CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Wirtschaftsinformatik

Agile fachmodellgetriebene Softwareentwicklung für mittelständische IT-Projekte

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Dipl.-Inform. Thomas Biskup

Gutachter:

Prof. Dr. Jorge Marx Gómez
Prof. Dr. Kurt Schneider

Tag der Disputation: 13.11.2009

Danksagung

Vor und während der Erstellung dieser Arbeit habe ich von vielen Seiten Unterstützung erfahren. Ich möchte diese Stelle nutzen, um allen zu danken, die mich durch ihr Interesse, ihr bohrendes Nachfragen und ihre vielfältigen Verdienste dazu motiviert haben, diese Arbeit berufsbegleitend fertigzustellen. Ich hatte zu Beginn selbst daran gezweifelt, dass dies möglich sein könnte – ich danke euch allen dafür, dass ihr mir geholfen habt, das Gegenteil zu beweisen. Besonderer Dank gebührt

- meinem Betreuer und Erstgutachter Prof. Dr. Jorge Marx Gómez für genau das richtige Maß an Betreuung, Motivation und Anschub in den schwierigen Phasen,
- meinem Zweitbetreuer Prof. Dr. Kurt Schneider und seinem Lehrstuhl insbesondere für seine fruchtbaren Anregungen zur Verifikation meiner Ergebnisse und für seine besonders engagierte Hilfsbereitschaft auch unter Zeitdruck,
- den (ehemaligen und jetzigen) Mitarbeitern des Abteilung Wirtschaftsinformatik 1 der Universität Oldenburg für die freundliche Aufnahme und vielfältige Unterstützung, insbesondere Dr. Nico Brehm, Dr. Liane Haak und Andreas Solsbach, die mir immer wieder mit Rat und Tat zur Seite standen,
- Prof. Dr.-Ing. Axel Hahn und Prof. Dr.-Ing. Jürgen Sauer für ihr Feedback insbesondere in den frühen Phasen der Arbeit,
- meiner Familie, insbesondere meinem Vater – der mir diesen Weg überhaupt erst ermöglicht hat – und meiner Mutter, die nicht aufgehört hat zu fragen „was die Arbeit denn macht“ sowie Melanie, die meine Doktorarbeit lange Zeit als meine Freundin und auf den letzten Metern auch als meine Frau mit viel Geduld und germanistischem Können unterstützt hat,
- meinem Informatik-Lehrer am Max-Planck-Gymnasium in Gelsenkirchen-Buer, Rainer Grabowski, der sagte „Mach' das! Klar!“, als ich ihn fragte, ob er glaube, dass ich mich für ein Informatik-Studium eigne und das Ergebnis seines Vertrauens leider nicht mehr miterleben konnte,
- Dr. Rainer Fehling als meinem ersten Projektleiter bei der PRO DV Software

AG, der Kundenorientierung und Moderation zu einem wichtigen Aspekt meiner Arbeit gemacht hat,

- Matthias Graus als einem der besten Projektleiter, die ich je auf Kundenseite getroffen habe und von dem man jeden Tag wieder etwas über Agilität lernen kann – auch wenn er das so nie nennen würde,
- allen ehemaligen und aktuellen Kollegen bei QuinScape, insbesondere (ohne Beachtung der Reihenfolge):
  - meinen Geschäftsführungskollegen Dr. Norbert Jesse und Dr. Gero Presser für viele strategische Diskussionen rund um IT-Projekte und Wissenschaft sowie für den Ansporn, titeltechnisch mit ihnen gleichziehen zu müssen, um in der Geschäftsführung optisch einheitlicher zu wirken,
  - Nils Wloka für faszinierende und kontroverse philosophische und technische Diskussionen und dafür, wie er Agilität nicht nur propagiert sondern auch lebt,
  - Holger Spielmann für seine Finger in den vielen wunden Punkten bei der Erstellung dieser Arbeit, Diskussionen über das sinnvolle Gleichgewicht zwischen Agilität und formaler Spezifikation sowie eine neue Einstellung zur Zeit,
  - Sascha Steiger für kritische Zweifel an vielen Stellen, die zu einer besseren Argumentation motivierten,
  - André Dierker und Martin Schröder für ihre unermüdliche Hilfe bei LaTeX- und TeX-Fragen – ohne euch sähe die Arbeit wesentlich hässlicher aus,
  - Philip Moston für seine englische Sprachunterstützung in letzter Sekunde,
  - allen bei QuinScape für ihr kontinuierliches Interesse am Stand meiner Arbeit, weil sie mich damit gezwungen haben, immer wieder neue Inhalte zu produzieren und so schließlich zum Ziel zu kommen,
- den Kollegen der BALVI GmbH (vornehmlich und alphabetisch geordnet Ralf Barth, Dr. Ulrich Bieker, Dr. Rainer Fehling, Dr. Klaus-Dieter Hasse, Kathrin Rützel, Christian Schneider und Dr. Detlef Stolp) für die Möglichkeit, auf Grundlage von PIKAS eine langjährige Produktstrategie aufzusetzen sowie stets anregende und interessante Diskussionen rund um dieses Thema,
- Manfred Stetz von United Planet für den Satz „Die Kunst liegt in der Beschränkung“ zu einem Zeitpunkt, als ich das Universum noch mit roher Kraft aus den Angeln heben wollte – mittlerweile verlasse ich mich lieber auf den richtigen Hebel,

- Dr. Eike Riedemann für seine Unterstützung bei der ursprünglichen Bewusstseinsfindung im Rahmen der Entscheidung für eine zunächst interne und dann externe Promotion sowie für seine Hilfe bei Fragen zur Geschichte der Qualität von Softwareprojekten (und für die Erfahrungen, die ich als studentische Hilfskraft beim Schreiben eines sehr langen Buches sammeln konnte),
- *Within Temptation*, *Bruce Springsteen* und *Maria Mena* für die musikalische Unterstützung in den schwierigsten Phasen der Arbeit sowie *Iced Earth* für *High Water Mark*, das gerade spielte, als ich zum ersten Mal mit allen Inhalten dieser Arbeit durch war und *Bushido featuring Karel Gott* für das grandiose *Für immer jung*, das einen ganzen Tag lang während der abschließenden Korrekturen spielte und last but not least
- der ADOM- und JADE-Community (http://www.adom.de) als Dank für das Verständnis, dass ADOM und JADE „für ein paar kurze Jahre“ hinter der Dissertation zurückstehen mussten – jetzt wird alles anders.

Der Weg, der mich seit dem Jahr 2000 zurück aus der Industrie über die Universität in die selbständige Unternehmertätigkeit mit berufsbegleitenden Promotionsaktivitäten geführt hat, war sehr lang – aber die Erfahrungen, die ich dabei sammeln sowie die Menschen, mit denen ich seitdem arbeiten durfte, waren jeden einzelnen Schritt wert! Ich hoffe und würde mich sehr freuen, wenn andere nun den Staffelstab aufnehmen und auf Grundlage dieser Arbeit weitere Schritte unternehmen, um Software-Entwicklung zu einer effektiveren Tätigkeit werden zu lassen.

Thomas Biskup
Dezember 2009, Witten

Zusammenfassung

Die Entwicklung moderner Softwaresysteme ist geprägt von hohem Zeitdruck, kurzen Innovationszyklen, kontinuierlich wechselnden Anforderungen und vielfältigen technischen Fragestellungen. Zwar gibt es verschiedene technische, methodische und konzeptionelle Ansätze, um diesen Herausforderungen zu begegnen, dennoch steigen IT-Projektkosten bei stagnierender Qualität: IT-Projekte verzögern sich oder scheitern gänzlich, Anforderungen werden unzureichend erfüllt und Lösungen sind bereits beim ersten Release wieder veraltet.
Dies betrifft insbesondere den Mittelstand: Ihm fehlen im Gegensatz zu Großkonzernen die Ressourcen für umfangreiche Projekte, kontinuierliche technische Evolution, Weiterbildung der Mitarbeiter und für die Einführung komplexer Lösungspakete. Parallel dazu herrscht im Mittelstand jedoch ein vergleichbarer Erfolgsdruck wie in Großkonzernen. Denn um im globalen Wettbewerb zu überleben, müssen alle Unternehmensprozesse stetig optimiert werden – unabhängig von der Größe des Unternehmens. Dazu gehört auch die kontinuierliche Anpassung der IT-Systeme – eine Herausforderung, die nur eine sinkende Zahl mittelständischer Unternehmen erfolgreich bewältigt.

Diese Arbeit stellt Conceptual Programming (CP) *als integriertes Softwareentwicklungsmodell vor, das vier erfolgreiche Methoden moderner Softwareentwicklung neu kombiniert:*

- *kundenorientierte Projektstrukturen zur Fokussierung von Anwenderbedürfnissen,*
- *domänenorientierte Modelle für die effektive Kooperation zwischen Domänenexperten und IT-Experten,*
- *modellbasierte generative Software-Entwicklung, um Projektkosten zu reduzieren und gleichzeitig die Qualität der Lösungen zu verbessern sowie*
- *agile Vorgehensmodelle, die unvollständige Anforderungen und stetigen Wandel berücksichtigen.*

So wird die Entwicklung komplexer, individueller und dennoch gut anpassbarer IT-Systeme unter Berücksichtigung mittelständischer Rahmenbedingungen möglich.

Zwei neue Methoden integrieren die einzelnen Bestandteile innerhalb dieses Vorgehensmodells:

- Conceptual Model Driven Software Development (CMDSD) *beschreibt Werkzeuge zur effizienten Realisierung komplexer und dennoch flexibler IT-Lösungen, die auch von reinen Domänenexperten effektiv benutzt werden können.*
- Concept-Driven Project Management (CDPM) *ermöglicht die Zusammenarbeit von IT-Experten mit Domänenexperten insbesondere auch in der Implementierungsphase. So wird die Qualität von Lösungen verbessert und der Erstellungsaufwand gesenkt.*

Die Arbeit illustriert exemplarisch für eine horizontale Domäne (den Mittelstand) die Kernidee, wie durch Beschränkung auf eine zusätzliche vertikale Problemdomäne (Portalentwicklung) leistungsfähigere Konzepte und Werkzeuge zur Entwicklung komplexer Lösungen entworfen werden können. Wir verifizieren diese Kernidee durch Entwicklung eines Prototyps, der mittlerweile auch in der realen Geschäftswelt eingesetzt wird.

Abstract

Modern software development is characterized by high time pressure, short innovation cycles, continuously changing requirements and a host of technical challenges. Despite there being a multitude of technical, methodical and conceptual approaches to counter these challenges, the costs of IT projects rise while the level of quality seems to remain static: IT projects continue to be late or even fail completely, requirements and expectations remain unfulfilled and solutions age even before their first release.
Small to medium sized companies (SMEs) are particularly pressured by these circumstances: In contrast to large corporations they lack the resources required to continuously evolve their solutions, train employees in the latest technologies and introduce new software systems. Nonetheless SMEs today face the same challenges as large corporations. To survive in global markets while facing global competition all company processes need to evolve continuously no matter what company size we look at. This entails a constant evolution of IT systems, too. SMEs successfully stand up to this challenge less and less.

In this thesis we introduce Conceptual Programming (CP) *as an integrated software development approach which combines four existing and successful software development methodologies in a new way:*

- *customer-oriented project structures more effectively tailored to handle customer needs,*
- *domain driven modeling focused on optimizing communication between domain experts and IT experts,*
- *model-driven and generative software development in order to lower project costs and increase solution quality and*
- *agile methodologies in order to compete with incomplete and constantly changing requirements.*

By doing so in a particular way we are going to facilitate the development of complex individual software solutions while at the same time respecting the particular

constraints of SMEs. Two new methods integrate the overall aspects of this new methodology:

- Conceptual Model Driven Software Development (CMDSD) *describes tools that are used to implement complex yet flexible IT solutions more efficiently and which nonetheless remain usable by pure domain experts.*
- Concept Driven Project Management (CDPM) *facilitates the cooperation between IT experts and domain experts even during the implementation phase of projects thereby increasing the solution quality and reducing implementation costs.*

This work explains the underlying core ideas by example: we extend a particular horizontal domain („SMEs“) by adding a second vertical domain („portals“) and show how to design more efficient concepts and tools for developing complex solutions in such a constrained domain context. We verify our core ideas by developing a prototype solution to support our new software development approach. The prototype at the time of this writing also is used in real world projects.

Inhaltsverzeichnis

Abbildungsverzeichnis xv

Grundlegende Konzepte xvi

I Einführung und Grundlagen 1

1 Einführung 2
- 1.1 Motivation . . . 2
- 1.2 Kernthese . . . 7
- 1.3 Vorgehensweise . . . 8
- 1.4 Leistung . . . 11
- 1.5 Gliederung . . . 12

2 Der deutsche Mittelstand und IT 15
- 2.1 Der deutsche Mittelstand . . . 16
  - 2.1.1 Situation der IT im deutschen Mittelstand . . . 18
  - 2.1.2 IT-Projekte im deutschen Mittelstand . . . 19
- 2.2 Potenziale zur Optimierung mittelständischer IT-Projekte . . . 20
  - 2.2.1 Kundenorientierung . . . 24
  - 2.2.2 Domänenfokussierung . . . 31
  - 2.2.3 Automatisierung . . . 36
  - 2.2.4 Agilität . . . 40
- 2.3 Zusammenfassung . . . 43

3 Technische und methodische Grundlagen 47
- 3.1 Kundenorientierte Disziplinen der Softwareentwicklung . . . 49
  - 3.1.1 Convergent Engineering (CE) . . . 49
  - 3.1.2 Intentional Programming (IP) . . . 54
  - 3.1.3 Concept Programming . . . 58
  - 3.1.4 Convergent Architecture (CA) . . . 60
  - 3.1.5 Fluent Interfaces . . . 67
  - 3.1.6 Language Workbenches . . . 68
  - 3.1.7 Weitere Ansätze . . . 69
- 3.2 Domänenfokussierte Verfahren zur Softwareentwicklung . . . 75
  - 3.2.1 Programmiersprachen der vierten Generation (4GL) . . . 76
  - 3.2.2 Computer Aided Software Engineering (CASE) . . . 77
  - 3.2.3 Language Oriented Programming (LOP) . . . 79
  - 3.2.4 Rapid Application Development (RAD) . . . 80

