produktlinien-Einführungsprojekt wichtigen Bestandteile.

Item. Generell ist die Datenbank in DOORS wie eine Ordnerstruktur in einem Dateisystem organisiert. In Abbildung 7.3 ist die Struktur dargestellt. Die Elemente dieser Struktur sind *Item*s. Jedes Element hat eine eindeutige Identifikation (*uniqueID*), über die es Datenbankweit gezielt adressierbar ist. Dazu gibt es zu jedem Element einen Namen (*name*), eine Beschreibung (*description*) und eine URL (*url*) über welche des Element z.B. über die Eingabeaufforderung aufgerufen werden kann. Der vollständige Namen *fullName* eines Elementes kann über die Ordnerstruktur abgeleitet werden. Elemente können gelöscht sein, wobei DOORS zwischen „Soft-Delete“, bei dem ein Element nur als gelöscht gekennzeichnet wird, und „Hard-Delete“, bei dem es wirklich gelöscht wird und nicht mehr in der Datenstruktur vorhanden ist, unterscheidet.

Wie bei einem Dateisystem gibt es Dateien (*Module*s) und Ordner (*Folder*), die wiederum andere Dateien und Ordner enthalten können. Jedes *item* kann in einem Ordner liegen, der als *parentFolder* zugänglich ist. Eine Ausnahme ist die Wurzel der Ordnerstruktur, die als einziges Element keinen übergeordneten Ordner besitzt. Ordner können

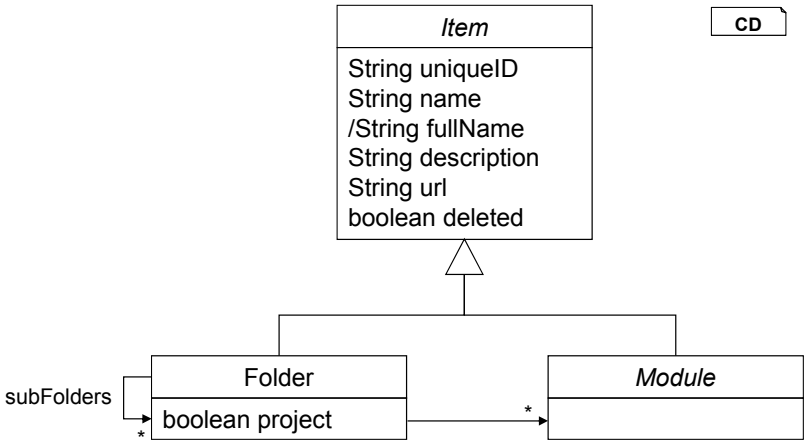


Abbildung 7.3: Übersicht über die Ordnerstruktur in DOORS

als Projekt gekennzeichnet sein, um Projekte in der Ordnerstruktur darzustellen. Sie sind so über ihren Namen direkt an der Wurzel aufgehängt, jedoch haben sie weiterhin einen anderen Ordner als Vaterknoten.

Module. *Module*s sind Basisobjekte für alle Module und beinhalten Informationen zu einem Modul, die versions- und typunabhängig gültig sind. In Abbildung 7.4 sind die Abhängigkeiten eines Moduls dargestellt. Jedes Modul kann mehrere festgelegte Versionsstände haben, die als *Baseline* bezeichnet werden. Diese hat jeweils eine Major-Version (*major*) und eine Minor-Version (*minor*). Zusammen mit einem Suffix (*suffix*) ergibt sich der Versions-String einer *Baseline*. So ist der Versions-String „1.0 Stand nach Freigabe“ der der Baseline mit der Major-Version „1“, der Minor-Version „0“ und dem Suffix „Stand nach Freigabe“.

Eine konkrete Version eines Moduls ergibt sich, wenn es mit einer bestimmten Baseline aufgerufen wird. Diese Instanz wird als *ModuleVersion* bezeichnet. Unabhängig von der Existenz der Baselines kann die aktuelle Version (*currentVersion*) des Moduls aufgerufen werden, die dem letzten Bearbeitungsstand entspricht. Konzepte, die Verzweigungen oder Zusammenführungen in der Versionshistorie erlauben, gibt es in DOORS nicht, so dass die aktuelle Version immer eindeutig ist.

ModuleVersion. Die Version (*version*) einer *ModuleVersion* lässt aus der zugehörigen Baseline ableiten. Es gibt drei verschiedene Arten von *ModuleVersion*:

- *FormalModule*: Diese Module enthalten eine Sammlung von Anforderungen. So kann sowohl ein Lastenheft für ein System in einem *FormalModule* abgelegt sein als auch die Spezifikation einer Software-Funktion.

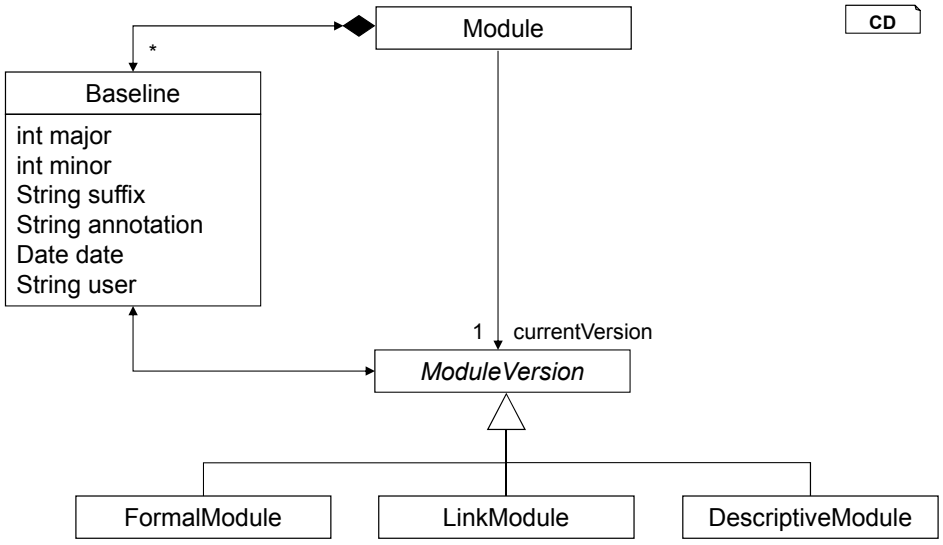


Abbildung 7.4: Übersicht über ein Modul in DOORS

- *LinkModule*: Diese Module enthalten Links zwischen einzelnen Anforderungen in den *FormalModule*s. Diese Links dienen beispielsweise dazu, Anforderungen für eine Spezifikation zu einer Anforderung in einem Lastenheft zurückzuverfolgen.

- *DescriptiveModule*: Diese Module können zur Ablage von Freitext-Informationen genutzt werden und werden im Allgemeinen nur sehr selten bis gar nicht verwendet. Im Rahmen des Produktlinien-Einführungsprojekts spielen sie keine Rolle und werden nicht weiter betrachtet.

Allen drei Arten ist der in Abbildung 7.5 dargestellte Aufbau gemein. Die Modul-Versionen sind mit einem Attribut- und Typ-System ausgestattet. In Modulen sind eine Reihe von Attributen (*Attribute*) vorhanden. Diese sind jeweils anhand ihres Namens für ein Modul eindeutig definiert. Sie sind mögliche Attribute für das *Module* (*moduleAttributes*) und/oder für die enthaltenen Anforderungsobjekte (*objectAttributes*). Alle Eigenschaften eines Moduls werden anhand ihres Namens in den *attributeValues* abgelegt. Diese *Values* ordnen jeweils einem Attribut einen Wert zu.

Jedes *Attribute* hat einen Namen, eine Beschreibung (*description*) und kann einen Standardwert (*defaultValue*) haben. Es gibt vordefinierte System-Attribute (*system*). Zudem gibt es die Möglichkeit, Attribute durch die in DOORS integrierte Erweiterungssprache DXL definieren zu lassen. Der notwendige Quellcode dafür ist in der *dxl*-Eigenschaft abgelegt.

Jedes Attribute ist einem Typ (*Type*) zugeordnet. Diese Typen sind ebenfalls modulweit vorhanden und sind anhand ihres Namens definiert. Sie haben einen Basistyp (*Base-*

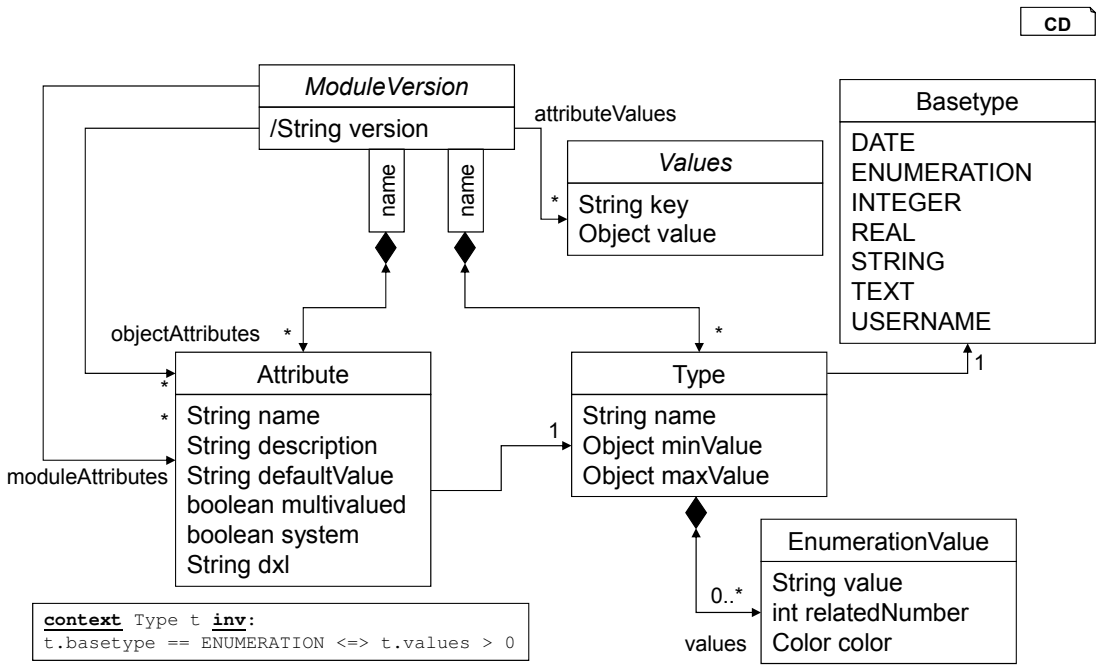


Abbildung 7.5: Übersicht über eine *ModuleVersion* in DOORS

Type), der Einfluss auf die mögliche Belegung von weiteren Werten hat. Diese sind in Tabelle 7.1 festgehalten. Für Zeichenketten sind die Basistypen *STRING* und *TEXT* vorgesehen. Bei den Basistypen für ein Datum (*DATE*) und für Zahlwerte (*INTEGER*, *REAL*) kann optional ein minimaler Wert (*minValue*) und ein maximaler Wert (*maxValue*) festgelegt werden. Im Falle des Basistyps einer Aufzählung (*ENUMERATION*) sind mögliche Werte des Typs definiert. Diese werden als eine Reihe von *EnumerationValue*s abgelegt. Jede hat jeweils einen eindeutigen Wert (*value*), eine zugeordnete Nummer (*relatedNumber*), die für zwei *EnumerationValue*s identisch sein kann, und optional eine von 36 vordefinierten Farben. Benutzernamen haben einen eigenen Basistyp (*USERNAME*). Attribute mit diesem Typ repräsentieren DOORS-Benutzer.

FormalModule. Das Element zur Definition von Anforderungssammlungen ist das *FormalModule*. Der Aufbau ist in Abbildung 7.6 dargestellt. Es enthält eine Reihe von Anforderungsobjekten (*ReqObject*). Ein Anforderungsobjekt ist eindeutig durch seine Nummer (*absNo*) innerhalb des Moduls identifizierbar. Zudem kann es über eine eigene URL (*url*) wie *Item*s referenziert werden. Die Werte der für die im Modul für Objekte festgelegten Attribute sind in Zuordnung *attributeValues* abgelegt. Ein Anforderungsobjekt kann wie ein Modul gelöscht sein (*deleted*) und es gibt ebenfalls die Unterscheidung zwischen „Soft-Delete“ und „Hard-Delete“.

Das erste Objekt (*first*) bzw. die Objekte der obersten Hierarchiestufe (*top*) sind gezielt

Tabelle 7.1: Mögliche Werte von Typen in Abhängigkeit des Basistyps

BaseType	minValue	maxValue	enumValues
DATE	optional	optional	nicht möglich
ENUMERATION	nicht möglich	nicht möglich	verpflichtend
INTEGER	optional	optional	nicht möglich
REAL	optional	optional	nicht möglich
STRING	nicht möglich	nicht möglich	nicht möglich
TEXT	nicht möglich	nicht möglich	nicht möglich
USERNAME	nicht möglich	nicht möglich	nicht möglich

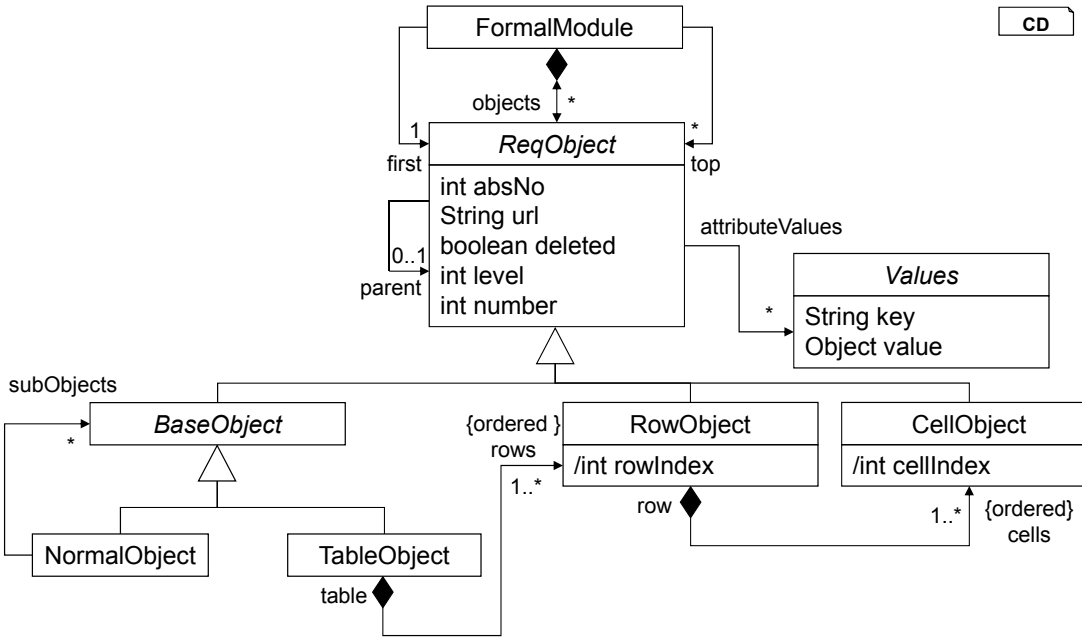


Abbildung 7.6: Übersicht über ein Formalmodul in DOORS

ansprechbar. Die Hierarchie wird so abgebildet, dass Objekte teilweise wieder selber Objekte enthalten können. Somit kann für jedes Anforderungsobjekt die Hierarchieebene (*level*) und die dazugehörige Kapitelnummer (*number*) abgeleitet werden. Die Hierarchie in DOORS ist in Abbildung 7.7 dargestellt, die beispielhaft ein Formalmodul zeigt. Es können zudem alle Objekte eines Formalmoduls in Form einer Liste (*objects*) ohne Beibehaltung der Hierarchie abgefragt werden.

Über die Hierarchie lassen sich vier verschiedene Objekt-Arten identifizieren. *Normal-Objects* sind normale Anforderungsobjekte, die keine speziellen Eigenschaften haben. Für die Darstellung von Tabellen in DOORS existieren *TableObject*s. Da beide als

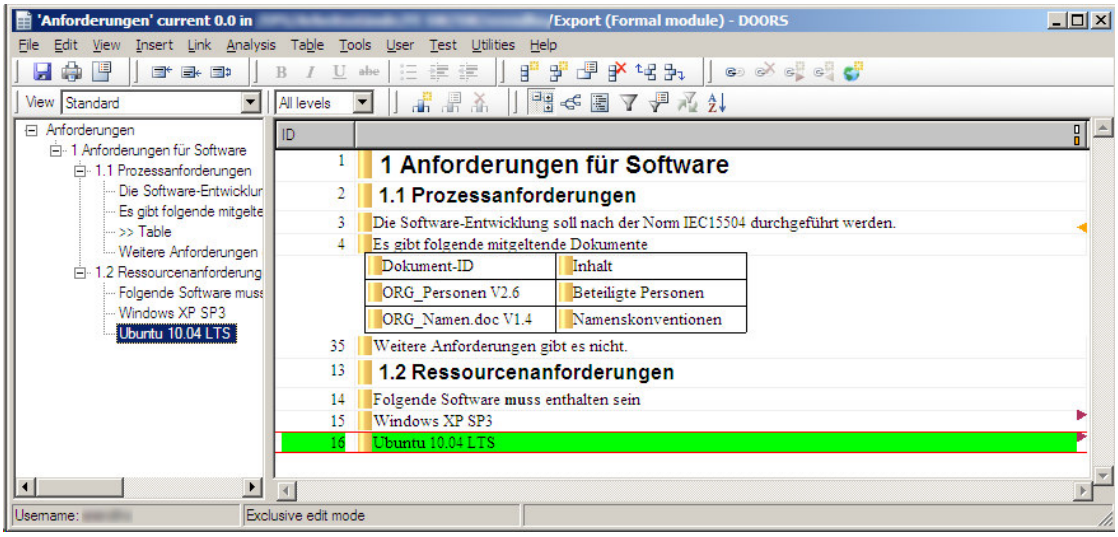


Abbildung 7.7: Beispiel für ein DOORS-Formalmodul

untergeordnete Objekte (*subObjects*) von ersterem existieren können, werden diese als Basis-Objekte (*BaseObjects*) betrachtet. In einem Tabellen-Objekt sind mehrere Reihen-Objekte (*RowObject*) enthalten. Diese enthalten wiederum Zellen-Objekte (*CellObject*), die die einzelnen Zellen einer Reihe darstellen.

View. Für die Anzeige eine Formalmoduls werden in DOORS Sichten (*Views*, siehe Abbildung 7.8) verwendet. Sichten sind streng genommen kein Bestandteil des Datenmodells, sondern waren ursprünglich nur zur Anzeige von Informationen gedacht. Da diese aber oft dazu verwendet werden, einen bestimmten Datenstand zu kennzeichnen, sind sie selbst bedeutungstragend und wurden deshalb mit in diese Betrachtung aufgenommen. Eine Sicht hat einen eindeutigen Namen (*name*), Spalten (*Column*) und kann einen Filter (*filter*) besitzen, der bestimmte Objekte von der Ansicht ein- oder ausschließt. Daneben können weitere Anzeigeoptionen konfiguriert werden, die jedoch im Rahmen der Entwicklungsprojekte des Produktlinien-Einführungsprojekts selten verwendet werden und deshalb hier nicht betrachtet werden.

In einer Sicht können in der benutzten DOORS-Version mindestens eine bis maximal 64 Spalten definiert werden. Ein Modul kann beliebig viele Sichten enthalten und mindestens immer die vordefinierte Ansicht, die aus zwei Spalten, eine mit Object Identifier und eine mit Object Heading bzw. Object Text, besteht. Es kann ein Standard-Sicht für ein Modul als auch für den aktuellen Nutzer festgelegt werden. Jede Spalte hat außer der abgeleiteten Nummer (*number*) noch einen Spaltentitel (*title*). Es gibt drei verschiedene Spaltentypen in DOORS. Die Spalte kann ein Attribut des jeweiligen Objektes anzeigen (*AttributeColumn*). Dieser Spaltentyp enthält eine Referenz auf das anzuzeigende Attri-

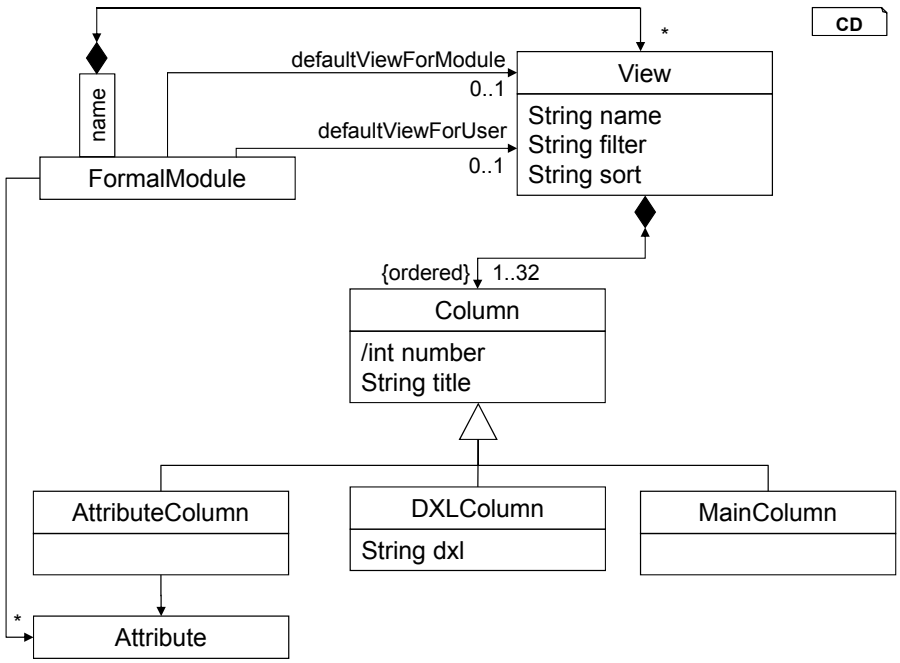


Abbildung 7.8: Übersicht über eine Sicht in DOORS

but. Eine weitere Möglichkeit ist die Anzeige von beispielsweise verlinkten Attributen durch ein DXL-Skript (*DXLColumn*). Hier ist das für die Anzeige notwendige Skript als Information hinterlegt (*dxl*). Der dritte mögliche Typ *MainColumn* ermöglicht die Anzeige von Object Heading bzw. Object Text in der zugehörigen Formatierung (z.B. Kapitelnummern). Dieser Spaltentyp wird auch in der vordefinierten Sicht verwendet.

FormattedText. Die Werte für Attribute von Objekten der Basistypen *STRING* oder *TEXT* können formatiert sein. Auf diese Weise wird auch die Ablage von OLE-Objekten in DOORS ermöglicht. Eine Übersicht über die Möglichkeiten der Textformatierung ist in Abbildung 7.9 gegeben.

Formatierter Text (*FormattedText*) enthält mindestens einen bis beliebig viele Absätze (*FormattedParagraph*). Ein Absatz kann Teil einer Aufzählung sein, was mit dem Attribute *bullet* angezeigt wird. Jeder Absatz besteht aus einem oder mehreren Fragmenten (*FormattedFragment*). Ein Fragment kann Text (*text*) oder OLE-Objekte beinhalten, was anhand des Attributs *ole* unterschieden wird. Die Formatierungsmöglichkeiten für den Text sind Fettschrift (*bold*), Kursivschrift (*italic*), Unterstreichung (*underline*), Durchstreichung (*strike*), Hochstellung (*superscript*) und Tiefstellung (*subscript*). Dem Text kann ein Zeilenumbruch folgen (*newline*) oder er kann eine URL repräsentieren (*url*). Aus diesen Informationen kann der Inhalt für beliebige Zielsprachen (z.B. HTML) aufbereitet werden.

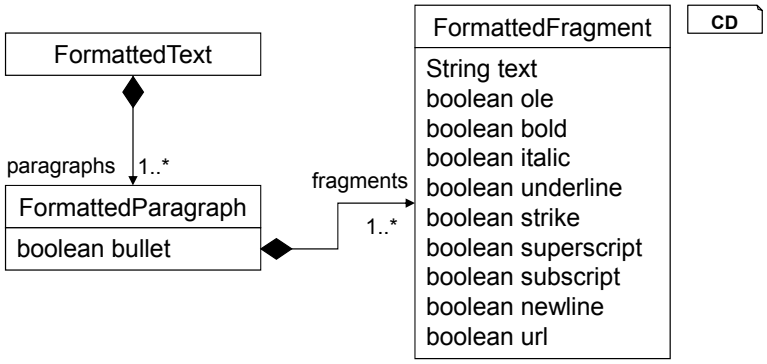
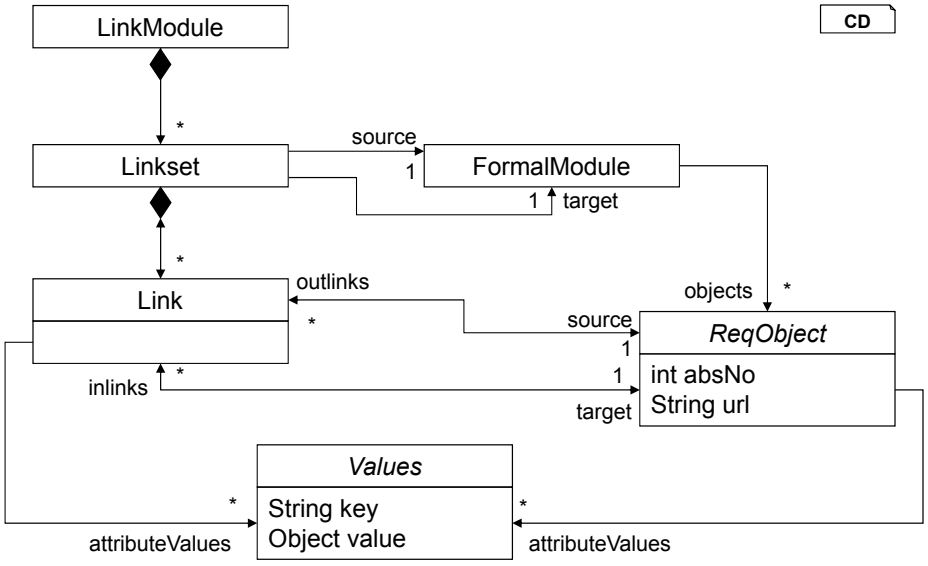


Abbildung 7.9: Übersicht über die Textformatierung in DOORS

LinkModule. Ein wesentliches Konzept in DOORS sind Links. Links verknüpfen gerichtet zwei Anforderungsobjekte. Diese werden, wie in Abbildung 7.10 gezeigt, in Linkmodulen (*LinkModule*) gespeichert. Die Art der Verknüpfung, z.B. „erfüllt“ oder „testet“ kann durch verschiedene Linkmodule frei definiert werden. Innerhalb eines Linkmoduls werden die Links in *Linksets* organisiert. Linksets haben genau ein Quellmodul *source* und ein Zielmodul *target*. Für jeden Link im Linkset gibt es deshalb ein Quellobjekt aus dem Quellmodul (*source*) und ein Zielobjekt aus dem Zielmodul (*target*). Ein Link kann wie Objekte spezifische Attribute haben (*attributeValues*).




```
context ReqObject r inv:
r instanceof  NormalObject || r instanceof CellObject
```

```
context Link l inv:
l.source.formalmodule == l.linkset.source && l.target.formalmodule == l.linkset.target
```

Abbildung 7.10: Übersicht über Linkmodul in DOORS

7.3.2 Artefakte identifizieren und Typ der Artefakte bestimmen

In DOORS lassen sich eine ganze Reihe an unterschiedlichen Typen von Artefakten finden. Über alle Lenkungsprojekte hinweg wurden die Formalmodule untersucht und die Ergebnisse sind in Tabelle 7.2 festgehalten. Von oben nach unten steigen in den Dokumenten die Detaillierung und der Fokus auf bestimmte Eigenschaften. Das spiegelt sich auch im Inhalt wider, der zunehmend formaler wird.

Tabelle 7.2: Artefakte in DOORS

Typ	Inhalt	Beispiele
Anforderungssammlung	Text	Lastenhefte, Pflichtenhefte
Architektur	Text, Tabellen	Systemarchitektur, Softwarearchitektur
Konzept	Text, Tabellen	Unit-Konzepte
Spezifikation	Text, Tabellen	Unit-Spezifikationen
Datentyplexikon	fester Aufbau	Datentyplexika für mehrere Softwarestände
Parameterlexikon	fester Aufbau	Parameterelexika für mehrere Softwarestände
Signallexikon	fester Aufbau	Signallexika für mehrere Softwarestände
Unitschnittstellen	fester Aufbau	Schnittstellenlexika für mehrere Softwarestände
Bussignalschnittstelle	fester Aufbau	Diagnoseschnittstellenlexika für mehrere Softwarestände
Diagnoseschnittstelle	fester Aufbau	Diagnoseschnittstellenlexika für mehrere Softwarestände
Testparameterlexikon	fester Aufbau	Testparameterlexika für mehrere Softwarestände

In den folgenden Abschnitten werden Beispiele gezeigt, wie die Modelle für die Inhalte aussehen und welche Informationen dort abgelegt werden sollen. Dabei wird vor allem auf die Links zwischen den Anforderungsobjekten eingegangen, da mit diesen die einzelnen Modelle verknüpft werden. Aus den Erkenntnissen wird abschließend ein allgemeines Modell für Objektstrukturen abgeleitet.

Lastenheft und Pflichtenheft

Bei diesem Beispiel für Artefakte mit rein textuellem Inhalt aus dem Kontext des Projekts bei Volkswagen gibt es zwei Formalmodule. Eines ist ein Lastenheft, welches beispielsweise von einem Kunden geliefert wurde, und eines ist ein Pflichtenheft, welches die Umsetzung der Anforderungen beschreibt. Für diese Art der Informationen ist DOORS ursprünglich konzipiert worden. Ein typisches Modell für diese Inhalte mit natürlichsprachlichen Text ist in Abbildung 7.11 dargestellt. Prinzipiell lassen sich mehrere Lastenhefte und Pflichtenhefte verknüpfen, die gezeigte Darstellung stellt einen Ausschnitt dar.