3.2.5 Architectured Rapid Application Development (ARAD) 82
3.2.6 Software-Produktlinien 83
3.2.7 Domain Driven Design (DDD) 87
3.2.8 Domain Specific Languages (DSL) 94
3.3 Modellgetriebene Verfahren der Softwareentwicklung 94
3.3.1 Model Driven Architecure (MDA) 95
3.3.2 Model Driven Software Development (MDSD) 103
3.3.3 Software Factories 105
3.3.4 Generative Programmierung (GP) 109
3.3.5 Domain-Specific Modeling (DSM) 112
3.3.6 Model Driven Engineering (MDE) 114
3.3.7 Business Process Modeling (BPM) 116
3.3.8 ARIS 116
3.3.9 Agile Modeling (AM) 118
3.4 Vorgehensmodelle 121
3.4.1 Familien von Vorgehensmodellen 121
3.4.1.1 Phasen-, Wasserfall- und Schleifenmodelle 122
3.4.1.2 Prototypische Vorgehensmodelle 123
3.4.1.3 Inkrementelle, evolutionäre, rekursive und iterative Modelle 125
3.4.2 Präskriptive oder plangetriebene Vorgehensmodelle 127
3.4.3 Agile Vorgehensmodelle 129
3.5 Zusammenfassung 132
3.6 Herleitung des CP-Lösungsansatzes 135

II Conceptual Programming (CP) 139

4 Conceptual Programming (CP) 140
4.1 Werte, Prinzipien und Methoden als Systematik 141
4.1.1 Werte 142
4.1.2 Prinzipien 143
4.1.3 Methoden 144
4.1.4 Der Zusammenhang zwischen Werten, Prinzipien und Methoden 144
4.2 CP als Paradigma 145
4.2.1 Werte 145
4.2.1.1 Wirtschaftlichkeit 146
4.2.1.2 Flexibilität 148
4.2.1.3 Zugänglichkeit 150
4.2.2 Prinzipien 152
4.2.2.1 Kundenorientierung 152
4.2.2.2 Domänenfokussierung 152
4.2.2.3 Automatisierung 154
4.2.2.4 Agilität 154
4.2.3 Methoden 155

4.2.3.1 Conceptual Model Driven Software Development (CMDSD) . . . 156
4.2.3.2 Concept Driven Project Management (CDPM) . . . 157
4.2.3.3 Produktgremium . . . 157
4.3 Weitere Aspekte von CP . . . 158
4.3.1 Entwurfssystematik für CMDSD und CDPM . . . 158
4.3.2 Möglichkeiten zur Validierung eines CP-basierten Entwurfs . . 159
4.3.3 Adaption von CP . . . 160
4.4 Zusammenfassung . . . 161

5 Conceptual Model Driven Software Development (CMDSD) 163
5.1 Ziele . . . 164
5.2 Das fachmodellgetriebene Manifest (Teil I) . . . 165
5.3 CMDSD im Überblick . . . 168
5.4 CMDSD als Paradigma . . . 171
5.4.1 Werte . . . 172
5.4.2 Prinzipien . . . 172
5.4.2.1 Vereinfachung . . . 174
5.4.2.2 Problemorientierung . . . 175
5.4.2.3 Uniformität . . . 177
5.4.2.4 Standardkonformität . . . 179
5.4.2.5 Kompaktheit . . . 180
5.4.2.6 Abstraktion . . . 181
5.4.2.7 Lokalität und Linearität . . . 190
5.4.2.8 Tradition und Innovation . . . 191
5.4.2.9 Inkrementelle Evolution . . . 195
5.4.2.10 Rapid Prototyping . . . 195
5.4.3 Methoden . . . 198
5.4.3.1 Entwicklungsleitung durch einen Chief Programmer . 198
5.4.3.2 Entwicklung eines Architekturstils . . . 199
5.4.3.3 MDSD-basierte Software-Produktlinien . . . 199
5.4.3.4 Unternehmenskonformes Projektmanagementmodell . 202
5.4.3.5 Archetypisierung . . . 203
5.4.3.6 Komplexitätsbewertung durch das Pareto-Prinzip . . 209
5.4.3.7 Domänenabgrenzung und Modellbildung durch Domain Driven Design . . . 211
5.4.3.8 Strukturierte Modellbildung . . . 213
5.4.3.9 Dekomposition von Komplexität . . . 224
5.4.3.10 Werkzeuginhärente Archetypen . . . 228
5.4.3.11 Intention statt Imperativ . . . 228
5.4.3.12 Kapselung von Technologien . . . 230
5.4.3.13 Erhaltung der Migrationsfähigkeit . . . 231
5.4.3.14 Differenziere deine Lösung . . . 232
5.4.3.15 Realisiere wissenskonforme Zugänglichkeit . . . 232
5.5 Weitere Aspekte von CMDSD . . . 233
5.5.1 Ausführbarkeit von Modellen . . . 233

5.5.2 Behandlung der Modelltransformationen . . . 234
5.5.3 Behandlung von Variabilität . . . 235
5.5.4 Adaption von CMDSD . . . 237
5.6 Zusammenfassung . . . 238

6 Concept Driven Project Management (CDPM) 241
6.1 Ziele . . . 243
6.2 Das fachmodellgetriebene Manifest (Teil II) . . . 244
6.3 CDPM im Überblick . . . 245
6.4 CDPM als Paradigma . . . 248
6.4.1 Werte . . . 248
6.4.2 Prinzipien . . . 248
6.4.2.1 Respekt . . . 250
6.4.2.2 Fairness . . . 252
6.4.2.3 Ehrlichkeit und Offenheit . . . 253
6.4.2.4 Uniformität . . . 255
6.4.2.5 Kompetenz statt Komplexität . . . 255
6.4.2.6 CMDSD-basiertes Vorgehen . . . 256
6.4.2.7 Konsequenz . . . 256
6.4.2.8 Direktheit . . . 257
6.4.2.9 Explizivität und Transparenz . . . 258
6.4.2.10 Bedarfsorientierung . . . 258
6.4.2.11 Min-Max . . . 261
6.4.2.12 Start small, then grow . . . 261
6.4.3 Methoden . . . 262
6.4.3.1 Philosophischer Projekt-Kickoff . . . 266
6.4.3.2 CP-Coaching . . . 267
6.4.3.3 Analyse durch Design . . . 269
6.4.3.4 Rapid Development . . . 271
6.4.3.5 Conceptual Pair Programming . . . 273
6.4.3.6 Refactoring . . . 276
6.4.3.7 Lebendes Angebot . . . 276
6.4.3.8 Agiles Glossar . . . 278
6.5 Weitere Aspekte von CDPM . . . 279
6.5.1 Teamorganisation . . . 279
6.5.2 Rollen . . . 279
6.5.2.1 Domänenexperten . . . 280
6.5.2.2 CMDSD-Experten . . . 281
6.5.2.3 Mediatoren . . . 282
6.5.2.4 Protokollführer . . . 282
6.5.3 Organisatorische Einführung von CDPM . . . 283
6.5.4 Strategisches Anforderungsmanagement . . . 286
6.5.5 Besonderheiten der Analysephasen . . . 287
6.5.6 Adaption von CDPM . . . 289
6.6 Zusammenfassung . . . 290

7 Abgrenzung zu existierenden Ansätzen 292
7.1 Einzelbetrachtung . . . 292
7.1.1 *RAD . . . 292
7.1.2 Convergent Engineering . . . 293
7.1.3 Convergent Architecture . . . 295
7.1.4 Software-Produktlinien . . . 295
7.1.5 Domain Driven Design . . . 297
7.1.6 Intentional Programming . . . 298
7.1.7 Concept Programming . . . 299
7.1.8 Language Workbenches . . . 299
7.1.9 Model-Driven Engineering . . . 302
7.1.10 Conceptual Programming nach Reiss . . . 302
7.1.11 Abgrenzung zu Conceptual Programming nach CRL . . . 305
7.2 Zusammenfassung . . . 305

III Verifikation des CP-Ansatzes 306

8 CP am Beispiel „Portalsysteme“ 310
8.1 CMDSD am Beispiel „Portalsysteme“ . . . 311
8.1.1 Schritt 1: Auswahl der Domäne . . . 311
8.1.2 Schritt 2: Identifikation des Architekturstils . . . 315
8.1.3 Schritt 3: Planung der CMDSD-Werkzeugentwicklung . . . 319
8.1.4 Schritt 4: Identifikation der Archetypen . . . 323
8.1.4.1 Formularbasierte Anwendung . . . 324
8.1.4.2 FAQ . . . 325
8.1.4.3 Dokumentenliste . . . 326
8.1.4.4 Forum . . . 326
8.1.4.5 Blog . . . 327
8.1.4.6 Aufgabenverwaltung . . . 328
8.1.4.7 Kalender . . . 328
8.1.4.8 Ressourcenverwaltung . . . 329
8.1.4.9 Wiki . . . 329
8.1.4.10 Kombinationsanwendungen . . . 330
8.1.5 Schritt 5: Identifikation der Anforderungen . . . 330
8.1.5.1 CI-Konformität . . . 330
8.1.5.2 Flexible Navigation . . . 330
8.1.5.3 Mehrstufige Workflows . . . 331
8.1.5.4 Hintergrundprozesse . . . 331
8.1.5.5 Personalisierung . . . 331
8.1.5.6 Datenintegration . . . 331
8.1.5.7 Geschäftslogik . . . 332
8.1.5.8 Zustände und Übergänge . . . 332
8.1.5.9 Tagging . . . 332
8.1.5.10 Suche . . . 332
8.1.5.11 Portlets . . . 333

8.1.5.12 Rollenbasierte Rechte . . . 333
8.1.6 Schritt 6: Domänenabgrenzung . . . 333
8.1.7 Schritt 7: Identifikation der Modelle . . . 334
8.1.7.1 Top-Down-Systematik . . . 335
8.1.7.2 Modellhierarchie . . . 335
8.1.7.3 Scopes . . . 337
8.1.8 Schritt 8: Modelldefinition und -reduktion . . . 338
8.1.8.1 Primäre Modelle . . . 339
8.1.8.2 Sekundäre Modelle . . . 361
8.1.8.3 Tertiäre Modelle . . . 370
8.2 CDPM am Beispiel „Portalsysteme“ . . . 371
8.3 Zusammenfassung . . . 372

9 Bewertung der Ergebnisse 374
9.1 Ergebnisse der CMDSD-Nutzung . . . 375
9.1.1 Kosten der Erstellung des CMDSD-Werkzeugs . . . 377
9.1.2 Nutzen der Erstellung des CMDSD-Werkzeugs . . . 379
9.1.2.1 Reduzierung von Einarbeitungs- und Weiterbildungskosten . . . 379
9.1.2.2 Reduzierung von Umsetzungsaufwänden . . . 382
9.1.2.3 Reduzierung von Migrationsaufwänden . . . 385
9.1.2.4 Steigerung der Umsetzungsqualität . . . 387
9.1.3 Bewertung des Kosten-Nutzen-Verhältnisses . . . 389
9.2 Ergebnisse der CDPM-Nutzung . . . 397
9.3 Weitere Projekte auf Basis des CP-Konzepts . . . 398
9.3.1 WODOMI . . . 399
9.3.2 BALVI iPi . . . 401
9.3.3 IKT.NRW . . . 402
9.4 Weitere Einsatzfelder . . . 403
9.4.1 Massive Multiplayer Online Role-Playing Games . . . 403
9.4.2 Semantic-Web-Framework-Ansätze . . . 404
9.4.3 Datenbasiertes Publizieren (Databased Publishing) . . . 404
9.4.4 Föderierte ERP-Systeme (FERP) . . . 405
9.5 Zusammenfassung . . . 406

10 Schlussbetrachtung 408
10.1 Abschließende Zusammenfassung . . . 408
10.2 Diskussion der Ergebnisse . . . 411
10.3 Ausblick . . . 413
10.3.1 Formale Evaluierung von CDPM . . . 413
10.3.2 Katalogunterstützung für domänenspezifische CP-Instanzen . 413
10.3.3 Ergonomische Optimierung von CMDSD . . . 414
10.3.4 Test von Fachmodellen . . . 414
10.3.5 Rapid Prototyping . . . 415
10.3.6 Domänenspezifische Auswahl von Projektmanagementmodellen 416
10.3.7 Übertragung von CP auf Projekte außerhalb des Mittelstands 416

IV Anhang **418**

Literaturverzeichnis **419**

Index **444**

Abbildungsverzeichnis

2.1 Einsatz von Individualsoftware 18
2.2 Integrierter Optimierungsansatz durch Conceptual Programming (CP) 21
2.3 Erfolgsfaktoren in IT-Projekten 21
2.4 Gründe für das Scheitern von IT-Projekten 26
2.5 Qualitätsverbesserungen in IT-Projekten 27
2.6 Qualitätsverluste durch Kommunikationsprobleme 29
2.7 Kostenüberschreitungen in IT-Projekten 30
2.8 Terminüberschreitungen in IT-Projekten 31
2.9 Wichtigkeit von und Zufriedenheit mit Kriterien bei der Auswahl von IT-Dienstleistern 36
2.10 Integrierter Optimierungsansatz durch Conceptual Programming (Zusammenfassung) 45

3.1 Essentielle Aktivitäten für Software-Produktlinien 85
3.2 Beispiel für ein domänenspezifischen Modell 113
3.3 Wasserfallmodell 122
3.4 Prototypisches Vorgehensmodell 124
3.5 Inkrementelles Vorgehensmodell 126

4.1 Zusammenhänge zwischen Werten, Prinzipien und Methoden 145
4.2 Wechselwirkungen zwischen CMDSD und CDPM 156

5.1 Prinzipien des CMDSD 174

6.1 Prinzipien des CDPM 251
6.2 Mikroebene des CDPM 264
6.3 Makroebene des Concept Driven Project Management 265
6.4 Feedback bei Anwendung der T-Technik 288

8.1 Unternehmensportale im Überblick 314
8.2 Schichtenmodell für PIKAS 316
8.3 Modellrepräsentationen in PIKAS 319
8.4 Logische Modellhierarchie in PIKAS (Auszug) 337
8.5 Datenkontexte in PIKAS 338
8.6 Grafische Repräsentation von Navigationsmodellen 341
8.7 Grafische Repräsentation von Prozessmodellen 346
8.8 Prozessmodel am Beispiel des FAQ-Archetypen 347
8.9 Beispiel für Domänentypen und Relationen 353

9.1 Aufwandsvergleich PIKAS zu klassischen Ansätzen 385

Grundlegende Konzepte

3.1 Identifikation von Konzepten 48
3.2 Enge Kommunikation 50
3.3 Erweiterbarkeit 55
3.4 Höherwertige Repräsentationsformen 56
3.5 Fokussierung zwecks Realisierbarkeit 57
3.6 Systematische Modellbildung 60
3.7 Architekturstil 61
3.8 Prinzipien- und methodenbasierte Definition 62
3.9 Ganzheitliche Betrachtungsweise 62
3.10 Geschlossene Menge sinnvoller Alternativen 62
3.11 Wiederverwendung 63
3.12 Entwicklungsstrukturen 64
3.13 Entwicklungsprozess 64
3.14 Organisationsstruktur 64
3.15 Werkzeugunterstützung 64
3.16 Technologieprojektionen 65
3.17 Spezifizität 65
3.18 Flexibilisierung von Fachlichkeit 66
3.19 Iteratives und inkrementelles Vorgehen 66
3.20 Förderung kreativer Organisationen 67
3.21 Psychologisch vorteilhafte Strukturbildung 70
3.22 Kooperative, inkrementelle Vorgehensmodelle 72
3.23 Abstraktion 73
3.24 Strukturierte Werkzeugnutzung 73
3.25 Rapid Prototyping von Nutzerschnittstellen 73
3.26 Angemessenheit des Entwicklungsprozesses 74
3.27 Metrische Bewertung 74
3.28 Nebenläufigkeit 78
3.29 Portierbarkeit durch Abstraktion 80
3.30 Chief Programmer 80
3.31 Rapid Application Development 81
3.32 Automatisierung von Infrastruktur 83
3.33 Domain Engineering und Application Engineering 85
3.34 Eng fokussierte Software-Produktlinien 87
3.35 Agilität 88
3.36 Durchgängiges Domänenmodell 89
3.37 Domänenmodell als Handwerkszeug 90
3.38 Zusammenarbeit von Fach- und IT-Experten 91
3.39 Domain Driven Design als Konstruktionsschema 93