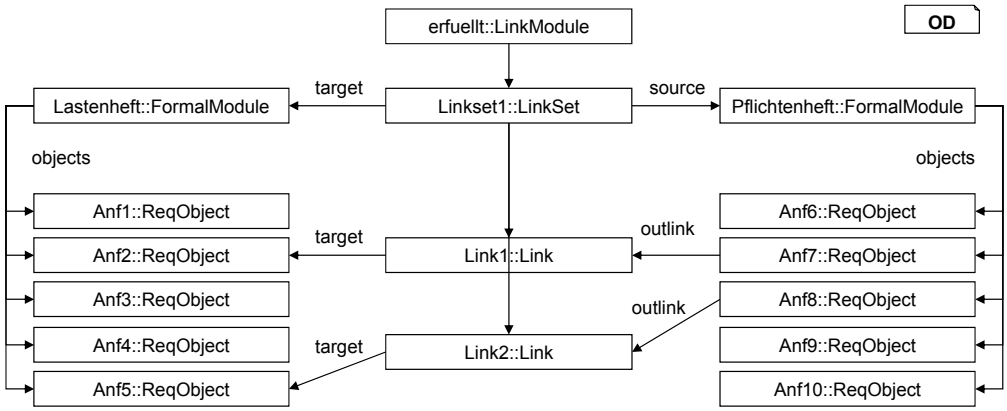


Abbildung 7.11: Lastenheft und Pflichtenheft in DOORS

Auf der linken Seite ist ein Lastenheft als Formalmodul modelliert, welches in diesem Fall fünf Anforderungen als Anforderungsobjekte hat (Anf1-Anf5). Das zugehörige Pflichtenheft als Formalmodul befindet sich auf der rechten Seite der Darstellung. Es hat ebenfalls fünf Anforderungen als Anforderungsobjekte (Anf6–Anf10).

Um Lastenheft und Pflichtenheft miteinander in Beziehung zu setzen, werden Links genutzt. Dazu gibt es oben in der Mitte ein Linkmodul. Die Links in diesem Modul zeigen an, welche Anforderungen des Pflichtenhefts welche Anforderungen des Lastenhefts erfüllen. Dafür gibt es im Linkmodul ein Linkset, welches alle Links vom Pflichtenheft in das Lastenheft enthält. Dieses Linkset enthält im vorliegen Fall zwei Links. Anf7 des Pflichtenheftes erfüllt Anf2 im Lastenheft und Anf8 erfüllt Anf5.

Architektur

Als Beispiel für ein Artefakt mit Text und Tabellen wird im Folgenden ein Architekturdokument beschrieben. Ein Architekturdokument (Abbildung 7.12) besteht zum Teil

aus natürlichsprachlichen Text als auch aus formeller aufgebauten Inhalten. Der obere Teil der Abbildung gleicht der Beziehung zwischen einem Lastenheft und einem Pflichtenheft. Im unteren Teil wird durch die Objekt-Struktur (Anf3–Anf5) in DOORS eine Tabelle erzeugt, deren Inhalt mit einem Parameterlexikon bzw. dem Parameter Par1 verknüpft ist. Damit wird zum Beispiel in der Architektur beschrieben, dass für ein Architekturelement ein definierter Parametersatz verwendet wird.

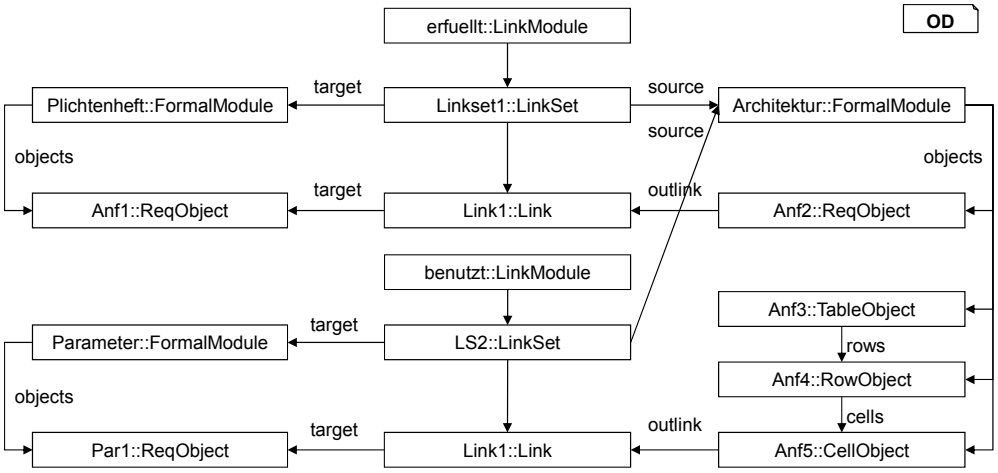


Abbildung 7.12: Architektur in DOORS

Parameterlexikon und Datentyplexikon

Ein Modell mit festem Aufbau stellt das Parameterlexikon mit Verknüpfung zum Datentyplexikon dar. Das Modell dazu ist in Abbildung 7.13 abgebildet.

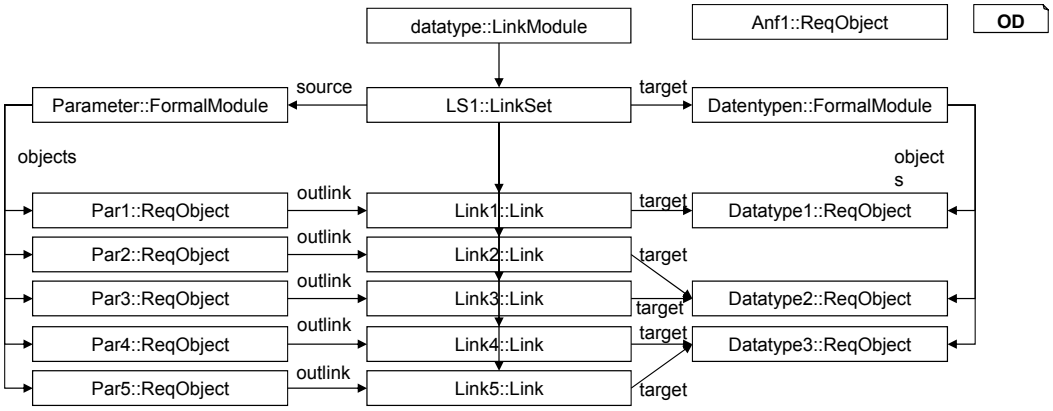


Abbildung 7.13: Parameterlexikon in DOORS

Auf der linken Seite ist ein Formalmodul mit fünf Parametern abgebildet. Dazu gibt es

auf der rechten Seiten ein Formalmodul mit drei Datentypen. Wie beim Lasten- und Pflichtenheft werden die Anforderungsobjekte beider Formalmodule über ein Linkmodul in Beziehung gesetzt. Dabei werden benutzte Datentypen nur einmal definiert und können dann für mehrere Parameter als Datentyp dienen.

Der große Unterschied ist, dass die Anforderungsobjekte nicht Text als Information besitzen, sondern mehrere Informationen dort modelliert werden. Wie die Informationen zu einem Parameter bzw. zu einem Datentyp abgelegt werden, ist in Abbildung 7.14 gezeigt. Das Anforderungsobjekt „Parameter1“ hat in diesem Fall den Namen Gainfactor (Verstärkungsfaktor), der in einer Berechnung mit einem anderen Wert multipliziert werden kann. Der physikalische Wert ist 0.5. Als Datentyp ist das Anforderungsobjekt „Datatype1“ zugeordnet. Dieses definiert einen im eingebetteten Bereich überwiegend verwendeten Fixpunkt-Datentyp, welcher als Basistyp einen 16-Bit Integer mit Vorzeichen besitzt. Mit dem Slope-Factor wird der Integer-Bereich durch Multiplikation auf den physikalischen umgerechnet.

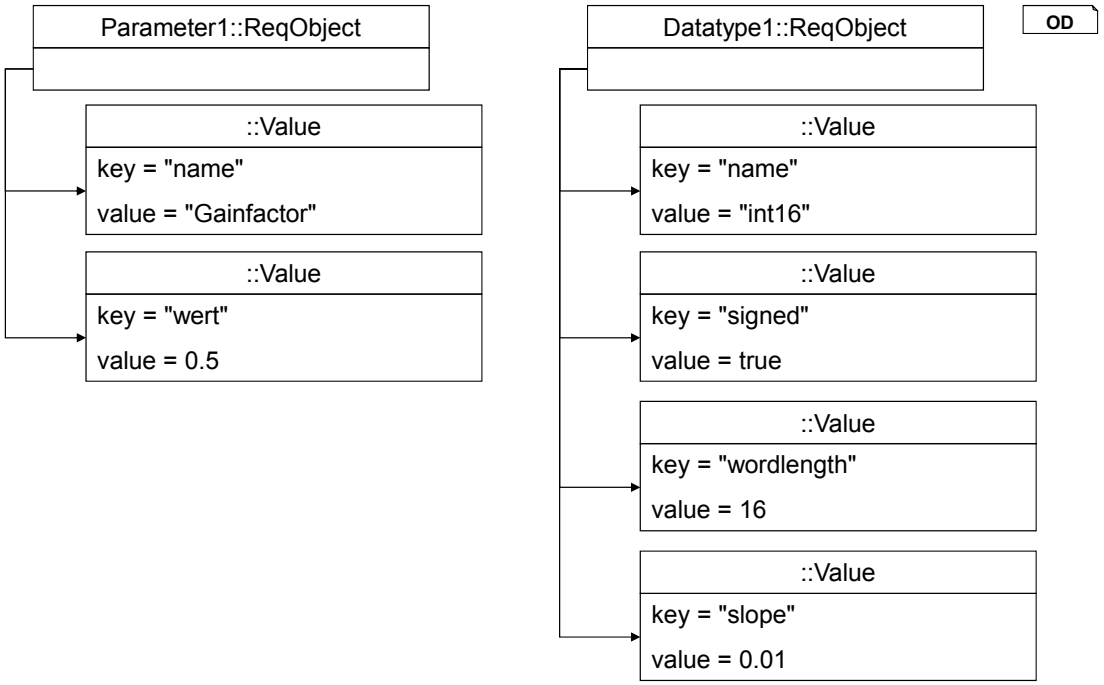


Abbildung 7.14: Informationen in den Anforderungsobjekten im Parameterlexikon

Die gezeigte Darstellung lässt sich auch auf Signale erweitern, die in einem eigenen Formalmodul modelliert sind. Darauf aufbauend lässt sich eine Schnittstellenbeschreibung aufbauen, in denen Signale und Parameter zu Komponenten zusammengefasst werden. Dies ist ein Beispiel von Daten, die bereits sehr formal in DOORS abgelegt sind und sich ideal für Generatoren nutzen lassen.

7.3.3 Modelleigenschaften der Artefakte untersuchen

Nachdem mögliche Kandidaten für die Nutzung als Modell im Entwicklungsprozess identifiziert wurden, wird über die Modelleigenschaften genauer untersucht, ob es sich wirklich um Modelle handelt. In Tabelle 7.3 ist das Ergebnis festgehalten. Es stellt sich heraus, dass besonders die Artefakte mit einem feinen Detaillierungsgrad und formellem Aufbau sich gut für eine systematische Nutzung als Modell eignen.

Tabelle 7.3: Untersuchung der Artefakte auf Modelleigenschaften

Typ	Merkmal			Fazit
	Abbildung	Verkürzung	Pragmatik	
Anforderungssammlung	ja	nein	nein	Nutzung als Modell aufwändig
Architektur	ja	teilweise	ja	Nutzung als Modell mit Einschränkungen möglich
Konzept	ja	teilweise	ja	Nutzung als Modell mit Einschränkungen möglich
Spezifikation	ja	teilweise	ja	Nutzung als Modell mit Einschränkungen möglich
Datentyplexikon	ja	ja	ja	Nutzung als Modell möglich
Parameterlexikon	ja	ja	ja	Nutzung als Modell möglich
Signallexikon	ja	ja	ja	Nutzung als Modell möglich
Unitschnittstellen	ja	ja	ja	Nutzung als Modell möglich
Bussignalschnittstelle	ja	ja	ja	Nutzung als Modell möglich
Diagnoseschnittstelle	ja	ja	ja	Nutzung als Modell möglich
Testparameterlexikon	ja	ja	ja	Nutzung als Modell möglich

7.3.4 Nutzbarkeit der Modellartefakte auswerten

Der Nutzen als Modell durch entsprechende Werkzeuge ist am ehesten bei den Lexika gegeben, da sie am ehesten den Modellmerkmalen entsprechen. Die Möglichkeit der Code-Generierung wurde für das Parameterlexikon schon genutzt. Zudem bietet sich an, Entwicklungsumgebungen wie Matlab/Simulink zu initialisieren.

Im Gespräch mit den Entwicklern wurde klar, dass aus den Lexika weitere Artefakte (außerhalb von DOORS) abgeleitet werden. Für die Datentypen und Signale können

Code-Generatoren entwickelt werden. Aus der Unitschnittstellenbeschreibung lässt sich sowohl Code für die Architektur als auch für die Entwicklungs- und Testumgebung auf einfache Weise erstellen, was bisher händisch gemacht wird. Die Testparameter aus dem Testparameterlexikon sind ebenfalls über einen Generator einfach in Testcode überführbar.

Die Entwickler äußerten ebenfalls den Wunsch, die Kommunikation mit den Kunden (also der Marke, die das Gesamtfahrzeug entwickelt) zu vereinfachen. Ansatzpunkte waren durch die Busschnittstelle des Lenksystems gegeben, welches regelmäßig mit der Busschnittstelle des Gesamtfahrzeugs abgeglichen werden muss, und durch die Diagnoseschnittstelle, die über ein festgelegtes Format an die Gesamtfahrzeugentwickler zurückgemeldet wird. Die Ergebnisse der Analyse sind in der Tabelle 7.4 zusammengefasst.

Tabelle 7.4: Nutzung der Artefakte als Modell

Typ	Nutzen	Aufwand
Anforderungssammlung	nein	-
Architektur	nein	-
Konzept	nein	-
Spezifikation	nein	-
Datentyplexikon	Code-Generierung für Datentypen	gering
Parameterlexikon	Code-Generierung für Parameter	schon vorhanden
Signallexikon	Code-Generierung für Signale	gering
Unitschnittstellen	Generierung von Architektur-Code, Konfiguration der Entwicklungs- und Testumgebung	mittel
Bussschnittstelle	Abgleich mit aktualisierter Fahrzeugbusschnittstelle	mittel
Diagnoseschnittstelle	Generierung/Anpassung des Austauschformats ODX	mittel
Testparameterlexikon	Generierung von Test-Code	mittel

Hinsichtlich der Variabilität wurde festgestellt, dass sich diese quer durch alle untersuchten Artefakte zieht. Die Anforderungssammlungen beschreiben sind teils produktspezifisch, teils projektspezifisch und decken einen unterschiedlichen Umfang an Varianten ab. Ebenso verhält es sich bei den Lexika und Schnittstellen. Bei diesen lassen sich über mehrere Produkte variable Anteile jedoch besser identifizieren und nutzen, da in diesen Modellen sich nur bestimmte Informationen ändern.

7.3.5 Artefakte definieren

Für die Lexika wurde sich von den Entwicklern gewünscht, dass falsche Informationen schon während der Modellierung in DOORS erkannt werden. Im Entwicklungsprozess tauchten immer wieder Abweichungen auf, die in den Lexika Änderungen notwendig machten. Dies war auch der Hauptgrund, weshalb die Code-Generierung noch nicht alle Lexika mit einschloss.

Deshalb lag in diesem Schritt ein besonderer Fokus darauf, festzulegen, wie bestimmte Sachverhalte im Modell beschrieben werden:

- Vermeidung redundanter Informationen: Da die Lexika voneinander abhängig sind (das Parameterlexikon bezieht seine Datentypen aus dem Datentyplexikon), müssen die Informationen über alle Lexika gleich sein. Es muss vermieden werden, dass eine Wortlänge (Anzahl an Bits) für einen Datentyp im Datentyplexikon anders definiert ist als im Parameter, der den Datentyp verwendet.

- Atomare Informationen verfügbar machen: Im Rahmen der Verarbeitung eines Modells ist es immer schwierig, wenn zu prüfende Informationen nicht atomar vorliegen. Basistypen der Datentypen werden beispielsweise gerne in der Form `SINT16` verkürzt, um anzuzeigen, dass es sich um einen Integer-Datentyp mit der Wortlänge 16 mit Vorzeichen handelt. Bei einer Überprüfung oder Generierung kann die Extraktion von Wortlänge und Vorzeichen zu Fehlern führen. Deshalb ist es notwendig sowohl die Wortlänge 16 als auch das Vorzeichen atomar im Modell vorzuhalten.

- Festlegen von Konventionen und Kontextbedingungen: In der Implementierung genutzte Namen müssen laut den Projektrichtlinien nur bestimmten Konventionen folgen. Die Lexika ermöglichen eine Reihe von semantischen Fehlern, die überprüft werden müssen. Zum Beispiel ist durch den verwendeten Datentyp das Maximum und das Minimum eines Parameterwertes festgelegt. Der Initialwert muss ebenfalls in diesem Bereich liegen. Bedingt durch die Verwendung von Fixpunkt-Datentypen sind auch die darstellbaren Werte definiert.

Nachdem das Metamodell für die einzelnen Modelltypen definiert ist, müssen diese gemäß der festgelegten Definition angepasst werden. Idealerweise wird parallel das Werkzeug entwickelt, welches die Modelle bezüglich der Definition prüft. So kann das Werkzeug auf die Modelle für den Test zurückgreifen. Auf Werkzeuge wird detailliert in Kapitel 8 eingegangen.

7.4 Definition von Variabilität in Modellen

In der Analyse der bestehenden Matlab/Simulink-Modelle bei Volkswagen wurde in Abschnitt 4.4 Variabilität in diesen identifiziert. Für eine Produktlinie müssen diese Modelle so erweitert werden, dass sie variable Anteile explizit darstellen können (Maßnahme M9, Festlegung der Variabilitätsrealisierung). Matlab/Simulink bietet bereits mehrere Konzepte für die Modellierung von Variabilität [WM14], die jeweils mit 150%-Modellen arbeiten (negative Variabilität). Mit dem Delta-Modeling-Ansatz [CHS10, SBB+10] gibt es ein Konzept, transformationale Variabilität in Modellen zu definieren. Beide Konzepte wurden für die Variabilitätsdarstellung in Architektur-Modellen untersucht. Zum Zeitpunkt der Untersuchung gab es keine Implementierung für Deltas in Matlab/Simulink, weshalb für den Vergleich das konzeptionell ähnliche MontiArc [HRR12] verwendet wurde. Textuelle Definitionsmöglichkeiten für Variabilität außerhalb von Matlab/Simulink wurden in [ES13] verglichen.

In den folgenden Abschnitten wird dargestellt, wie MontiArc mit 150%-Modellen (MontiArc150%, Abschnitt 7.4.1) und wie MontiArc mit dem Delta-Ansatz (Δ-MontiArc, Abschnitt 7.4.2) genutzt wird. Anschließend werden in Abschnitt 7.4.3 beide Modellierungstechniken anhand eines Beispiels verglichen und in Abschnitt 7.4.4 eine Entscheidung für das Projekt abgeleitet.

7.4.1 MontiArc für 150%-Modelle

MontiArc erlaubt die textuelle Modellierung von Architekturen mittels (kompositional aufgebauten) Komponenten. Komponenten besitzen typisierte Ports, zwischen denen Konnektoren definiert werden. Damit lässt sich ein System in beliebiger Detailtiefe modellieren. Ein Beispiel für eine MontiArc-Komponente ist in Listing 7.1 gegeben.

MA

```
1 package wipe;
2
3 component IntervalControl {
4     autoconnect port;
5     port
6         in IntervalSelection,
7         in VehicleSpeed vs,
8         out WipeCmd;
9     component IntervalCmdProcessor icp;
10    connect vs -> icp.VehicleSpeed;
11 }
```

Listing 7.1: MontiArc-Beschreibung der Komponente `IntervalControl` für eine Scheibenwischersteuerung

MontiArc-Komponenten sind wie Java in Packages organisiert (Zeile 1). Die Zeilen 3 bis 11 definieren die Komponente. Zunächst wird in Zeile 4 die `autoconnect`-Funktion aktiviert, die bewirkt, dass Ports mit gleichen Namen und kompatiblen Typen automatisch verbunden sind. Die Portdefinitionen der Komponente sind in den Zeilen 5 bis 8 zu sehen. Es werden zwei Ports als eingehende Ports definiert (Zeile 6, 7), von denen einer den Typ `IntervalSelection` und der andere den Typ `VehicleSpeed` hat und dazu als `vs` benannt ist. Als ausgehender Port der Komponenten existiert ein Port mit dem Typ WipeCmd (Zeile 8). Die Komponente enthält die außerhalb von `IntervalControl` definierte Komponente `IntervalCmdProcessor` (Zeile 9). Zusätzlich zu den automatisch erstellten Konnektoren ist der Konnektor vom Eingangsport `vs` zum einem Eingangsport der inneren Komponente modelliert (Zeile 10). Zur besseren Übersicht ist in Abbildung 7.15 eine grafische Repräsentation des Modells dargestellt.

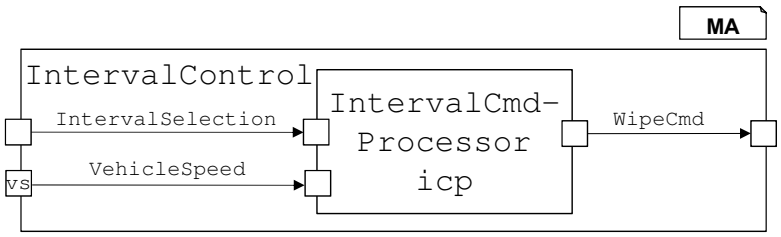


Abbildung 7.15: Grafische Repräsentation der Komponente `IntervalControl`

Für die Modellierung als 150%-Modell wurden die in MontiArc vorhandenen Annotationen genutzt. Ein Beispiel dafür ist in Listing 7.2 auf Basis des Listings 7.1 zu sehen. Es ergeben sich unterschiedliche Varianten, je nachdem ob das Feature `Rainsensor` ausgewählt ist oder nicht. Über das Feature wird eine Variante des Systems konfiguriert, welches eine Scheibenwischersteuerung auf Basis der Regenintensität enthält. In Zeile 8 wird bei Auswahl des `Rainsensor`-Features ein Eingangsport mit dem Typ `RainSensorStat` hinzugefügt. Ebenso gibt es eine neue Komponente, die den neuen Eingang benutzt (Zeile 11). Die zusätzlichen Verbindungen werden automatisch über die `autoconnect`-Anweisung hergestellt.

Nicht dargestellt sind die Annotationen im Modell von `IntervalCmdProcessor`, damit abhängig vom Feature `Rainsensor` der Port für `RainIntensity` vorhanden ist. In Listing 7.3 bzw. Abbildung 7.16 wird das Modell nach Auflösung der Variabilität dargestellt.

7.4.2 DeltaMontiArc

Bei der transformationalen Variabilitätsmodellierung mit dem Delta-Modeling-Ansatz werden bestehende Modelle nicht modifiziert, sondern durch sogenannte Deltas im Rahmen der Auflösung der Variabilität transformiert. Dies hat den Vorteil, dass sich Va-

```
MA
1 package wipe;
2
3 component IntervalControlWithRainSensor {
4     autoconnect port;
5     port
6         in IntervalSelection,
7         in VehicleSpeed vs,
8         <<variant = "RainSensor">> in RainSensorStat,
9         out WipeCmd;
10    component IntervalCmdProcessor icp,
11              <<variant = "RainSensor">> RainEval;
12    connect vs -> icp.VehicleSpeed;
13 }
```

Listing 7.2: Beispiel für annotative Modellierung von Variabilität in MontiArc

```
MA
1 package wipe;
2
3 component IntervalControlWithRainSensor {
4     autoconnect port;
5
6     port
7         in IntervalSelection,
8         in VehicleSpeed vs,
9         in RainSensorStat,
10        out WipeCmd;
11
12    component IntervalCmdProcessor icp,
13              RainEval;
14
15    connect vs -> icp.VehicleSpeed;
16 }
```

Listing 7.3: Variante der Scheibenwischersteuerung mit Regensensor

riabilität dokumentieren lässt, ohne das ursprüngliche Modell zu ändern. In der Praxis ist dieser transformationale Ansatz insbesondere sinnvoll, da neue Produktvarianten oft durch Modifikation von bestehenden Produkten entstehen.

Delta-Modeling wurde schon für mehrere Modelle und Programmiersprachen angewandt, unter anderem für Klassendiagramme [SBB+10] und Java [Sch10, SD10], und lässt sich auf eine Vielzahl weiterer Sprachen anwenden. Eine allgemeine Methodik zur Erweiterung einer Sprache in Form einer MontiCore-Grammatik um die Delta-Modeling Elemente wird in [HHK+13] beschrieben. Für die Variabilitätsmodellierung in Architekturen mit Delta-Modeling wurde Δ-MontiArc, basierend auf MontiArc, entwickelt [HRRS11,

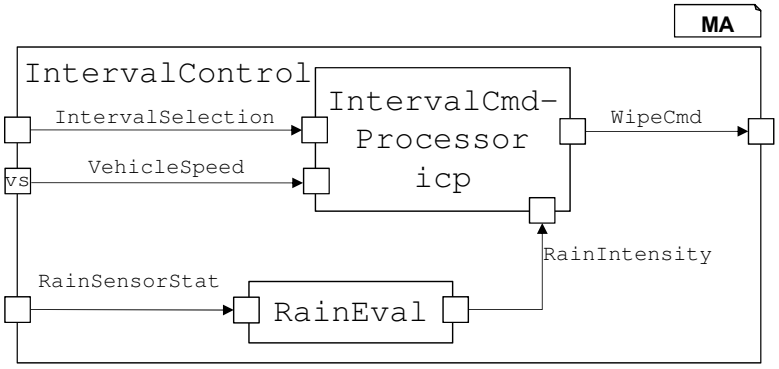


Abbildung 7.16: Grafische Repräsentation der Variante Scheibenwischersteuerung mit Regensensor

HKR+11a]. Mit dem Sprachframework MontiCore [GKR+08, KRV08, Kra10, KRV10], auf dessen Basis bereits MontiArc entwickelt wurde, lassen sich die Erweiterungen des Delta-Modeling einfach hinzufügen. Zusätzlich wurden die Kontextbedingungen bei der Anwendung von Deltas erfasst. Somit besteht die Möglichkeit, eine komplette Produktfamilie mittels Δ-MontiArc zu analysieren [HKR+11b] und zu prüfen, ob alle mit den vorliegenden Deltas modellierbaren Modelle valide sind. Mit dem in [MR14a, MR14b] gezeigten Ansatz lassen sich zu bestehenden Modellen Deltas ableiten.

In Δ-MontiArc werden Varianten durch Transformationen bzw. Deltas, basierend auf einem sogenannten Kernmodell, definiert. Das Kernmodell ist in diesem Beispiel die Komponente `IntervalControl` aus Listing 7.1 bzw. Abbildung 7.15. Ein Delta beschreibt, wie ein Modell modifiziert werden muss, um eine Variante zu erhalten. Als Änderungsmöglichkeiten stehen das Hinzufügen, Entfernen und Modifizieren von Komponenten, Ports und Konnektoren zur Verfügung.

Varianten können durch die Anwendung beliebig vieler Deltas erzeugt werden. Dabei ist die Reihenfolge der Deltas nicht festgelegt, so dass eine unterschiedliche Reihenfolge bei der Anwendung der Deltas zu unterschiedlichen Ergebnissen führen kann. Zur Vermeidung von unterschiedlichen Ergebnissen kann über Application Order Contraints für ein Delta festgelegt werden, welche anderen Deltas schon vorher angewendet sein müssen oder welche noch nicht angewendet sein dürfen. Dies ist wichtig, wenn ein durch ein Delta hinzugefügtes Element in einem weiteren Delta modifiziert wird.

Ein textuelles Beispiel für ein Delta ist im Listing 7.4 dargestellt. In den Zeilen 2 bis 5 wird die Komponente `IntervalControl` modifiziert, indem der eingehende Port `RainSensorStat` (Zeile 3) und die Komponente `RainEval` (Zeile 4) hinzugefügt werden. Die innere Komponente `IntervalCmdProcessor` erhält zusätzlich den eingehenden Port `RainIntensity` (Zeilen 6 bis 8). Zuletzt werden in Zeile 9 durch `expand autoconnect` zusätzliche Konnektoren durch die `autoconnect`-Funktion erstellt, die

die neuen Ports verbinden. Eine grafische Repräsentation des Deltas ist in Abbildung 7.17 gezeigt.

```
1 delta DRainSensor {                                          ΔMA
2   modify component IntervalControl {
3     add port in RainSensorStat;
4     add component RainEval;
5   };
6   modify component IntervalCmdProcessor {
7     add port in RainIntensity;
8   };
9   expand autoconnect;
10 }
```

Listing 7.4: Delta für das Hinzufügen des Regensensors zur Scheibenwischersteuerung

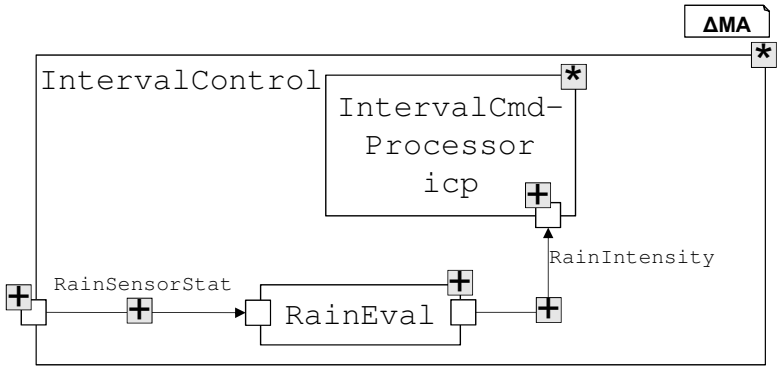


Abbildung 7.17: Grafische Repräsentation des Deltas `DRainSensor`

Nach Anwendung des Deltas entsteht `IntervalControlWithRainSensor` als neue Variante. Diese Komponente ist identisch mit dem Ergebnis der Modellierung mit dem 150%-Modell in Listing 7.3 bzw. Abbildung 7.16. Die Varianten, jeweils mit dem bzw. ohne das Feature `RainSensor`, lassen sich durch Anwendung oder Nichtanwendung des vorgestellten Deltas modellieren und um weitere Varianten mittels Deltas erweitern.

7.4.3 Vergleich der Ansätze

Um die beiden Ansätze miteinander zu vergleichen, wurden zwei eigenentwickelte Beispiele genutzt. Die Beispiele sind an Modelle angelehnt, wie sie in Komplexität und Struktur in Projekten oft vorkommen, entstammen aber keinem Projektkontext.