3.40 Gleichförmige Abstraktion 99
3.41 Kognitive Zugänglichkeit 101
3.42 Modellgetriebene Vorgehensweise 105
3.43 Massentaugliche Werkzeuge 106
3.44 Visuelle Modellierung 107
3.45 Kurze Kommunikationswege 109
3.46 Feature-Modellierung zur Domänenabgrenzung 111
3.47 Generative Programmierung 112
3.48 Exklusive Domänenmodellierung 114
3.49 Makro- und Mikroprozesse der Modellbildung 115
3.50 Enger Rahmen 115
3.51 Werte, Prinzipien und Methoden 118
3.52 Agile Modeling 120
3.53 Ausführbare Prototypen 125
3.54 Inkrementelles, evolutionäres, rekursives und iteratives Vorgehen 127

Abkürzungsverzeichnis

4GL	4th Generation Languages, Programmiersprachen der vierten Generation
AC-MDSD	Architecture Centric Model Driven Software Development
AM	Agile Modeling
ARAD	Architectured Rapid Application Development
ARIS	Architektur integrierter Informationssysteme
BPM	Business Process Modeling
CA	Convergent Architecture
CASE	Computer Aided Software Engineering
CE	Convergent Engineering
CDPM	Concept Driven Project Management
CIM	Computation Independent Model
CMDSD	Conceptual Model Driven Software Development
CMMI	Capability Maturity Model Integration
CP	Conceptual Programming
CPP	Conceptual Pair Programming
CRUD	Create, Read, Update, Delete – die elementaren Datenpflegeoperationen
CWM	Common Warehouse Metamodel
DDD	Domain Driven Design
DSDM	Dynamic Systems Development Method
DSL	Domain Specific Language
DSM	Domain Specific Modeling
ERP	Enterprise Resource Planning
FDD	Feature Driven Development
GP	Generative Programmierung
GUID	Globally Unique Identifier
IP	Intentional Programming
KMU	Abkürzung für „Kleine und mittelständische Unternehmen“, vgl. SME
LOP	Language Oriented Programming
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MOF	Meta Object Facility

OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
PSP	Personal Software Process
RAD	Rapid Application Development
RUP	Rational Unified Process
SME	Abkürzung für „Small to medium sized Enterprises“, vgl. KMU
SPL	Software-Produktlinie
SW-CMM	Capability Maturity Model for Software
TSP	Team Software Process
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	Extreme Programming
YAGNI	You Ain't Gonna Need It

Teil I

Einführung und Grundlagen

Kapitel 1

Einführung

„Die Zukunft erkennt man nicht,
man schafft sie.“
— Stanislaw Brzozowski (1878–1911), Polnischer Philosoph

1.1 Motivation

Wirtschafts- und Gesellschaftssysteme sind angesichts einer zunehmenden Globalisierung einem kontinuierlichen Wandel unterworfen. Wirtschaftsunternehmen müssen sich diesem Wandel stellen, um weiter konkurrenzfähig zu bleiben und unter den sich verändernden Rahmenbedingungen operieren zu können. Unternehmen müssen daher ihre Prozesse kontinuierlich überwachen, überprüfen, anpassen und erweitern ([Tayl95], S. 1ff.). Da Unternehmensprozesse in vielen Fällen eng mit Softwarelösungen gekoppelt sind[1], bedingen Prozessänderungen neue Anforderungen auf der Softwareseite, um existierende Lösungen an neue Rahmenbedingungen anzupassen oder neue Funktionalitäten umzusetzen. Mittelständische Unternehmen sind davon ebenso betroffen wie Großkonzerne (siehe [Stee02]) und stehen hier sogar unter einem besonderen Druck (siehe [Glas06, ITPR07]).

Der kontinuierliche Wandel zwingt Unternehmen, Innovationen und Modifikationen während des laufenden Kerngeschäfts einzuführen, was natürlich hohe Kosten verursacht. Auf Softwareseite wird das Problem dadurch verschärft, dass die zugrunde liegenden Basistechnologien, -werkzeuge und -plattformen ebenfalls einer ständigen

[1] Angefangen bei der Adressverwaltung und endend bei komplexen Prozessteuerungssystemen. Mittlerweile hängen 80% der Exporte Deutschlands vom Einsatz moderner IT-Lösungen ab (siehe [BKPS04], S. 1 sowie [BMBF00]).

Evolution unterworfen sind. Umgekehrt zeigen langlaufende Untersuchungen, dass nur solche Unternehmen ihre Marktanteile behaupten oder vergrößern, die diesen kontinuierlichen Wandel als Chance begreifen ([Pete87]).

Großunternehmen begegnen dieser Problematik durch hohen Ressourceneinsatz in der Form von Geld, Fachexpertise und Personal. Dementsprechend etablieren sich hier zunehmend modellbasierte Verfahren zur Beschreibung komplexer Geschäftsprozesse (Model Driven Architecture/MDA, Business Process Modeling/BPM, usw.). Die Komplexität der resultierenden Projekte führt allerdings auch zu Einführungszeiten von durchschnittlich zwei bis fünf Jahren ([Tayl95], S. 7). Für mittelständische Unternehmen gestaltet sich die Situation schwieriger, da Ressourcen hier begrenzter sind, Reaktionen auf den Markt schneller erfolgen müssen und Software auf Managementebene nicht immer denselben Stellenwert hat wie bei Großunternehmen. Schließlich ist Software dort bei der Planung und Abwicklung hochverteilter und komplexer Prozesse absolut notwendig. Software hat im mittelständischen Umfeld häufig noch einen sehr pragmatisch orientierten Hilfsmittelcharakter. Gerade bei kleinen und mittleren Unternehmen ist häufig noch die Erinnerung an die Zeiten äußerst präsent, in denen man gänzlich ohne Computer ausgekommen ist. Zudem werden die geringeren, verfügbaren Ressourcen überwiegend dazu verwendet, das Kerngeschäft zu stärken und zu flexibilisieren.

Aufgrund dieser Rahmenbedingungen sind auch die typischen oder vermeintlichen Lösungen, die für Großkonzerne existieren, nur in geringem Maße für den Mittelstand geeignet: Lizenzen sind häufig viel zu teuer, das notwendige technische Hintergrundwissen zur Implementierung derartig komplexer IT-Lösungen ist nicht in ausreichendem Maße vorhanden und es stehen auch nicht genügend zeitliche und personelle Ressourcen zur Verfügung, um umfassende Lösungen in aufwändigen Projekten einzuführen. Der einzige Ausweg ist: Die Unternehmen müssen sich für „Lösungen von der Stange“ entscheiden, also konfigurierbare Standardprodukte, die im Wesentlichen unverändert oder nur mit geringen Anpassungen in den Produktivbetrieb übernommen werden. Besonders hart betroffen sind Domänen, für die kaum Standardsoftware verfügbar ist, z. B. im Automobilbereich ([BKPS04], S. 1). Hier sind kleine und mittlere Unternehmen ihren großen Abnehmern geradezu ausgeliefert und müssen sich deren Systemvorgaben alternativlos beugen. Aus strategischer Unternehmersicht ist diese Entscheidung aber sehr problematisch, weil Unternehmen durch die implizite Standardisierung ihre individuellen Stärken verlieren und ihr Profil verwässern. Gerade diese individuellen Stärken und Profile machen aber die Stärke des Mittelstandes aus. Somit ist mittelfristig die Überlebensfähigkeit des

gesamten Mittelstandes in einer zunehmend global operierenden Wirtschaft bedroht. Doch wie können durch eine Ökonomisierung der zugrunde liegenden Softwareprozesse mittelständische Unternehmensprozesse gestärkt und so die Existenz mittelständischer Unternehmen auch langfristig gesichert werden?

Um diese Frage zu beantworten, müssen zunächst die Hauptbereiche unserer Untersuchung näher erläutert werden. Dazu gehören:

- Technologie- und architekturzentrierte Werkzeuge und Methoden
- Generalisierte Modelle
- Codezentrierte Entwicklungsprozesse
- Präskriptive und statische Vorgehensmodelle

Aus der Betrachtung dieser Hauptprobleme leiten wir Anforderungen an eine neue, bessere Entwicklungssystematik ab, die wir in den nachfolgenden Kapiteln dieser Arbeit als Motivation für die Entwicklung unseres „Conceptional Programming"-Ansatzes nutzen.

Technologie- und architekturzentrierte Modelle und Werkzeuge

Die Mehrzahl moderner Vorgehensmodelle, Modellierungsverfahren und Entwicklungsumgebungen orientiert sich primär an den Bedürfnissen von Entwicklern und Architekten. Es gibt vielfältige Hilfsmittel, um komplexe, technische Funktionalitäten zu beschreiben; verständliche Darstellungen für Domänenexperten sind aber die Ausnahme, da sich die meisten Arbeiten auf Ebene komplexer, formaler Modelle oder auf Quelltextebene bewegen[2]. Das erschwert die Kommunikation zwischen Domänen- und IT-Experten und kann zu drastischen Qualitätsverlusten bei der Umsetzung von System-Anforderungen führen.

[2]Im Rahmen dieser Arbeit wird folgende prinzipielle Unterscheidung zwischen *Domänenexperten* und *IT-Experten* vorgenommen: Domänenexperten besitzen ausgewiesene Expertise in ihrer Fachdomäne, aber nur geringe oder keine Detailkenntnisse über IT-technische Frage- und Problemstellungen. IT-Experten sind ausgewiesene technische Experten, die aber nur geringes oder kein Domänenwissen besitzen. Unter dem Begriff der Domäne werden im Sinne dieser Arbeit also Nicht-IT-Themen gefasst. Diese vereinfachende Generalisierung stellt keine problematische Einschränkung dar, da nur selten Experten beide Themenkreise gleichermaßen im notwendigen Detail beherrschen. Dies ist leider um so zutreffender für den Mittelstand, da hier wesentlich häufiger eine Vielzahl von Aufgaben in Personalunion zusammengefasst werden, so dass eine tiefgreifende Spezialisierung üblicherweise in maximal einer Domäne umsetzbar ist.

Um die Projektkommunikation zu vereinfachen, muss ein Paradigma für die Software-Entwicklung konzipiert werden, das die Informationsübertragung zwischen Domänen- und IT-Experten auf möglichst wenige Schritte minimiert und so eine enge gemeinsame Kommunikation ermöglicht.

Generalisierte Modelle

Weit verbreitete Modellierungsverfahren und Vorgehensmodelle wie UML oder die MDA sind gegenwärtig sehr generisch aufgestellt. Sie konzentrieren sich nicht auf spezielle Domänen und verfügen somit nur über sehr allgemeine Modellelemente. Daher können detaillierte domänenspezifsche Problemstellungen – wenn überhaupt – häufig nur über komplexe Hilfskonstrukte oder Erweiterungen abgebildet werden. Das erhöht wiederum die Gesamtkomplexität der Modelle, die daher von Domänenexperten nur unzureichend auf ihre Korrektheit überprüft werden können. Somit werden Fehler erst spät entdeckt und verursachen hohe Folgekosten (siehe u. a. [Ried97], Kapitel 2.5 und [HaKn08]). Die Konsequenz: Individualentwicklungen sind insbesondere in einem Umfeld mit extrem beschränkten IT-Budgets sehr riskant und somit unattraktiv. Die bedrohliche Alternative: Standardlösungen mit geringer Individualisierung. Zudem erschwert der generische Ansatz die weitestgehende Automatisierung der anschließenden Programmgenerierung, wodurch wiederum zusätzliche Entwicklungskosten entstehen. Möglicherweise werden Modelle sogar als reine Dokumentation angesehen, was im Projektverlauf häufig zu Diskrepanzen zwischen der Dokumentation und dem Codestand führt, da Entwickler nur schwer zu motivieren sind, entkoppelte Informationen konsistent zu halten. Inkonsistente und unvollständige Dokumentation erhöht abermals die Wartungs- und Betriebskosten von Software ([Deri06], S. 4).

Um Anforderungen leichter überprüfen zu können, muss der Abstraktionsgrad von zugrunde liegenden Dokumentations- und Modellierungsmechanismen auf ein möglichst informelles und dennoch präzises Niveau angehoben werden.

Codezentrierte Entwicklungsprozesse

Entwicklungsprozesse sind überwiegend codezentriert. In der Regel werden IT-Systeme in der Realisierungsphase immer noch manuell programmiert. Das liegt zum einen an den unzureichenden Möglichkeiten zur Automatisierung dieser Vorgänge aufgrund zu universeller und generischer Modelle (siehe dazu die Erläuterungen im vorausgehenden Abschnitt). Zum anderen wird weithin angenommen, dass einzig manuelles Programmieren die notwendige Flexibilität bei der Implementierung von Systemen bietet. Letztlich führt das in der Umsetzung aber zu erhöhten Zeitaufwänden und zusätzlichen Fehlerquellen, da manuelle und repetitive Programmiertätigkeiten naturgemäß fehleranfälliger sind als Verfahren, bei denen weite Teile des Systems automatisiert erzeugt werden.

> *Um repetitive, zeitaufwändige und fehleranfällige Tätigkeiten zu reduzieren, ist eine Alternative zu code-zentrierten Vorgehensweisen zu definieren.*

Präskriptive und statische Vorgehensmodelle

Präskriptive und statische Vorgehensmodelle erschweren den Umgang mit wechselnden und neuen Anforderungen im Projektverlauf. Für kleinere Projekte, bei denen Feinspezifikationen häufig erst während der Implementierung vollendet werden, ist eine hohe Dynamik im Anforderungsmanagement charakteristisch. In späteren Phasen sind kostenintensive Probleme also vorprogrammiert. Zudem fordern präskriptive Verfahren aufwändige Dokumentation, die gerade bei kleinen Projekten nicht leistbar ist. In Konsequenz verzichten viele mittelständische Unternehmen vollständig auf ein formales Projektmanagement, was sich wiederum in vielen gescheiterten Projekten niederschlägt und letztlich dazu führt, dass Individualentwicklungen zur Betonung von Unternehmensstärken nicht mehr als Option in Frage kommen.