Beispiel 1: Steuergerätesoftware für ein Antiblockiersystem (ABS)

Das ABS-Beispiel deckt fünf Varianten ab. Es existiert jeweils eine Variante für Motorräder, drei Varianten für PKW und eine Variante für LKW. Die PKW-Varianten unterscheiden sich hinsichtlich der zusätzlichen Ausstattung mit einer Anti-Schlupf-Regelung (ASR, verhindert das Durchdrehen der Reifen beim Anfahren) bzw. mit einem elektronischen Stabilitätsprogramms (ESP, stabilisiert das Fahrzeug im Falle eines möglichen Über- oder Untersteuerns). Das Beispiel für das 150%-Modell besteht aus fünf MontiArc-Modellen:

- `Main.arc` für die Gesamtsoftware
- `ABS.arc` für die ABS-Funktionalität
- `BrakePressureController.arc` zur Ansteuerung der einzelnen Radbremsen
- `AccelerationDetection.arc` für die ASR Funktionalität
- `SpinningWheelDetection.arc` für die ESP-Funktionalität

Das Beispiel für das Δ-MontiArc-Modell enthält in Bezug auf Funktionalität die gleichen fünf Modelle, jedoch ohne jegliche Variabilität. Das Kernmodell bildet das System PKW mit ABS, welches sich mit den folgenden vier Deltas modifizieren lässt:

- `TwoWheel.delta` beschreibt die Modifikation zu einem System für ein Motorrad
- `SixWheel.delta` beschreibt die Modifikation zu einem System für einen LKW
- `ASR.delta` fügt die ASR-Funktionalität hinzu
- `ESP.delta` fügt auf Basis der ASR-Funktionalität die ESP-Funktionalität hinzu

Beispiel 2: Steuersoftware für einen Multicopter

Das Beispiel ist eine Erweiterung des Beispiels der studentischen Arbeit in [Kut11]. Für diese Software können insgesamt acht Varianten modelliert werden. Diese ergeben sich durch die Variation der Rotorenanzahl (standardmäßig vier, optional sechs oder acht), durch die Möglichkeit, die Höhe zu halten in der Konfiguration mit vier Rotoren, und zusätzlich für alle Varianten durch die Option, den Kunstflugmodus auszuschalten. Das Modell besteht aus den folgenden Dateien:

- `FlightController.arc` enthält die Gesamtsoftware
- `AccEval.arc` modelliert einen Beschleunigungssensor
- `GyroEval.arc` beschreibt einen Lagesensor
- `PressureEval` stellt einen Luftdrucksensor zur Ermittlung der Höhe dar
- `SteeringCmdProcessor` wertet die Steuerbefehle und Sensoren aus, ermittelt daraus, wie die Motoren angesteuert werden müssen. Dazu sind die folgenden Komponenten enthalten:
  - `HeightAdaptor` in SteeringCmdProcessor
  - `HeightComparator` in SteeringCmdProcessor
  - `PowerCalculator` in SteeringCmdProcessor
- `OutputProcessor` ist eine Treiberstufe für die Motoren, die die Ansteuerung für die einzelnen Motoren übernimmt.

Im Δ-MontiArc-Modell sind diese Modelle ebenfalls ohne Variabilität enthalten. Das Kernmodell ist Abbildung 7.18 dargestellt und kann durch folgende Deltas modifiziert werden.

- `HexoCopter.delta` erhöht die Anzahl der Rotoren auf sechs
- `OctoCopter.delta` erhöht die Anzahl der Rotoren auf acht
- `HeightHold.delta` fügt die Höhenkontrolle hinzu
- `PressureSensor.delta` enthält die Modifikationen für das Hinzufügen des Höhensensors und wird vom Delta `HeightHold` benötigt
- `RemoveHHFlightMode.delta` entfernt den Kunstflugmodus

Zum Vergleich der beiden Beispiele wurden der Umfang und die Verständlichkeit ermittelt. Für den Umfang wurden die Anzahl der Dateien und der Lines of Code (LOC) ermittelt. Bei der Ermittlung der LOC werden Leerzeilen nicht berücksichtigt. Auch sonst wurde bei der Implementierung der Beispiele darauf geachtet, dass sich nicht Verfälschungen durch zusätzliche Zeilenumbrüche ergeben. Die Variabilitätsinformationen wurden wie in den Listings 7.1 (MontiArc-Modelle), 7.2 (MontiArc150%-Modelle) und 7.4 (Delta-Modelle) definiert.

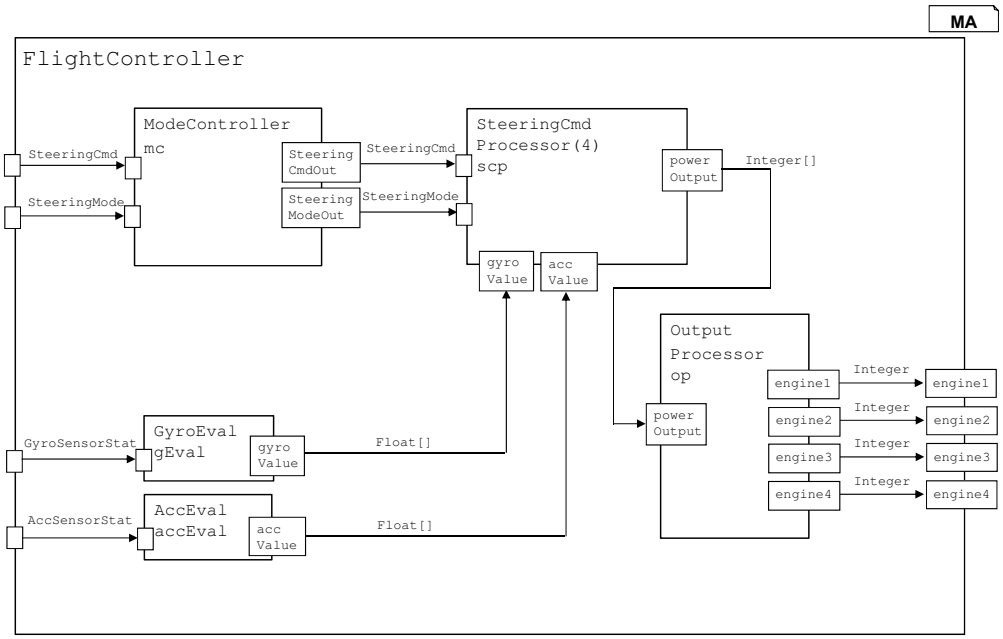


Abbildung 7.18: MontiArc-Modell der Standardvariante des `FlightControllers`

Für die Beurteilung der Verständlichkeit wurde untersucht, wie sich die Variabilität auf die einzelnen Dateien verteilt und wie groß die Dateien sind. Die eigene Erfahrung war, dass verteilte Variabilität schwerer zu erfassen ist als (auf einzelne Artefakte) konzentrierte Variabilität. Generell sind größere Dateien schwerer zu erfassen als kleinere. Für die Beispiele sind die ermittelten Werte sind Tabelle 7.9 dargestellt.

Tabelle 7.9: Vergleich des Modellierungsumfangs

	ABS		**Multicopter**	
	150%	Δ-MA	150%	Δ-MA
# Dateien	5	9	9	14
LOC über alle Dateien	93	127	100	148
# Dateien mit Gemeinsamkeiten	3	5	5	9
# Dateien mit Variabilität	0	4	0	5
# Dateien mit beidem	2	0	4	0
maximale LOC einer Datei	55	27	26	20
∅ LOC pro Dateien	≈18,60	≈14,11	≈11,11	≈10,57
Anteil LOC für Variabilität	≈68,82%	≈62,20%	≈32,00%	≈ 46,62%

Vom Umfang her ist die Anzahl der Dateien im Multicopter-Beispiel höher. Das spiegelt sich auch in der Anzahl der Zeilen wider. In beiden Beispielen sind die Modelle für die Δ-MontiArc-Implementierung größer.

Bei der Verständlichkeit ist der Δ-MontiArc-Ansatz konzeptbedingt höher als beim Ansatz mit einem 150%-Modell. Die Variabilität ist auf einzelne Dateien beschränkt und wird nicht mit den gemeinsamen Anteilen vermischt. Aus dem Verhältnis der Dateien mit Gemeinsamkeiten zu Dateien mit Variabilität bzw. Dateien mit beiden Anteilen lässt sich keine Aussage herleiten.

Die Verständlichkeit bezüglich der Größe der Dateien ist auf Seiten der Δ-MontiArc-Implementierung etwas besser. Sowohl die maximale Anzahl an Zeilen der größten Datei als auch die durchschnittliche Anzahl an Zeilen pro Datei sind im Δ-MontiArc-Ansatz kleiner, wobei der Unterschied im MultiCopter-Beispiel marginal ist. Bezüglich des Anteils der Variabilität über alle Zeilen zeigen sich im ABS-Beispiel Vorteile für Δ-MontiArc, im MultiCopter-Beispiel jedoch für den Ansatz mit dem 150%-Modell.

Zusammenfassend war auf dieser Analyse keine eindeutige Entscheidung für einen der beiden Modellierungsansätze möglich. Deshalb wurde in einem zweiten Experiment die Evolution der Modelle mitberücksichtigt. Dies war vor allem durch die Projekte bei Volkswagen motiviert, die auch im Laufe der Zeit evolvierten. Für dieses Experiment wurde das ABS-Beispiel sowohl für den Ansatz MontiArc150% als auch für Δ-MontiArc detailliert und durch typische Evolutionsszenarien erweitert sowie auf die Punkte Umfang und Verständlichkeit untersucht.

Die Untersuchung wurde detailliert in [HRRS12] durchgeführt und beschrieben. Bei der Untersuchung wurde vor allem die Änderung über die Zeit (Evolution) untersucht. Dass dies bei der Modellierung von Variabilität wichtig ist, wurde in [SSA14a, SSA14b] bestätigt. Als Evolutionsszenarien wurden folgende angenommen:

- Szenario 1: Einführung einer neuen Variante
- Szenario 2: Entfernung einer existierenden Variante
- Szenario 3: Änderung einer existierenden Variante (z.B. durch Fehlerbehebungen)

Zusätzlich wurden drei Refactoring-Szenarien untersucht, die mit dem Δ-MontiArc-Ansatz möglich sind:

- Szenario 4: Zusammenführung von Deltas
- Szenario 5: Änderung des Kernmodells mit Entfernung der Variante für das Kernmodell; da hier eine Variante entfernt wird, kann diese Änderung auch in MontiArc150% analog durchgeführt werden.
- Szenario 6: Änderung des Kernmodells mit Erhaltung der Variante für das Kernmodell über inverse Deltas

Es wurden wie beim ersten Experiment der Umfang und die Verständlichkeit der Modelle miteinander verglichen. Die Ergebnisse der ersten Evaluation wurden dabei im Wesentlichen bestätigt. Bezüglich des Modellumfangs gibt es für beide Ansätze nur marginale Unterschiede, wobei MontiArc150% etwas besser abschneidet. Das Verständnis ist nach den zu Grunde gelegten Kriterien wiederum beim Δ-MontiArc-Ansatz besser, da die Variabilitätsinformationen wie im ersten Experiment konzentriert vorliegen.

7.4.4 Entscheidung im Produktlinien-Einführungsprojekt

Aus dem Vergleich lässt sich keine tragfähige Entscheidung für einen der beiden Modellierungsansätze herleiten. Zwar lässt der Vergleich einen kleinen Vorteil der Delta-Modellierung erkennen (zumal die bestehenden Modelle nicht geändert werden müssen), jedoch ist die Frage, ob dieser auch nach Übertragung der Ansätze in Matlab/Simulink besteht. Zu diesem Zeitpunkt lag im Produktlinien-Einführungsprojekt noch kein Werkzeug vor, mit dem die Modellierung von Deltas in Matlab/Simulink möglich war. Aus diesem Grund gab es von Seiten der Projektverantwortlichen die Entscheidung, eine neuerliche Evaluation durchzuführen, wenn es ein Werkzeug für die Delta-Modellierung in Simulink gibt.

Ein solches Werkzeug wurde mittlerweile in [HKM+13] als Delta-Simulink vorgestellt. Die Evaluation eines weit fortgeschrittenen Prototyps des Werkzeugs durch die Entwickler bei Volkswagen fand im Rahmen eines kontrollierten Experimentes statt, welches in [KRR15] eingehend beschrieben ist. Hier ergab sich, dass mit Delta-Simulink schneller und einfacher modelliert werden kann und dies auch besser von den Nutzern angenommen wird als der Ansatz mit dem 150%-Modell. Subjektiv fanden die Teilnehmer der Evaluation den Delta-Ansatz schwieriger zu verstehen, was jedoch auf die neue und bisher ungewohnte Arbeitsweise zurückzuführen ist. Eine letztendliche Entscheidung für einen der beiden Modellierungsansätze wurde jedoch bisher nicht getroffen.

7.5 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Modelle im Entwicklungsprozess stärker berücksichtigt werden können. Dafür müssen oft gar keine neuen Modelle erschaffen, sondern nur bestehende Daten als Modelle erkannt und genutzt werden. Dazu wurde eine Methodik vorgestellt, wie bisher implizit in der Entwicklung genutzte Modelle identifiziert (Maßnahme M6, Identifikation und Nutzung von Modellen) und für die weitere Nutzung optimiert werden können. Dies wird durch die eingehende Analyse der Artefakte des Entwicklungsprozesses erreicht. Somit lässt sich mit relativ wenig Aufwand ein wesentlicher Schritt in Richtung einer Produktlinie machen.

Für die untersuchten Projekte bei Volkswagen wurde gezeigt, wie sich diese Methodik anwenden lässt. Dazu wurde insbesondere auf das Werkzeug DOORS eingegangen, da hier eine Reihe potentieller Modelle vermutet wurde. Mit einem Metamodell wurden die in DOORS vorhandenen Informationen greifbar gemacht und mehrere Modellierungssprachen, wie ein Parameterlexikon, identifiziert.

Als Realisierung der Maßnahme M9 (Festlegung der Variabilitätsrealisierung) wurden verschiedene Möglichkeiten diskutiert, wie Variabilität in Matlab/Simulink-Modellen definiert werden kann. Dabei wurde der Fokus auf den bisher meist verwendeten Ansatz mit 150%-Modellen gelegt und dieser mit dem neuen Ansatz der Delta-Modellierung verglichen. Es zeigte sich, dass bei der Kapselung von Varianten die Delta-Modellierung Vorteile hat, auch wenn die zugrunde liegenden Artefakte größer sind. Für die Projekte bei Volkswagen ließ sich jedoch keine endgültige Entscheidung ableiten.

Kapitel 8

Erstellung von Werkzeugen

Die im vorherigen Kapitel identifizierten Modellierungssprachen und ihre Modelle können ihr volles Potential nur mit dazu passenden Werkzeugen entfalten. Zum einen werden Werkzeuge benötigt, um die Konsistenz von Modellen zu prüfen und um eventuelle Modellierungsfehler aufzudecken (Prüfwerkzeuge), zum anderen ermöglichen sie die Generierung weiterer Artefakte aus der Modellierungssprache (Generierungswerkzeuge). Die Entwicklung dieser Werkzeuge zur Ersetzung eines bisher manuell durchgeführten Prozesses wird mit der Maßnahme M7 (Einsatz von Werkzeugen) angestrebt. Dabei wird soweit möglich auf bereits vorhandene Infrastruktur (z.B. DOORS) zurückgegriffen.

Für die Entwicklung von Werkzeugen existieren einige Rahmenbedingungen, die durch die Einflussfaktoren aus Kapitel 5 gegeben sind:

- Personen und Material (E1.1): Die Ressourcen für die Werkzeugentwicklung müssen zur Verfügung stehen.

- Innovationskultur (E2.3): Der Personenkreis, der das Werkzeug einsetzen soll, muss auch gewillt sein, es einzusetzen.

- Regularien (E2.4): Das Werkzeug muss den Qualitäts- und Sicherheitsanforderungen genügen, damit die damit entwickelte Software weiterhin den zu erfüllenden Normen entspricht.

- Einsatz von Modellen (E2.5): Modelleinsatz ist ein grundlegender Baustein für Werkzeuge und wurde im Kapitel 7 behandelt.

- Technische Infrastruktur (E2.6): Das Werkzeug muss in die bestehende technische Infrastruktur einbettbar sein.

- Bindung von Variabilität (E4.2): Der Bindungszeitpunkt der Variabilität muss feststehen.

Um mit den Modellen adäquat umgehen zu können, müssen geeignete Schnittstellen für die Werkzeuge entworfen werden. Für Modellinstanzen, die in Form von Dateien auf der Festplatte existieren, müssen entsprechende Parser entwickelt oder generiert werden. Daten, die in externen Werkzeugen liegen, benötigen Schnittstellen, damit einfach auf diese Daten zugegriffen werden kann. Dabei unterstützt ein vorher definiertes Metamodell. Dies ist sehr wichtig, da durchaus unterschiedliche Personen den Zugriff und die Verarbeitung der Informationen realisieren. Maßnahme M8 (Anbindung bestehender Datenbanken) realisiert diesen Zugriff.

In diesem Kapitel wird eine Methodik vorgestellt, wie Werkzeuge für den Einsatz in einer Produktlinie konzipiert und implementiert werden können (Abschnitt 8.1). Die entsprechende Anwendung bei Volkswagen wird in Abschnitt 8.2 gezeigt. In Abschnitt 8.3 wird das Kapitel durch mehrere Beispiele, wie Werkzeuge im Produktlinien-Einführungsprojekt entwickelt und eingesetzt wurden, abgerundet.

8.1 Methodik zur Entwicklung von Werkzeugen für den Produktlinieneinsatz

Die Entwicklung von Werkzeugen für den Einsatz in einer Produktlinie unterscheidet sich nicht groß von der Entwicklung von Werkzeugen ohne Produktlinienkontext. Der Hauptunterschied bei Produktlinien-Werkzeugen ist, dass die Berücksichtigung von Variabilität bei der Entwicklung eine Rolle spielt. Insofern gliedert sich die Entwicklung in drei Schritte, die in den folgenden drei Abschnitten detailliert werden:

- Werkzeug definieren: Funktionsumfang dokumentieren, Rahmenbedingungen berücksichtigen, Umgang mit Variabilität festlegen
- Zugriff auf die Quellen definieren: Metamodell für die Quellsprache nutzen, Parser für weitere Informationen erstellen
- Werkzeug erstellen: Implementierung, Test und Dokumentation durchführen

8.1.1 Werkzeug definieren

Zur Definition des Werkzeugs müssen die Punkte berücksichtigt werden, die im Folgenden beschrieben sind.

Anforderungen für das Werkzeug

Ein Werkzeug wird über die Anforderungen, die es erfüllen soll, definiert. Dafür ist eine besondere Berücksichtigung der Einflussfaktoren Technische Infrastruktur (Einflussfaktor E2.6) und Innovationskultur (Einflussfaktor E2.3) notwendig. Grundsätzlich müssen durch die Anforderungen folgende Fragen beantwortet werden:

- Was sind Quellsprachen und was sind Zielsprachen: Aus einem oder mehreren Quellartefakten sollen ein oder mehrere generierte Artefakte erzeugt werden. Wenn generierte Artefakte durch handcodierte Artefakte ergänzt werden, muss vermieden werden, dass generierte Anteile später händisch bearbeitet werden, da beim erneuten Generieren aufgrund einer Modelländerung die händischen Änderungen verloren gehen. Vorgehensweisen für die Trennung und Integration von handcodierten und generierten Artefakten werden in [GHK+15, MR15] beschrieben.

- Wie werden aus den Quellsprachen die Zielsprachen generiert: Dies bildet die Kernfunktion des Generators ab.

- Wer nutzt das Werkzeug: Je größer der Personenkreis, desto einfacher und zugänglicher muss die Nutzung sein. Die bei Volkswagen gemachten Erfahrungen zeigen, dass für die erfolgreiche Einführung von Werkzeugen eine hohes Maß an Benutzbarkeit hilfreich ist.

- Wo soll das Werkzeug laufen: Ein Werkzeug, welches durch einen Entwickler ausgeführt werden soll, hat andere Anforderungen an die Umgebung als ein Werkzeug, welches im Hintergrund auf einem Server Routine-Aufgaben erledigen soll. Außerdem muss klar definiert sein, zu welchen anderen Systemen und Werkzeugen Verbindungen bestehen.

- Welche Anforderungen gelten nur für das konkrete Werkzeug und welche Anforderungen gelten für eine Werkzeugfamilie: Anforderungen, die für eine Werkzeugfamilie gelten oder gelten könnten, müssen für eine bessere Wiederverwendbarkeit in der Implementierung besonders berücksichtigt werden. So entsteht analog zu der Software-Produktlinie eine Werkzeug-Produktlinie [RR15].

Prüfwerkzeug oder Generator

Werkzeuge nehmen generell zwei wichtige Funktionen wahr. Zum einen sollen sie die Eingabeartefakte prüfen (Prüfwerkzeug), zum anderen weitere Artefakte generieren (Generatoren). Beide Funktionen können von einem Werkzeug oder von getrennten Werkzeugen durchgeführt werden. Letzteres ist sinnvoll, wenn aus einem Artefakt mehrere

Artefakte generiert werden. Die Prüfung muss dann nur einmal erfolgen.

Werkzeuge sollen zur Unterstützung der Entwicklung dienen und nicht dem Entwickler Entwurfsentscheidungen abnehmen, indem es falsche Spezifikationen korrigiert. Angewendet auf Prüfwerkzeuge bedeutet dies, dass diese Werkzeuge nur dazu dienen, eventuelle Fehler dem Entwickler mitzuteilen, sie jedoch nicht selber korrigieren (siehe Abbildung 8.1). Die Korrektur von Fehlern ist dem Entwickler vorbehalten, da dies Änderungen an den Anforderungsdokumenten nach sich ziehen kann, was von denen mit dieser Methodik entwickelten Werkzeugen nicht geleistet werden kann. Ein bei einer Modellprüfung auftretender Fehler kann früh im Entwicklungsprozess fehlerhafte Artefakte anzeigen, die im weiteren Entwicklungsprozess zu Folgefehlern führen würden. Deshalb müssen Fehler, die in dieser Prüfung auftreten, sofort behandelt werden.

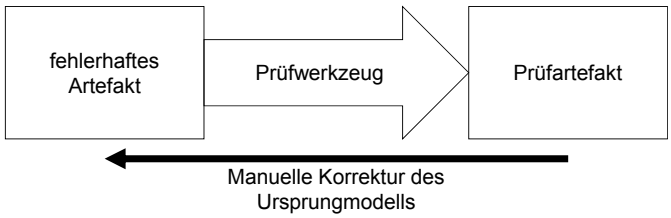


Abbildung 8.1: Behebung von Auffälligkeiten bei Prüfwerkzeugen

Sind die Modelle geprüft, können durch einen Generator (in dem das Prüfwerkzeug selber enthalten sein kann) weitere Artefakte generiert werden. Hierbei können im Vergleich zu dem manuellen Prozess nicht nur fehlerhafte sondern auch unvollständige Artefakte sichtbar werden, die bisher mit Wissen und Erfahrung des Entwicklers vervollständigt werden. An dieser Stelle ist es wichtig, zwischen fehlerhaften und unvollständigen Artefakten zu unterscheiden (siehe Abbildung 8.2). Fehlerhafte Eingangsartefakte werden korrigiert. Bei unvollständigen Artefakten muss der Generator angepasst werden, indem das Wissen und die Erfahrung des Entwicklers im Generator berücksichtigt wird oder weitere Eingangsartefakte angezogen werden.

Unvollständige Artefakte können auch durch Anpassung vervollständigt werden. Das zieht jedoch in der Regel die Erweiterung der Modellierungssprache und/oder deren Kontextbedingungen nach sich, was im Rahmen der Modelldefinition (Abschnitt 7.2.5) schon geschehen sein sollte. Für alle bestehenden Modelle muss anschließend geprüft werden, ob die Sprachdefinition auch noch für sie passt.

Qualitätssicherung

Zur Qualitätssicherung ist die Überprüfung der funktionalen Eigenschaften (Test) und der formalen Eigenschaften (Review) des Werkzeugs erforderlich. Für den Test ist es

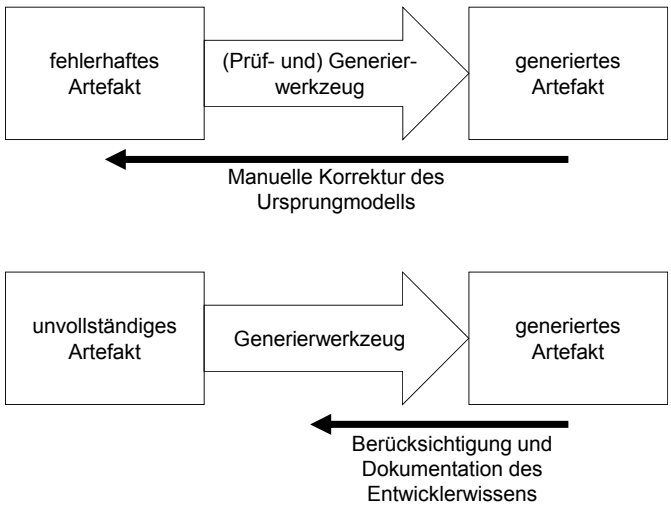


Abbildung 8.2: Behebung von Auffälligkeiten bei Generatoren

wichtig das Testziel vorzugeben, z.B. ein bestehendes Modell, welches fehlerfrei verarbeitet werden muss oder in dem bestimmte Fehler erkannt werden müssen. Bei Generatoren spielt der Einflussfaktor Regularien (Einflussfaktor E2.4) eine große Rolle. Es müssen die Modellierungs- und Kodierungs-Richtlinien eingehalten werden, um qualitativen Anforderungen zu genügen. Darüber hinaus muss der generierte Code weiter qualitätsgesichert werden, um die anvisierte Funktion des Codes sicherzustellen. Dies ist besonders bei sicherheitskritischen Anwendungen wichtig und wird durch Tests sichergestellt.

Ein alternativer Weg wäre die Zertifizierung des Werkzeugs durch unabhängige Organisationen. Dies ist jedoch mit erheblichem Aufwand (Kosten durch Zertifizierungsorganisation, Erstellung zusätzlicher Dokumentation für Zertifizierung) verbunden, der sich rentieren muss. Um dieses Risiko zu vermeiden, wird der generierte Code deshalb wie handcodierter Code behandelt und getestet. Mit einem nach dieser Methodik entwickelten Werkzeug können dazu Testumgebungen erstellt werden. Die Tests in dieser Testumgebung müssen durch Entwickler geschrieben werden.

Bindung von Variabilität

Im Kontext von Produktlinien spielt die Bindung von Variabilität (Einflussfaktor E4.2) in den Werkzeugen eine wichtige Rolle. Prinzipiell gibt es hierfür zwei Wege:

- Bindung der Variabilität innerhalb des Werkzeugs (Abbildung 8.3): Durch die Kenntnis der variablen Anteile im Werkzeug kann dieses potentiell auch mit der Variabilität umgehen. Das ermöglicht die teilweise Bindung von Variabilität und das Reagieren auf unvollständige Bindung der Variabilität. Der Nachteil ist, dass

die Variabilitätsbehandlung im Werkzeug direkt realisiert werden muss (für jedes Werkzeug, welches die gebundene Variabilität benötigt).

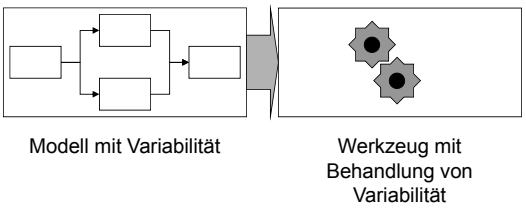


Abbildung 8.3: Bindung der Variabilität im Werkzeug

- Bindung der Variabilität als separater Schritt außerhalb des Werkzeugs (Abbildung 8.4): Durch diese Konstellation kann das Werkzeug auch in einem Entwicklungsprozess ohne Variabilität eingesetzt werden, was sich besonders im Rahmen einer Produktlinien-Einführung positiv auswirkt, da bestehende Modelle und Prozesse nur minimal geändert werden müssen. Die Nutzung von Standard-Werkzeugen wird so erleichtert.

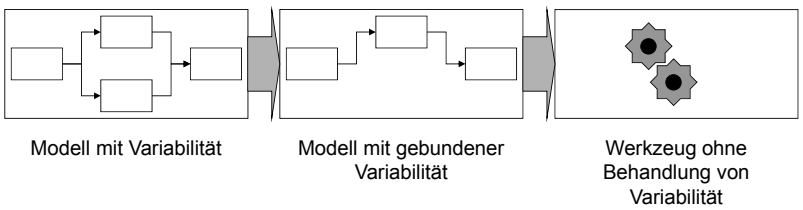


Abbildung 8.4: Bindung der Variabilität außerhalb des Werkzeugs

8.1.2 Zugriff auf die Quellen definieren

Im Fokus stehen beim Zugriff auf die Quellen die in Kapitel 7 identifizierten Modelle. Die Modelle liegen teilweise als Datei vor, teilweise sind sie in Datenbanken abgelegt, die nur über spezielle Programme (z.B. DOORS) genutzt werden können. In diesen Datenbanken stellen Ausschnitte jeweils einzelne Modelle dar. Für den Zugriff bieten sich abhängig vom Ablageort unterschiedliche Verfahren an:

- Daten in anderen Systemen: Der optimale Weg ist die Nutzung einer Schnittstelle, die die Daten anhand des in Abschnitt 7.2.1 dokumentierten Metamodells ausgibt. Über einen Service kann durch das Metamodell navigiert werden und die gewünschten Informationen können abgefragt werden. Durch die Nutzung des Metamodells können Modellinhalte unverändert weiter verarbeitet werden. Insbesondere ist eine Überarbeitung der in Abschnitt 7.2.5 dokumentierten Kontextbedingungen nicht notwendig.

- DSL in Text-Dateien: Hierfür bietet sich die Implementierung der DSL in einem Sprachwerkzeug an, z.B. MontiCore [MC]. Aus der DSL-Definition wird durch das Sprachwerkzeug eine Klassenstruktur erstellt, die dem definierten Metamodell in weiten Teilen entspricht. Abhängig von den Funktionen des Sprachwerkzeugs sind noch eventuelle Anpassungen notwendig. Ansonsten gilt auch hier, dass Kontextbedingungen nicht überarbeitet bzw. nur im Detail angepasst werden müssen.