> *Um den Projekterfolg zu maximieren und die Infrastrukturkosten zu minimieren, ist ein Vorgehensmodell zu spezifizieren, das sich möglichst nahtlos in die zuvor formulierten Anforderungen integriert.*

Zusammenfassung

Wir haben in diesem Abschnitt verdeutlicht, warum die *Flexibilisierung von Software-Lösungen* absolut notwendig ist, damit mittelständische Unternehmen in einem von stetigem Wandel geprägten globalen Markt weiterhin erfolgreich agieren und so langfristig erfolgreich sein können. Dabei haben wir die besonderen *Probleme – ausgelöst durch die beschränkten Ressourcen des Mittelstands* – herausgearbeitet. Sie gefährden die Überlebensfähigkeit einer ganzen Klasse von Unternehmen. Nachfolgend haben wir die Hauptprobleme definiert, für die eine Lösung gefunden werden muss, um die Wettbewerbsfähigkeit des Mittelstandes durch bezahlbare und hinreichend flexible Softwarelösungen sicherzustellen: *technologie- und architekturzentrierte Vorgehensmodelle* behindern die effektive Formulierung von Anforderungen. *Generalisierte Modellierungsverfahren* erschweren die Überprüfung von Detailspezifikationen sowie die Automatisierung von Entwicklungsprozessen. *Codezentrierte Vorgehensweisen* erfordern Zeit und fördern Fehlerquellen. *Präskriptive und statische Vorgehensmodelle* beschränken den Projekterfolg in einem hochdynamischen Umfeld.

1.2 Kernthese

In dieser Dissertation wird ein domänenorientiertes Paradigma entwickelt, das die Kommunikation im Rahmen der Anforderungsanalyse und Spezifikation maßgeblich optimiert und eine weitestgehende automatisierte Generierung von Softwaresystemen direkt aus der Spezifikation heraus erlaubt. Dazu werden modellbasierte Vorgehensweisen genutzt, um aus der Systemspezifikation Anwendungen zu erzeugen. Rapid Prototyping ist ein integraler Bestandteil des Paradigmas, weil es die korrekte und kurzfristige Definition und Überprüfung von Anforderungen unterstützt und so Folgekosten aufgrund von Fehlentwicklungen vermeidet. Entwickler können sich dabei auf höherwertige Aufgaben konzentrieren und werden in der effizienten Umsetzung von Standardanforderungen unterstützt. Gleichzeitig können Domänenexperten aufgrund der Domänenfokussierung der zugrunde liegenden Modelle unmittelbar am Umsetzungsprozess mitwirken und so auch kurzfristig Feedback liefern.

Auf diese Weise lassen sich die Entwicklungskosten komplexer Lösungen auch für ein mittelständisches Projektumfeld so weit reduzieren, dass Individualentwicklungen in einem nachvollziehbaren und bezahlbaren Rahmen möglich werden. Daraus resultieren flexiblere Lösungen, so dass Investitionen durch generative Verfahren, die eine

einfache Technologie-Evolution ermöglichen, gesichert werden. Den Managementrahmen dafür definieren wir durch die Anpassung agiler Projektmanagementmethoden an generative und modellbasierte Technologien.

Die Kernleistung dieser Arbeit liegt also in der:

> *Entwicklung eines auf ausführbaren Fachmodellen basierenden Paradigmas (inklusive eines Vorgehensmodells), aufgrund dessen komplexe IT-Lösungen im mittelständischen Umfeld zuverlässig und anforderungsgerecht unter Berücksichtigung der existierenden Ressourcenbeschränkungen realisiert werden können.*

Das in den nachfolgenden Kapiteln entwickelte Paradigma bezeichnen wir als *Conceptual Programming (CP)* und unterteilen es in die Bereiche *Conceptual Model Driven Software Development (CMDSD)* und *Concept Driven Project Management (CDPM)*. CMDSD beschreibt die Entwicklung von Werkzeugen, die die Grundlage für Projektarbeiten bilden. CDPM definiert ein Vorgehensmodell für Projekte, auf dessen Basis CMDSD-basierte Softwarelösungen gemeinsam mit Kunden spezifiziert und kosten- sowie bedarfsgerecht umgesetzt werden können.

1.3 Vorgehensweise

Die vorliegende Dissertation stellt vier Paradigmenwechsel vor, die unmittelbar die in Abschnitt 1.1 angeführten Probleme betreffen und in Kombination zu einer neuen Entwicklungsmethodologie in Form des Conceptual Programming führen. Das meint den Wechsel

- von *technologie- und architekturzentrierten* zu *kundenorientierten* Werkzeugen und Methoden,
- von *allgemeinen* zu *domänenspezifischen* Modellen,
- von *manueller Codezentrierung* zu *modellbasierter Generierung bzw. Automatisierung* und
- von *präskriptiven* zu *integrierten agilen* Projektmanagementmethoden.

Um diese Paradigmenwechsel zu fördern, geben wir eine Referenzmodellierung für ein Metamodell vor, dass Werkzeug- und Methodeneinsatz im Hinblick auf diese

Ziele hin optimiert und evaluieren unsere Referenzmodellierung anschließend anhand eines beispielhaften prototypischen Projekts. Nachfolgend charakterisieren wir kurz den Kern des jeweiligen Paradigmenwechsels.

Von technologie- und architekturzentrierten zu kundenorientierten Werkzeugen und Methoden

Konsequente Kundenorientierung ist das Kernmotiv aller hier vorzustellenden Ansätze. Kundenorientierung hilft dabei, sich weniger auf technische denn auf für Kunden unmittelbar verständliche, nachvollziehbare und für deren Tagesgeschäft wichtige Aspekte eines Problems zu fokussieren. Beispielsweise ist es für Domänenexperten relativ unwichtig, durch welche Technologien Anforderungen tatsächlich realisiert werden, solange auch alle impliziten, nicht-funktionalen Forderungen erfüllt sind. Letzteres erfordert aber keine Berücksichtigung in allen Domänenmodellen, sondern kann z. B. durch globale Metamodelle gesteuert werden[3]. So wird die erforderliche ökonomische Grundhaltung gelegt für bezahlbare, in ihrer Komplexität beherrschbare und dennoch äußerst flexible IT-Lösungen für den Mittelstand.

Zudem dient Kundenorientierung bei der Definition eines neuen Vorgehensmodells als Leitlinie, mit deren Hilfe eine konsequente nicht-technische Sicht der Problemdomäne gewährleistet werden kann.

Von allgemeinen zu domänenspezifischen Modellen

Eine hochgradige Domänenspezialisierung ist erforderlich, weil sie die Bandbreite denkbarer technischer Umsetzungen auf eine überschaubare Zahl domänentypischer Szenarien reduziert. Technische Freiheiten werden im Anschluss aufgrund von Best Practices zu dem jeweiligen Szenario mit sinnvoll erscheinenden Standardeinstellungen vorbelegt. Auf diese Weise ist während der Detailmodellierung domänenspezifischer Zusammenhänge kein Wechsel zwischen technischem und fachlichem Kontext erforderlich. Technische Details werden also so weit verborgen, dass Domänenexperten auf einer technologisch abstrakten und hohen fachlichen Ebene über die konkreten Anforderungen und Bedürfnisse mit den IT-Experten diskutieren können. Auf

[3] Im Extremfall könnte sich daher ein Modell in Form einer simplen Checkliste verbergen, dass nicht-funktionale Charakteristika wie Antwortzeitverhalten, erwartete Zahl paralleler Anwender, etc. abfragt und basierend darauf die notwendige technische Infrastruktur über geeignete Generierungsvorschriften ableitet. Insbesondere ist es aber nicht notwendig, explizite Metainformationen an allen Domänenmodellen zu erzwingen – solche Informationen können in vielen Fällen automatisch abgeleitet werden, wie wir in späteren Kapiteln zeigen werden.

diese Weise lassen sich flexibel parametrisierbare und erweiterbare domänenspezifische Standardkomponenten mit hoher Wiederverwendbarkeit definieren. Durch den konsequenten Einsatz dieser Komponenten lassen sich Entwicklungszeiten abermals erheblich reduzieren, da technische Entscheidungen statt in jedem Einzelfall nur einmal auf globaler Ebene getroffen werden. Außerdem schafft ein durchgängig verwendetes domänenspezifisches Vokabular zusätzliche Nähe und eine solide Vertrauensbasis zwischen Kunden und Dienstleistern. Dies wiederum stärkt die Flexibilität im Umgang mit sich verändernden Anforderungen.

Studien belegen, dass Unternehmen insbesondere dann durch IT-Systeme nachhaltig unterstützt werden, wenn die Einführung dieser Systeme durch eine enge Kopplung von Software Engineering und Business Engineering begleitet wird (siehe [Glec93]). Das gerade setzt aber eine Domänenfokussierung voraus, um eine gemeinsame sprachliche Grundlage für Domänen- und IT-Experten zu finden. Zudem gibt es umfangreiche Erfahrungen zu domänenspezifischer Modellierung und domänengetriebenen Design, die den Wert dieser Fokussierung unterstreichen (siehe das Standardwerk [Evan04a] zum Thema *Domain-Driven Design*).

Von manueller Codezentrierung zu modellbasierter Generierung bzw. Automatisierung

Domänenspezifische Modelle mit einem hinreichend hohen Abstraktionsgrad werden genutzt, um Lösungen möglichst umfassend zu generieren. Im Idealfall sind die Modelle direkt ausführbar. In jedem Fall verbessert diese Vorgehensweise den Entwicklungsprozess hinsichtlich Zeit, Stabilität und Zukunftssicherheit. Aufgrund der modellbasierten (automatischen) Generierung werden repetitve Tätigkeiten im Vergleich zu manuellen, codezentrierten Verfahren vollständig ausgeblendet und vielfältige klassische Fehlerquellen beseitigt. Selbst im Vergleich zu modernen Wizard-gestützten Systemen können die Entwicklungs-, Wartungs- und Betriebskosten von IT-Lösungen stärker optimiert werden, da fehleranfällige und zeitraubende Einstellschritte entfallen. Zudem gewährleistet die Verwendung von Standardmodellen die Zukunftssicherheit der konzipierten Lösungen, da bei richtiger Konstruktion der Modelle lediglich die Generierungsvorschriften für die ausführbaren Systeme angepasst werden müssen, um neuen technologischen Trends und Erfordernissen gerecht zu werden.

Von präskriptiven zu integrierten agilen Projektmanagementmethoden

Im Gegensatz zu stark präskriptiven Vorgehensmodellen reduzieren agile Projektmanagementmethoden den Verwaltungsoverhead auf die essentiellen Tätigkeiten[4]. Agile Methoden erhöhen insbesondere für kleine und mittlere Projekte sowohl die Qualität der resultierenden Lösungen als auch die Erfolgswahrscheinlichkeiten maßgeblich. Darüber hinaus ermöglichen agile Methoden das effiziente Arbeiten in einem hochgradig dynamischen Umfeld wie es für den Mittelstand typisch ist: Denn hier findet die eigentliche Feinspezifikation eines Systems sehr oft im Rahmen der Implementierung statt. Daher sind neue Anforderungen die Regel und nicht die Ausnahme.

1.4 Leistung

Die Leistung dieser Arbeit setzt sich aus den folgenden Hauptbestandteilen zusammen:

- *Identifikation der Notwendigkeit eines Paradigmenwechsels* im Software Engineering, um die strategische Überlebensfähigkeit von mittelständischen IT-Unternehmen durch geeignete IT-Lösungen zu sichern,
- *Ermittlung der besonderen Rahmenbedingungen mittelständischer IT-Projekte* als Entscheidungsgrundlage bei der Entwicklung neuer Verfahren und Vorgehensmodelle,
- *Adaption von Softwareproduktlinien* auf die Rahmenbedingungen kleiner und mittlerer IT-Projekte,
- *Verschmelzung codezentrierter (agiler) und dokumentenzentrierter (modellgetriebener) Verfahren* zur Entwicklung von Software,
- *Definition eines integrierten Vorgehensmodells* zur Entwicklung von komplexen, individuellen IT-Lösungen für mittelständische Unternehmen,
- *Integration agiler Projektmanagementmethoden mit generativen, fachmodellgetriebenen Ansätzen*,

[4] Zu beachten ist, dass agile Verfahren mindestens ebenso viel Disziplin erfordern wie präskriptive Verfahren(siehe [BoTu03a], Beobachtung 3 bzw. in deutlich ausführlicherer Form die Erläuterungen in [BoTu03b]). Agil ist also nicht gleichzusetzen mit *weniger Arbeit im Entwicklungsprozess* sondern mit *disziplinierterer Arbeit an den wesentlichen Artefakten des Entwicklungsprozesses.*

- *Definition eines Wertesystems mit daraus resultierenden Prinzipien und Methoden* sowohl für die Erstellung geeigneter Werkzeuge als auch für deren Projekt-Nutzung,
- *Entwicklung des Konzepts der Archetypisierung von Domänen*, das als Fundament für die konzeptionelle Modellbildung der erforderlichen Fachmodelle sowie für die Effizienzsteigerung im eigentlichen Entwicklungsprozesses dient und die
- *Definition eines Reduktionsverfahrens* zur fachmodellspezifischen Überarbeitung von standardisierten Modellen.

1.5 Gliederung

Die Arbeit gliedert sich in drei Hauptteile mit jeweils eigenen Unterkapiteln.

Teil I motiviert Problemstellung und Ausgangspunkte für einen neuen Lösungsansatz.

Kapitel 1: Einführung gibt einen Überblick über die Arbeit, beschreibt das zu lösende Problem, den Lösungsansatz und die Gliederung der weiteren Teile und Kapitel.

Kapitel 2: Der deutsche Mittelstand und IT führt den deutschen Mittelstand mit seinen besonderen Rahmenbedingungen für IT-Projekte ein. Dabei werden die Probleme identifiziert, die verhindern, dass Vorgehensmodelle und Technologien aus dem Umfeld von Großunternehmen erfolgreich für die besonderen Rahmenbedingungen des Mittelstands adaptiert werden können. Aus diesen Problemen werden Optimierungspotenziale abgeleitet.

Kapitel 3: Technische und methodische Grundlagen gibt einen Überblick über den Stand der Forschung im Software Engineering zu den identifizierten Optimierungsschwerpunkten aus Kapitel 2. Zudem werden Praktiken aus realen Projekten beschrieben. Aus diesen Beschreibungen leiten wir Ansätze ab, die mittelständische Unternehmen in die Lage versetzen, Individualprojekte unter Berücksichtigung aller Rahmenbedingungen erfolgreich umzusetzen und so die zuvor beschriebenen Optimierungspotenziale zu realisieren.