- XML in Text-Dateien: XML hat als GPL (General Purpose Language) nicht die Möglichkeit syntaktisch auf ein bestimmtes Metamodell beschränkt zu werden. Metamodelle können mittels Schema-Dateien (XSD – XML Schema Definition) implementiert werden. Über spezielle Generatoren können aus diesen Schema-Dateien Klassenstrukturen erstellt werden (z.B. JiBX [JIBX]), die analog zu den Klassen aus der DSL verwendet werden können. Die Kontextbedingungen können ebenfalls direkt oder mit leichten Anpassungen übernommen werden.

- Komplexe Daten in Binärdateien: In diese Kategorie fallen beispielsweise die verschiedenen Office-Formate wie die von Microsoft Word, Microsoft Excel, etc. Da diese Formate, bedingt durch ihren Funktionsumfang, sehr komplex ausfallen, ist die Entwicklung einer eigenen Schnittstelle häufig sehr zeitaufwändig. Oft existieren jedoch schon frei verfügbare Implementierungen, die sich nutzen lassen. Die komplexen Daten sind in einem eigenen Metamodell definiert, welches in der Regel nicht mit dem in Abschnitt 7.2.1 dokumentierten Metamodell übereinstimmt. Aus diesem Grund ist der Zugriff auf diese Daten mit mehr Aufwand verbunden.

8.1.3 Werkzeug erstellen

Die Erstellung der Werkzeuge besteht im Wesentlichen aus der Umsetzung der Anforderungen für das Werkzeug. Für diese müssen technische Realisierungen konzipiert und implementiert werden. Dazu gehört auch, eine entsprechende Werkzeugarchitektur zu entwerfen und technische Details zu klären.

Die Adapter, die im vorherigen Schritt entwickelt oder ausgesucht wurden, müssen ebenfalls in der Realisierung berücksichtigt werden. Da auf einem Metamodell mehrere Werkzeuge (eine Werkzeugfamilie) basieren können, macht es für die Wiederverwendbarkeit Sinn, das Werkzeug modular zu entwickeln. Das heißt, dass für alle Werkzeuge die Grundfunktionalität gemeinsam entwickelt wird und sich die Werkzeuge nur in ihrer Funktion unterscheiden. Bei Volkswagen hat sich herausgestellt, dass eine Benutzerschnittstelle, die wiedererkannt wird, die Einführung und Benutzung neuer Werkzeuge mit gleicher Anwendungslogik erleichtert.

Zur Erstellung des Werkzeugs gehören auch qualitativ hochwertige Tests. Idealerweise wird dazu ein Testmodell erstellt, welches alle Teile des Metamodells abdeckt. Dies gibt

Sicherheit bei Weiterentwicklungen und Fehlerbehebungen am Werkzeug.

Zuletzt gehört als wichtiger Punkt zur Erstellung des Werkzeugs auch eine entsprechende Dokumentation. Am wichtigsten für die Anwender ist natürlich eine Einführung, wie das Werkzeug bedient wird. Dazu gehört auch, die notwendigen Modelle zu erklären. Es empfiehlt sich, schon während der Entwicklung des Werkzeugs die Anwender mit einzubeziehen, da die Erfahrung bei Volkswagen zeigt, dass sich dadurch gute Empfehlungen für die Werkzeugbedienung ergeben.

8.2 Konstruktion konkreter Werkzeuge bei Volkswagen

In diesem Abschnitt wird die gerade definierte Methodik an einem Beispiel bei Volkswagen demonstriert. Dazu wird zu der in Abschnitt 7.3 identifizierten Busschnittstelle in DOORS ein Werkzeug entwickelt. Die Entwicklung von hochkomplexen mechatronischen Systemen wie elektromechanischen Lenksysteme benötigt eine große Entwicklungsmannschaft. Aus diesem Grund sind Teile der Entwicklung (Baugruppen) an externe Firmen vergeben worden. Andersherum ist die ein Lenksystem nur eine Komponente im Gesamtfahrzeug. Damit die Integration von Baugruppen in die Komponente und der Komponente in das Gesamtfahrzeug möglichst problemlos erfolgen kann, sind genaue Abstimmungen über die Busschnittstellen und die Einhaltung dieser notwendig. Änderungen an den Busschnittstellen haben einen erheblichen Einfluss. Wenn Nachrichten nicht oder unvollständig durch das System interpretiert werden, kann ein ungewöhnliches Verhalten des Lenksystems auftreten, welches nicht auf die Ursache hindeutet.

Um nachvollziehen zu können, welche Auswirkungen sich konkret ergeben haben, müssen die externen Daten mit den internen Daten abgeglichen werden. Die identifizierten Änderungen müssen in den eigenen Dokumenten anschließend weiter verarbeitet werden. Für den späteren Einsatz in Produktlinien kann es sich als schwierig herausstellen, einen solchen Abgleich für verschiedene Varianten und Versionen durchzuführen, insbesondere, wenn der Kommunikationspartner keine Produktlinien einsetzt. Das dazu entwickelte Werkzeug wird im folgenden „DBCCheck“ genannt.

8.2.1 Werkzeug definieren

Zur Definition des Werkzeugs DBCCheck wird nach der Beschreibung aus Abschnitt 8.1.1 vorgegangen. Es werden die Anforderungen an das Werkzeug festgehalten, ob es sich um ein Prüfwerkzeug oder Generator handelt, welche Qualitätskriterien erfüllt sein müssen und wie mit der Variabilität umgegangen werden soll. Die Definition des Werkzeugs wurde zusammen mit der Lenkungsentwicklung bei Volkswagen durchgeführt. Im

Folgenden werden die abgestimmten Anforderungen aufgeführt.

Das Werkzeug soll den bisher manuell durchgeführten Schritt Entwicklungsprozess unterstützen, der in Abbildung 8.5 zu sehen ist. Eingangsartefakte sind die durch ein Lenkungsprojekt bei Volkswagen genutzte Busschnittstelle aus DOORS und die vom Auftraggeber definierten Bussignale in einer DBC-Datei. Das DBC (data base CAN)-Format dient zur Beschreibung von möglichen Nachrichten auf einem CAN-Bus [DBC]. Über dieses Format kann festgelegt werden, in welchen Botschaften einzelne Bits bestimmte Werte darstellen. Das Format ist eine DSL, deren Instanzen in einer Text-Datei abgelegt sind.

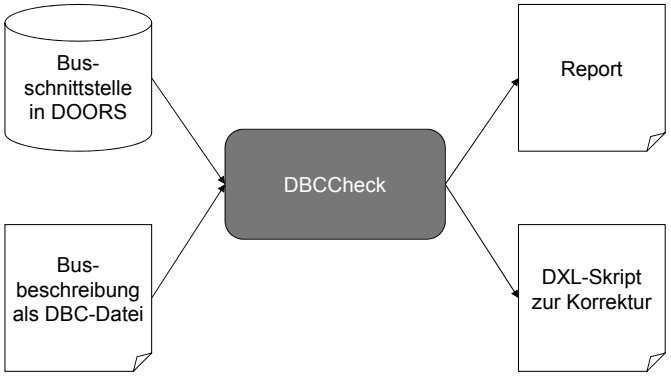


Abbildung 8.5: Das Werkzeug DBCCheck

Als Ausgabe des Werkzeugs entsteht ein Report, in dem die Änderungen gut farblich hervorgehoben werden sollen. Auf diese Weise kann man sich schnell einen Überblick über mögliche Fehler schaffen und gezielt reagieren. Ein weiteres Ergebnis des Abgleichs ist ein DXL-Skript, welches die Werte in DOORS korrigieren kann. Dies wird nicht automatisch getan, da Änderungen nur von Entwicklern durchgeführt werden sollen, wobei das Skript unterstützt.

Die Struktur der untersuchten Daten ist in Abbildung 8.6 zu sehen. In der Busschnittstelle als auch in der DBC-Datei sind die Daten in Botschaften eingeteilt, die wiederum aus mehreren Signalen aufgebaut sind. Eine Botschaft ist durch seine ID eindeutig identifizierbar. Diese, der Name und die Länge in Byte (DLC – Data Length Code) der Botschaft werden beim Abgleich gegenübergestellt. ID und DLC werden in DOORS hexadezimal angegeben, jedoch in der DBC-Datei als dezimaler Long-Wert gespeichert, so dass beim Vergleich eine Umrechnung erfolgen muss.

Die Signale auf dem Bus (beispielsweise ein Lenkwinkel) besitzen ebenfalls mehrere Attribute. In der DBC-Datei wird zu jedem Signal der Name gespeichert, welches Startbit in der Botschaft es hat, die Länge, die Byte-Order (BidEndian bzw. LittleEndian) und ob das Signal ein Vorzeichen hat. Die Interpretation des Signals ist ebenfalls festgelegt und erfolgt wie im Rest der Software mit Fixpunkt-Datentypen. Zur Berechnung der

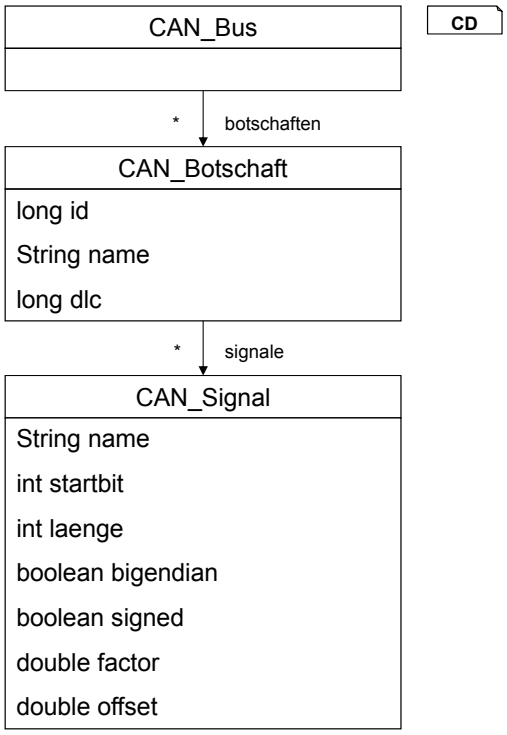


Abbildung 8.6: Daten zur Beschreibung des CANs

physikalischen Werte werden ein Faktor, der erst mit dem Integer-Rohwert auf dem Bus multipliziert wird, und ein Offset, der anschließend addiert wird, angegeben. So lassen sich das physikalische Minimum und Maximum berechnen.

Im Report sollen die Daten aus der DBC-Datei den Daten der Busschnittstelle aus DOORS gegenüber gestellt werden. Abweichungen durch fehlende Botschaften und Signale sollen farblich hervorgehoben werden. Das DXL-Skript soll die Befehle enthalten, die das DOORS-Dokument auf Basis der DBC-Datei so anpassen kann, dass die Daten für beide Dokumente gleich sind.

Das Werkzeug soll durch die Softwareentwickler der Lenkungsentwicklung eingesetzt werden. Da der Personenkreis unterschiedlichste Werkzeuge einsetzt, besteht hinsichtlich Funktion und Benutzung von Werkzeugen ein großer Erfahrungsschatz. Deshalb ist eine gute Nutzbarkeit auch noch gegeben, wenn Dinge mit Fachbegriffen beschrieben sind. Da die Entwickler auch technisch sehr gut mit dem CAN-Bus vertraut sind, erwarten diese im Report einen hohen Detailgrad bezüglich der abgeprüften Daten.

Die Ausführung des Werkzeugs DBCCheck soll am besten aus DOORS heraus möglich sein. Daraus ergibt sich, dass das Werkzeug auf dem Rechner des Entwicklers läuft. Außer DOORS sind keine weiteren Werkzeuge im Rahmen der Umsetzung von DBCCheck

betroffen. Als Implementierungsprache soll aufgrund ihrer freien Verwendbarkeit und der Entwicklererfahrung Java [Java] genutzt werden.

Hinsichtlich der Wiederverwendbarkeit gibt es mehrere Bestandteile, die auch in weiteren Werkzeugen sinnvoll sind:

- Erstellung eines Reports: Zur Dokumentation von Ungenauigkeiten und für die Nachvollziehbarkeit der Tätigkeiten des Werkzeugs dient ein Report, der die Ergebnisse in einem passenden Format zugänglich macht. Es soll ein einfacher Mechanismus existieren, um festgestellte Abweichungen, Warnungen oder Informationen zu hinterlegen.

- Bereitstellung einer GUI: Es soll eine generische GUI existieren, die mit einfachen Mitteln um Optionen für das Werkzeug erweitert werden kann. Die Options-Einstellungen sollen speicherbar sein, so dass ein einfacher Aufruf des Werkzeugs mit der gleichen Konfiguration möglich ist. Für die Rückmeldung an den Nutzer wurde eine Fortschrittsanzeige gewünscht.

DBCCheck ist primär dazu gedacht, die Busschnittstelle zu prüfen und auf Basis dieser Prüfung Korrekturen durchzuführen. Aus diesem Grund ist das Werkzeug als Prüfwerkzeug einzuordnen. Die DXL-Datei, die den Korrekturschritt unterstützt, ist kein Artefakt, welches im weiteren Entwicklungsprozess Anwendung findet.

Da das Werkzeug einen bisher manuellen Schritt mit händischer Anpassung unterstützt, beschränkt sich die Qualitätssicherung auf den Test des Werkzeugs. Langfristig kann das Werkzeug im Prozess verankert werden, indem es dort explizit erwähnt wird. Wenn das Werkzeug sich im Betrieb bewährt hat, kann der erstellte Report explizit im Prozess im Rahmen des weiterhin von Entwicklern durchgeführten Review angezogen werden.

Die Variabilitätsbindung im Rahmen einer Produktlinie soll außerhalb des Werkzeugs geschehen. Das erleichtert die Implementierung des Werkzeugs, da aufgrund des Einflussfaktors E1.1 (Personen und Material) nur wenige Ressourcen zur Verfügung stehen. Die Anforderungen an das Werkzeug sind in Tabelle 8.1 zusammengefasst.

8.2.2 Zugriff auf die Quellen definieren

Für DBCCheck gibt es nach Abbildung 8.5 zwei Quellartefakte, für die die notwendigen Schnittstellen entwickelt werden müssen, zum einen die Busschnittstelle in DOORS, zum anderen die Busbeschreibung in der DBC-Datei. Für beide wurde eine Anbindung an das Werkzeug entworfen, was im Folgenden detailliert wird.

Tabelle 8.1: Anforderungen an DBCCheck

Name	Beschreibung	wiederverwendbar
A1	Quellsprachen sind das DOORS-Metamodell und die DSL zur DBC-Beschreibung. Zielsprachen sind zum einen ein Report im Excel-Format und zum anderen ein DXL-Skript zur Korrektur	teilweise
A2	Alle Botschaften und Signale der DBC-Datei müssen mit dem Datenbestand in DOORS abgeglichen werden. Wird ein Unterschied erkannt, so ist dieser als Änderung im Report zu markieren. Fehlt eine Botschaft oder ein Signal, dann ist die Änderung als fehlend zu markieren. Anschließend muss für alle Botschaften und Signale in DOORS überprüft werden, ob sie auch in der DBC-Datei vorhanden sind. Ist dies nicht der Fall, so ist die Änderung als entfernt zu markieren.	nein
A3	Anwender sind die Softwareentwickler der Lenkungsentwicklung. Diese wünschen eine grafische Oberfläche für das Werkzeug.	ja
A4	Das Werkzeug soll auf dem lokalen Rechner des Softwareentwicklers aus DOORS heraus ausgeführt werden.	ja
A5	Das Werkzeug ist ein Prüfwerkzeug.	nein
A6	Die Qualitätssicherung erfolgt auf Basis der aktuellen Daten aus DOORS und der vorhandenen DBC-Dateien.	nein
A7	Die Variabilität soll außerhalb des Werkzeugs gebunden werden.	nein

Adapter zu DOORS

Da der Zugriff auf DOORS durch mehrere Werkzeuge notwendig ist, wurde ein genereller Adapter entworfen, der für alle in DOORS abgelegten Modelle genutzt werden kann. Basierend auf dem in Abschnitt 7.3.1 dargestellten Metamodell wurde ein Adapter entworfen, der es erlaubt, dieses in Java mit den Daten aus DOORS zu nutzen. Da der Adapter auch in anderen Werkzeugen zum Einsatz kommen sollte, bei deren Anwendung keine direkte Verbindung zur DOORS-Datenbank besteht, sollte eine Online- (mit Verbindung zum DOORS-Server) und eine Offline-Anbindung (ohne Verbindung zur DOORS-Server) realisiert werden. Szenario hierfür ist beispielsweise die Anpassungen des in Abschnitt 7.3 identifizierten Parameterlexikons mit anschließendem Ausführen

eines Generators auf der Teststrecke. Der Adapter realisiert nur einen lesenden Zugriff auf die Daten in DOORS, da Modelländerungen stets durch den Entwickler und nicht automatisiert durch Werkzeuge durchgeführt werden sollen.

Für beide Zugriffsmöglichkeiten soll eine Schnittstelle dienen, damit das Werkzeug an sich nicht durch den Wechsel der Datenbasis geändert werden muss. Die in Abschnitt 7.3.1 dargestellten Klassen wurden dazu als Interfaces realisiert. Attribute dieser Klassen werden durch entsprechende get-Methoden realisiert. Ebenso sind für eine Navigation entlang der gerichteten Assoziationen ebenfalls get-Methoden erstellt worden. Über diese Interfaces kann später von den Nutzern durch das Datenmodell navigiert werden. Die Interfaces werden jeweils einmal für die Online-Variante und einmal für die Offline-Variante implementiert wie in Abbildung 8.7 am Beispiel von *FormalModule* zu sehen ist. Das Interface *FormalModule* wird von der Klasse *_Online_FormalModule* für den direkten Zugriff auf DOORS implementiert. Für den Zugriff auf einen exportierten Datenstand wird die Implementierung des Interfaces in der Klasse *_Offline_FormalModule* verwendet.

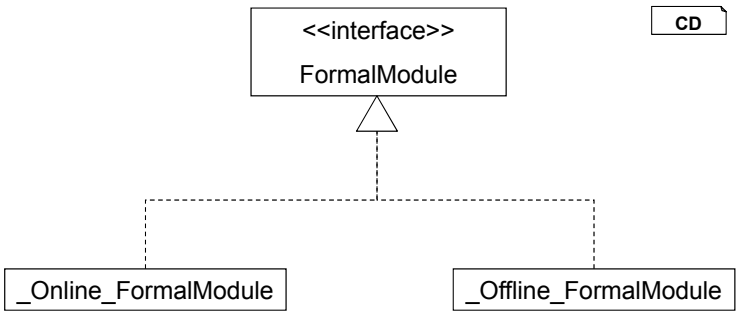


Abbildung 8.7: Implementierung der Interfaces im DOORS-Adapter

Für abstrakte Klassen innerhalb des Metamodells entstehen in der Implementierung ebenfalls Interfaces. Die Implementierung für die Zugriffsmöglichkeiten auf Basis dieses Interfaces erfolgt in Form abstrakter Klassen. Als Beispiel dazu ist die Implementierung von *Module* in Abbildung 8.8 gezeigt. Die abstrakte Klasse *Item* aus Abbildung 7.3 wird durch das entsprechende Interface dargestellt und hat als Implementierung die abstrakten Unterklassen *_Online_Item* und *_Offline_Item*, die den für alle *Item*s gemeinsamen Anteil an Methoden für die jeweilige Zugriffsmöglichkeit enthalten. Das Interface *Module* erweitert das Interface *Item* wie im Metamodell. Die Implementierung von *Module* (*_Online_Module* und *_Offline_Module*) erweitert die abstrakten *Item*-Klassen und implementiert nur den spezifischen Anteil für *Module*.

Um den Adapter vollständig zu testen, wurde eine umfangreiche JUnit-Testsuite [JUnit] aufgebaut, die sowohl Online-Adapter als auch Offline-Adapter testet. Da die Tests gegen das Metamodell implementiert sind, können die Tests komplett für beide Adapter-Varianten wiederverwendet werden. Somit kann überprüft werden, ob sich beide Vari-

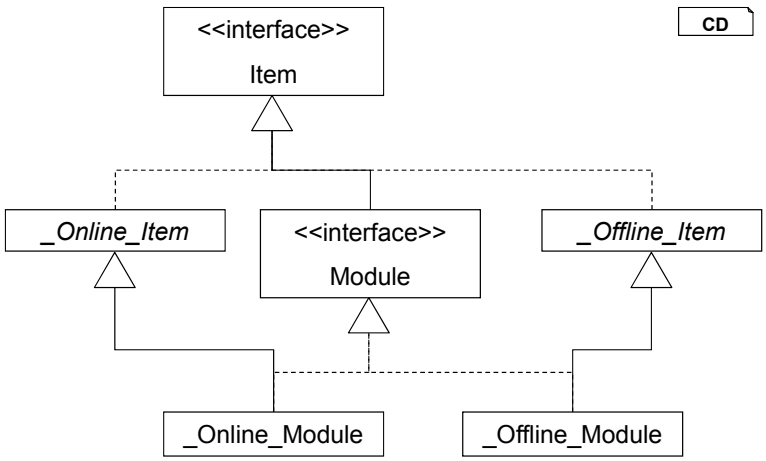


Abbildung 8.8: Implementierung der abstrakten Interfaces im DOORS-Adapter

anten gleich verhalten.

Caching. Um den Speicherverbrauch des Adapters zu minimieren, werden nur die Elemente des Datenmodells initialisiert, die gerade benötigt werden. Instanzen werden zentral an einem Cache gespeichert, so dass Mehrfachinstanzierungen von einem Datenelement vermieden werden. Dies schafft vor allem im Online-Adapter einen Geschwindigkeitszuwachs beim Zugriff, da dort die Kommunikation zwischen DOORS und Java das Nadelöhr ist. Als Nachteil entsteht dadurch, dass Änderungen während der Laufzeit des Adapters nicht berücksichtigt werden können. Im Einsatz spielt diese Einschränkung keine Rolle, da zur Ausführungszeit eines Werkzeugs sich die Datenbasis in der Regel nicht ändert.

Um die Initialisierung zu steuern, sind sämtliche Klassen des Metamodells mit privaten Konstruktoren ausgestattet. Dafür gibt es eine statische get-Methode, die die Aufgabe des Konstruktors übernimmt, jedoch vor der Erstellung einer neuen Instanz prüft, ob eine Instanziierung notwendig ist oder ob das benötigte Element sich bereits im zentralen Cache befindet, der als Singleton nach [GHJV95] implemntiert ist. Es wird außerdem geprüft, ob das angeforderte Element im Datenmodell vorhanden ist, so dass nur existente Daten instanziiert werden.

Das Caching kann für die Anforderungsobjekte im Detail gesteuert werden. Da oft mehrere Attributwerte für ein Anforderungsobjekt benötigt werden, können diese beim Online-Adapter in einem Schritt aus der Datenbank extrahiert werden, ohne für jeden eine einzelne (zeitaufwändige) Kommunikation aufzubauen. Im Offline-Adapter hat dies keine Auswirkungen, da dort kein Kommunikationsaufbau notwendig ist.

Constraints. Da die Kommunikation über den Online-Adapter relativ langsam im Vergleich zum Offline-Adapter ist, wurden die übertragenen Daten auf ein Minimum reduziert. Diese Optimierung findet bei der Iteration über eine Anforderungsobjektmenge Anwendung, in der nur bestimmte Objekte mit gewissen Eigenschaften letztendlich verarbeitet werden. Wenn beispielsweise 100 Anforderungsobjekte in einem Modul vorhanden sind, von diesen aber nur 20 den Wert „für Test“ für das Attribut „Zweck“ enthalten, dann gäbe es einen Zugriff für das Erstellen der Liste und jeweils einen Zugriff auf das Attribut „Zweck“ der 100 Elemente, also 101 Zugriffe. Letztendlich müssen aber nur die 20 benötigten Elemente übertragen werden, womit das Optimum bei 21 Zugriffen läge.

Um dies durch den Anwender aktiv steuern zu können, gibt es für Listen von Anforderungsobjekten die Möglichkeit der Nutzung von Constraints. Diese werden direkt in DOORS ausgewertet und ersparen somit die Übertragung von nicht notwendigen Elementen. Es werden vom Adapter die in Tabelle 8.2 dargestellten Constraints angeboten. Constraints erweitern die abstrakte Oberklasse `Constraint`.

Tabelle 8.2: Constraints für die Einschränkung von Listen

Constraint	Funktion
`ContainsConstraint`	überprüft, ob ein Attribut einen Wert enthält
`ContainsNotConstraint`	überprüft, ob ein Attribut einen Wert nicht enthält
`EqualsConstraint`	überprüft, ob ein Attribut mit einem Wert übereinstimmt
`EqualsNotConstraint`	überprüft, ob Attribut nicht mit einem Wert übereinstimmt
`AndConstraint`	verknüpft zwei Constraints mit *UND*
`OrConstraint`	verknüpft zwei Constraints mit *ODER*

Zur Verdeutlichung ist in Listing 8.1 das oben genannte Beispiel als Java-Code ohne Anwendung eines Constraints dargestellt. Der Java-Code in Listing 8.2 hat die gleiche Funktion, verwendet jedoch ein Constraint. Im ersten Listing wird die komplette Liste an Objekten geholt und jeweils einzeln das Attribut geprüft. Das zweite Listing optimiert den Zugriff, indem diese Prüfung als Constraint für die Liste mit übergeben wird.

Für die Implementierung der Constraints werden zwei Methoden in der abstrakten Oberklasse `Constraint` definiert, die durch das Framework benötigt werden. Die Online-Variante benutzt die Methode `getDXL`, um aus dem gegeben Constraint einen angepassten DXL-String zu erstellen, der einen logischen Ausdruck repräsentiert. Bei der Offline-Variante wird die Methode `accepts` genutzt, um für ein Anforderungsobjekt zu prüfen, ob das Constraint erfüllt ist.

```java
FormalModule fm = doors.getInstance().getModule("/TestModule").
    getCurrentVersion();
for (ReqObject ro : fm.getObjects()) {
  if (ro.getAttributeValue("Zweck").contains("für Test")) {
    // do something
  }
}
```

Listing 8.1: Iteration ohne Constraints

```java
FormalModule fm = doors.getInstance().getModule("/TestModule").
    getCurrentVersion();
for (ReqObject ro : fm.getObjects(new ContainsConstraint("Zweck",
       "für Test"))) {
  // do something
}
```

Listing 8.2: Iteration mit Constraints

Technische Realisierung des Online-Adapters. Der Online-Adapter stellt den aktuellen Datenbestand einer laufenden DOORS-Instanz zur Verfügung. Dazu wird eine Kommunikation über die COM-Automation-Schnittstelle von DOORS aufgebaut. Über diese werden dann spezifische DXL-Skripte gesendet, die in DOORS ausgeführt werden und die gewünschten Informationen zurücksenden.

Für die Kommunikation wurde die JACOB-Bibliothek [Jacob] verwendet. Diese Bibliothek erlaubt, eine Kommunikation über die COM-Automation-Schnittstelle eines Programms aufzubauen und es damit zu steuern. Damit war es möglich auf eine laufende Instanz von DOORS zuzugreifen. Die Anmeldung an die Datenbank erfolgt auf dem vom Nutzer gewohnten Weg und es sind keine Änderungen am Programm selber notwendig.

Die DXL-Skripte, die zur Kommunikation genutzt werden, haben oft einen ähnlichen Aufbau. Diese Struktur, sowie Templates zur Initialisierung von Datenbankteilen wie Modulen, sind im Adapter in Hilfsmethoden ausgelagert, die diesen Teil einheitlich zur Verfügung stellen. Zusätzlich ist ein Debug-Modus implementiert, der bei der Fehlerbehebung in der Entwicklung des Adapters genutzt werden kann. Für den Einsatz des Adapters ist dieser deaktiviert, da das Debugging die Kommunikation zwischen Java und DOORS verlangsamt.

Technische Realisierung des Offline-Adapters. Der Offline-Adapter ermöglicht den Zugriff auf einen festen Datensatz aus DOORS. Dieser Adapter ist insbesondere dafür gedacht, mit Daten zu arbeiten, ohne direkt mit dem DOORS-Server verbunden zu sein.

Somit kann auch ohne Netzwerkzugriff mit vorher exportierten Daten gearbeitet werden. Dies ist für die Entwicklung von Transformationen hilfreich, da sich geänderte Daten vom Server gezielt übernehmen lassen. So lassen sich neue Spezialfälle einer Transformation nach und nach aufnehmen oder gezielt behandeln.

Der Adapter gliedert sich in einen Export-Teil, der einmalig in DOORS ausgeführt werden muss und die Daten in einer Dateistruktur ablegt, sowie einem Import-Teil, welcher diese Dateistruktur im Rahmen des Metamodells bereitstellt. Der Export-Teil wird durch ein DXL-Skript realisiert. Dieses bildet die DOORS-Ordnerstruktur auf der Festplatte ab, indem es für die Ordner in DOORS jeweils Ordner auf der Festplatte anlegt und die einzelnen Module in Dateien exportiert. Der Inhalt der Dateien ist in einer domänenspezifischen Sprache (DSL) abgelegt, die speziell für diesen Einsatzzweck entwickelt wurde.

Ausgangsbasis für jeden Export ist ein Formalmodul in DOORS. In diesem Modul wird das DXL-Skript aufgerufen. Dieses exportiert zunächst die Moduldefinition, also vorhandene Typen und Attribute. Anschließend werden die Werte der Modulattribute exportiert. Danach erfolgt das Exportieren der Anforderungsobjekte. Hierbei wird die Hierarchie beibehalten, so dass Objekte unter anderen Objekten gruppiert werden, wie es bei Tabellen der Fall ist. Anschließend werden die Viewdefinitionen und die im Modul enthaltenen Links exportiert. Dabei werden die benutzten Linkmodule gesammelt und anschließend deren Definitionen in eigene Dateien exportiert. Das vollständige DXL-Skript zum Export ist im Anhang unter Abschnitt A.1 zu finden.

Beim Export wurde darauf geachtet, dass die Informationen möglichst kompakt und ohne Redundanzen abgelegt werden. Besonders beim späteren Parsen mit MontiCore können zu große Dateien das Einlesen verzögern. Deshalb werden OLE-Objekte, die in den Anforderungen enthalten sind, separat in eigene Dateien exportiert. Teilweise ist der Export durch die Gegebenheiten des DXL-Skripts bestimmt. So lassen sich Links nur von einem Formalmodul aus ansprechen, obwohl sie in Linkmodulen abgelegt sind.

Der Import-Teil nutzt das DSL-Framework MontiCore, welches den Export parsen kann und den Inhalt im Rahmen des Metamodells in Java zur Verfügung stellt. Basis für diese Verarbeitung ist eine MontiCore-Grammatik. Diese ist vollständig im Anhang unter Abschnitt A.2 abgebildet. Aus dieser werden durch MontiCore anschließend Klassen für die abstrakte Syntax, ein Parser und weitere Teile für die Verarbeitung von Eingabedateien generiert.

Der eingelesene DOORS-Datensatz kann mit Hilfe des MontiCore-Frameworks unter Benutzung von Workflows weiterverarbeitet werden. Es ist so beispielsweise möglich eine Transformation nach RIF [RIF] zu erstellen. Dabei können dann gezielt Informationen so aufbereitet werden, wie es für andere Tools nötig ist, die dieses Format verarbeiten. Eine Migration von DOORS zu einem anderen Anforderungswerkzeug lässt sich auf diese

Weise ebenfalls realisieren. Die Option der Weiterverarbeitung wurde nicht im Produktlinien-Einführungsprojekt genutzt, sondern sie soll nur als Beispiel dienen, dass mit dem entwickelten Adapter eine Reihe weiterer Anwendungen möglich sind.

Adapter zum DBC-Format

Das DBC-Format ist eine textuelle Repräsentation der Daten, die in einer domänenspezifischen Sprache beschrieben sind. Im Format selber können noch weitere Informationen abgelegt werden, nicht nur die für DBCCheck benötigten Daten. So können komplette Netzwerke beschrieben und Kommentare hinterlegt werden.

Da es sich bei den Daten um Instanzen einer DSL in Textdateien handelt, wurde diese DSL in einem Sprachwerkzeug definiert. Mittels MontiCore wurde eine Grammatik entwickelt, um die DBC-Dateien in eine Objektstruktur überführen zu können. Die Grammatik für das DBC-Format ist im Anhang unter Abschnitt B.1 abgelegt.

Mit MontiCore wurden aus der Grammatik die Java-Klassen und ein Parser generiert, mit dem die DBC-Dateien eingelesen werden konnten und für das Werkzeug als Java-Objekte bereitgestellt wurden. Der so entwickelte Adapter wurde für eine bessere Wiederverwendung als eigenes Modul bereitgestellt, welches auch in anderen Werkzeugen Einsatz finden kann. Das gleiche Vorgehen lässt sich auf weitere Formate im Rahmen der Steuergeräteentwicklung anwenden, beispielsweise auf das ASAP-2 Format des ASAM-Standards [ASA10] für die Steuergerätekalibrierung.

8.2.3 Werkzeug erstellen

Für die Implementierung sind im Produktlinienkontext besonders die wiederverwendbaren Anforderungen interessant. Nach Tabelle 8.1 trifft dies auf die Anforderungen A1, A3 und A4 zu.

Realisierung von Anforderung A1

Der wiederverwendbare Teil dieser Anforderung ist die Erstellung des Reports. Es wurden dabei drei wesentliche Kategorien von Meldungen benutzt:

- Abweichungen: Inkorrekte Eingaben, die nicht automatisiert behandelt werden können und zu fehlerhaften Artefakten führen. Beispiel: Verstoß gegen Namenskonventionen, die in C zu einem Kompilierungsfehler führen.

- Warnung: Eingaben, die in bestimmten Kontexten Unklarheiten provozieren könnten, jedoch im Werkzeug behandelt werden können. Beispiel: Gemischte Benutzung von Punkt und Komma für die Dezimaltrennung bei Fließkommazahlen.

- Information: Positive geprüfte Eigenschaften des Dokuments, die in einem Report auftauchen sollen.

Um eine einfache Übersicht über Abweichungen, Warnungen und Informationen zu erhalten, die im Rahmen der Werkzeuganwendung aufgetreten sind, wurde ein Excel-Report verwendet. Dieser kann die erhaltenen Informationen tabellarisch darstellen und ermöglicht mittels Farbgebung bestimmte Ergebnisse hervorzuheben. Zudem ist Excel eines der standardmäßig verwendeten Formate bei Volkswagen.

Für diesen Report wurde die POI-Bibliothek [POI] genutzt, die auch das Einlesen von Microsoft Office Formaten erlaubt. Die Anbindung an die POI-Bibliothek wurde vereinfacht. Es wurde ein Standard-Kopf für Excel-Reports bereitgestellt, der wesentliche Informationen des Checks beinhalten (z.B. Versionsnummern, Artefaktnamen). Für die einfache Erstellung der Reports wurden Methoden zum Einfügen von neuen Fehlern, Warnungen und Informationen inklusive passender Formatierungen angeboten, die sich bei Bedarf einfach anwenden und erweitern lassen. Ebenso lassen sich direkt Links zu DOORS einbetten, um schnell zu den relevanten Anforderungen navigieren zu können.

Für die Erfassung der Ergebnisse wurde eine eigene Klasse implementiert, die über `addError`, `addWarning` und `addInfo` die Aufnahme von Fehlermeldungen erlaubt. Für alle Methoden müssen der Name des DOORS-Moduls, in dem das Ereignis aufgetreten ist, die aktuelle Nummer des DOORS-Objektes, ein Text zur leichteren Zuordnung des Objektes (z.B. Object Text des Objektes), sowie eine (Fehler-)Meldung übergeben werden. Es existiert zusätzlich jeweils eine erweiterte Variante der Methoden, über die noch weitere Daten zu dem Ereignis übergeben werden können.

Realisierung von Anforderung A3

Die generische GUI soll die Informationen enthalten, die immer für ein Werkzeug benötigt werden. Ein Beispiel für eine GUI ist in Abbildung 8.9 gezeigt. Diese enthält die folgenden Elemente:

- Standardelemente (z.B. Schließen-Button)

- Pfad für Ausgaben (+ Durchsuchen und Pfad öffnen)

- Konfiguration (+ Öffnen des DOORS-Elements + Start)

- Fortschrittsbalken
- Speicherung der Einstellungen (als Funktion)

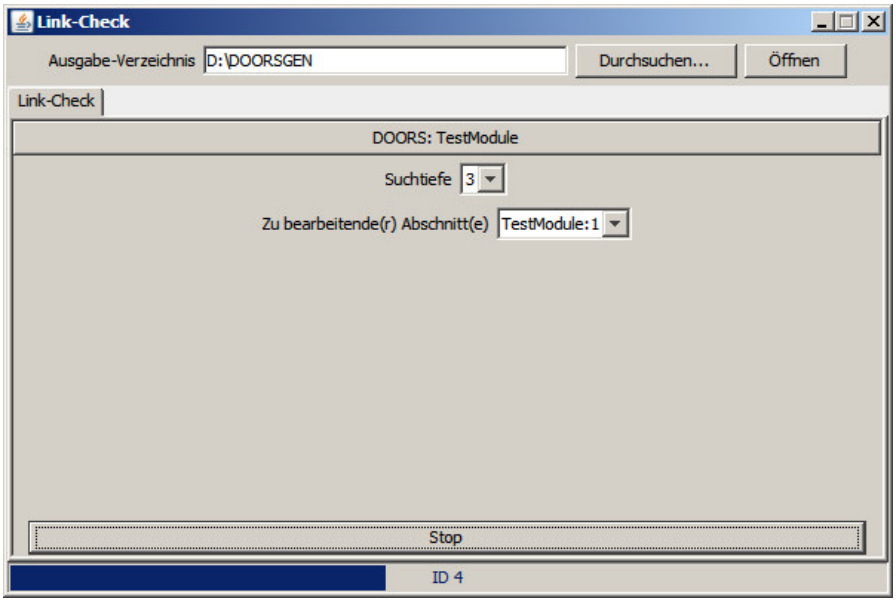


Abbildung 8.9: GUI eines Werkzeugs

Die GUI an sich wurde so konzipiert, dass diese zunächst unabhängig vom konkreten Anwendungsfall aufgebaut wird. Immer vorhanden ist das Ausgabeverzeichnis (in dem z.B. ein Report erstellt wird) im oberen Teil und die Statusleiste im unteren Teil. Das Ausgabeverzeichnis kann textuell oder über Schaltflächen eingestellt bzw. geöffnet werden. Die Statusleiste dient gleichzeitig als Fortschrittsanzeige.

Konkrete Anwendungsfälle sind Generierungs- oder Prüfaufgaben, die das Werkzeug realisieren soll. Diese werden in separate Klassen ausgelagert (sogenannten Tasks). Die Tasks zeigen sich dann als Tabs im Werkzeug. Aufgrund der sensiblen realen Daten für „DBCCheck“, wird der GUI an einem fiktiven Beispiel des Tasks „Link-Check“ beschrieben. In der Abbildung 8.9 wurde der Standard-GUI der Task „Link-Check“ hinzugefügt.

Folgende Methoden müssen für einen Task implementiert werden:

- `getTitle`: Diese Methode gibt zurück, wie der Title des Tasks und somit auch des zugehörigen Tabs sein soll. Das ermöglicht den Namen auch dynamisch erzeugen zu lassen. Ein Beispiel für die Implementierung der Methode ist in Listing 8.3 abgebildet.

```java
@Override
public String getTitle() {
  return "Link-Check";
}
```

Listing 8.3: Beispiel für getTitle

- `getDoorsEntryPoint`: Mit dieser Methode wird der Pfad zu dem zugrunde liegenden DOORS Formalmodul abgefragt. Der Name wird als Schaltfläche als erstes Element im Tab angezeigt. Bei Druck auf die Schaltfläche wird das entsprechende Dokument geöffnet. Mit dem Beispiel aus Listing 8.4 wird in Abbildung 8.9 die Schaltfläche mit „DOORS: TestModule“ beschriftet.

```java

@Override
public String getDoorsEntryPoint() {
  return "/TestModule";
}
```

Listing 8.4: Beispiel für getDoorsEntryPoint

- `fillOptionPane` In dieser Methode wird der Einstellungsbereich des Tasks konfiguriert. Es gibt vordefinierte Elemente für Checkboxen, Comboboxen, Textfelder mit einer Erweiterung für Dateinamen und Elemente für Informationen. In Abbildung 8.9 gibt es zwei Optionen, die per Combobox eingestellt werden: Zum einen die Suchtiefe für Links und zum anderen, welche Abschnitte berücksichtigt werden sollen.

  Im Beispiel in Listing 8.5 wird zunächst eine Liste aufgebaut, aus der für den Check ein Wert ausgewählt werden soll (Zeile 3–6). Danach wird die Option initialisiert, wobei die Werteliste und eine Anfangsauswahl übergeben werden (Zeile 7). Die Funktion von getProperty wird weiter unten erklärt. Mit der vom Check bereitgestellten Methode `void addOption(String, Option)` wird diese Option dann anschließend der grafischen Oberfläche hinzugefügt. Die übergebene Zeichenkette stellt die Bezeichnung der Option dar. Eine weitere Option ist in den Zeilen 10–11 angegeben. Die möglichen Werte werden direkt aus dem DOORS-Dokument extrahiert.

- `startTask` Diese Methode enthält die Implementierung des Tasks. Unter Benutzung der vom DOORS-Adapter angebotenen Methoden kann die Funktionalität realisiert werden. In der GUI wird zum Ausführen des Tasks am unteren Ende des Tabs eine Schaltfläche zum Starten angezeigt. Wenn der Task läuft, lässt sich dieser mit der gleichen Schaltfläche stoppen, weshalb diese in Abbildung 8.9 mit

```java
1  @Override
2  public void fillOptionPane() {
3    List<Integer> depths = new ArrayList<Integer>();
4    depths.add(1);
5    depths.add(2);
6    depths.add(3);
7    depth = new ComboBoxOption(depths, getProperty(DEPTH));
8    addOption("Suchtiefe", depth);
9
10   start = new ComboBoxOption(getDoors().getFormalModule(
        getDoorsEntryPoint()).getTop(), getProperty(START));
11   addOption("Zu bearbeitende(r) Abschnitt(e)", start);
12 }
```

Listing 8.5: Beispiel für fillOptionPane

„Stop“ beschriftet ist.

Ein Beispiel für diese Methode ist in Listing 8.6 abgebildet. Dabei wird auch gezeigt, mit welchen Mitteln die Fortschrittsanzeige und der Report erstellt werden können. In Zeile 4 wird ein Excel-Report initialisiert. Danach wird in Zeile 6 der Fortschrittsbalken durch die Methoden `setProgressMax(int max)` auf die Anzahl der zu überprüfenden Objekte gesetzt. Der Status des Fortschrittsbalkens wird in Zeile 11 durch `setProgressStatus` gesetzt und das Weiterlaufen des Fortschrittsbalkens wird durch `setProgressNext()` in Zeile 12 erreicht. Informationen für den Report werden diesem in Zeile 17 übergeben.

- `saveProperties` In dieser Methode werden Einstellungen angegeben, die gespeichert werden sollen. Listing 8.7 zeigt ein Beispiel für diese Speicherung. In Zeile 3 und Zeile 4 werden mittels der Methode `void saveProperty(String, Object)` Paare definiert, die zu speichernde Werte anzeigen. Der erste Parameter definiert einen Schlüssel (der im Beispiel für die gesamte Klasse als Konstante verfügbar ist) und einen Wert, der dem Schlüssel zugeordnet wird. Über die Methode `String getProperty(String)` können diese Werte bei der Initialisierung mit `fillOptionPane` genutzt werden, wie es im Listing 8.5 in den Zeilen 7 und 10 zu sehen ist.

Realisierung von Anforderung A4

Die GUI lässt sich über die Kommandozeile starten. Über DXL-Dateien können in DOORS die Menüs erweitert und Kommandozeilenaufrufe abgesetzt werden. Auf diese Weise lässt sich die GUI auf gleichem Wege innerhalb und außerhalb von DOORS

```java
@Override
public void startCheck() {
  FormalModule m = ((ReqObject) start.getSelection()).getModule
      ();
  initExcelSheet(m, "Link-Prüfung");
  int size = m.getObjects().size();
  setProgressMax(size);
  inspectsubs((ReqObject) start.getSelection(), (Integer) depth.
      getSelection());
}

protected void inspectsubs(ReqObject parent, int depth) {
  setProgressStatus(parent.getAbsNo());
  setProgressStatusNext();
  if (parent instanceof NormalObject || parent instanceof
      CellObject) {
    for (ReqObject sub : parent.getSubObjects()) {
      List<Link> outlinks = sub.getOutLinks();
      if (outlinks != null && outlinks.size() > 0) {
        errorhandler.addInfo(sub.getAbsNo(), sub.getModule().
            getName(), sub.getMainColumn(), "Outlink gefunden");
      }
      inspectsubs(sub, depth - 1);
    }
  }
}
```

Listing 8.6: Beispiel für startTask

```java
@Override
public void saveProperties() {
  saveProperty(DEPTH, depth.getSelection());
  saveProperty(START, start.getSelection());
}
```

Listing 8.7: Beispiel für das Speichern von Einstellungen

starten.

8.2.4 Erfahrungen beim Einsatz des Werkzeuges

Durch die einfache Möglichkeit das Abgleichs mit einem wesentlich geringeren Fehlerpotential und durch die Berücksichtigung der Entwicklerwünsche wurde das Werkzeug gut angenommen. Ein positiver Nebeneffekt war, dass bisher nicht erkannte unkritische Ab-

weichungen erkannt wurden und so zukünftige Fehler vermieden wurden. Die entwickelte Standard-GUI erlaubt, zukünftige Werkzeuge mit minimalem Aufwand zu entwickeln, was sich schon ab dem zweiten Werkzeug rentiert hat. Die so gewonnenen Ressourcen konnten für andere Arbeiten verwendet werden.

8.3 Weitere Beispiele für den Einsatz im Produktlinien-Einführungsprojekt

Wie schon erwähnt wurden im Rahmen des Produktlinien-Einführungsprojekts weitere Werkzeuge entwickelt, die den Entwicklungsprozess unterstützen. Für die Aktualisierung von Diagnoseinformationen aus DOORS heraus wurde das zweite Werkzeug „ODXGen“ entwickelt, welches in Abschnitt 8.3.1 beschrieben ist. Die Schnittstelle im Quellcode zwischen Domain und Application Engineering spielt für Produktlinien eine wichtige Rolle. Das Werkzeug „EVFGen“ in Abschnitt 8.3.2 wurde zu diesem Zweck, angepasst an die Gegebenheiten des Produktlinien-Einführungsprojekts, entwickelt. Zuletzt wird in Abschnitt 8.3.3 das Werkzeug „TestbedGen“ vorgestellt, welches erlaubt Testumgebungen für Artefakte des Domain Engineering zu generieren und so den Aufwand für den Test zu reduzieren.

8.3.1 Generierung von Diagnose-Daten aus DOORS (ODXGen)

Dieses Werkzeug „ODXGen“ bildet einen Generator auf Basis der in Abschnitt 7.3.2 erkannten Sprache für die Diagnoseschnittstelle dar. Statt diese Information händisch über externe Werkzeuge in den Entwicklungsprozess zu bringen, lässt sich viel einfacher ein Generator einsetzen. Als zusätzliche Herausforderung sollte bei dieser Generierung ein bestehendes Artefakt ergänzt und geändert werden, so dass die Generierung sehr gezielt mit der bestehenden Datei umgehen musste.

Die Arbeitsweise des Werkzeugs ist in Abbildung 8.10 schematisch zu sehen. ODX (Open Diagnostic Data Exchange) ist ein standardisiertes Format, welches zur Beschreibung der Steuergeräte-Diagnose zum Einsatz kommt [ODX08]. Sämtliche Informationen sind in einer XML-Struktur abgelegt. Es sind mögliche Ereignisse bzw. Ursachen und Reaktionsmöglichkeiten definiert. Diese Daten sind später in dem Werkzeug zur Diagnose eines Fehlers oder einer Auffälligkeit notwendig.

Für die Entwicklung sind diese Informationen in einem DOORS-Dokument abgelegt, die bisher händisch durch ein spezielles Werkzeug in das ODX-Format übertragen werden. Da dies ziemlich aufwändig ist und im Wesentlichen nur Einträge von DOORS in bestimmte Felder im Werkzeug kopiert werden mussten, wurde ein Weg gesucht, dies zu

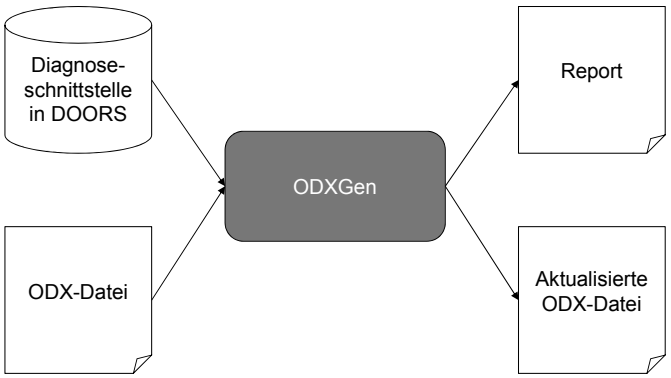


Abbildung 8.10: Das Werkzeug ODXGen

automatisieren. Die Informationen zu den Ereignissen machen nur einen Teil der ODX-Datei aus. Deshalb konnte diese nicht einfach von Grund auf generiert werden, sondern es musste eine bestehende Datei aktualisiert werden.

Wie beim Werkzeug DBCCheck mussten die Informationen aus DOORS und die bestehenden Daten aus der ODX-Datei zusammengeführt und die einzelnen Ereignisse korrekt zugeordnet werden. Die Änderungen zur bestehenden ODX-Datei wurden ermittelt und im Report dargestellt. Der Report ermöglicht einerseits die Dokumentation der Aktualisierung, andererseits werden die Daten selber nochmal geprüft. So werden Warnungen erzeugt, wenn die Daten offensichtlich unvollständig sind. Die Anforderungen A1, A3 und A4 gelten auch für dieses Werkzeug und die Realisierung wurde mit der gleichen GUI, jedoch mit einer anderen Task realisiert.

Für die Anbindung des ODX-Formats wurde DOM4J [DOM4J] als allgemeine XML-Bibliothek für Java verwendet. Durch die Unterstützung von XPath [XPath10] ist die Navigation durch die XML-Datei besonders einfach. Die Nutzung des allgemeinen (also nicht sprach-/xsd-spezifischen) Adapters war seitens der Lenkungsentwicklung gewünscht, um den Änderungsumfang beim Umstieg auf eine neuere Version des ODX-Formats so gering wie möglich zu halten. Letztendlich bietet ein allgemeiner Adapter auch die Möglichkeit, alle auf XML basierenden Sprachen einzubinden, z.B. auch das für den Anforderungsaustausch entwickelte Requirement Interchange Format (RIF) [RIF].

Der Einsatz des automatischen Updaters brachte einige Optimierungen. Die Aktualisierung wurde von mehreren Manntagen auf wenige Sekunden verkürzt. Durch das systematische Übertragen ist die Fehleranfälligkeit geringer, als wenn die Daten per Copy und Paste aktualisiert werden. Die Erkennung von Fehlern hat leichte Vorteile bei der automatisierten Lösung. Systematische potenzielle Fehlerquellen können gezielt überprüft werden. Bei der händischen Übertragung fallen eventuelle Fehler noch ins Auge, jedoch ist die Wahrscheinlichkeit höher, dass ein Fehler übersehen wird. Außerdem ermöglicht der ODX-Updater einen übersichtlichen Vergleich zwischen der ODX-Version

vor der Aktualisierung und der nach der Aktualisierung durch den erstellten Report. Bei der händischen Aktualisierung ist dies höchstens über einen unkomfortablen Baseline-Vergleich in DOORS möglich

8.3.2 Generierung von Schnittstellen-Code (EVFGen)

Ein Produkt entsteht im Application Engineering nicht allein durch das Zusammenfügen der Artefakte des Domain Engineering. Dazu ist noch Schnittstellencode (im Produktlinien-Einführungsprojekt C-Code) notwendig, der das Zusammenwirken der Artefakte ermöglicht. Der dafür entwickelte Generator „EVFGen“ unterstützt die Produktlinienentwicklung [BRRW10]. Die Funktion des Schnittstellencodes ist in Abbildung 8.11 dargestellt. Hauptaufgabe dieses Quellcodes ist die Weiterleitung von Methodenaufrufen aus der Basissoftware in die jeweiligen Funktionsmodule, die in Matlab/Simulink codiert sind. Es gibt jeweils eine Methode zur Initialisierung des Moduls beim Einschalten des Systems (init), zur Speicherung von Werten nach dem Ausschalten der Zündung (shutdown) und für den regelmäßigen Funktionsaufruf entsprechend des gewählten Taktrasters (step). Bei letzterem müssen für jede Ausführung ein- und ausgehende Signale koordiniert werden. Dazu existiert eine Möglichkeit, Werte vorzugeben, mit denen diese Signale manipuliert werden können, um die Sicherheitsfunktionen der Software zu testen.

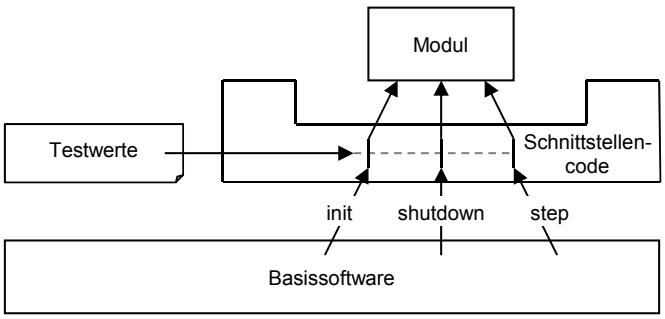


Abbildung 8.11: Aufgaben des Schnittstellencodes

Bei der Umsetzung des Generators sollten mehrere Ziele erfüllt werden:

- Gemeinsame und veränderliche Anteile sollen gut getrennt werden können.
- Die Anbindung an das Anforderungsmanagement soll variabel gestaltet sein. Das zu diesem Zeitpunkt eingesetzte Werkzeug zur Verwaltung von Anforderungen war DOORS. Der Generator sollte prinzipiell auch mit anderen Anforderungsmanagementwerkzeugen über eine variable Schnittstelle angewendet werden können.
- Die zu generierenden Artefakte sollen nicht nachbearbeitet werden. Daraus ergibt

sich, dass die generierten Artefakte fehlerfrei sein müssen und ohne weitere Anpassung kompiliert werden können.

- Für den Einsatz der Artefakte in der Lenkungssoftware ist es wichtig, dass diese die qualitativen Eigenschaften der bisher manuell erstellten Artefakte erfüllen. Dazu gehört die Misra-Kompatibilität [Mot04] sowie die Absicherung des generierten Quellcodes nach den Richtlinien zur Entwicklung von sicherheitskritischen Systemen [IEC10, ISO09].

Bei der Umsetzung des Generators wurde eine domänenspezifische Sprache entwickelt, die als Abstraktionsschicht für das Anforderungsmanagementsystem dient. Mittels MontiCore konnte basierend auf dieser DSL ein Generator erstellt werden. Für die Generierung von gemeinsamen Anteilen wurde die Templateengine Velocity [Vel] verwendet. In den Templates wurde der Quelltext so gestaltet, dass alle relevanten Richtlinien erfüllt wurden. Die veränderlichen Anteile waren weiterhin handcodiert und wurden in getrennten Dateien verwaltet.

So konnten mit EVFGen alle gesetzten Ziele erfüllt werden. Zusätzlich ergab sich eine Einsparungsmöglichkeit von 82% der Lines of Code für die DSL im Vergleich zum C-Code. Dies erleichtert die Fehlersuche und verringert die nötige Review-Zeit.

Beim praktischen Einsatz des Generators ergab sich jedoch, dass die Entwickler es einfacher fanden, den bekannten Quellcode zu modifizieren, als die Änderungen in einem unbekannten Modell vorzunehmen. Zusätzlich hatte EVFGen nur bei der Einführung neuer Module seine Stärken, da in diesem Fall der komplette Schnittstellencode für das Modul neu erstellt werden musste. Der überwiegende Änderungsanteil waren kleine punktuelle Änderungen, die beispielsweise durch zusätzliche Signale hervorgerufen wurden.

Deshalb wurde der Einsatzbereich des Generators geändert. Da besonders die Signaländerungen einen großen Impact haben und für jede Signaländerung auch der Manipulationsteil angepasst werden muss, ergab sich von den Entwicklern her der Wunsch, diesen Teil zu generieren. Änderungen eines Signals hatten Änderungen an mehreren Stellen des Manipulationscodes zur Folge. Um die Einsatzhürden für die Entwickler nochmals zu verringern, wurde als Basis für die Generierung ein Anforderungsdokument in DOORS verwendet. Dieses Anforderungsdokument wies schon eine sehr gute Strukturierung auf, so dass es als Modell genutzt werden konnte. Es waren nur kleine Änderungen notwendig, hauptsächlich um Prosa in eindeutige Attribute zu überführen, wodurch gleichzeitig die Modellqualität erhöht wurde.

Die gestellten Ziele wurden größtenteils auch mit dem neuen Generator erfüllt. Generierte und handcodierte Teile waren weiterhin getrennt, was eine kleine Überarbeitung des handcodierten Quellcodes notwendig machte. Die generierten Dateien mussten nicht nachträglich geändert werden und die notwendigen Richtlinien wurden erfüllt. Einzig

die Unabhängigkeit vom Anforderungsmanagementsystem war nicht mehr gegeben. Jedoch wurde dies Ziel durch die einfache Anwendbarkeit des Werkzeugs kompensiert. Die Qualität und die Möglichkeiten des Generators erlaubten es, ihn im laufenden Entwicklungsprozess einzusetzen.

8.3.3 Generierung von Test-Infrastruktur (TestbedGen)

Die Artefakte des Domain Engineering werden in den Lenkungsprojekten bei Volkswagen in Matlab/Simulink entwickelt und getestet. Zur Entwicklung, Code-Generierung und zum Test werden die sogenannten Module in einem dafür angepassten Testbed eingesetzt. Die Anpassung betrifft vor allem die Schnittstellen des Moduls, also die ein- und ausgehenden Signale. Eingangssignale müssen mit sinnvollen Daten stimuliert und Ausgangssignale gemessen werden.

Für das Hinzufügen, Löschen und Ändern von Signalen muss das Testmodell (siehe Abbildung 8.12) an mehreren Stellen angepasst werden. Dies ist teilweise nur umständlich über Kontextmenüs oder tiefer liegende Optionen möglich. Die notwendige manuelle gezielte Benennung von Teilen des Testmodells ist ebenfalls fehleranfällig. Zusätzlich zu den Änderungen müssen die enthaltenen Elemente lesbar angeordnet werden, um die Verständlichkeit zu gewährleisten.

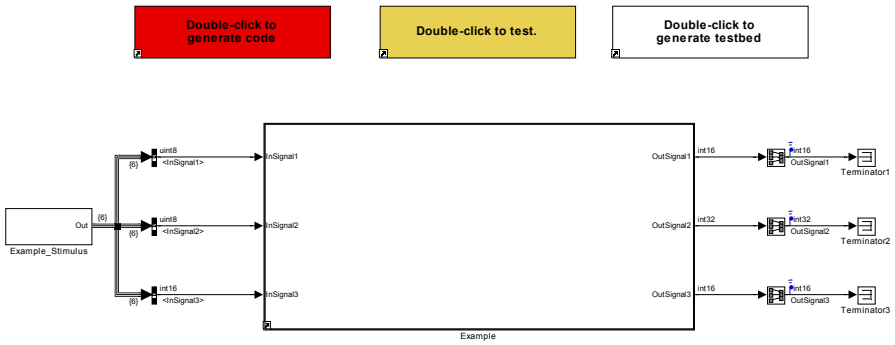


Abbildung 8.12: Testbed für Module

Um den Entwicklungsprozess zu optimieren und gleichzeitig den Entwicklern zu zeigen, dass die Optimierungen für eine Produktlinienentwicklung nicht immer zusätzliche Arbeiten nach sich ziehen, wurde für die Erstellung des Testbeds der Generator „TestbedGen“ erstellt [HRRW12]. Über einen zusätzlichen Button im bisherigen Testbed kann das gesamte Testbed, angepasst an das entsprechende Modul, neu erstellt werden. Alle Elemente werden durch den Generator an die richtige Stelle gesetzt, sodass alle Testmodelle gut lesbar sind und ein gleiches Aussehen haben. Um den Einsatz für die Entwickler möglichst einfach zu machen, wurde der Generator direkt in Matlab implementiert.

Die Generierung des Testmodells hat zu einer deutlichen Reduktion des Aufwands für Moduländerungen geführt. Durch den Generator konnte die umständliche, zeitaufwändige und fehleranfällige Erstellung der Test-Infrastruktur signifikant verringert werden. Mehrstündige Tätigkeiten wurden auf die wenigen Sekunden verkürzt, die der Generator benötigt.