Teil II: Conceptual Programming (CP) präsentiert den Hauptteil dieser Arbeit und leitet einen neuen Lösungsansatz zur Bewältigung der zuvor geschilderten mittelständischen Probleme her, der anschließend gegen existierende Ansätze abgegrenzt wird.

Kapitel 4: Conceptual Programming (CP) diskutiert die grundlegende Kernleistung der Arbeit und führt dabei das Konzept fachmodellgetriebener Programmierung ein. Insbesondere wird die Umsetzung der identifizierten Optimierungspotenziale sowie das Zusammenspiel der zwei konzeptionellen Hauptmethoden dargestellt.

Kapitel 5: Conceptual Model Driven Software Development (CMDSD) beschreibt das Konzept fachmodellgetriebener Softwareentwicklung als einen Teil der Lösung für das hier zu untersuchende Problem. Das Kapitel erläutert detailliert, wie sogenannte CMDSD-Werkzeuge als Software-Produktlinie konstruiert werden, damit sie dem Mittelstand als Grundlage für effiziente IT-Individualentwicklungen dienen können.

Kapitel 6: Concept Driven Project Management (CDPM) definiert ein Metamodell für fachmodellgetriebenes Projektmanagement. Das Metamodell nutzt das Potenzial der zuvor beschriebenen CMDSD-Werkzeuge und steuert alle Projektteilnehmer dahingehend, dass sie Individualentwicklungen fachmodellgetrieben umsetzen. Dazu vereint das Vorgehensmodell generative modellgetriebene Ansätze mit agilen Entwicklungsprozessen.

Kapitel 7: Abgrenzung zu existierenden Ansätzen vergleicht die den vorausgegangenen Kapiteln erzielten Ergebnisse mit dem Stand der Technik und arbeitet die neuen Errungenschaften von CP, CMDSD und CDPM heraus.

Teil III: Verifikation des CP-Ansatzes zeigt anhand eines prototypischen Beispiels, wie der CP-Ansatz in der Praxis für eine exemplarische Domäne erfolgreich eingesetzt werden kann. Zudem beschreibt er die konkreten experimentellen Ergebnisse, die sich bei der Evaluierung des vorgestellten Ansatzes ergeben haben.

Kapitel 8: CP am Beispiel Portalsysteme verifiziert die Thesen aus Kapitel 5 anhand eines praktischen Beispiels. Das allgemeine Konzept wird auf das Themengebiet „Portalsysteme“ übertragen. Zusätzlich stellt das Kapitel den exemplarischen

Entwurf eines entsprechenden CMDSD-Systems namens PIKAS (für *Portal Implementation Kit – Application Studio*) sowie die Adaption des CDPM-Modells auf die gewählte Fachdomäne vor.

Kapitel 9: Bewertung der Ergebnisse beschreibt die Erfahrungswerte, die wir bei der Implementierung und Anwendung des im vorausgehenden Kapitels beschriebenen prototypischen Systems gesammelt haben. Dabei entwickeln wir eine Formelsammlung zur Potenzial- und Machbarkeitsanalyse für die CP-Planungsphase. Zudem werden verschiedene Fachdomänen und weitere auf CP basierende, aktuell laufende Projekte vorgestellt.

Kapitel 10: Schlussbetrachtung beendet die eigentliche Arbeit mit einer Zusammenfassung und Bewertung der erzielten Ergebnisse sowie einem Ausblick auf vertiefende Forschungsthemen.

Die Anhänge enthalten die Literaturreferenzen sowie das Stichwortverzeichnis.

Kapitel 2

Der deutsche Mittelstand und IT

„The road to hell is paved with good intentions and littered with sloppy analysis.“
— Unix Fortune Cookie, Quelle unbekannt

Zentrales Anliegen dieser Dissertation ist es, erfolgreiche Vorgehensmodelle und Methoden für die besonderen bereits in Kapitel 1 angeführten mittelständischen Rahmenbedingungen zu adaptieren. Ziel ist dabei, mittelständische Unternehmen in die Lage zu versetzen, ihre prozessunterstützenden IT-Systeme sowohl individualisiert (angepasst auf die Stärken einer spezifischen Unternehmung) als auch flexibel (kontrolliert anpassbar an die sich stetig ändernden Anforderungen, die aus der globalen Wirtschaft resultieren) zu entwerfen, zu implementieren und zu betreiben. Aktuelle Forschungsergebnisse wie z. B. [Coll01] zeigen, das erfolgreiche Unternehmen u. a. durch den Einsatz einer richtigen Mischung von Agilität und Disziplin in ihren Prozessen bestechen. Entsprechend wollen wir in diesem Abschnitt die aus unserer Sicht richtige Mischung für die Software-Entwicklung im Mittelstand basierend auf den besonderen Rahmenbedingungen des Mittelstandes und den aktuell erfolgreichsten Methoden und Ansätzen aus agilen und plan-basierten (disziplinierten) Modellen herausarbeiten. In den späteren Kapiteln sollen sie durch einen Rahmen für ein besonders effektives kombiniertes Zusammenspiel für die Praxis einsetzbar gemacht werden.

In Abschnitt 2.1 charakterisieren wir dazu den deutschen Mittelstand basierend auf existierenden Studien und Analysen. Aus dieser Charakterisierung leiten sich Ansatzpunkte für eine detailliertere Untersuchung der besonderen Rahmenbedingungen ab, die in mittelständischen Unternehmen für IT-Projekte und -Lösungen vorherrschen.

Die so gewonnenen Ansatzpunkte vertiefen wir in Abschnitt 2.2 und belegen ihre Stichhaltigkeit durch weitere Studien, Untersuchungen und Erfahrungsberichte.

Die einzelnen Potenziale werden überblickartig dargestellt und basierend auf existierenden Erfahrungswerten hinsichtlich ihrer Erfolgsaussichten charakterisiert. Dabei stellen wir bereits Möglichkeiten für die Übertragung der vorgestellten Technologien auf die zuvor beschriebenen Rahmenbedingungen vor. Die Ergebnisse liefern eine Partitionierung der zentralen Rahmenbedingungen und Beschränkungen, die bei der Durchführung mittelständischer IT-Projekte besondere Aufmerksamkeit erfordern.

So erhalten wir die Grundlage für die detaillierte Betrachtung bereits existierender Lösungsansätze im nachfolgenden Kapitel 3, aus denen im weiteren Verlauf der Arbeit ein neues Vorgehensmodell entwickelt wird.

2.1 Der deutsche Mittelstand

Das deutsche Institut für Mittelsstandsforschung (siehe [InMB06a]) definiert Mittelstand wie folgt (siehe [InMB06b]):

> *Zur wissenschaftlichen und politischen Auseinandersetzung mit dem Mittelstand (KMU) gehört in jedem Fall die Strukturierung des Unternehmensbestandes nach Größenklassen. In Ermangelung weiterer typisierender Daten beschränkt sich die Beschreibung von Unternehmensgrößenstrukturen in der Regel auf die Merkmale „Umsatz“ und „Zahl der Beschäftigten“. Gegebenenfalls sind zusätzlich qualitative Erklärungs- und Beschreibungsmerkmale einzubeziehen.*
>
> *Als Konsequenz der Einführung des Euro und der Umstellung der amtlichen Statistik auf Euro-Werte hat sich das IfM Bonn entschlossen, seine quantitative Mittelstandsdefinition anzupassen. Bei der Festlegung der Umsatzgrößenklassen orientiert sich das IfM Bonn an der KMU-Definition der Europäischen Union.*
>
> ***Mittelstandsdefinition des IfM Bonn[1]***
>
Unternehmensgröße	***Zahl der Beschäftigten***	***Umsatz €/ Jahr***
klein	*bis 9*	*bis unter 1 Million*
mittel	*10 bis 499*	*1 bis 50 Millionen*
groß	*500 und mehr*	*50 Millionen und mehr*

[1] seit Einführung des Euro (01.01.2002)

Diese Definition orientiert sich bei den Umsatzwerten an der KMU-Definition der Europäischen Union (siehe [EuKo03]). Die EU formuliert die zentrale Bedeutung von kleinen und mittleren Unternehmen für Europa (siehe [EuCo03], S. 5) wie folgt:

> *Micro, small and medium-sized enterprises (SMEs) play a central role in the European economy. They are a major source of entrepreneurial skills, innovation and employment. In the enlarged European Union of 25 countries, some 23 million SMEs provide around 75 million jobs and represent 99% of all enterprises.*

Kleine und mittelständische Unternehmen stellen entsprechend einen Großteil der wirtschaftlichen Leistungskraft Deutschlands (siehe [KlPi04], S. 15). In [GüKa04] finden sich die folgenden beeindruckenden Zahlen zur Wirtschaftskraft des deutschen Mittelstands für das Jahr 2003 (S. 15ff.):

- Es gibt in Deutschland circa 3,38 Millionen kleine und mittelständische Unternehmen.
- In diesen Unternehmen arbeiten circa 19,98 Millionen Menschen. KMUs beschäftigen damit 70,2% aller Arbeitnehmer, die in Privatunternehmen arbeiten. Zudem werden 81,9% aller Auszubildenden in KMUs ausgebildet.
- Diese Unternehmen machen 99,7% aller zur Umsatzsteuer angemeldeten Unternehmen aus.
- Sie erwirtschaften 41,2% der Umsatzsteuereinnahmen des Staates.

Die Zahlen für das Jahr 2002 aus der gleichen Studie ([GüKa04], S. 16) zeigen aber auch, welche engen Grenzen diesen Unternehmen in ihrer Investitionskraft gesetzt sind:

- Auf die Gesamtmenge der KMU gesehen, erwirtschaftet ein Unternehmen im Durchschnitt gerade mal einen Jahresumsatz von ca. 599.600 €.
- Nimmt man nach obiger Definition die Kleinunternehmen (also die mit weniger als einer Million Euro Jahresumsatz) aus dieser Rechnung heraus, steigt der durchschnittliche Jahresumsatz auf 4.424.970 Euro.

Diese Zahlen lassen erahnen, wie viel Budget nach Abzug aller Kosten sowie der notwendigen Investitionen in das Kerngeschäft verbleiben, um IT-Projekte durchzuführen, auch wenn uns zu diesem Zeitpunkt leider keine präziseren Aussagen zu diesem Thema vorliegen.

2.1.1 Situation der IT im deutschen Mittelstand

IT-Projekte im Mittelstand sind ein schwieriges Geschäft: Mittelständler geben laut aktueller Studien circa 5–6% ihres Gesamtbudgets für IT-Projekte aus (siehe [Leie06]). In Kombination mit den oben dargestellten Zahlen und den Aussagen von Anbietern z. B. in [Leie06] zeigt sich die Notwendigkeit einfach skalierbarer, ebenso leistungsfähiger wie anpassbarer, komponentenbasierter Systeme, um effizient Geschäfte mit dem Mittelstand zu machen. Geschäfte, die sowohl dem Mittelständler im Vergleich zu Standardlösungen einen Mehrwehrt als auch dem Dienstleister eine sinnvolle Marge erbringen. Denn nur unter diesen Voraussetzungen werden entsprechende Verfahren und Lösungen auch angeboten werden. Laut einer Studie

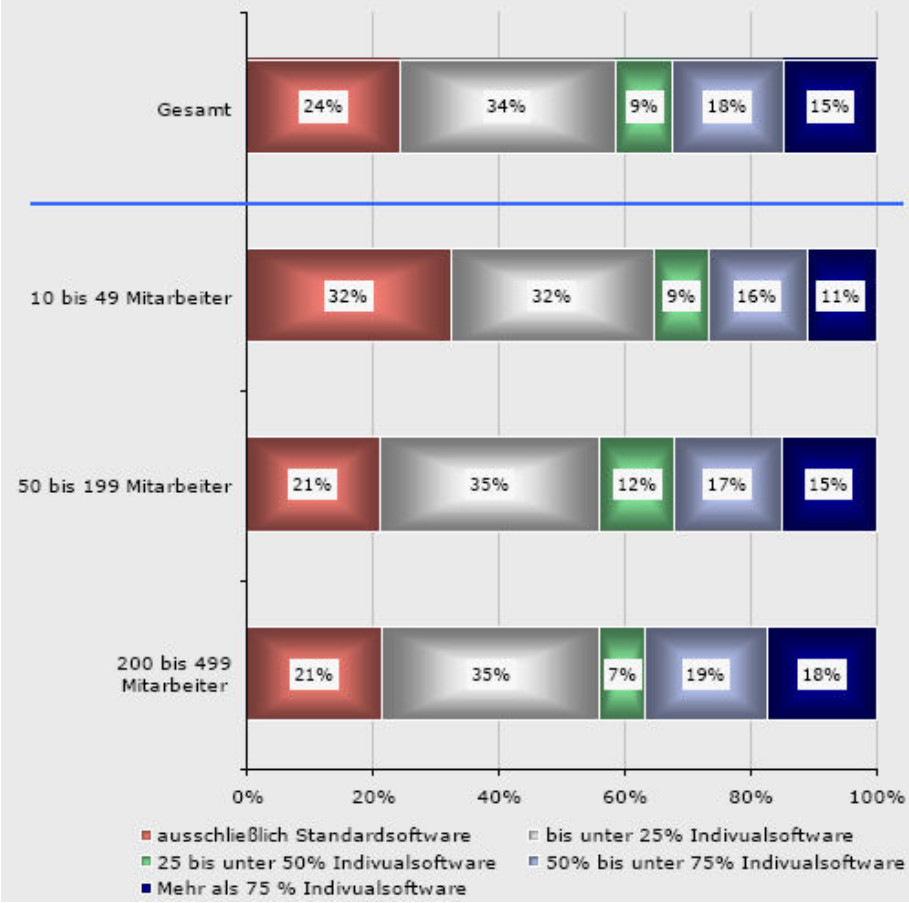


Abbildung 2.1: Einsatz von Individualsoftware (aus [Tech05])

der Firma TechConsult setzt der Mittelstand bereits in erheblichem Maß Individualsoftware ein (siehe dazu [Tech05], S. 11 bzw. Abbildung 2.1). Wir betrachten hier nur die Gesamtsicht: 24% der Unternehmen setzen ausschließlich auf Standardsoftware, 33% zu mehr als 50% auf Individualsoftware. Ein eindeutiger Beleg für die

bereits mehrfach formulierte Notwendigkeit, Kosten bei Individualentwicklungen zu reduzieren und die Qualität der Lösungen dabei signifikant zu steigern.

Besonderes Problem kleinerer und mittelständischer Unternehmen ist der enge finanzielle Rahmen, der Grenzen hinsichtlich der Einführung komplexer Lösungen setzt. So sind zum Beispiel nur wenige KMU-Unternehmen in der Lage, angemessene ERP-Lösungen einzuführen (siehe [LüMS05], S. 19). In diesem Zusammenhang spricht man z. B. auch vom *ERP-KMU-Dilemma* (siehe auch [Rich06], S. 1).