Für die Entwickler bedeutete der Generator eine Vermeidung eines repetitiven einfachen Vorgangs, so dass diese sich auf anspruchsvollere Tätigkeiten konzentrieren konnten. Das einheitliche Erscheinungsbild verhindert, dass man sich in einem noch nicht bekannten Testmodell erst einmal zurecht finden muss. Somit ist es ein weiterer Vorteil, dass sich neue Entwickler besser in den bestehenden Entwicklungsprozess einarbeiten können.

8.4 Zusammenfassung

Die praktische Umsetzung der Ansätze für Produktlinien ist mit sehr viel Aufwand verbunden. Es sind viele kleine Schritte notwendig, um das Team für die Idee zu gewinnen, den anfallenden Aufwand für das Produktlinien-Einführungsprojekt bestmöglich zu kompensieren und Meilensteine in den Lenkungsprojekten nicht zu verzögern. Der Fokus der Arbeiten lag vor allem darin, den Werkzeugeinsatz im bestehenden Prozess zu erhöhen und dabei die Basis für die Entwicklung mittels Produktlinien zu schaffen. Die Berücksichtigung von Variabilität ist in einem weiteren Schritt wie in Abbildung 8.4 möglich.

Für den erhöhten Werkzeugeinsatz wurde eine Schnittstelle zu DOORS entwickelt, die die Nutzung der Informationen im Anforderungsmanagement vereinfacht. Auf Basis der in Abschnitt 7.3.2 identifizierten Sprachen wurden spezifische Adapter entwickelt, die Zugriff auf die verschiedensten Artefakte realisieren. Diese Schnittstelle zu DOORS verringert die technischen Hürden für die Entwicklung von Werkzeugen, die direkt Informationen aus DOORS beziehen.

Für die Werkzeugentwicklung ist jedoch nicht nur die Schnittstelle allein entscheidend, sondern es kommt auch darauf an, dass grundsätzliche Dinge wie grafische Oberfläche, Konfiguration und Fortschrittsanzeige vorhanden sind. Das dafür entwickelte Framework ermöglicht die einfache Entwicklung von Werkzeugen, bei denen für diese Punkte bereits Standardimplementierungen vorliegen. Somit kann sich ein Werkzeugentwickler auf die eigentliche Funktion des Werkzeugs konzentrieren.

Zuletzt wurden noch einige praktische Beispiele für Werkzeuge diskutiert. Eine wesentliche Erkenntnis ist, dass sich Werkzeuge entlang des gesamten Entwicklungsprozesses einsetzen lassen, sei es zum Abgleich von Anforderungen (DBCCheck in Abschnitt 8.2), zur Generierung von Quellcode (ODXGen in Abschnitt 8.3.1 und EVFGen in Abschnitt 8.3.2) oder zur Unterstützung des Tests (TestbedGen in Abschnitt 8.3.3). Alle Werk-

zeuge wurden erfolgreich im bestehenden Entwicklungsprozess integriert und haben so einen wesentlichen Beitrag für das Produktlinien-Einführungsprojekt geleistet.

Für die Entwicklung von Werkzeugen ist es wichtig eng mit den zukünftigen Nutzern der Werkzeuge zusammenzuarbeiten. Beim Generator EVFGen konnte erst durch Rückmeldungen der Nutzer ein wirklich unterstützendes Werkzeug entwickelt werden. Dieses Beispiel zeigt auch, wie herausfordernd die Einbringung von Forschungsergebnissen in bestehende Entwicklungen ist. Mit diesen praktischen Werkzeugen sind jedoch die Hürden für zukünftige Änderungen deutlich verringert worden.

Kapitel 9

Zusammenfassung und Ausblick

In dieser Arbeit wurden praktische Methoden und Techniken zur Einführung von Software-Produktlinien in einem industriellen Umfeld behandelt. Dieses Kapitel dient dazu, die wesentlichen Ergebnisse zusammenzufassen und einen Ausblick auf zukünftige Forschungen in diesem Bereich zu geben. Aufgrund der gegebenen Möglichkeiten konnten nicht alle Maßnahmen umgesetzt werden, woraus sich ein Erweiterungspotential ergibt.

9.1 Zusammenfassung

Zunächst wurde in Kapitel 2 in die Thematik der Software-Produktlinien eingeführt. Dabei wurden wesentliche Techniken und erfolgreiche Umsetzungen von Produktlinien dargestellt. Es hat sich gezeigt, dass es viele Ansätze für Produktlinien und deren Einführung gibt, jedoch die Umsetzung eines Ansatzes von vielen Faktoren abhängig ist. Eine Umsetzung für eine Einführung lässt sich aufgrund anderer Bedingungen schwer auf einen anderen Kontext übertragen. Deshalb ist in der Regel die Kombination von mehreren Teilen verschiedener Umsetzungen notwendig, um einem bestimmten Kontext gerecht zu werden.

In Kapitel 3 wurde das Produktlinien-Einführungsprojekt vorgestellt, welches die Überführung von mehreren Softwareentwicklungen für elektromechanische Lenksysteme bei Volkswagen in eine Softwareproduktlinie beinhaltet. Daraus wurden wesentliche Maßnahmen abgeleitet und beschrieben, die in diesem Kontext zur Einführung einer Produktlinie beitragen. Die Maßnahmen können nicht singulär ausgeführt werden, sondern benötigen in der Regel andere Maßnahmen, die vor ihnen ausgeführt werden müssen.

Da nicht alle Maßnahmen umgesetzt werden können, wurden in Kapitel 4 zunächst die Softwarestände genauer untersucht und damit die ersten Analyse-Maßnahme durchgeführt. Anhand einer Abstraktion der Softwarestände auf Systeme mit Komponenten und Datenflüssen sowie darauf basierenden Kennzahlen wurde die technische Variabilität

der produktidentifizierenden Funktionen ermittelt. Dies ermöglicht eine quantifizierbare Aussage über Gemeinsamkeiten und Unterschiede der Softwarestände, die auch im folgenden zum Projekt-Tracking verwendet werden kann.

Ebenso wichtig für die Auswahl der Maßnahmen ist die genaue Untersuchung der Einflussfaktoren für das Produktlinien-Einführungsprojekt in Kapitel 5. Eine Reihe unterschiedlicher Einflussfaktoren wurde dabei aufgeführt und für den Kontext bei Volkswagen bewertet. Dies erfolgte in den Kategorien quantitative und qualitative Einflussfaktoren, sowie Einflussfaktoren bedingt durch das Unternehmen und durch die Variabilität.

Aufgrund der gemeinsamen Analyse der technischen Variabilität und der Einflussfaktoren konnten in Kapitel 6 die Maßnahmen ausgewählt werden, die mit den gegebenen Bedingungen den größten Benefit brachten. Die konkreten Maßnahmen beinhalten eine Erweiterung und Schärfung der modellbasierten Entwicklung, die die wesentliche Grundlage für eine Software-Produktlinie darstellt. Durch diese Maßnahmen kann später in den Modellen die Variabilität besser berücksichtigt werden.

Für die modellbasierte Softwareentwicklung wurde in Kapitel 7 auf die verwendeten Modelle eingegangen. Dabei wurde beschrieben, wie sich weitere Sprachen identifizieren und beispielsweise über Metamodelle nutzen lassen. Besonders das System DOORS hat sich bei Volkswagen als Quelle für Modelle verschiedenster Arten und damit auch Modellierungssprachen ergeben. Für die Einbringung der Variabilität wurde gezeigt, mit welcher Methodik sich eine optimale Darstellungsform für Variabilität (positiv, negativ oder transformational) ermitteln lässt.

Modelle allein bringen für eine Entwicklung wenig, wenn nicht dazugehörige Werkzeuge erstellt werden. Kapitel 8 beschreibt die Entwicklung von Werkzeugen, die für die spätere Anwendung in einer Software-Produktlinie gedacht sind. Dabei hat sich ergeben, dass insbesondere der Zugriff auf die Quellinformationen durch die in Vorkapitel entwickelten Metamodelle sehr einfach realisierbar ist. Dies wurde anhand des Java-DOORS Adapters gezeigt, der Basis für eine Reihe der bei Volkswagen entwickelten Werkzeuge ist.

9.2 Ausblick

Die Etablierung einer Produktlinie ist mit den umgesetzten Maßnahmen noch lange nicht abgeschlossen. Vielmehr wurden wichtige Vorarbeiten gemacht, die die Anwendung weiterer Produktlinientechniken erst erlauben. Erst durch den Fokus Identifikation bzw. Auszeichnung von Modellen kann die Definition der Variabilität sinnvoll erfolgen. Der Einsatz von Generatoren muss dann unter Berücksichtigung der Variabilität erfolgen.

Die Behandlung von Variabilität, welche für Produktlinien eines der wesentlichen Ele-

mente ist, kann noch intensiver untersucht und umgesetzt werden. Insbesondere der Delta-Ansatz [SBB+10, Sch10] scheint für den betrachteten Kontext im Produktlinien-Einführungsprojekt vielversprechend zu sein, da die Variabilitätsinformationen separat zu den bestehenden Modellen abgelegt werden können und so die laufende Entwicklung nicht beeinträchtigt wird. Ebenso wurde nur auf die technischen Möglichkeiten der Variabilitätsbindung eingegangen, ohne jedoch den Bindungsprozess und Bindungszeitpunkt zu beschreiben. Dies wäre erst bei einer vollständigen Umsetzung des Produktlinienansatzes notwendig, was im Rahmen dieser Arbeit nicht möglich war.

Generell kann auf der bestehenden Arbeit durch die Umsetzung weiterer Maßnahmen aufgebaut werden. Die im Rahmen dieser Arbeit umgesetzten Maßnahmen sind diejenigen, die einen großen Benefit im Entwicklungsprozess bringen, um das Thema Software-Produktlinien zu verankern. Mit den weiteren Maßnahmen aus Abschnitt 3.2 kann der Entwicklungsprozess weiter für Produktlinien angepasst werden. Eventuell ist die Definition ergänzender Maßnahmen notwendig, da der bisherige Stand auch für die aufbauenden Arbeiten von Grund auf neu bewertet werden kann.

Die Beschreibung der gemachten Erfahrungen bei der Einführung von Produktlinien ist weiterhin notwendig. Es wird nie einen richtigen Weg geben, sondern mehrere, um das Ziel zu erreichen. In der Praxis ist ein großer Erfahrungsschatz im Umgang mit möglichen Produktlinien sehr wichtig, um verschiedene Situationen im Produktlinienkontext bewerten zu können. Dieser Erfahrungsschatz wird durch die Beschreibung von bisher erfolgten praktischen Umsetzungen ständig erweitert.

Insgesamt trägt diese Arbeit dazu bei, einen Einblick in die praktische Ausgestaltung von Software-Produktlinien zu bringen. Die theoretischen Konzepte können noch so gut sein, aber sie bringen wenig, wenn sie nicht in der Praxis angewendet werden können. Das Ziel dieser Arbeit war, diesen Praxisbezug herzustellen und zu zeigen, wie sich bestimmte Herausforderungen bei der Einführung von Produktlinien bewältigen lassen. Insofern ist auch folgenden Produktlinien-Einführungsprojekten geholfen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[ABM00] Atkinson, Colin ; Bayer, Joachim ; Muthig, Dirk: Component-Based Product Line Development: The KobrA Approach. In: *Proceedings of the First Software Product Line Conference (SPLC)*, 2000, S. 289–309

[AC04] Antkiewicz, Michal ; Czarnecki, Krzysztof: FeaturePlugin: Feature Modeling Plug-In for Eclipse. In: *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, 2004, S. 67–72

[AJTK09] Apel, Sven ; Janda, Florian ; Trujillo, Salvador ; Kästner, Christian: Model Superimposition in Software Product Lines. In: *Proceedings of the International Conference on Model Transformation (ICMT)*, 2009, S. 4–19

[ALMK08] Apel, Sven ; Lengauer, Christian ; Möller, Bernhard ; Kästner, Christian: An Algebra for Features and Feature Composition. In: *12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, 2008, S. 36–50

[AP10] Arenz, Andrea ; Potrykus, Sven: Modellbasierte Funktionsentwicklung. In: *ATZ - Automobiltechnische Zeitschrift* 112 (2010), Nr. 1, S. 28–32

[ASA10] ASAM: *Data Model for ECU Measurement and Calibration*. Feb 2010

[ASB+08] Alves, Vander ; Schwanninger, Christa ; Barbosa, Luciano ; Rashid, Awais ; Sawyer, Peter ; Rayson, Paul ; Pohl, Christoph ; Rummler, Andreas: An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 67–76

[Aut] AUTOSAR development cooperation: *AUTOSAR - AUTomotive Open System ARchitecture*. `http://www.autosar.org/`

[BAB+00] Boehm, Barry W. ; Abts, Chris ; Brown, A. W. ; Chulani, Sunita ; Clark, Bradford K. ; Horowitz, Ellis ; Madachy, Ray ; Reifer, Donald J. ; Steece, Bert: *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000

[Bat05] Batory, Don: Feature Models, Grammars, and Propositional Formulas. In: *Proceedings of the 9th International Software Product Line Conference (SPLC)*, 2005, S. 7–20

[BBM05] Berg, Kathrin ; Bishop, Judith ; Muthig, Dirk: Tracing Software Product Line Variability: From Problem to Solution Space. In: *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT)*, 2005, S. 182–191

[BBRC06] Batory, Don ; Benavides, David ; Ruiz-Cortés, Antonio: Automated Analysis of Feature Models: Challenges Ahead. In: *Communications of the ACM* 49 (2006), S. 45–47

[BCK03] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice*. 2. Auflage. Addison-Wesley Professional, 2003

[BCO96] Brownsword, Lisa ; Clements, Paul ; Olsson, Ulf: Successful Product Line Engineering: A Case Study / Carnegie-Mellon Software Engineering Institute. 1996 (CMU/SEI-96-TR-016). – Forschungsbericht

[Beu03] Beuche, Danilo: *Composition and Construction of Embedded Software Families*, Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg, Diss., 2003

[BFK+99] Bayer, Joachim ; Flege, Oliver ; Knauber, Peter ; Laqua, Roland ; Muthig, Dirk ; Schmid, Klaus ; Widen, Tanya ; DeBaud, Jean-Marc: PuLSE: A Methodology to Develop Software Product Lines. In: *Proceedings of the Symposium on Software Reusability (SSR)*, 1999, S. 122–131

[BG14] Benavides, David ; Galindo, José A.: Variability Management in an Unaware Software Product Line Company: An Experience Report. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2014, S. 5:1–5:6

[BHJ+03] Birk, Andreas ; Heller, Gerald ; John, Isabel ; Schmid, Klaus ; Müller, Klaus: Product Line Engineering: The State of the Practice. In: *IEEE Software* 20 (2003), S. 52–60

[BK04] Buhr, Kerstin ; Kolagari, Ramin T.: Softwarebasierte Produktlinien - Szenarien für Automobilhersteller und Zulieferer. In: *Softwaretechnik Trends* 24 (2004), S. 30–33

[BKPS04] Böckle, Günter (Hrsg.) ; Knauber, Peter (Hrsg.) ; Pohl, Klaus (Hrsg.) ; Schmid, Klaus (Hrsg.): *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt, 2004

[BM07] Braganca, Alexandre ; Machado, Ricardo J.: Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. In: *Proceedings of the 11th International Software Product Line Conference (SPLC)*, 2007, S. 3–12

[Boe88] Boehm, Barry W.: A Spiral Model of Software Development and Enhancement. In: *IEEE Computer* 21 (1988), Nr. 5, S. 61–72

[Bos99] Bosch, Jan: Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. In: *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA)*, 1999, S. 321–340

[Bos00] Bosch, Jan: *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000

[Bra07] Bragança, Alexandre: *Methodological Approaches and Techniques for Model Driven Development of Software Product Lines*, University of Minho, Diss., 2007

[BRR10a] Berger, Christian ; Rendel, Holger ; Rumpe, Bernhard: Measuring the Ability to Form a Product Line from Existing Products. In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010, S. 151–154

[BRR+10b] Berger, Christian ; Rendel, Holger ; Rumpe, Bernhard ; Busse, Carsten ; Jablonski, Thorsten ; Wolf, Fabian: Product Line Metrics for Legacy Software in Practice. In: *Proceedings of the 14th International Software Product Line Conference (SPLC)* Bd. 2, 2010, S. 247–250

[BRRW10] Berger, Christian ; Rendel, Holger ; Rumpe, Bernhard ; Wolf, Fabian: Generative Softwareentwicklung zur Optimierung der Konstruktion eingebetteter Softwaresysteme am Beispiel einer Lenkungssteuerung. In: *Proceedings des 11. Braunschweiger Symposiums Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel (AAET)*, 2010, S. 246–257

[BSRC10] Benavides, David ; Segura, Sergio ; Ruiz-Cortés, Antonio: Automa-

ted Analysis of Feature Models 20 Years Later: A Literature Review. In: *Information Systems* 35 (2010), S. 615–636

[BSTRC07] Benavides, David ; Segura, Sergio ; Trinidad, Pablo ; Ruiz-Cortés, Antonio: FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In: *Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2007, S. 129–134

[BTRC05] Benavides, David ; Trinidad, Pablo ; Ruiz-Cortés, Antonio: Automated Reasoning on Feature Models. In: *Proceedings of 17th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2005, S. 491–503

[Bur93] Burkhard, Neil: Reuse-Driven Software Processes Guidebook. Version 02.00.03 / Software Productivity Consortium. 1993 (SPC-92019). – Forschungsbericht

[C95] Norm ISO/IEC 9899:1990/Amd 1:1995 March 1995. *C Integrity*

[CA05] Czarnecki, Krzysztof ; Antkiewicz, Michal: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *Generative Programming and Component Engineering*. Springer Berlin / Heidelberg, 2005, S. 422–437

[CANoe] Vector: *CANoe*. `http://vector.com/vi_canoe_de.html`, Abruf: 24.08.2014

[CBUE02] Czarnecki, Krzysztof ; Bednasch, Thomas ; Unger, Peter ; Eisenecker, Ulrich: Generative Programming for Embedded Software: An Industrial Experience Report. In: *Generative Programming and Component Engineering*. Springer Berlin / Heidelberg, 2002, S. 156–172

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

[CFGK05] Conrad, Mirko ; Fey, Ines ; Grochtmann, Matthias ; Klein, Torsten: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei Daimler-Chrysler. In: *Informatik - Forschung und Entwicklung* 20 (2005), Nr. 1-2, S. 3–10

[CH06] Czarnecki, Krzysztof ; Helsen, Simon: Feature-based Survey of Model

Transformation Approaches. In: *IBM Systems Journal* 45 (2006), S. 621–645

[Che76] Chen, Peter P.: The Entity-Relationship Model: Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), S. 9–36

[CHE04] Czarnecki, Krzysztof ; Helsen, Simon ; Eisenecker, Ulrich: Staged Configuration using Feature Models. In: *Proceedings of the third International Software Product Line Conference (SPLC)*, 2004, S. 266–283

[CHE05] Czarnecki, Krzysztof ; Helsen, Simon ; Eisenecker, Ulrich: Formalizing Cardinality-based Feature Models and their Specialization. In: *Software Process: Improvement and Practice*, 2005, S. 7–29

[CHS10] Clarke, Dave ; Helvensteijn, Michiel ; Schaefer, Ina: Abstract Delta Modeling. In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, 2010, S. 13–22

[CJMN06] Clements, Paul C. ; Jones, Lawrence G. ; McGregor, John D. ; Northrop, Linda M.: Getting There From Here: A Roadmap for Software Product Line Adoption. In: *Communications of the ACM* 49 (2006), S. 33–36

[CKK06] Czarnecki, Krzysztof ; Kim, Chang Hwan P. ; Kalleberg, Karl T.: Feature Models are Views on Ontologies. In: *Proceedings of the 10th International Software Product Line Conference (SPLC)*, 2006, S. 41–51

[CKMM08] Carbon, Ralf ; Knodel, Jens ; Muthig, Dirk ; Meier, Gerald: Providing Feedback from Application to Family Engineering - The Product Line Planning Game at the Testo AG. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 180–189

[CMB10] Clements, Paul ; McGregor, John ; Bass, Len: Eliciting and Capturing Business Goals to Inform a Product Lines Business Case and Architecture. In: *Software Product Lines: Going Beyond*. Springer Berlin / Heidelberg, 2010, S. 393–405

[CN02] Clements, Paul ; Northrop, Linda: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002

[CSW08] Czarnecki, Krzysztof ; She, Steven ; Wasowski, Andrzej: Sample Spaces and Feature Models: There and Back Again. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 22–31

[CW07] Czarnecki, Krzysztof ; Wasowski, Andrzej: Feature Diagrams and Logics: There and Back Again. In: *Proceedings of the 11th International Software Product Line Conference (SPLC)*, 2007, S. 23–34

[Cza04] Czarnecki, Krzysztof: Overview of Generative Software Development. In: *Proceedings of Unconventional Programming Paradigms (UPP)*, 2004, S. 326–341

[DBC] Vector Informatik GmbH: *DBC-Kommunikations-Datenbasis für CAN*. `http://www.vector.com/vi_candb_de.html`, Abruf: 03.08.2011

[DGR11] Dhungana, Deepak ; Grünbacher, Paul ; Rabiser, Rick: The DOPLER Meta-tool for Decision-oriented Variability Modeling: A Multiple Case Study. In: *Automated Software Engineering* 18 (2011), S. 77–114

[Dij70] Dijkstra, Edsger W.: *Notes on Structured Programming*. April 1970. – circulated privately

[DK01] Deursen, Arie van ; Klint, Paul: Domain-Specific Language Design Requires Feature Descriptions. In: *Journal of Computing and Information Technology* 10 (2001), S. 1–17

[DM07] Donegan, Paula M. ; Masiero, Paulo C.: Design Issues in a Component-based Software Product Line. In: *Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse*, 2007

[DMH02] Dincel, Ebru ; Medvidovic, Nenad ; Hoek, André van d.: Measuring Product Line Architectures. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE 01)*, 2002, S. 346–352

[DOM4J] Sourceforge: *dom4j*. `http://dom4j.sourceforge.net/`, Abruf: 03.08.2011

[Dor] Rational, IBM: *DOORS*. `http://www.ibm.com/software/awdtools/doors/`, Abruf: 18.02.2012

[DRG07] Dhungana, Deepak ; Rabiser, Rick ; Grünbacher, Paul: Decision-Oriented Modeling of Product Line Architectures. In: *Proceedings of the 6th Working IEEE/IFIPO Conference on Software Architecture (WICSA)*, 2007

[DSB04] Deelstra, Sybren ; Sinnema, Marco ; Bosch, Jan: Experiences in Software Product Families: Problems and Issues During Product Derivation. In: *Proceedings of the Third International Software Product Line Conference (SPLC)*, 2004, S. 165–182

[DSB05] Deelstra, Sybren ; Sinnema, Marco ; Bosch, Jan: Product Derivation in Software Product Families: A Case Study. In: *Journal of Systems and Software* 74 (2005), S. 173–194

[DSF07] Djebbi, Olfa ; Salinesi, Camille ; Fanmuy, Gauthier: Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. In: *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE)*, 2007, S. 301–306

[DSW14] Damiani, Ferruccio ; Schaefer, Ina ; Winkelmann, Tim: Delta-oriented Multi Software Product Lines. In: *Proceedings of the 18th International Software Product Line Conference (SPLC)*, 2014, S. 232–236

[Ecl] Eclipse Foundation: *Eclipse*. `http://www.eclipse.org/`, Abruf: 07.11.2011

[EM05] Etxeberria, Leire ; Mendieta, Goiuria S.: Product-Line Architecture: New Issues for Evaluation. In: *Proceedings of the 9th International Software Product Line Conference (SPLC)*, 2005, S. 174–185

[ES13] Eichelberger, Holger ; Schmid, Klaus: A Systematic Analysis of Textual Variability Modeling Languages. In: *Proceedings of the 17th International Software Product Line Conference (SPLC)*, 2013, S. 12–21

[FHR08] Fieber, Florian ; Huhn, Michaela ; Rumpe, Bernhard: Modellqualität als Indikator für Softwarequalität: eine Taxonomie. In: *Informatik-Spektrum* 31 (2008), Oktober, Nr. 5, S. 408–424

[Flo08] Florenz, Bastian: *Software and System Architecture Evaluation and Analysis in the Automotive Domain*, Technische Universität Carolo Wilhemina zu Braunschweig, Diss., 2008

[FPDF95] Frakes, William ; Prieto-Díaz, Rubén ; Fox, Christopher: DARE: Domain Analysis and Reuse Environment. In: *Proceedings of the 7th Annual Workshop on Software Reuse*, 1995

[FPDF98] Frakes, Bill ; Prieto-Díaz, Rubén ; Fox, Christopher: DARE: Domain

Analysis and Reuse Environment. In: *Annals of Software Engineering* 5 (1998), S. 125–141

[GBS01] Gurp, Jilles V. ; Bosch, Jan ; Svahnberg, Mikael: On the Notion of Variability in Software Product Lines. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001, S. 45–54

[GE10] Gustavsson, Håkan ; Eklund, Ulrik: Architecting Automotive Product Lines: Industrial Practice. In: *Software Product Lines: Going Beyond*. Springer Berlin / Heidelberg, 2010, S. 92–105

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

[GHK+07] Grönniger, Hans ; Hartmann, Jochen ; Krahn, Holger ; Kriebel, Stefan ; Rumpe, Bernhard: View-Based Modeling of Function Nets. In: *Proceedings of the Object-oriented Modelling of Embedded Real-TimeSystems (OMER4) Workshop*, 2007

[GHK+08a] Grönniger, Hans ; Hartmann, Jochen ; Krahn, Holger ; Kriebel, Stefan ; Rothhardt, Lutz ; Rumpe, Bernhard: Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In: *Proceedings of 4th International Congress on Embedded Real-Time Systems (ERTS)*, 2008

[GHK+08b] Grönniger, Hans ; Hartmann, Jochen ; Krahn, Holger ; Kriebel, Stefan ; Rothhardt, Lutz ; Rumpe, Bernhard: View-Centric Modeling of Automotive Logical Architectures. In: *Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008

[GHK+15] Greifenberg, Timo ; Hölldobler, Katrin ; Kolassa, Carsten ; Look, Markus ; Mir Seyed Nazari, Pedram ; Müller, Klaus ; Navarro Perez, Antonio ; Plotnikov, Dimitri ; Reiss, Dirk ; Roth, Alexander ; Rumpe, Bernhard ; Schindler, Martin ; Wortmann, Andreas: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In: *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015, S. 74–85

[GKPR08] Grönniger, Hans ; Krahn, Holger ; Pinkernell, Claas ; Rumpe, Bernhard: Modeling Variants of Automotive Systems using Views. In: *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, 2008, S. 76–89

[GKR+08] Grönniger, Hans ; Krahn, Holger ; Rumpe, Bernhard ; Schindler, Martin ; Völkel, Steven: MontiCore: a Framework for the Development of Textual Domain Specific Languages. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008, S. 925–926

[GKRS06] Grönniger, Hans ; Krahn, Holger ; Rumpe, Bernhard ; Schindler, Martin: Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In: *Modellierung*, 2006, S. 67–81

[GL00] Gannod, Gerald C. ; Lutz, Robyn R.: An Approach to Architectural Analysis of Product Lines. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, 2000, S. 548–557

[GLS08] Gruler, Alexander ; Leucker, Martin ; Scheidemann, Kathrin: Calculating and Modeling Common Parts of Software Product Lines. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 203–212

[Gom04] Gomaa, Hassan: *Designing Software Product Lines with UML.* Addison Wesley, 2004

[Grö10] Grönniger, Hans: *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*, RWTH Aachen University, Diss., 2010

[Han12] Handelsblatt: *Autos nach dem Prinzip Lego.* `http://www.handelsblatt.com/auto/test-technik/die-neuen-produktions-baukaesten-autos-nach-dem-prinzip-lego/7176548.html`. Version: 2012, Abruf: 22.11.2012

[HDM03] Hoek, André van d. ; Dincel, Ebru ; Medvidovic, Nenad: Using Service Utilization Metrics to Assess the Structure of Product Line Architectures. In: *Proceedings of the 9th International Symposium on Software Metrics (METRICS)*, 2003, S. 298–308

[HFT04] Hein, Andreas ; Fischer, Thomas ; Thiel, Steffen: Fahrerassistenzsysteme bei der Robert Bosch GmbH. In: *Software-Produktlinien: Methoden, Einführung und Praxis.* dpunkt, 2004, S. 193–205

[HHK+13] Haber, Arne ; Hölldobler, Kathrin ; Kolassa, Carsten ; Look, Markus ; Müller, Klaus ; Rumpe, Bernhard ; Schäfer, Ina: Engineering Delta Modeling Languages. In: *Proceedings of the 17th International Software Product Line Conference (SPLC)*, 2013, S. 22–31

[HKK04] Hardung, Bernd ; Kölzow, Thorsten ; Krüger, Andreas: Reuse of Software in Distributed Embedded Automotive Systems. In: *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*, 2004, S. 203–210

[HKM+13] Haber, Arne ; Kolassa, Carsten ; Manhart, Peter ; Mir Seyed Nazari, Pedram ; Rumpe, Bernhard ; Schaefer, Ina: First-Class Variability Modeling in Matlab/Simulink. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2013, S. 11–18

[HKO+06] Her, Jin S. ; Kim, Ji H. ; Oh, Sang H. ; Rhew, Sung Y. ; Kim, Soo D.: A Framework for Evaluating Reusability of Core Asset in Product Line Engineering. In: *Information and Software Technology* 49 (2006), S. 740–760

[HKR+11a] Haber, Arne ; Kutz, Thomas ; Rendel, Holger ; Rumpe, Bernhard ; Schaefer, Ina: Delta-oriented Architectural Variability Using MontiCore. In: *Proceedings of the First International Workshop on Software Architecture Variability (SAVA)*, 2011

[HKR+11b] Haber, Arne ; Kutz, Thomas ; Rendel, Holger ; Rumpe, Bernhard ; Schaefer, Ina: Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models. In: *Proceedings of VARY International Workshop*, 2011, S. 43–52

[HMA10] Hamza, Haitham S. ; Martinez, Jabier ; Alonso, Carmen: Introducing Product Line Architectures in the ERP Industry: Challenges and Lessons Learned. In: *Proceedings of the 14th International Software Product Line Conference (SPLC)* Bd. 2, 2010, S. 263–265

[HMPO+08] Haugen Øystein ; Møller-Pedersen, Birger ; Oldevik, Jon ; Olsen, Gøran K. ; Svendsen, Andreas: Adding Standardized Variability to Domain Specific Languages. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 139–148

[HR04] Harel, David ; Rumpe, Bernhard: Meaningful Modeling: What's the Semantics of "Semantics"? In: *Computer* 37 (2004), Nr. 10, S. 64–72

[HRR12] Haber, Arne ; Ringert, Jan Oliver ; Rumpe, Bernard: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems / RWTH Aachen. Version: february 2012. `http://aib.informatik.`

rwth-aachen.de/2012/2012-03.pdf. 2012 (AIB-2012-03). – Forschungsbericht

[HRRS11] Haber, Arne ; Rendel, Holger ; Rumpe, Bernhard ; Schaefer, Ina: Delta Modeling for Software Architectures. In: *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, 2011, S. 1–10

[HRRS12] Haber, Arne ; Rendel, Holger ; Rumpe, Bernhard ; Schaefer, Ina: Evolving Delta-oriented Software Product Line Architectures. In: *Foundations of Computer Software, Development, Operation and Management of Large-Scale Complex IT Systems*, 2012, S. 183–208

[HRRW12] Hopp, Christian ; Rendel, Holger ; Rumpe, Bernhard ; Wolf, Fabian: Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In: *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik*, 2012, S. 181–192

[IEC91] Norm ISO/IEC 9126 December 1991. *Software Engineering - Product Quality - Part 1: Quality Model*

[IEC06] Norm ISO/IEC 15504 März 2006. *SPICE (Software Process Improvement and Capability Determination)*

[IEC10] Norm DIN EN 61508 April 2010. *Funktionale Sicherheit sicherheitsbezogener elektrischer,elektronischer und programmierbarer elektronischer Systeme*

[IEE91] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. In: *IEEE Std 610* (1991), Jan, S. 1–217

[ISO09] Vornorm DIS 26262 Juli 2009. *Road vehicles – Functional safety*

[Jacob] Sourceforge: *JACOB - Java COM Bridge*. http://sourceforge.net/projects/jacob-project/, Abruf: 19.10.2011

[Java] Oracle: *Java*. http://www.java.com/de/, Abruf: 03.08.2011

[JE09] John, Isabel ; Eisenbarth, Michael: A Decade of Scoping - A Survey. In: *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009, S. 31–40

[JH05] Joshi, Anjali ; Heimdahl, Mats: Model-Based Safety Analysis of Simulink

Models Using SCADE Design Verifier. In: *Computer Safety, Reliability, and Security*. Springer Berlin / Heidelberg, 2005, S. 122–135

[JIBX] SOURCEFORGE: *JiBX*. `http://jibx.sourceforge.net/`, Abruf: 15.06.2014

[JSB+08] JABLONSKI, Thorsten ; SCHUMANN, Heiko ; BUSSE, Carsten ; HAUSSMANN, Heiko ; HALLMANN, Udo ; DREYER, Dirk ; SCHÖTTLER, Frank: Die neue elektromechanische Lenkung APA-BS. In: *ATZElektronik* 1 (2008), S. 30–35

[JUnit] JUNIT: *JUnit*. `http://junit.org/`, Abruf: 06.09.2015

[KAK08] KÄSTNER, Christian ; APEL, Sven ; KUHLEMANN, Martin: Granularity in Software Product Lines. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008, S. 311–320

[KBB+02] KNAUBER, Peter ; BERMEJO, Jesus ; BÖCKLE, Günter ; PRADO LEITE, Julio Cesar S. ; LINEN, Frank van d. ; NORTHROP, Lind ; STARK, Michael ; WEISS, David M.: Quantifying Product Line Benefits. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE 01)*, 2002, S. 155–163

[KCH+90] KANG, Kyo C. ; COHEN, Sholom ; HESS, James ; NOWAK, William ; PETERSON, Spencer: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon Software Engineering Institute. 1990 (CMU/SEI-90-TR-21). – Forschungsbericht

[KCH+92] KANG, Kio C. ; COHEN, Sholom G. ; HOLIBAUGH, Robert ; PERRY, James ; PETERSON, Spencer: A Reuse-based Software Development Methodology / Carnegie-Mellon Software Engineering Institute. 1992 (CMU/SEI-92-SR-004). – Forschungsbericht

[KKL+98] KANG, Kyo C. ; KIM, Sajoong ; LEE, Jaejoon ; KIM, Kijoo ; KIM, Gerard J. ; SHIN, Euiseob: FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. In: *Annals of Software Engineering* 5 (1998), S. 143–168

[KLD02] KANG, Kyo C. ; LEE, Jaejoon ; DONOHOE, Patrick: Feature-oriented Product Line Engineering. In: *IEEE Software* 19 (2002), S. 58–65

[KN06] KISHI, Tomoji ; NODA, Natsuko: Formal Verification and Software Product Lines. In: *Communication of the ACM* 49 (2006), S. 73–77

[KPRR04] Kamsties, Erik ; Pohl, Klaus ; Reis, Sacha ; Reuys, Andreas: Anforderungsbasiertes Testen. In: *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt, 2004, S. 119–136

[Kra10] Krahn, Holger: *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*, RWTH Aachen University, Diss., 2010

[KRR15] Kolassa, Carsten ; Rendel, Holger ; Rumpe, Bernhard: Evaluation of Variability Concepts for Simulink in the Automotive Domain. In: *Proceedings of the 48th Hawaii International Conference on System Sciences* (2015), S. 5373–5382

[Kru02] Krueger, Charles: Eliminating the Adoption Barrier. In: *IEEE Software* 19 (2002), S. 29–31

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. 3. Auflage. Addison-Wesley Longman, Amsterdam, 2003

[KRV08] Krahn, Holger ; Rumpe, Bernhard ; Völkel, Steven: MontiCore: Modular Development of Textual Domain Specific Languages. In: *Proceedings of Tools Europe* Bd. 11, Springer-Verlag Berlin-Heidelberg, 2008 (Lecture Notes in Business Information Processing)

[KRV10] Krahn, Holger ; Rumpe, Bernhard ; Völkel, Steven: MontiCore: A Framework for Compositional Development of Domain Specific Languages. In: *International Journal on Software Tools for Technology Transfer (STTT)* 12 (2010), S. 353–372

[Kut11] Kutz, Thomas: *Entwicklung von Softwarearchitektur-Produktlinien mittels Delta-Modellierung*, RWTH Aachen University, Masterarbeit, 2011

[LL10] Ludewig, Jochen ; Lichter, Horst: *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2010

[LLE07] Lau, Kung-Kiu ; Ling, Ling ; Elizondo, Perla V.: Towards Composing Software Components in Both Design and Deployment Phases. In: *Proceedings of the 10 International Symposium on Component-Based Software Engineering (CBSE)*, 2007, S. 274–282

[MA02] Muthig, Dirk ; Atkinson, Colin: Model-Driven Product Line Architectures. In: *Proceedings of the Second International Conference on Software Product Lines (SPLC)*, 2002, S. 110–129

[MA09] Montagud, Sonia ; Abrahão, Silvia: Gathering Current Knowledge about Quality Evaluation in Software Product Lines. In: *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009, S. 91–100

[MBRB10] Mengi, Cem ; Babur, Önder ; Rendel, Holger ; Berger, Christian: Model-driven Configuration of Function Net Families in Automotive Software Engineering. In: *Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE)*, 2010, S. 49–60

[MC] Lehrstuhl Software Engineerng, RWTH Aachen: *MontiCore*. `http://www.monticore.de/`, Abruf: 27.01.2015

[Men12] Mengi, Cem: *Automotive Software - Prozesse, Modelle und Variabilität*, RWTH Aachen University, Diss., 2012

[MHS05] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and How to Develop Domain-Specific Languages. In: *ACM Computing Surveys* 37 (2005), S. 316–344

[MLF+10] Marinho, Fabiana ; Lima, Fabrício ; Filho, João F. ; Rocha, Lincoln ; Maia, Marcio ; Aguiar, Saulo de ; Dantas, Valéria ; Viana, Windson ; Andrade, Rossana ; Teixeira, Eldânae ; Werner, Cláudia: A Software Product Line for the Mobile and Context-Aware Applications Domain. In: *Software Product Lines: Going Beyond*. Springer Berlin / Heidelberg, 2010, S. 346–360

[MLM+13] Malavolta, Ivano ; Lago, Patricia ; Muccini, Henry ; Pelliccione, Patrizio ; Tang, Antony: What Industry Needs from Architectural Languages: A Survey. In: *IEEE Transactions on Software Engineering* 39 (2013), Nr. 6, S. 869–891

[MO07] Mebane, Holt ; Ohta, Joni T.: Dynamic Complexity and the Owen Firmware Product Line Program. In: *Proceedings of the 11th International Software Product Line Conference (SPLC)*, 2007, S. 212–222

[Mot04] Motor Industry Software Reliability Association: *MISRA-C 2004: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2004

[MR14a] Müller, Klaus ; Rumpe, Bernhard: A Model-Based Approach to Impact Analysis Using Model Differencing. In: *Proceedings of the 8th International*

Workshop on Software Quality and Maintainability (SQM) Bd. 65, 2014 (Electronic Communications of the EASST)

[MR14b] Müller, Klaus ; Rumpe, Bernhard: User-Driven Adaptation of Model Differencing Results. In: *Proceedings of International Workshop on Comparison and Versioning of Software Models (CVSM)* Bd. 34, 2014 (GI Softwaretechnik-Trends 2), S. 25–29

[MR15] Mir Seyed Nazari, Pedram ; Rumpe, Bernhard: Using Software Categories for the Development of Generative Software. In: *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015, S. 498–503

[MS10] Mannion, Mike ; Savolainen, Juha: Aligning Business and Technical Strategies for Software Product Lines. In: *Software Product Lines: Going Beyond*. Springer Berlin / Heidelberg, 2010, S. 406–419

[NA10] Nolan, Andy ; Abrahão, Silvia: Dealing with Cost Estimation in Software Product Lines: Experiences and Future Directions. In: *Software Product Lines: Going Beyond*. Springer Berlin / Heidelberg, 2010, S. 121–135

[Nei80] Neighbors, James M.: *Software construction using components*, Department Information and Computer Science, University of California, Diss., 1980

[NK08] Noda, Natsuko ; Kishi, Tomoji: Aspect-Oriented Modeling for Variability Management. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 213–222

[ODX08] Norm ISO 22901-1 November 2008. *Road vehicles – Open diagnostic data exchange (ODX)*

[OMG10a] Object Management Group: *OMG Systems Modeling Language (OMG SysML) Version 1.2*. May 2010. – OMG document number formal/2010-05-05

[OMG10b] Object Management Group: *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3*. May 2010. – OMG document number formal/2010-05-03

[OMG10c] Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3*. May 2010. – OMG document number formal/2010-05-05

[OMG11] Object Management Group: *Business Process Model and Notation (BPMN)*. January 2011. – OMG document number formal/2011-01-03

[Pan11] Pander, Jürgen: *Was bringt Spritspartechnik?* http://www.spiegel.de/auto/aktuell/0,1518,771630,00.html. Version: 2011, Abruf: 27.04.2011

[Par76] Parnas, David L.: On the Design and Development of Program Families. In: *IEEE Transactions on Software Engineering* 2 (1976), Nr. 1, S. 1–9

[PBL05] Pohl, Klaus ; Böckle, Günter ; Linden, Frank van der: *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer Berlin / Heidelberg, 2005

[PFR02] Pree, Wolfgang ; Fontoura, Marcus ; Rumpe, Bernhard: Product Line Annotations with UML-F. In: *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*. London, UK : Springer Berlin / Heidelberg, 2002. – ISBN 3–540–43985–4, S. 188–197

[PH11] Pfeffer, Peter (Hrsg.) ; Harrer, Manfred (Hrsg.): *Lenkungshandbuch: Lenksysteme, Lenkgefühl, Fahrdynamik von Kraftfahrzeugen*. Vieweg+Teubner Verlag, 2011

[Pil98] Piller, Frank Thomas: *Kundenindividuelle Massenproduktion: die Wettbewerbsstrategie der Zukunft*. Hanser, München, 1998

[POI] Apache: *Apache POI - the Java API for Microsoft Documents*. http://poi.apache.org/, Abruf: 03.08.2011

[PSZ08] Peng, Xin ; Shen, Liwei ; Zhao, Wenyun: Feature Implementation Modeling Based Product Derivation in Software Product Line. In: *Proceedings of the 10th International Conference on Software Reuse (ICSR)*, 2008, S. 142–153

[Pur] pure systems: *pure::variants*. http://www.pure-systems.com/pure_variants.49.0.html, Abruf: 03.11.2011

[RBSP02] Riebisch, Matthias ; Böllert, Kai ; Streitferdt, Detlef ; Philippow, Ilka: Extending Feature Diagrams with UML Multiplicities. In: *6th World Conference on Integrated Design & Process Technology (IDPT)*, 2002

[RGDM03] Raith, Ch. ; Gesele, F. ; Dick, W. ; Miegler, M.: Vernetzte Produkt-

entwicklung am Beispiel Audi dynamic steering. In: *VDI-Berichte* 1789 (2003), S. 185–205

[RIF] Herstellerinitative Software (HIS): *Requirements Interchange Format*. `http://www.automotive-his.de/rif`, Abruf: 27.04.2011

[RR15] Roth, Alexander ; Rumpe, Bernhard: Towards Product Lining Model-Driven Development Code Generators. In: *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015, S. 539–545

[RRW13] Ringert, Jan Oliver ; Rumpe, Bernhard ; Wortmann, Andreas: MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In: *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, 2013

[RRW14] Ringert, Jan Oliver ; Rumpe, Bernard ; Wortmann, Andreas: *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014 (Aachener Informatik-Berichte, Software Engineering 20)

[Rum11] Rumpe, Bernhard: *Modellierung mit UML*. 2. Auflage. Springer Berlin / Heidelberg, 2011

[Rum12] Rumpe, Bernhard: *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. 2. Auflage. Springer Berlin / Heidelberg, 2012

[SBB+10] Schaefer, Ina ; Bettini, Lorenzo ; Bono, Viviana ; Damiani, Ferruccio ; Tanzarella, Nico: Delta-Oriented Programming of Software Product Lines. In: *Proceedings of the 14th International Software Product Line Conference (SPLC)*, 2010, S. 77–91

[Sch04] Schreiber, Annette: Transitionsprozesse. In: *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt, 2004, S. 139–163

[Sch10] Schaefer, Ina: Variability Modelling for Model-Driven Development of Software Product Lines. In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010, S. 85–92

[SD10] Schaefer, Ina ; Damiani, Ferruccio: Pure Delta-Oriented Programming. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*, 2010, S. 49–56

[Sdd15] Soares, Larissa Rocha ; do Carmo Machado, Ivan ; de Almeida, Eduardo Santana: Non-Functional Properties in Software Product Lines: A Reuse Approach. In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2015, S. 67:67–67:74

[SHT06] Schobbens, Pierre-Yves ; Heymans, Patrick ; Trigaux, Jean-Christophe: Feature Diagrams: A Survey and a Formal Semantics. In: *Proceedings of IEEE International Conference on Requirements Engineering (RE)*, 2006, S. 139–148

[Simulink] MathWorks: *Simulink*. `http://de.mathworks.com/products/simulink/`, Abruf: 19.01.2015

[SJ04] Schmid, Klaus ; John, Isabel: A Customizable Approach to Full Lifecycle Variability Management. In: *Science of Computer Programming* 53 (2004), S. 259–284

[SPK06] Sugumaran, Vijayan ; Park, Sooyong ; Kang, Kyo C.: Software Product Line Engineering. In: *Communications of the ACM* 49 (2006), S. 29–32

[SRG11] Schmid, Klaus ; Rabiser, Rick ; Grünbacher, Paul: A Comparison of Decision Modeling Approaches in Product Lines. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, 2011, S. 119–126

[SRVK10] Sprinkle, Jonathan ; Rumpe, Bernhard ; Vangheluwe, Hans ; Karsai, Gabor: Metamodelling: State of the Art, and Research Challenges. In: *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2010, S. 59–78

[SSA14a] Seidl, Christoph ; Schaefer, Ina ; Assmann, Uwe: Capturing Variability in Space and Time with Hyper Feature Models. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2014, S. 6:1–6:8

[SSA14b] Seidl, Christoph ; Schaefer, Ina ; Assmann, Uwe: Integrated Management of Variability in Space and Time in Software Families. In: *Proceedings of the 18th International Software Product Line Conference (SPLC)*, 2014, S. 22–31

[SSDJ12] Schöttler, Frank ; Schillak, Christian ; Dannehr, Boris ; Jablonski,

Thorsten: Neuartige elektromechanische Lenkung für ein Premiumfahrzeug. In: *ATZElektronik* 1 (2012), S. 40–44

[SSS14] Schuster, Sven ; Schulze, Sandro ; Schaefer, Ina: Structural Feature Interaction Patterns: Case Studies and Guidelines. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2014, S. 14:1–14:8

[Sta73] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Springer Berlin / Heidelberg, 1973

[STB+04] Steger, Mirjam ; Tischer, Christian ; Boss, Birgit ; Müller, Andreas ; Pertler, Oliver ; Stolz, Wolfgang ; Ferber, Stefan: Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In: *Proceedings of the Third International Software Product Line Conference (SPLC)*, 2004, S. 34–50

[Syn] Rational, IBM: *Synergy*. `http://www.ibm.com/software/awdtools/synergy/`, Abruf: 18.02.2012

[TH02] Thiel, Steffen ; Hein, Andreas: Modeling and Using Product Line Variability in Automotive Systems. In: *IEEE Software* 19 (2002), S. 66–72

[Thi02] Thiel, Steffen: On the Definition of a Framework for an Architecting Process Supporting Product Family Development. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE 01)*, 2002, S. 125–142

[THSC06] Trigaux, Jean-Christophe ; Heymans, Patrick ; Schobbens, Pierre-Yves ; Classen, Andreas: Comparative semantics of Feature Diagrams: FFD vs. vDFD. In: *Proceedings of Workshops on Comparative Evaluation in Requirements Engineering*, 2006, S. 36–47

[VAC+09] Vogel, Oliver ; Arnold, Ingo ; Chughtai, Arif ; Ihler, Edmund ; Kehrer, Timo ; Mehlig, Uwe ; Zdun, Uwe: Architektur-Mittel (WOMIT). Version: 2009. In: *Software-Architektur*. Spektrum Akademischer Verlag, 2009. – ISBN 978–3–8274–1933–0, 125–310

[Vel] Apache: *Velocity*. `http://velocity.apache.org/`, Abruf: 20.10.2011

[VG07] Voelter, Markus ; Groher, Iris: Handling Variability in Model Trans-

formations and Generators. In: *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, 2007

[VK05] Verlage, Martin ; Kiesgen, Thomas: Five Years of Product Line Engineering in a Small Company. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, S. 534–543

[VMod] Industrieanlagen-Betriebsgesellschaft mbH: *Das V-Modell*. http://www.v-modell.iabg.de/, Abruf: 18.10.2011

[Vol08] Volkswagen AG: *Volkswagen Konzernvorstand treibt Zukunftsprojekt „Strategie 2018“ voran*. http://www.volkswagenag.com/content/vwcorp/info_center/de/news/2008/11/group_strategy_2018.html. Version: 2008, Abruf: 15.12.2013

[Wei98] Weiss, David M.: Commonality Analysis: A Systematic Process for Defining Families. In: *Proceedings of ARES Workshop on Development and Evolution of Software Architectures for Product Families*, 1998, S. 214–222

[WM14] Weiland, Jens ; Manhart, Peter: A Classification of Modeling Variability in Simulink. In: *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2014, S. 7:1–7:8

[WSB+08] White, Jules ; Schmidt, Douglas C. ; Benavides, David ; Trinidad, Pablo ; Ruiz-Cortés, Antonio: Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In: *Proceedings of the 12th International Software Product Line Conference (SPLC)*, 2008, S. 225–234

[XPath10] Dyck, Michael ; Robie, Jonathan ; Snelson, John ; Chamberlin, Don: XML Path Language (XPath) 2.0 (Second Edition) / W3C. 2010. – W3C Recommendation. – http://www.w3.org/TR/2010/REC-xpath20-20101214/

[YGM06] Yoshimura, Kentaro ; Ganesan, Dharmalingam ; Muthig, Dirk: Defining a Strategy to Introduce a Software Product Line using Existing Embedded Systems. In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT)*, 2006, S. 63–72

[ZC03] Zubrow, David ; Chastek, Gary: Measures for Software Product Lines / Carnegie-Mellon Software Engineering Institute. 2003 (CMU/SEI-2003-TN-031). – Forschungsbericht

[ZDW+08] Zhang, Tao ; Deng, Lei ; Wu, Jian ; Zhou, Qiaoming ; Ma, Chunyan: Some Metrics for Accessing Quality of Product Line Architecture. In: *Proceedings of the International Conference on Computer Science and Software Engineering*, 2008, S. 500–503

[ZHJ03] Ziadi, Tewfik ; Hélouët, Loïc ; Jézéquel, Jean-Marc: Towards a UML Profile for Software Product Lines. In: *Proceedings of the 5th International Workshop on Product Familiy Engineering (PFE)*, 2003, S. 129–139