2.1.2 IT-Projekte im deutschen Mittelstand

Wie bereits in Abschnitt 2.1.1 festgestellt, unterliegen IT-Projekte im Mittelstand rigiden Budgetbegrenzungen und sind somit gegenwärtig nur dann für Dienstleister interessant, wenn in signifikantem Maß Standardlösungen eingesetzt bzw. in modifizierter Form eingeführt werden können.

IT-Projekte stellen für den Mittelstand immer wieder einen Kraftakt dar, insbesondere auch aufgrund personeller Engpässe (siehe [Gron06], S. 9). Die Abhilfe ist eine Konzentration auf die Unterschiede zwischen wettbewerbsrelevanten und nichtwettbewerbsrelevanten Prozessen. Letztere können durch Standardsoftware abgebildet werden – für Erstere ist es von entscheidender Bedeutung, dass unterstützende Software-Produkte optimal an die Stärken des jeweiligen Unternehmens angepasst werden. Auch hier können Standardlösungen wichtige Beiträge liefern, wenn Sie zwei wichtige Voraussetzungen erfüllen (siehe wiederum [Gron06], S. 9):

1. Die Standardlösungen müssen an individuelle Unternehmenskontexte anpassbar sein. So können die wettbewerbsrelevanten Stärken des jeweiligen Unternehmens betont werden.
2. Die zur Anpassungen erforderlichen Maßnahmen müssen in einem überschaubaren Kostenrahmen realisierbar sein.

Für Mittelständler sind also insbesondere die Lösungen interessant, die in hohem Maße die wettbewerbsrelevanten Vorteile eines Unternehmens in den Vordergrund stellen. Gerade hier sind aber Anpassungen und umfangreiche Konfigurationen erforderlich. Somit ist der Grundkonflikt bereits offensichtlich: es existiert ein Spannungsverhältnis zwischen strikten Budgetrestriktionen einerseits und der Notwendigkeit zur Entwicklung individueller wettbewerbsrelevanter (und damit potenziell aufwändiger) Lösungen.

2.2 Potenziale zur Optimierung mittelständischer IT-Projekte

Wie wir bereits in der Einleitung dieser Arbeit erläutert haben, sehen wir herausragende Potenziale zur Optimierung von Kosten und Leistung bei mittelständischen IT-Projekten in der Übertragung erfolgreicher Methoden und Ansätze aus IT-Projekten bei Großunternehmen. Die größte Herausforderung dabei ist, die Komplexität dieser Verfahren und Methoden soweit zu reduzieren, dass Projekte bei ihrer Anwendung in einem Kostenrahmen zu realisieren sind, der auch zum mittelständischen Umfeld passt. Die Reduktion der Komplexität sowie die daraus resultierenden Kostenersparnisse werden wir durch vier miteinander zu kombinierende Paradigmenwechsel erreichen (siehe auch Abschnitt 1.3). Im Einzelnen ist dies der Wechsel

- von *technologie- und architekturzentrierten* zu *kundenorientierten* Werkzeugen und Methoden,
- von *allgemeinen* zu *domänenspezifischen* Modellen,
- von *manueller Codezentrierung* zu *modellbasierter Generierung bzw. Automatisierung* und
- von *präskriptiven* zu *integrierten agilen* Projektmanagementmethoden.

Im Folgenden werden wir durch Studien und Erfahrungsberichte belegen, welches Potenzial diesen einzelnen Paradigmenwechseln innewohnt. Zunächst wollen wir aber die Lösungsansätze betrachten, die wir verfolgen und in Form des Conceptual Programming (CP, siehe Teil II) zu einem neuen und leistungsfähigen Vorgehensmodell verschmelzen werden. Bei unserer Definition von Conceptual Programming werden wir uns auf die effiziente Integration der Einzelaspekte fokussieren wie in Abbildung 2.2 angedeutet. Die einzelnen Lösungsansätze bezeichnen wir im weiteren Verlauf mit den Oberbegriffen *Kundenorientierung* (Abschnitt 2.2.1, S. 24ff.), *Domänenfokussierung* (Abschnitt 2.2.2, S. 31ff.), *Automatisierung* (Abschnitt 2.2.3, S. 36) und *Agilität* (Abschnitt 2.2.4, S. 40).

Betrachtet man die in Tabelle 2.3 dargestellten Erfolgsfaktoren für IT-Projekte[2] und dort die Faktoren, die von zentraler Bedeutung für die technische und methodische Steigerung der Effizienz von IT-Projekten sind, fällt sofort auf, dass technische Fragestellungen eine sehr niedrige Relevanz für den Projekterfolg besitzen (Rang 9

[2] Siehe insbesondere [Hart06] für die diesbezüglichen Ergebnisse zu über 50.000 untersuchten IT-Projekten seit 1994.

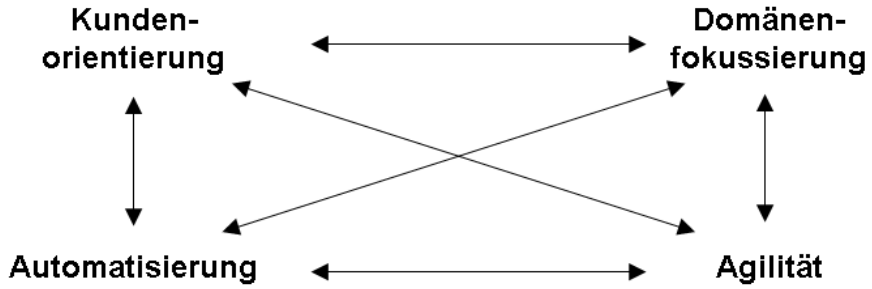


Abbildung 2.2: Integrierter Optimierungsansatz durch Conceptual Programming (CP)

und 10, teilweise Rang 5), während kommunikative und organisatorische Themen im Vordergrund stehen (Rang 1–5). Dies bestätigt unsere nachfolgend detaillierter erläuterte Forderung nach einer absoluten *Kundenorientierung* bei der Gestaltung von IT-Werkzeugen und Prozessen (siehe Abschnitt 2.2.1, S. 24ff.).

Rang	***Erfolgsfaktor***
1	User Involvement
2	Executive Management Support
3	Clear Business Objectives
4	Optimizing Scope
5	Agile Process
6	Project Manager Expertise
7	Financial Management Resources
8	Skilled Resources
9	Formal Methodology
10	Standard Tools and Infrastructure

Abbildung 2.3: Erfolgsfaktoren in IT-Projekten (nach [Hart06])

Die Studie belegt zudem, dass die zukünftige Entwicklungen zur Verbesserung der Qualität von Software sowie der Erfolgsaussichten von IT-Projekten die fachliche und soziologische Seite der Probleme und Lösungen fokussieren müssen. Technologische Fragen haben in den Hintergrund zu treten, da sie als „im Prinzip gelöst“ wahrgenommen werden. Fachliche Aspekte und der möglichst unmittelbare Dialog mit den Fachexperten, aus deren Reihen die konkreten Anforderungen kommen, werden in den Vordergrund treten müssen, um weitere nachhaltige Erfolge in der Steigerung von Softwarequalität und der Kostenoptimierung von Software-Projekten erzielen zu können. In [CoHa05] wird ab Seite VII ff. eine Sicht der historischen Veränderungen im Software-Entwicklungsprozess präsentiert, die eine grundlegende Modifikation

der existierenden Vorgehensmodelle motiviert: Es wird argumentiert, dass sich die sozialen Komponenten der Software-Entwicklungsmethodiken mittlerweile in ihrer vierten Ausprägung in der Praxis wiederfinden:

- Der Stil der ersten Generation war geprägt durch die „IT-Urzeit“, in der Systeme noch „maschinell“ programmiert wurden (durch Hebel und Schalter).
- Der Stil der zweiten Generation ist geprägt durch Programmiersprachen, die es Wissenschaftlern erlaubt haben, alleine oder in kleinen Teams zu arbeiten.
- Der Stil der dritten Generation (geerbt aus der Hardware-Entwicklung) ist geprägt durch formale Prozesse und Methoden mit einem deutlich sichtbaren expliziten Management und einer stark hierarchischen Strukturierung.
- Der Stil der vierten Generation ist geprägt durch den Abbau und das Zusammenfalten von Hierarchien sowie den Aufbau dynamischer sozialer und kommunikativer Strukturen. Dieser Stil wird gegenwärtig als *agil* bezeichnet.

Dies untermauert unsere nachfolgend dargestellte Forderung nach erhöhter *Agilität* in mittelständischen IT-Projekten (siehe Abschnitt 2.2.4, S. 40ff.).

Zudem ist die Wechselwirkung zwischen Produktionsprozessen, Werkzeugen und Produkten zu berücksichtigen: Werkzeuge werden an sich verändernde bzw. veränderte Prozesse angepasst (oder erlauben umgekehrt die Ausprägung neuer effektiverer Prozesse). Die Qualität von Produkten wird durch den Einsatz optimaler Produktionsprozesse und Werkzeuge dominiert. Das heißt: Erforderliche Qualitätssteigerungen müssen sich in Änderungen von Prozessen oder Werkzeugen widerspiegeln. Moderne agile Prozesse setzen erstmals den Kunden und die Interaktion mit dem Kunden in den Mittelpunkt des Software-Erstellungsprozesses. Der Prozess und das Produkt müssen sich flexibel gegenüber den Anforderungen des Kunden verhalten, ebenso wie dem Kunden eine flexible Einstellung gegenüber dem technisch Sinnvollen und Machbaren abgefordert wird. Diese Kombination hat sich in den letzten Jahren als äußerst effektiv erwiesen – für Projekte kleinerer bis mittlerer Größe existieren dazu mittlerweile auch Studien (siehe unsere Ausführungen in Abschnitt 3.4.3).

Auch in der IT muss sich daher eine solche Umstellung nachhaltig auf die verwendeten Werkzeuge und Prozesse auswirken, durch die Kunden unmittelbar eine fühlbare Qualitätssteigerung erleben und erfahren. Nach Ansicht des Autors nach wird der kommende fünfte Stil von Software-Entwicklung erstmalig nachhaltig aus den Erfolgen des vierten prozessorientierten Stils und nicht mehr durch die zugrunde liegenden Technologien geprägt sein. Hard- und Software werden allerdings als erfolgskritische Enabler diesen Stil der fünften Generation zum Erfolg führen müssen,

da es gilt, hinreichend ausgefeilte und kundenorientierte Werkzeuge und Prozesse zu definieren, die wiederum das eigentliche Endprodukt hinsichtlich Kosten und Qualität optimieren. Um die Vorteile eng aufeinander abgestimmter Werkzeuge und Prozesse konsequent nutzen zu können, werden wir eine hohe *Domänenfokussierung* fordern (genauer ist dies beschrieben in Abschnitt 2.2.2 ab Seite 31). Zudem werden wir unser Konzept des *Conceptual Programming* aus Kapitel 4 unterteilen in die Teilkonzepte des *Conceptual Model Driven Software Development (CMDSD)* in Kapitel 5 und des *Concept Driven Project Management (CDPM)* in Kapitel 6.

Erste verhaltene Ansätze dazu sind inzwischen auch bei den großen Softwareproduzenten zu erkennen, die zunehmend erkennen, dass die wachsende Komplexität der Anwendungswelt ein tiefergehendes Prozessverständnis erfordert. So sagt zum Beispiel Björn Interthal, zuständig bei SAP Deutschland für die Beraterausbildung: „Die einzelnen Module verlieren an Bedeutung, künftig steht der Lösungsgedanke im Vordergrund“ und „Das ist nicht nur ein Update, sondern ein Paradigmenwechsel“ (siehe [Seeg05]). Mag es sich dabei auch um von Marketing geprägte Aussagen handeln, ist ihr Kern dennoch zutreffend und in weiten Teilen der IT-Industrie beobachtbar: Das Verständnis für die Fachprozesse rückt in den Vordergrund, Details der technischen Implementierung werden zunehmend unwichtiger.

Auch andere aktuell wieder auflebende Trends untermauern diese These: Die aktuellste und stärkste Entwicklung im Software Engineering – das modellbasierte Generieren von Software[3] – hat ihre Wurzeln bereits in der Frühzeit der Informatik: Schon Anfang der 70er Jahre wurde u. a. von Parnas die Trennung von Spezifikation und Implementierung als grundlegendes Abstraktionsprinzip gefordert (siehe [Parn72]). Genau diese Trennung bildet aber die Grundlage der heute propagierten, modellbasierten Generierung von Software: Diese entschärft die Trennung und verringert die Kluft zwischen Spezifikation und Implementierung, indem sie (Teile der) Spezifikationen zu Grundlagenartefakten der Implementierung werden lässt – entweder durch Code-Erzeugung und anschließende manuelle Ergänzung aus Modellen heraus, durch direkte Umsetzung in ausführbaren Code oder gar durch die unmittelbare (Ausführung durch) Interpretation der Modellspezifikationen. Auf der Reduktion der Trennung von Spezifikation und Implementierung fußt auch unsere Forderung von modellbasierter *Automatisierung* (siehe dazu Abschnitt 2.2.3 ab Seite 36).

Die vorangegangenen Ausführungen machen deutlich, dass die Übertragung der bei Großunternehmen bereits etablierten und bewährten Verfahren zur Entwicklung und

[3] Siehe z. B. [Fisc05a], Zitat: „*Microsofts Vision geht jedoch weiter: Fachspezifische Entwürfe und Prozesse sollen in automatischen Produktionslinien zu Code verarbeitet werden.*“. Details finden sich in [GSCK04].

Zuschneidung hochkomplexer Systeme auch auf die Rahmenbedingungen typischer IT-Projekte im Mittelstand absolute Priorität hat. Nur so kann sichergestellt werden, das kleine und mittlere Unternehmen von den Möglichkeiten leistungsfähiger IT-Systeme in einem vergleichbaren Maße wie Großunternehmen profitieren können. Klar ist, das dabei Kompromisse erforderlich sein werden. Genauso klar ist aber, dass die Abwendung von standardisierten und starren IT-Lösungen „out of the box“ hin zu anpassbaren und zuschneidbaren Baukastensystemen eine einzigartige Chance offenbart: Kleine und mittlere Unternehmen können mit für sie angepassten Systemen in einem zunehmend globalen Markt konkurrenzfähig bleiben und zudem Wachstumspotenziale erschließen, die Ihnen ansonsten verwehrt blieben[4]. Die ökonomische Notwendigkeit dieses Weges steht außer Frage, schätzt doch z. B. eine Studie der Info-Tech Research Group (siehe [Micr06]) den Schaden durch unzureichendes IT-Management in der EU auf circa 120 Milliarden Euro (in den USA sogar auf 130 Milliarden Euro).

Nachfolgend vertiefen wir die bisher hergeleiteten Forderungen nach *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität*.