Anhang A

DOORS-Offline-Adapter

A.1 DXL-Skript für den DOORS-Export

DXL

```
1  // Export for MontiCore
2
3  /* Struktur in eine Datei schreiben */
4
5  // prevent dxl timeout dialog
6  pragma runLim, 0
7  pragma encoding, "UTF-8"
8
9  string FOLDEROUT
10 string indent = ""
11 Stream fileout
12 int progress = 0
13 Skip accessedItems = createString
14
15 //-------------format-----------------------------------
16 string formatFolder(Folder folder){
17     string base = fullName (folder)
18     string result = ""
19     int i = 1
20     while (i < length base){
21         string c = base[i:i]
22         if (c == "/"){
23             result = result "."
24         }
25         else if (c == " " || c == "-" || c == ".") {
26             result = result "_"
27         }
28         else if (c == "ä") {
29             result = result "ae"
30         }
31         else if (c == "Ä") {
```

```
32              result = result "Ae"
33          }
34          else if (c == "ö") {
35              result = result "oe"
36          }
37          else if (c == "Ö") {
38              result = result "Oe"
39          }
40          else if (c == "ü") {
41              result = result "ue"
42          }
43          else if (c == "Ü") {
44              result = result "Ue"
45          }
46          else if (c == "ß") {
47              result = result "ss"
48          }
49          else {
50              result = result c
51          }
52          i++
53      }
54      return result
55  }
56
57  string formatModule(string modname){
58      string result = ""
59      int i = 0
60      while (i < length modname){
61          string c = modname[i:i]
62          if (c == "/" || c == "-" || c == "." || c== " ") {
63              result = result "_"
64          }
65          else if (c == "ä") {
66              result = result "ae"
67          }
68          else if (c == "Ä") {
69              result = result "Ae"
70          }
71          else if (c == "ö") {
72              result = result "oe"
73          }
74          else if (c == "Ö") {
75              result = result "Oe"
76          }
77          else if (c == "ü") {
78              result = result "ue"
79          }
80          else if (c == "Ü") {
```

```
81              result = result "Ue"
82          }
83          else if (c == "ß") {
84              result = result "ss"
85          }
86          else if (c == "-") {
87            result = result "_";
88          }
89          else {
90              result = result c
91          }
92          i++
93      }
94      return result
95  }
96
97  //-------------formatnvc-----------------------------------
98  Buffer formatnvc(string input){
99      Buffer source = create
100     source = input
101     int sourcelength = length source
102     Buffer result = create
103     int i = 0
104     while (i < sourcelength){
105         string c = source[i:i]
106         if (c == "\n"){
107             result += "\\n"
108         }
109         else if (c == "\r") {
110             result += "\\r"
111         }
112         else if (c == "\"") {
113             result += "\\\""
114         }
115         else if (c == "\\") {
116             result += "\\\\"
117         }
118         else {
119             result += c
120         }
121         i++
122     }
123     return result
124 }
125
126 //-------------format default values
      ----------------------------------
127 Buffer formatdefaultvalues(string input){
128     Buffer source = create
```

```
129     source = input
130     int sourcelength = length source
131     Buffer result = create
132     int i = 0
133     while (i < sourcelength){
134         string c = source[i:i]
135         if (c == "\n"){
136             result += "\", \""
137         }
138         else if (c == "\r") {
139             result += "\\r"
140         }
141         else if (c == "\"") {
142             result += "\\\""
143         }
144         else if (c == "\\") {
145             result += "\\\\"
146         }
147         else {
148             result += c
149         }
150         i++
151     }
152     return result
153 }
154
155 //-------------getPackage-----------------------------------
156 string getpackage(Folder folder) {
157   if (null(folder)){
158     return null string
159   } else {
160     string result = formatFolder folder
161     return result
162   }
163 }
164
165 //-------------doindent-------------------------------------
166 void doindent() {
167     indent = indent "    "
168 }
169
170 //-------------unindent-------------------------------------
171 void unindent() {
172     if (length indent >= 4){
173         indent = indent[4:]
174     }
175 }
176
177 //-------------makebasedir----------------------------------
```

```
178 string makebasedir(string package) {
179   if(null(package)){
180     return FOLDEROUT
181   } else {
182     int i = 0
183     string basedir = FOLDEROUT
184     string newdir = ""
185     while (i<length package){
186         if (package[i] == '.'){
187             basedir = basedir "/" newdir
188             // no fail if dir exists
189             noError()
190             mkdir basedir
191             lastError()
192             newdir = ""
193         }
194         else {
195             newdir = newdir package[i:i]
196         }
197         i++
198     }
199     if (newdir != ""){
200         basedir = basedir "/" newdir
201         // no fail if dir exists
202         noError()
203         mkdir basedir
204         lastError()
205     }
206     return basedir
207   }
208 }
209
210 void exportFolder(Folder f){
211   if (!null(f)){
212     string uid = uniqueID(f)
213     if (!find(accessedItems,uid)){
214       string package = null
215       string basedir
216       if (type(f)=="Project"){
217         basedir = makebasedir(null)
218       }
219       else{
220         package = getpackage(getParentFolder(f))
221         basedir = makebasedir(package)
222       }
223       Folder parent = getParentFolder(f)
224       string modname = formatModule name f
225       if (null parent){
226         modname = "Root_Folder"
```

```
227       } else {
228         exportFolder(parent)
229       }
230       fileout = write basedir "/" modname ".mce"
231       if (!null(package)) {
232         fileout << "package " package ";\n\n"
233       }
234       fileout << "folder "
235       if (type(f)=="Project"){
236         fileout << "project "
237       }
238       fileout << "\"" modname "\" " (isDeleted(f)?"deleted ":"") "
              {\n"
239       doindent()
240
241       // OriginalName
242       fileout << indent "originalname \"" (name f) "\"\n"
243       // UniqueId
244       fileout << indent "uniqueid \"" uniqueID(f) "\"\n"
245       // parent folder
246       if (!null(package) || (type(f)=="Project")) {
247         fileout << indent "parentfolder \"" fullName(
                getParentFolder(f)) "\"\n"
248       }
249
250       // DOORS URL
251       fileout << indent "url \"" getURL(f) "\"\n"
252       // Description
253       fileout << indent "description \"" description(f) "\"\n"
254
255       unindent()
256       fileout << indent "}\n"
257
258       close fileout
259       put(accessedItems,uid,f)
260     }
261   }
262 }
263
264 //-------------converttomcrichtext
      ------------------------------------
265 string converttomcrichtext(string rtfstring, Object o) {
266   string result = "richtext {"
267   int olecounter = 1
268
269   RichTextParagraph rp
270   RichText rt
271   bool bullets = false
272   for rp in rtfstring do {
```

```
273     result = result "paragraph " (rp.isBullet?"bullet ":"") " {"
274     for rt in rp do {
275       result = result "text "
276       result = result (rt.isOle?"ole ":"")
277       result = result (rt.bold?"bold ":"")
278       result = result (rt.italic?"italic ":"")
279       result = result (rt.underline?"underline ":"")
280       result = result (rt.strikethru?"strikethrough ":"")
281       result = result (rt.superscript?"superscript ":"")
282       result = result (rt.subscript?"subscript ":"")
283       result = result (rt.newline?"newline ":"")
284       result = result (rt.isUrl?"url ":"")
285
286       string rtcontent = ""
287       if (rt.isOle) {
288         string package = getpackage(getParentFolder(module o))
289         string oledir = makebasedir(package)
290         oledir = oledir "/" formatModule name module o
291         noError()
292         mkdir oledir
293         lastError()
294         Stream oleout = write oledir  "/" o."Absolute Number" "_"
              olecounter ".ole"
295         oleout << oleRtf (rt.getEmbeddedOle)
296         close oleout
297         rtcontent = (formatModule name module o) "/" o."Absolute
              Number" "_" olecounter ".ole"
298         olecounter++
299       } else {
300         rtcontent = stringOf formatnvc rt.text
301       }
302       result = result  "{\"" rtcontent "\"}"
303     }
304     result = result "}"
305   }
306   return result "}"
307 }
308
309 //-------------writemoduledefinition
      ----------------------------------
310 void writemoduledefinition(Module m, Stream outstream){
311     // Typdefinitionen
312     AttrType modAttrTypeDef
313     for modAttrTypeDef in m do {
314         outstream << indent "typedef \"" modAttrTypeDef.name "\"
              {\n"
315         doindent()
316         outstream << indent "basetype " modAttrTypeDef.type "\n"
317         if (modAttrTypeDef.type == attrInteger) {
```

```
318             if (modAttrTypeDef.minValue) {
319                 int minvalue = modAttrTypeDef.minValue
320                 outstream << indent "minvalue \"" minvalue "\"\n"
321             }
322             if (modAttrTypeDef.maxValue) {
323                 int maxvalue = modAttrTypeDef.maxValue
324                 outstream << indent "maxvalue \"" maxvalue "\"\n"
325             }
326         }
327         if (modAttrTypeDef.type == attrReal) {
328             if (modAttrTypeDef.minValue) {
329                 real minvalue = modAttrTypeDef.minValue
330                 outstream << indent "minvalue \"" minvalue "\"\n"
331             }
332             if (modAttrTypeDef.maxValue) {
333                 real maxvalue = modAttrTypeDef.maxValue
334                 outstream << indent "maxvalue \"" maxvalue "\"\n"
335             }
336         }
337         if (modAttrTypeDef.type == attrDate) {
338             if (modAttrTypeDef.minValue) {
339                 Date minvalue = modAttrTypeDef.minValue
340                 outstream << indent "minvalue \"" stringOf (
                        dateAndTime(minvalue)) "\"\n"
341             }
342             if (modAttrTypeDef.maxValue) {
343                 Date maxvalue = modAttrTypeDef.maxValue
344                 outstream << indent "maxvalue \"" stringOf (
                        dateAndTime(maxvalue)) "\"\n"
345             }
346         }
347         if (modAttrTypeDef.type == attrEnumeration){
348             outstream << indent "values "
349             string sep = ""
350             int i = 0
351             for (i=0; i<modAttrTypeDef.size; i++) {
352               outstream << sep modAttrTypeDef.values[i] ":" (
                      getRealColorOptionForTypes()?modAttrTypeDef.
                      colors[i]:getRealColor(modAttrTypeDef.colors[i]))
                       ":\"" modAttrTypeDef.strings[i] "\""
353               sep = ", "
354             }
355             outstream << "\n"
356         }
357         unindent()
358         outstream << indent "}\n"
359     }
360
361     // Attributdefinitionen
```

```
362     AttrDef modAttrDef
363     string attformodule = ""
364     string attforobject = ""
365     string sepmod = ""
366     string sepobj = ""
367     for modAttrDef in m do {
368       if (modAttrDef.module) {
369           attformodule = attformodule sepmod "\"" modAttrDef.name
                "\""
370           sepmod = ", "
371       }
372       if (modAttrDef.object) {
373           attforobject = attforobject sepobj "\"" modAttrDef.name
                "\""
374           sepobj = ", "
375       }
376       outstream << indent "attrdef \"" modAttrDef.name "\"" (
              modAttrDef.system?" system":"") (modAttrDef.multi?"
              multivalued":"") " {\n"
377       doindent()
378       outstream << indent "type \"" modAttrDef.typeName "\"\n"
379       outstream << indent "description \"" modAttrDef.description
              "\"\n"
380       if (modAttrDef.defval) {
381           AttrType modAttrTypeDef = modAttrDef.type
382           if (modAttrTypeDef.type == attrInteger) {
383               int val = modAttrDef.defval
384               outstream << indent "default \"" val "\"\n"
385           }
386           else if (modAttrTypeDef.type == attrReal) {
387               real val = modAttrDef.defval
388               outstream << indent "default \"" val "\"\n"
389           }
390           else if (modAttrTypeDef.type == attrDate) {
391               Date val = modAttrDef.defval
392               outstream << indent "default \"" stringOf (
                      dateAndTime(val)) "\"\n"
393           }
394           else if (modAttrTypeDef.type == attrUsername ||
                  modAttrTypeDef.type == attrEnumeration ||
                  modAttrTypeDef.type == attrString || modAttrTypeDef.
                  type == attrText) {
395               string val = modAttrDef.defval
396               outstream << indent "default " converttomcrichtext(
                      val,null) "\n"
397           }
398       }
399       if (modAttrDef.dxl) {
400         outstream << indent "dxl \"" (formatnvc (modAttrDef.dxl))
```

```
            "\"\n"
401       }
402       unindent()
403       outstream << indent "}\n"
404     }
405
406     outstream << indent "moduleattributes: " attformodule ";\n"
407     outstream << indent "objectattributes: " attforobject ";\n"
408
409     // Attributinhalte
410     string modAttrName
411     for modAttrName in attributes m do {
412         AttrDef ad = find(m, modAttrName)
413         if (!(null m.modAttrName) && !(m.modAttrName ""==="")){
414             string s = richTextWithOle(m.modAttrName)
415             outstream << indent "\"" modAttrName "\" \"" s "\"\n"
416         }
417     }
418 }
419
420 void exportLinkMod(Module lm){
421   if (!null(lm)){
422     string uid = uniqueID(lm)
423     if (!find(accessedItems,uid)){
424       string package = getpackage(getParentFolder(lm))
425       string basedir = makebasedir(package)
426       string modname = formatModule name lm
427       Stream fileout2 = write basedir "/" modname ".mce"
428       fileout2 << "package " package ";\n\n"
429       fileout2 << "linkmodule \"" modname "\" {\n"
430       doindent()
431
432       // OriginalName
433       fileout2 << indent "originalname \"" (name lm) "\"\n"
434       // UniqueId
435       fileout2 << indent "uniqueid \"" uniqueID(lm) "\"\n"
436       if (!null(package)) {
437         // parent folder
438         fileout2 << indent "parentfolder \"" fullName(
              getParentFolder(lm)) "\"\n"
439       }
440       // DOORS URL
441       fileout2 << indent "url \"" getURL(lm) "\"\n"
442       // Description
443       fileout2 << indent "description \"" description(lm) "\"\n"
444       // Version
445       fileout2 << indent "version \"" version(lm) "\"\n"
446
447       writemoduledefinition(lm, fileout2)
```

```
448
449       if (!isVisible lm){
450         lm = read(fullName(lm), true)
451       }
452       Object o
453       for o in lm do {
454         Object source
455         Object target
456         Linkset ls = linkset o
457         load ls
458         refresh lm
459         getSource(ls,source)
460         getTarget(ls,target)
461         fileout2 << indent "linkset \"" fullName (module (source))
                "\" -> \""  fullName (module (target)) "\"\n"
462       }
463
464       unindent()
465       fileout2 << indent "}\n"
466
467       close fileout2
468       put(accessedItems,uid,lm)
469     }
470   }
471 }
472
473 //-------------writeobject-----------------------------------
474 bool writeobject(Object o, Module m) {
475     if (progressCancelled) {
476         if (confirm("Exit loop?")) {
477             progressStop
478             return false
479         }
480     }
481     progressMessage "Exportiere " o."Absolute Number" ""
482     fileout << indent "reqobject " (isDeleted(o)?"deleted ":"") "
            {\n"
483     doindent()
484     progressStep ++progress
485     // DOORS URL
486     fileout << indent "url \"" getURL(o) "\"\n"
487     fileout << indent "level " level(o) "\n"
488     fileout << indent "number \"" number(o) "\"\n"
489     string objAttrName
490     for objAttrName in m do {
491         AttrDef ad = find(current Module, objAttrName)
492         if (!(null o.objAttrName)){
493             AttrType at = find(current Module, ad.typeName)
494
```

```
495             string s
496             // only convert html for Integer and Real attributes
497             if (at.type == attrInteger || at.type == attrReal ) {
498                 fileout << indent "\"" objAttrName "\" \"" o.
                        objAttrName "\"\n"
499             }
500             else if (at.type == attrDate) {
501                 Date value = o.objAttrName
502                 if (null value){
503                   fileout << indent "\"" objAttrName "\" \"\"\n"
504                 }
505                 else {
506                   fileout << indent "\"" objAttrName "\" \""
                        stringOf (dateAndTime(value)) "\"\n"
507                 }
508             }
509             else {
510                 fileout << indent "\"" objAttrName "\" "
                        converttomcrichtext(richTextWithOle(o.
                        objAttrName),o) "\n"
511             }
512         }
513     }
514
515     Object subo
516     for subo in o do {
517         if (!writeobject(subo, m)) {
518             return false
519         }
520     }
521
522     unindent()
523     fileout << indent "}\n"
524     return true
525 }
526
527 //-------------Auslesen-----------------------------------
528 void writeout(DB progressDB) {
529     Object o
530     Module m = current
531     load view "Standard view"
532     filtering off
533     showDeletedObjects (true)
534
535     string currentview = currentView m
536     int objectcount = 0
537     for o in all m do {
538         objectcount++
539     }
```

```
540     progressStart(progressDB, "MC Export", "", objectcount);
541     progressMessage "Initialisierung"
542
543     string package = getpackage(getParentFolder(m))
544
545     string basedir = makebasedir(package)
546
547     exportFolder(getParentFolder(m))
548
549     string modname = formatModule name m
550     fileout = write basedir "/" modname ".mce"
551
552     progressMessage "Auslesen"
553
554     fileout << "package " getpackage(getParentFolder(m)) ";\n\n"
555     fileout << "module \"" modname "\" {\n"
556     doindent()
557
558     // OriginalName
559     fileout << indent "originalname \"" (name m) "\"\n"
560     // UniqueId
561     fileout << indent "uniqueid \"" uniqueID(m) "\"\n"
562     // UniqueId of parent folder
563     fileout << indent "parentfolder \"" fullName(getParentFolder(m
            )) "\"\n"
564     // DOORS URL
565     fileout << indent "url \"" getURL(m) "\"\n"
566     // Description
567     fileout << indent "description \"" description(m) "\"\n"
568     // Version
569     fileout << indent "version \"" version(m) "\"\n"
570
571     // Baselines
572     Baseline b
573     for b in m do {
574         fileout << indent "baseline  {\n"
575         doindent()
576         fileout << indent "major " major(b) "\n"
577         fileout << indent "minor " minor(b) "\n"
578         fileout << indent "suffix \"" suffix(b) "\"\n"
579         fileout << indent "annotation \"" annotation(b) "\"\n"
580         fileout << indent "date \"" stringOf (dateAndTime(dateOf b
                )) "\"\n"
581         fileout << indent "user \"" user(b) "\"\n"
582         unindent()
583         fileout << indent "}\n"
584     }
585
586     writemoduledefinition(m, fileout)
```

```
587
588     for o in top m do {
589         writeobject(o, m)
590     }
591
592     // Views
593     string viewName
594     for viewName in views m do {
595         fileout << indent "viewdef \"" viewName "\" {\n"
596         doindent()
597         load view viewName
598         Filter f = current
599         if (!null f) {
600             fileout << indent "filterrule \"" stringOf(m ,f) "\"\n
                    "
601         }
602         Sort s = current
603         if (!null s) {
604             fileout << indent "sortrule \"" stringOf(s) "\"\n"
605         }
606         Column c
607         for c in m do {
608             string att = attrName(c)
609             if (null att) {
610                 if (main(c)) {
611                     fileout << indent "maincolumn\n"
612                 } else {
613                     fileout << indent "dxlcolumn \"" title c "\"
                            \"" (formatnvc (dxl (c))) "\"\n"
614                 }
615             } else {
616                 fileout << indent "attributecolumn \"" title c "\"
                        \"" att "\"\n"
617             }
618         }
619         unindent()
620         fileout << indent "}\n"
621     }
622     fileout << indent "defaultviewformodule \""
            getDefaultViewForModule(m) "\"\n"
623     fileout << indent "defaultviewforuser \""
            getDefaultViewForUser(m) "\"\n"
624
625     //Links
626     for o in m do {
627         // loading all source modules (otherwise incoming links
                are not detected)
628         ModName_ srcModRef
629         for srcModRef in o<-"*" do {
```

```
630             read(fullName(srcModRef), false)
631         }
632
633         Link l
634         AttrDef modAttrDef
635         for l in o <- "*" do {
636             Module linkmod = module(l)
637             exportLinkMod(linkmod)
638             string lm = fullName(linkmod)
639             ModName_ sourceMod = source l
640             string sourceModName = formatModule name sourceMod
641             string fullsourceModName =  fullName sourceMod
642             if (fullsourceModName != fullName m){
643                 fileout << indent "link {\n"
644                 doindent()
645                 fileout << indent "SourceModule \""
                        fullsourceModName "\"\n"
646                 fileout << indent "LinkModule \"" lm "\"\n"
647                 fileout << indent "SourceAbsoluteNumber "
                        sourceAbsNo(l) "\n"
648                 fileout << indent "TargetAbsoluteNumber "
                        targetAbsNo(l) "\n"
649                 for modAttrDef in linkmod do {
650                     if (modAttrDef.object) {
651                         string att = modAttrDef.name
652                         if (!(l.att ""=="")) {
653                             fileout << indent "\"" att "\" \"" l.
                                    att "\"\n"
654                         }
655                     }
656                 }
657                 unindent()
658                 fileout << indent "}\n"
659             }
660         }
661         for l in o -> "*" do {
662             Module linkmod = module(l)
663             exportLinkMod(linkmod)
664             string lm = fullName(linkmod)
665             ModName_ targetMod = target l
666             string targetModName = formatModule name targetMod
667             string fulltargetModName =  fullName targetMod
668             fileout << indent "link {\n"
669             doindent()
670             if (fulltargetModName != fullName (m)){
671                 fileout << indent "TargetModule \""
                        fulltargetModName "\"\n"
672             }
673             fileout << indent "LinkModule \"" lm "\"\n"
```

```
674             fileout << indent "SourceAbsoluteNumber " sourceAbsNo(
                    l) "\n"
675             fileout << indent "TargetAbsoluteNumber " targetAbsNo(
                    l) "\n"
676             for modAttrDef in linkmod do {
677                 if (modAttrDef.object) {
678                     string att = modAttrDef.name
679                     if (!(l.att ""=="")) {
680                         fileout << indent "\"" att "\" \"" l.att "
                                \"\n"
681                     }
682                 }
683             }
684             unindent()
685             fileout << indent "}\n"
686         }
687     }
688
689     unindent()
690     fileout << indent "}\n"
691
692     progressMessage "Speichern"
693     close fileout
694
695     progressMessage "Modul öffnen"
696     close (m, false)
697
698     progressStop
699 }
700
701 //------------main
      ------------------------------------------------------
702 if ((current Module) == null){
703     ack "Es ist kein Modul gewählt"
704 }
705 else {
706     string modulename = (current Module)."Name"
707     // Formalmodule nicht leer, hat Objekte
708     if ((current Object) == null) {
709         ack "Es sind keine Objekte in der aktuellen View vorhanden
                "
710     }
711     else {
712         Object o = current
713         DB parseBox = create("MontiCore Export", styleCentered|
                styleFixed)
714         label(parseBox, "Zielverzeichnis angeben")
715         DBE folderIn = field(parseBox, "Zieldatei", "D:\\temp",
                40)
```

```
716         void doGet(DB parseBox) {
717             FOLDEROUT = get folderIn
718             writeout(parseBox)
719             hide(parseBox)
720             infoBox "finished"
721         }
722         apply(parseBox, "Exportieren", doGet)
723         show parseBox
724     }
725 }
```

Listing A.1: DXL-Skript für den Export aus DOORS

A.2 DSL für den DOORS-Import

MG

```
1  package de.se_rwth.doors;
2
3  grammar DoorsExport {
4
5    options {
6      compilationunit Item
7    }
8
9    token INT = ('-')? ('0'..'9')+ : int;
10
11   enum Basetype = "Integer" | "Real" | "String" | "Text" | "Date"
         | "Enumeration" | "Username" | "Bool";
12
13   interface Item;
14
15   ast Item =
16     Name
17     originalname:STRING
18     uniqueid:STRING
19     parentfolder:STRING
20     url:STRING
21     description:STRING;
22
23
24   Folder implements Item =
25     "folder" (["project"])? name:STRING (["deleted"])? "{"
26     // Name with Umlauts etc.
27     "originalname" originalname:STRING
28     "uniqueid" uniqueid:STRING
29     ("parentfolder" parentfolder:STRING)?
30     "url" url:STRING
```

```
31    "description" description:STRING
32  "}";
33
34  interface Module extends Item;
35
36  ast Module =
37    modversion:STRING
38    moduleTypes:TypeDefinition*
39    attributes:AttributeDefinition*
40    moduleAttributes:STRING*
41    objectAttributes:STRING*
42    moduleValues:AttributeValue*;
43
44
45  DescriptiveModule implements Module =
46    "descriptivemodule" name:STRING (["deleted"])? "{"
47    // Name with Umlauts etc.
48    "originalname" originalname:STRING
49    "uniqueid" uniqueid:STRING
50    "parentfolder" parentfolder:STRING
51    "url" url:STRING
52    "description" description:STRING
53    "version" modversion:STRING
54  "}";
55
56  LinkModule implements Module =
57    "linkmodule" name:STRING (["deleted"])? "{"
58    // Name with Umlauts etc.
59    "originalname" originalname:STRING
60    "uniqueid" uniqueid:STRING
61    "parentfolder" parentfolder:STRING
62    "url" url:STRING
63    "description" description:STRING
64    "version" modversion:STRING
65    moduleTypes:TypeDefinition*
66    attributes:AttributeDefinition*
67    "moduleattributes" ":" (moduleAttributes:STRING ("," 
          moduleAttributes:STRING)*)? ";"
68    "objectattributes" ":" (objectAttributes:STRING ("," 
          objectAttributes:STRING)*)? ";"
69    moduleValues:AttributeValue*
70    LinkSet*
71  "}";
72
73  LinkSet =
74    "linkset" from:STRING "->" to:STRING;
75
76  FormalModule implements Module =
77    "module" name:STRING (["deleted"])? "{"
```

```
78      // Name with Umlauts etc.
79      "originalname" originalname:STRING
80      "uniqueid" uniqueid:STRING
81      "parentfolder" parentfolder:STRING
82      "url" url:STRING
83      "description" description:STRING
84      "version" modversion:STRING
85      Baseline*
86      moduleTypes:TypeDefinition*
87      attributes:AttributeDefinition*
88      "moduleattributes" ":" (moduleAttributes:STRING ("," moduleAttributes:STRING)*)? ";"
89      "objectattributes" ":" (objectAttributes:STRING ("," objectAttributes:STRING)*)? ";"
90      moduleValues:AttributeValue*
91      ReqObject*
92      ViewDefinition*
93      "defaultviewformodule" defaultviewformodule:STRING
94      "defaultviewforuser" defaultviewforuser:STRING
95      Link*
96    "}";
97
98    Baseline =
99      "baseline" "{"
100       "major" major:INT
101       "minor" minor:INT
102       "suffix" suffix:STRING
103       "annotation" annotation:STRING
104       "date" date:STRING
105       "user" user:STRING
106     "}";
107
108   TypeDefinition =
109     "typedef" name:STRING "{"
110     "basetype" Basetype
111     // only for Integers, Reals and Dates
112     ("minvalue" minvalue:STRING)?
113     ("maxvalue" maxvalue:STRING)?
114     // only for Enumerations
115     ("values" values:EnumValue ("," values:EnumValue)*)?
116     "}";
117
118   EnumValue =
119     relatednumber:INT ":" color:INT ":" (value:STRING| FormattedText);
120
121   AttributeDefinition =
122     "attrdef" name:STRING (["system"])? (["multivalued"])? "{"
123     "type" type:STRING
```

```
124     "description" description:STRING
125     ("default" (default:STRING|htmldefault:FormattedText) )?
126     ("dxl" dxl:STRING)?
127     "}";
128
129   ReqObject =
130     "reqobject" (["deleted"])? "{"
131     "url" url:STRING
132     "level" level:INT
133     "number" number:STRING
134     AttributeValue*
135     subobjects:ReqObject*
136     "}";
137
138   AttributeValue =
139     definition:STRING (value:STRING|FormattedText);
140
141   FormattedText =
142     "richtext" "{"
143     FormattedParagraph*
144     "}";
145
146   FormattedParagraph =
147     "paragraph"
148     (["bullet"])?
149     "{"
150     FormattedFragment*
151     "}";
152
153   FormattedFragment =
154     "text"
155     (["ole"])?
156     (["bold"])?
157     (["italic"])?
158     (["underline"])?
159     (["strikethrough"])?
160     (["superscript"])?
161     (["subscript"])?
162     (["newline"])?
163     (["url"])?
164     "{"
165     text:STRING
166     "}";
167
168   ViewDefinition =
169     "viewdef" name:STRING "{"
170     FilterRule?
171     SortRule?
172     Column*
```

```
173     "}";
174
175   FilterRule =
176     "filterrule" rule:STRING;
177
178   SortRule =
179     "filterrule" rule:STRING;
180
181   interface Column;
182
183   AttributeReference implements Column =
184     "attributecolumn" name:STRING definition:STRING;
185
186   MainColumn implements Column =
187     "maincolumn";
188
189   DXLScriptContainer implements Column =
190     "dxlcolumn" name:STRING dxl:STRING;
191
192   Link =
193     "link" "{"
194       // if source and target are in the current module, both
             values are null
195       ("SourceModule" sourcemodname:STRING | "TargetModule"
             targetmodname:STRING)?
196       "LinkModule" linkmodname:STRING
197       "SourceAbsoluteNumber" sourcenumber:INT
198       "TargetAbsoluteNumber" targetnumber:INT
199       AttributeValue*
200     "}";
201 }
```

Listing A.2: MontiCore-Grammatik für den Import der Datensätze aus DOORS

Anhang B

Grammatiken

B.1 DSL für das DBC-Format

MG

```
1  package de.se_rwth.dbcparser;
2
3  grammar DBC {
4
5    token NUMBER = ('-')? ('0'..'9')+ ('.' ('0'..'9')+)? ('E
       ''-'('0'..'9')+)?;
6
7    ident STRING = '"'! (ESC|~('"'))* '"'!;
8
9    enum Type = "INT" | "FLOAT" | "STRING" | "ENUM" | "HEX";
10
11   DBC =
12     "VERSION" dbcversion:STRING
13     ( NS |
14
15       EnvironmentVariable |
16       EnvironmentVariableDataLength |
17       Bus |
18       BusUnit |
19       BusObject |
20
21       EnvironmentVariableAttribute |
22       BusAttribute |
23       BusUnitAttribute |
24       BusObjectAttribute |
25       SignalAttribute |
26       SignalValueType |
27       BusAttributeDefault |
28
29       BusUnitEnvironmentVariable |
30       BusUnitTXBusObject|
31       BusUnitRXSignal |
32       BusUnitAttributeDefault |
```

```
33
34      BusObjectSender |
35
36      EnvironmentVariableComment |
37      BusComment |
38      BusUnitComment |
39      BusObjectComment |
40      SignalComment |
41
42      EnvironmentVariableValue |
43      BusValue |
44      BusUnitValue |
45      BusObjectValue |
46      SignalValue |
47      BusUnitRXSignalValue |
48      ValueTable |
49      Value
50    )*;
51
52  // Namespace?
53  NS =
54    "NS_" ":" (
55    name:"NS_DESC_" |
56    name:"CM_" | // done
57    name:"BA_DEF_" | // done
58    name:"BA_" | // done
59    name:"VAL_" | // done
60    name:"CAT_DEF_" |
61    name:"CAT_" |
62    name:"FILTER" |
63    name:"BA_DEF_DEF_" | // done
64    name:"EV_DATA_" |
65    name:"ENVVAR_DATA_" | // done
66    name:"SGTYPE_" |
67    name:"SGTYPE_VAL_" |
68    name:"BA_DEF_SGTYPE_" |
69    name:"BA_SGTYPE_" |
70    name:"SIG_TYPE_REF_" |
71    name:"VAL_TABLE_" | // done
72    name:"SIG_GROUP_" |
73    name:"SIG_VALTYPE_" |
74    name:"SIGTYPE_VALTYPE_" |
75    name:"BO_TX_BU_" | // done
76    name:"BA_DEF_REL_" | // done
77    name:"BA_REL_" | // done
78    name:"BA_DEF_DEF_REL_" | // done
79    name:"BU_SG_REL_" |
80    name:"BU_EV_REL_" |
81    name:"BU_BO_REL_" |
```

```
82    name:"SG_MUL_VAL_"
83    )*;
84
85  //-------------------------ELEMENTE-------------------
86
87  // Umgebungsvariable
88  EnvironmentVariable =
89    "EV_" evname:Name ":" x1:NUMBER "[" minvalue:NUMBER "|"
          maxvalue:NUMBER "]" unit:STRING initvalue:NUMBER access:
          NUMBER dummynodevector:Name sender:Name ("," sender:Name)
          *";";
90  // Bytelaenge extra codiert
91  EnvironmentVariableDataLength =
92    "ENVVAR_DATA_" evname:Name ":" length:NUMBER ";";
93
94  // Netzwerk
95  Bus =
96    "BS_" ":";
97
98  // Knoten (Steuergeraete)
99  BusUnit =
100   "BU_" ":" buname:Name*;
101
102 // Botschaft
103 BusObject =
104   "BO_" boid:NUMBER boname:Name ":" bytes:NUMBER sender:Name
105   Signal*;
106
107 // Signal
108 Signal =
109   "SG_" sgname:Name (multiplexer:["M"]|multiplexed:["m2"])? ":"
110   start:NUMBER "|" length:NUMBER "@" SignBO
111   "(" factor:NUMBER "," offset:NUMBER ")"
112   "[" minval:NUMBER "|" maxval:NUMBER "]"
113   unit:STRING
114   receiver:Name ("," receiver:Name)*;
115
116 enum SignBO = "0-" | "0+" | "1-" | "1+";
117
118 ast Signal =
119   method public boolean isSigned(){
120       return signBO == ASTConstantsDBC.CONSTANT0 ||signBO ==
              ASTConstantsDBC.CONSTANT2 ;
121   }
122   method public boolean isIntel(){
123       return signBO == ASTConstantsDBC.CONSTANT2 ||signBO ==
              ASTConstantsDBC.CONSTANT3 ;
124   }
125   method public boolean isMotorola(){
```

```
126         return signBO == ASTConstantsDBC.CONSTANT0 ||signBO ==
               ASTConstantsDBC.CONSTANT1 ;
127     };
128
129   //-------------------------ATTRIBUTE-------------------
130
131   // Attribut fuer Umgebungsvariable
132   EnvironmentVariableAttribute =
133     "BA_DEF_" "EV_" attname:STRING Type ((values:STRING ("," 
          values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))? ";";
134
135   // Attribut fuer Netwerk
136   BusAttribute =
137     "BA_DEF_" attname:STRING Type ((values:STRING ("," values:
          STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))? ";";
138
139   // Attribute fuer Knoten
140   BusUnitAttribute =
141     "BA_DEF_" "BU_" attname:STRING Type ((values:STRING ("," 
          values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))? ";";
142
143   // Attribute fuer Botschaften
144   BusObjectAttribute =
145     "BA_DEF_" "BO_" attname:STRING Type ((values:STRING ("," 
          values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))? ";";
146
147   // Attribute fuer Signal
148   SignalAttribute =
149     "BA_DEF_" "SG_" attname:STRING Type ((values:STRING ("," 
          values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))? ";";
150
151   SignalValueType =
152     "SIG_VALTYPE_" boid:NUMBER sgname:Name ":" value:NUMBER ";";
153
154   // Default-Wert von obigen Attributen
155   BusAttributeDefault =
156     "BA_DEF_DEF_" attname:STRING (value:STRING|value:NUMBER) ";";
157
158   //-------------------------ABGELEITETE ATTRIBUTE
          -------------------
159
160   // Umgebungsvariable fuer Knoten
161   BusUnitEnvironmentVariable =
162     "BA_DEF_REL_" "BU_EV_REL_" attname:STRING Type ((values:STRING
           ("," values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))?
          ";";
163
164   // Attribute fuer Knoten und TX-Botschaft
165   BusUnitTXBusObject =
```

```
166     "BA_DEF_REL_" "BU_BO_REL_" attname:STRING Type ((values:STRING
            ("," values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))?
           ";";
167
168   // Attribute fuer Knoten mit RX-Signal
169   BusUnitRXSignal =
170     "BA_DEF_REL_" "BU_SG_REL_" attname:STRING Type ((values:STRING
            ("," values:STRING)*)|(minvalue:NUMBER maxvalue:NUMBER))?
           ";";
171
172   // Attribute fuer Knoten mit RX-Signal
173   BusUnitAttributeDefault =
174     "BA_DEF_DEF_REL_" attname:STRING (value:STRING|value:NUMBER)
           ";";
175
176   // Sende-Knoten fÃ¼r Botschaft
177   BusObjectSender =
178     "BO_TX_BU_" boid:NUMBER ":" sender:Name ("," sender:Name)+
           ";";
179
180
181   //-------------------------KOMMENTARE-------------------
182
183   // Kommentar fuer Umgebungsvariable
184   EnvironmentVariableComment =
185     "CM_" "EV_" evname:Name content:STRING ";";
186
187   // Kommentar fuer Netzwerk
188   BusComment =
189     "CM_" content:STRING ";";
190
191   // Kommentar fuer Knoten
192   BusUnitComment =
193     "CM_" "BU_" buname:Name content:STRING ";";
194
195   // Kommentar fuer Botschaft
196   BusObjectComment =
197     "CM_" "BO_" boid:NUMBER content:STRING ";";
198
199   // Kommentar fuer Signal
200   SignalComment =
201     "CM_" "SG_" boid:NUMBER sgname:Name content:STRING ";";
202
203   //-------------------------WERTE-------------------
204   // Wert fuer Attribut von Umgebungsvariable
205   EnvironmentVariableValue =
206     "BA_" attribute:STRING "EV_" evname:Name (value:NUMBER|value:
           STRING) ";";
207
```

```
208   //Wert fuer Attribut von Netzwerk
209   BusValue =
210     "BA_" attribute:STRING (value:NUMBER|value:STRING) ";";
211
212   // Wert fuer Attribut von Knoten
213   BusUnitValue =
214     "BA_" attribute:STRING "BU_" buname:Name (value:NUMBER|value:
          STRING) ";";
215
216   // Wert fuer Attribut von Botschaft
217   BusObjectValue =
218     "BA_" attribute:STRING "BO_" boid:NUMBER (value:NUMBER|value:
          STRING) ";";
219
220   // Wert fuer Attribut von Signal
221   SignalValue =
222     "BA_" attribute:STRING "SG_" boid:NUMBER sgname:Name (value:
          NUMBER|value:STRING) ";";
223
224   // Wert fuer Attribute von Knoten mit RX-Signal
225   BusUnitRXSignalValue =
226     "BA_REL_" attribute:STRING "BU_SG_REL_" buname:Name "SG_" boid
          :NUMBER sgname:Name (value:NUMBER|value:STRING) ";";
227
228   // Wertetabelle
229   ValueTable =
230     "VAL_TABLE_" valuesname:Name (key:NUMBER value:STRING)+ ";";
231
232   // Zuordnung Wertetabelle -> Signal
233   Value =
234     "VAL_" boid:NUMBER sgname:Name (key:NUMBER value:STRING)+ ";";
235 }
```

Listing B.1: MontiCore-Grammatik für das DBC-Format

Anhang C

Lebenslauf

Name	Rendel
Vorname	Holger
Geburtstag	30.05.1982
Geburtsort	Braunschweig
Staatsangehörigkeit	deutsch

seit 2011	Mitarbeiter bei der Volkswagen AG
2009 – 2011	Wissenschaftlicher Mitarbeiter am Lehrstuhl Software Engineering der RWTH Aachen
2008	Wissenschaftlicher Mitarbeiter am Institut Software Systems Engineering der TU Braunschweig
2008	Abschluss als Diplom-Wirtschaftsinformatiker
2003 – 2008	Studium der Wirtschaftsinformatik an der TU Braunschweig
2002 – 2003	Zivildienst
2002	Abitur am Scharnhorstgymnasium in Hildesheim
1989 – 2002	Grundschule, Orientierungsstufe und Gymnasium

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P, which is described in the two books [Rum11] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06], [KRV10], [Kra10] allows the spe-

cification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems [HHK+14], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737. TU Munich, Germany, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin, 2te Edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin, 2te Edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.