2.2.1 Kundenorientierung

Wir fordern als ersten zentralen Bestandteil für das noch zu definierende Modell des *Conceptual Programming* eine absolute Kundenorientierung sowohl in IT-Prozessen und -Methoden als auch bei den verwendeten Werkzeugen und Technologien. Kundenorientierung wird bei allen weiteren Betrachtungen das zentrale Leitmotiv darstellen. In diesem Abschnitt werden wir darstellen, warum unserer Ansicht nach eine absolute Ausrichtung allen Denkens und Handelns auf die Kunden von IT-Dienstleistungen zwingend erforderlich ist. Die Notwendigkeit dafür belegt z. B. [Trat06], S. 30, der auf die Schwierigkeiten in der Benutzung existierender Entwicklungswerkzeuge hinweist: *„Der typische Anwender auf fachlicher Seite wird bei diesem Ansatz zu wenig geführt oder findet sich erst gar nicht in dem Werkeug zurecht“*[5]. Zudem werden wir aufzeigen, warum gegenwärtige Ansätze und Modelle diese Forderung nur in einem sehr eingeschränkten Maß unterstützen.

Die Entwicklung von Software im Allgemeinen hat bis zum heutigen Tag mit er-

[4] Aufgrund der *Economies of Scale* in Kombination mit den *Economies of Scope* werden derartige Baukastensysteme auch bezahlbar. Die Mechanismen hinter *Economies of Scale* und *Economies of Scope* werden in dieser Arbeit kurz auf Seite 39 vorgestellt.

[5] Dies erläutert er bei seinen Ausführungen zur modellbasierten Anforderungsanalyse mit gängigen UML-Werkzeugen.

staunlichen Qualitätsproblemen zu kämpfen, vergleicht man sie mit anderen Ingenieursdisziplinen (siehe [Gruh05, Gilb94]). Älteren Studien zufolge (siehe [Boeh81, Myer86, Ried97]) liegt die Zahl an Fehlern im Quelltext zwischen 30 und 100 je 1000 Anweisungen. Auch aktuelle Studien zu den Ergebnissen von IT-Projekten belegen dieses eher erschreckende Bild (siehe [Stan03a]): Immer noch fast 50% aller Softwareprojekte haben mit Budget-, Qualitäts- und Terminproblemen zu kämpfen und ca. 15% aller IT-Projekte werden erfolglos abgebrochen (siehe auch Abbildung 2.5 auf Seite 27). Die Gründe für die existierenden Qualitätsmängel sind seit langem bekannt und haben sich nicht grundsätzlich verändert (siehe z. B. [Zoll05]):

- Textbasierte Spezifikationen bieten einen (zu) großen Interpretationsspielraum.
- Schnittstellen mit anderen Systemen funktionieren nur selten auf Anhieb.
- Bis zu den ersten Tests vergeht immer noch sehr viel Zeit. Spät entdeckte Fehler verursachen entsprechend hohe Kosten bei der Behebung.
- Dokumentation und Softwarestand passen selten zusammen.
- Der Bruch zwischen Designern und Implementierern belastet Projekte unnötig.

Diese Einzelaspekte lassen sich wie folgt zusammenfassen:

- Das Anforderungsmanagement bereitet gravierende Probleme, da bei Verwendung von klassischen Methoden der Sofwaretechnologie der „Weg“ von der Erfassung der Anforderung bis hin zu ihrer Umsetzung relativ weit ist.
- Die erforderlichen Informationen wandern durch eine ganze Reihe von Händen, was aufgrund der hinlänglich bekannten Probleme aus der Informationstheorie zu den ebenso bekannten Qualitätsmängeln bei der Erstellung von Software führt[6]. Studien über die Gründe von Erfolg und Misserfolg von Projekten belegen diese Vermutung (so zum Beispiel [Bitp02, KiRS04, Stan03b]) tatsächlich: So zeigen zum Beispiel die Untersuchungen der Standish Group bei mittlerweile ca. 50.000 IT-Projekten, dass von den zehn häufigsten Gründen für das Scheitern eines Projektes vier (in Tabelle 2.4 fett dargestellt) in direktem Zusammenhang mit Schwächen im Anforderungsmanagement stehen. Werkzeuge, die das Requirements Engineering – insbesondere modellbasiert –

[6] Siehe dazu auch die nachfolgenden Ausführungen ab Seite 28 sowie Abbildung 2.6. Es wird explizit darauf hingewiesen, dass dies nicht der einzige Grund für Mängel in der Software-Qualität ist, hier aber aus Sicht des Autors einer der herausragenden Mängel liegt, da er so eng am Kunden – also am eigentlichen Domänenexperten und Lösungssuchenden – aufgehangen ist.

unterstützen, werden vornehmlich in internationalen Großunternehmen eingesetzt (siehe z. B. [Trat06], S. 29).

Das stützt unsere These, dass auch für mittelständische IT-Projekte eine bezahlbare modellbasierte Unterstützung notwendig ist. Dies wird gestärkt durch die Beobachtung, dass insbesondere in mittelständischen IT-Projekten stetig wechselnde Anforderungen an der Tagesordnung sind (siehe [Pößn05]) und somit die Konsistenz von Dokumentation und Implementation ohne modellbasierte generative Ansätze besonders schwer herzustellen bzw. zu halten sein wird – zumal die geringen IT-Budgets im Mittelstand wenig Raum für ausführliche Spezifikationsphasen lassen (siehe die Ausführungen in Abschnitt 2.1).

Rang	***Grund***	***%***
1	**Unvollständige Anforderungen**	13,1
2	**Mangelnde Nutzerbeteiligung**	12,4
3	Ressourcenknappheit	10,6
4	**Unrealistische Erwartungen**	9,9
5	Mangelnde Managementunterstützung	9,3
6	**Wechselnde Anforderungen**	8,7
7	Mangelnde Planung	8,1
8	Produkt obsolet	7,5
9	Unzureichendes IT-Management	4,3
10	Technologische Probleme	4,3

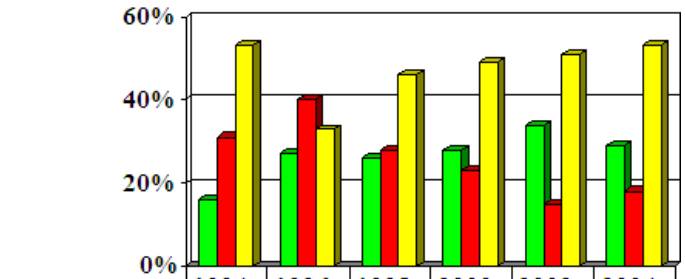
Abbildung 2.4: Gründe für das Scheitern von IT-Projekten (siehe [Stan03b])

Zudem zeigen aktuelle Befragungen, dass nicht allein Prozessverbesserungen für die zu beobachtende Steigerung der Software-Qualität verantwortlich sind: Aussagen großer Software-Hersteller belegen, dass mittlerweile nur noch ein Drittel aller Entwicklungsbudgets in die tatsächliche Weiterentwicklung von Systemen fließen und dagegen zwei Drittel der Mittel für Integration und Qualitätssicherung aufgewendet werden (müssen) - siehe z. B. [Livs05]:

> *„According to a distinguished engineer from IBM, a recent internal study revealed that only about one in three dollars that IBM spends on software product development goes toward new features. The other two-thirds are spent on non-value-added integration costs.*
>
> *IBM is no less efficient in software development than the rest of the industry. Mathematics is hard to argue with. The ratio of dollars spent on new features to dollars spent on certification and support goes down fast*

> *as the number of standard interfaces goes up. Two-thirds of non-value-added costs is already hard to swallow. What if that becomes four-fifths or nine-tenths?*
>
> *A number of senior software people know that we are in deep trouble and that we don't have a get-well plan, but it's not profitable to talk too much about it – except as a sell point for the new miracle product of the day.“*

Die bereits erzielte Qualitätssteigerung spricht dennoch für die vielen Initiativen und Bestrebungen, Softwarequalität zu verbessern (siehe [Hart06, SaCu03] sowie Abbildung 2.5). Zwar ist der Anteil erfolgreicher IT-Projekte von 1994 nach 2004 von 18% auf 31% gestiegen (siehe [Hart06]), doch der immer noch signifikante Anteil von Fehlschlägen und Problemen (53% aller Projekte bleiben in diesem Zeitraum unverändert problematisch) sowie der steigende Kostendruck ([Livs05]) implizieren, dass für eine weitere Qualitätssteigerung in Zukunft radikalere Wechsel bei der Philosophie zur Erstellung von Softwaresystemen erforderlich werden. Das Potenzial ist vorhanden – so kosten Software-Fehler laut einer Studie des National Institutes of Standards and Technology: *„Every year, according to a 2002 study by the National Institute of Standards and Technology, software failures cost $59.5 billion.“* (aus [Rose07], Seite 2). [GSCK04] zitieren sogar Verluste in Höhe von 81 Milliarden US-Dollar durch abgebrochene Projekte sowie weitere Verluste in Höhe von 59 Milliarden US-Dollar durch Budgetüberschreitungen (S. 3). Der Weg von der Anforderung zu ihrer Umsetzung muss verkürzt werden, was sich bereits aufgrund geringerer Verluste in der Kommunikationskette zwischen Anwendern und Entwicklern in Qualitätsverbesserungen niederschlagen wird (siehe die nachfolgenden Ausführungen sowie Abbildung 2.6 auf Seite 29).



	1994	1996	1998	2000	2002	2004
Succeeded	16%	27%	26%	28%	34%	29%
Failed	31%	40%	28%	23%	15%	18%
Challenged	53%	33%	46%	49%	51%	53%

Abbildung 2.5: Qualitätsverbesserungen in IT-Projekten (aus [Hart06])

In [Deri06], S. 27–28 wird die Bedeutung mangelnder Domänenkenntnisse und der damit verbundenen Menge von Kommunikationsschritten zwischen Kunde und resultierender Lösung wie in Abbildung 2.6 dargestellt betont: Verliert eine Anforderung vom Kunden bis zur letztlich erfolgten Implementierung auf ihrem Weg über den Kunden, Analysten, Designer und Implementierer pro Schritt nur 10% an Präzision[7], so wird sie mit weniger als 60% Genauigkeit abgebildet. Diese Kommunikationsfehler führen zu Fehlern in den resultierenden Systemen (siehe [Ried97], Abschnitt 2.1, S. 21ff.). Je später solche Fehler entdeckt werden, umso kostspieliger und aufwändiger wird es, sie zu beheben. Daher liegt es nahe, die Spezifikation bzw. Modellierung von Anforderungen so gut wie möglich für Domänenexperten nachprüfbar zu halten, damit diese frühzeitig auf Probleme hinweisen können. Der Lösungsansatz dazu liegt in der *Domänenfokussierung* und wird im folgenden Abschnitt 2.2.2 beschrieben. Das zugrunde liegende Prinzip lautet *Kundenorientierung*: Zuerst müssen die eigenen Kenntnisse, Werkzeuge und Prozesse so ausgerichtet werden, dass sie sich so nah wie irgend möglich an den Bedürfnissen des Kunden und damit dann erst in Folger auch natürlich an seiner Problemdomäne orientieren (siehe dazu auch die Ausführungen in [Trat06]). Hier zeigt sich auch die besondere Relevanz von Qualitätsverbesserungen für mittelständische IT-Projekte. In einem Umfeld kleiner Budgets und großer Ressourcenknappheit wirken sich Fehler erheblich drastischer aus als in Großprojekten, die variablere Spielräume zur Kompensation von Problemen bieten.

Zudem erzwingen die bereits mehrfach erwähnten geringen mittelständischen IT-Budgets geradezu den Erfolg von Projekten, da im Gegensatz zur Situation bei Großunternehmen selten zusätzliche Ressourcen vorhanden sind, um unzureichend erfolgreiche Projekte auch in Folgephasen weiterführen zu können. Der Mittelstand ist im Gegensatz zu Großunternehmen also viel stärker zum Erfolg verdammt. Faktisch werden also für den Mittelstand viele der in Abbildung 2.5 als „challenged“ bezeichneten Projekte eher als „failed“ bewertet werden (müssen), da sie letztlich aufgrund mangelnder Budgets für eine erfolgreichere Fortführung auch eingestellt werden. Damit könnte die Dunkelziffer für Fehlschläge nach neueren Zahlen (Stand 2004) bis zu 71% betragen!

Studien wie [Hart06] untermauern die Notwendigkeit von mehr Kundenorientierung und Domänenfokussierung: Technische Gründe sind nur noch in 4,3% der Fälle Schuld am Projektabbruch. Zudem zeigen Untersuchungen zu Kosten- und Zeitüberschreitungen (siehe die Abbildungen 2.7 und 2.8), dass deutliche Verbesserungen erzielt werden konnten. Dennoch sind die jeweiligen Überschreitungen immer noch

[7] ein optimistischer Wert gemessen an realen Projekterfahrungen

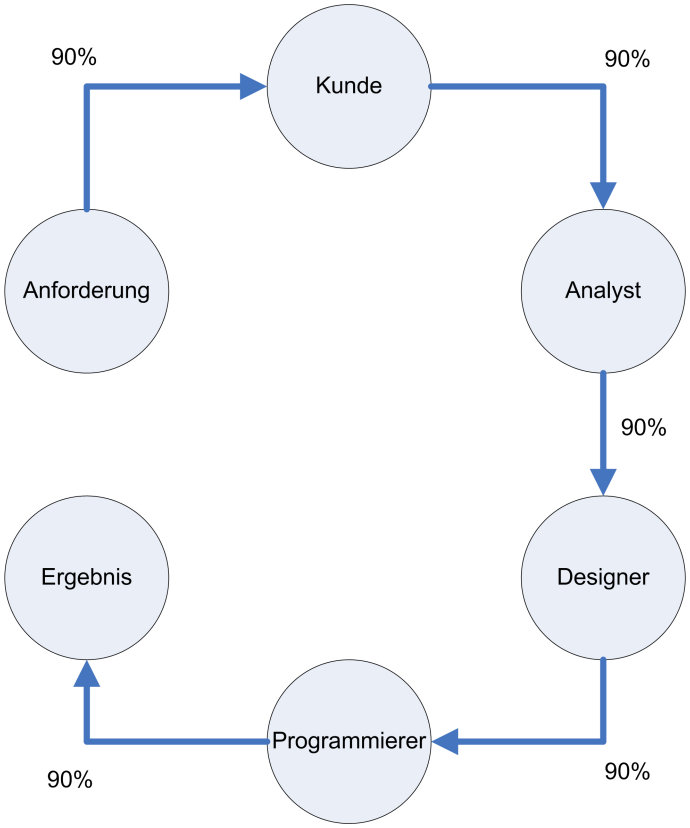


Abbildung 2.6: Qualitätsverluste durch Kommunikationsprobleme (aus [Deri06, Kris98])

signifikant und die Verbesserungen in den letzten sechs Jahren sogar gegenläufig: Allmählich ververstärken sich die Probleme von IT-Projekten wieder, was unserer Ansicht nach seine Ursache in der stetig wachsenden Komplexität der Projekte hat, zu deren Bewältigung die aktuellen Methoden, Technologien und Verfahren nur noch sehr begrenzt geeignet sind (siehe auch die auf Seite 26 zitierte Aussage aus [Livs05]).

Auch die Werte der *Agile Alliance* ([AgAl06]) unterstreichen die Bedeutung der Kundenorientierung. Die Mitglieder der Agile Alliance legen ihrem Handeln vier Prinzipien zugrunde ([AAGI06]), von denen zwei nachfolgend im direkten Zusammenhang zu den Ausführungen dieses Abschnitts stehen:

Individuals and interactions over processes and tools
Customer collaboration over contract negotiation

Wie bereits der allgemeinere Ausdruck „individuals“ impliziert, strebt agile Softwareentwicklung danach, Individuen über Prozesse und Werkzeuge zu stellen. Wir

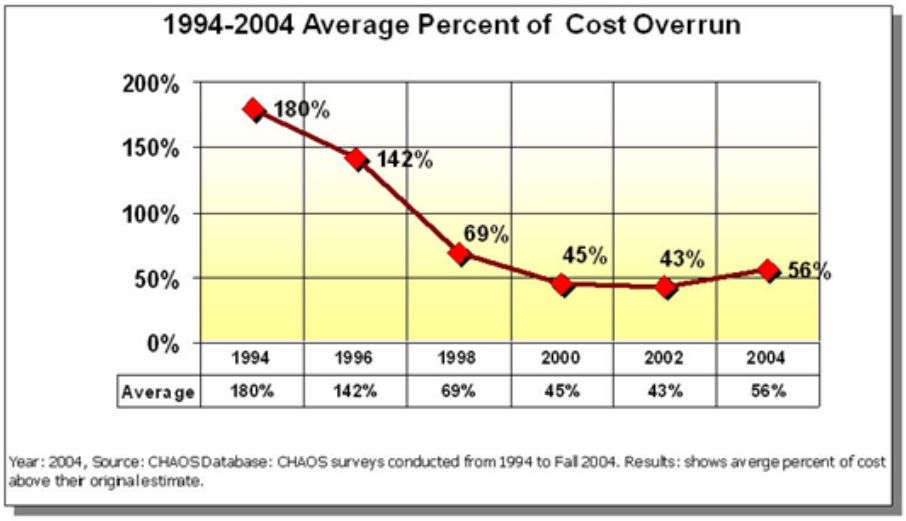


Abbildung 2.7: Kostenüberschreitungen in IT-Projekten (aus [Hart06])

unterstützen diesen Anspruch nachhaltig, sind aber wie oben ausgeführt der Meinung, dass im Rahmen dieser Arbeit und unter Berücksichtigung mittelständischer Rahmenbedingungen der Kunde das maßgebliche Individuum ist. Alle Aktivitäten und Werkzeuge müssen darauf ausgerichtet werden, dass

- Anforderungen kundengerecht dargestellt werden können,
- Entwicklungsprozesse von Kunden verstanden und beeinflusst werden können,
- Werkzeuge mit den Begrifflichkeiten einer kundengerechten Fachdomäne arbeiten.

Wie wir in den weiteren Ausführungen sehen werden, ist dies ein maßgeblicher Schlüssel, um Vorgehensweisen und Technologien aus Großprojekten derart herunterzuskalieren, dass sie auch in erheblich kleineren Projekten und unter deutlich rigideren Zeitrestriktionen erfolgreich eingesetzt werden können. Somit stellt das in Teil II definierte Modell des *Conceptual Programming* eine spezialisierte Form agiler Softwareentwicklung dar.

Wir sehen hier aber auch eine Abstraktionsmöglichkeit, die letztlich zur Vereinfachung von Prozessen und Werkzeugen führt: Wenn unsere Werkzeuge und Vorgehensmodelle optimal auf Kundenbedürfnisse ausgerichtet sind, sparen wir eine zusätzliche Optimierung hinsichtlich der klassischen Anforderungen zur Unterstützung von Entwicklern ein, z. B. weil wir weder Code-Wizards noch übermäßige Generalisierung fordern, die sich in erhöhtem Entwicklungsaufwand von Werkzeugen niederschlagen würden. Natürlich werden IT-Experten damit quasi gezwungen, sich in ihrem Den-

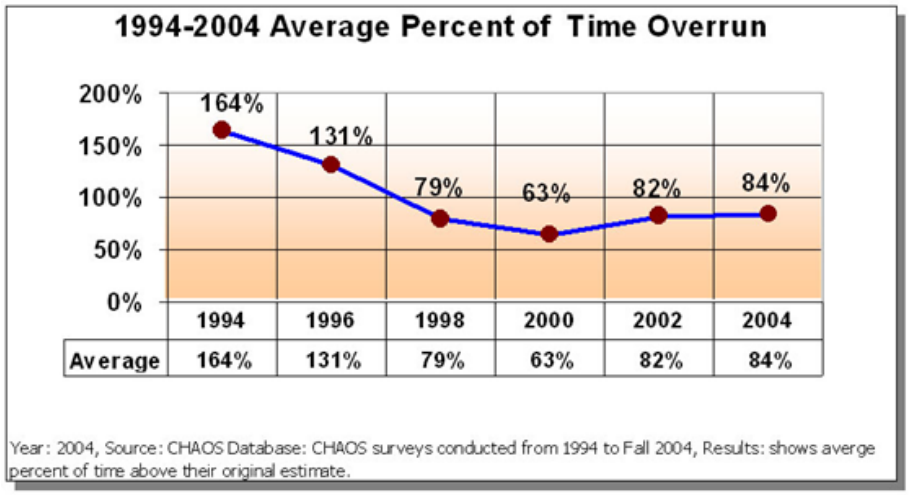


Abbildung 2.8: Terminüberschreitungen in IT-Projekten (aus [Hart06])

ken und Handeln näher an das Denken und Handeln von Domänenexperten und weg von technologischen Fragestellungen zu bewegen. Wie wir zeigen werden, führt gerade diese Notwendigkeit zu einer Verbesserung in der Qualität der resultierenden Lösungen.

Wir fassen zusammen: Kundenorientierung ist das grundlegende Prinzip, von dem ausgehend Methoden, Werkzeuge und Prozesse derart gestaltet werden müssen, dass die unmittelbaren Interessen, Kenntnisse und Bedürfnisse des Kunden den Entwurfsfokus bilden. Auf diese Weise können existierende Probleme reduziert und Optimierungspotenziale sowohl in der Anforderungsdefinition als auch der eigentlichen Umsetzung erschlossen werden. Das abstrakte Prinzip der Kundenorientierung wird in den folgenden drei Abschnitten durch die konkreten Prinzipien von Domänenfokussierung, Automatisierung und Agilität untermauert. In Kapitel 3 werden wir in Abschnitt 3.1 verschiedene besonders kundenorientierte Vorgehensmodelle der Software-Entwicklung analysieren, um aus den vorliegenden Erfahrungswerten eine Grundstruktur für den von uns zu entwickelnden mittelständischen Ansatz abzuleiten.

2.2.2 Domänenfokussierung

Industrieprozesse sind durch den Aufbau von Fertigungsstraßen effizienter geworden; die Standardisierung von Bauteilen hat dazu geführt, dass Produktionsprozesse von Arbeitern mit geringerem Ausbildungsstand effizient abgewickelt werden können –

insbesondere sind keine detaillierten technischen Kenntnisse erforderlich, sondern eine solide fachliche Ausbildung tritt in den Vordergrund. Zudem erleichtert eine entsprechende Standardisierung den Bau komplexerer Systeme. Bewährte Standardbausteine lösen Standardprobleme. Auch in der Software-Industrie gibt es hier große Optimierungspotenziale. Das bestätigt z. B. IBM in [PCHa93] auf S. 568: In einigen Projekten konnten durch die Einführung von entsprechenden Komponententechnologien Kosten in Höhe von 14–20% eingespart werden. Hewlett Packard (HP) ist es gelungen, durch den Einsatz wiederverwendbarer Software-Komponenten Entwicklungszeiten um 75% zu reduzieren (siehe [Lim98]). Nun darf man die Einsparpotenziale von Großprojekten nicht mit den Potenzialen von viel kleineren mittelständischen Projekten vergleichen – aber Parallelen lassen sich durchaus ziehen, da auch die von uns vorgeschlagene Kombination von modellgetriebener Generierung und domänenorientierter Entwicklung komplexere Projekte ermöglicht. In den zugrunde liegenden Komponenten werden ebenfalls komplexe Logiken enthalten sein. Somit ergeben sich zumindest tendenziell ähnliche Einsparpotenziale, da die Entwicklung der notwendigen Werkzeuge (siehe dazu mehr in Kapitel 5) durchaus komplexe Systeme entstehen lässt, die für eine eingeschränkte Domäne mit begrenztem Budget dennoch Ergebnisse ermöglichen, die früher Großprojekten vorbehalten waren. Fachexperten können sich mit einem solchen Ansatz mehr darauf konzentrieren, neue Probleme zu bearbeiten als immer wiederkehrende Aufgabenstellungen zu beschreiben und analysieren. Und trotzdem bleiben alle Möglichkeiten zur Optimierung erhalten: Detaillierte fachliche Modelle beschreiben die Funktionsweise von Bausteinen, Schnittstellen sind standardisiert und ermöglichen so, die Interna von Systemen von einer Systemgeneration zur nächsten zwecks Leistungssteigerung auszutauschen, ohne dass der Gesamtentwurf des Systems komplett überarbeitet werden müsste.

Zudem erlaubt diese Form der Standardisierung die effiziente und schnelle Massenfertigung von Varianten. So werden zum Beispiel in der Automobilindustrie auf den Fertigungsstraßen im Grunde ausschließlich Unikate gebaut, da es für jede Automarke durch die vielfältigen Wahlmöglichkeiten beim Kauf zehntausende Varianten ein- und desselben Standardmodells gibt (siehe [CzEi00], S. 2 unten).

In der Software-Industrie ist dieses Maß von Wiederverwendung aber immer noch nicht erreicht: In tausenden Betrieben werden täglich immer wieder die gleichen Probleme durch erneute Programmierung (sprich: komplette Neuerfindung) gelöst: Sei es das Programm zur Adressverwaltung, die Software zum Etikettendruck oder das Diskussionsforum für eine Abteilung – nur selten werden Standardkomponenten eingesetzt, häufig werden existierende Systeme als Vorbild für eine Neuimplementie-

rung genommen und manchmal wird komplett auf der grünen Wiese ohne Kenntnis des aktuellen Technik-Standes und der spezifischen Anforderungen gearbeitet. Und nicht nur die täglichen Probleme sind von dieser Symptomatik geprägt: Auch komplexe Systeme zeigen einen Trend zur Neuerfindung. Betrachtet man den z. B. den Content Management Markt, trifft man auf Hunderte von Produkten, die alle mehr oder minder die gleichen Eigenschaften vorweisen (alleine ca. 380 CMS-Lösungen bei [Cont08]). Schaut man sich ERP-Systeme an, gibt es insbesondere für kleinere und mittlere Firmen unzählige miteinander konkurrierende Produkte (siehe [Soft08]). Die meisten dieser Systeme setzen nicht auf Standardmodellen auf (wenn die fachlichen Zusammenhänge überhaupt explizit gemacht werden), sondern sind zumeist das Ergebnis unzähliger Einarbeitungs- und Entwicklungsstunden, die von den jeweiligen Dienstleistern und Anbietern im Verlauf der Produktgeschichte investiert wurden. Hubert berichtet in [Hube02], S. 15, von ähnlichen Beobachtungen: *„By comparison, if one looks at the IT industry today, one observes a whole lot of jet planes with helicopter propellers. This is so because designers are njot yet working at the level of architectural style. Instead, they reinvent, beginning at an extremely low level each time. It does not help if they can buy expensive components from the marketplace – the propellers and turbines of the IT world – if these are not applied properly.“* Und nicht zuletzt findet man daher auch immer wieder die gleichen gravierenden Fehler in diesen hochkomplexen Systemen – was nicht wirklich verwundert, denn es liegen eben keine geprüften Standardkomponenten zugrunde.

Komponentenorientierte Systeme sind bislang gescheitert, weil es nicht gelungen ist, hinreichend allgemeine Schnittstellen zu definieren, die eine tatsächliche Austauschbarkeit von Komponenten erlauben – stattdessen wird mit vergleichsweise primitiven Werkzeugen an hochkomplexen Problemen gearbeitet. Greenfield und Short ziehen eine Analogie zwischen der Art und Weise, wie in der Software-Industrie hochkomplexe und -abstrakte Konzepte durch Dinge wie *String*s und Schleifen beschrieben werden und der Art und Weise, wie die Hardware-Industrie bei einem vergleichbaren Herangehen arbeiten würde: Diese würde komplexe und moderne Prozessoren durch manuelles Zusammenlöten von Transistoren entwickeln anstatt sie auf Grundlage von ASICS zu beschreiben und dann automatisiert generieren zu lassen (siehe S. 4 von [GSCK04]). Komponententechnologien in der IT scheinen also noch zu sehr in technischen Details verhangen zu sein statt sich auf die höherwertigen Probleme zu konzentrieren.

Daher erhöhen wir den Abstraktionsgrad der Systeme durch eine gezielte Fokussierung der Problemdomäne und gehen sogar einen Schritt weiter: Wir werden nicht

fordern, Schnittstellen und Modelle zu definieren, die Probleme möglichst allgemeingültig lösen und dies insbesondere auch noch als offizieller Standard tun, sondern werden unsere Domänenfokussierung lediglich auf ein Produkt bzw. eine eingeschränkte Domäne konzentrieren. Schon Jackson hat festgestellt, dass die Ausdruckskraft mit zunehmender Spezifizität von Abstraktionen steigt (siehe [Jack99] bzw. [GSCK04], S. 59–60). Dort verwendete Modelle müssen auf Standardmodelle abbildbar sein, sie können aber durchaus implizite Spezifikationen beinhalten, ohne diese in expliziten Modellformalismen auszudrücken. Das erlaubt es uns, mit realistischem Aufwand und auch aus Sicht eines kleinen Dienstleisters standardisierte Lösungen für eine Problemdomäne anzubieten, ohne stetig die Komplexität eines globalen Modells berücksichtigen zu müssen. Erfahrungen aus der Praxis belegen, das bislang Produkte am Erfolgreichsten waren, wenn sie ein entsprechendes Vorgehen gewählt haben – die so definierten produktspezifischen Standards konnten sich mitunter zu globalen Standards entwickeln, weil die Produkte so erfolgreich waren, das andere Hersteller die entsprechenden Standards aufgegriffen und breitflächig unterstützt haben. Ein Beispiel dafür sind die Erweiterungsmöglichkeiten von Adobe Photoshop im Bereich grafischer Bildverarbeitung, die basierend auf einer Standardprogrammierschnittstelle der Firma Adobe eine Vielzahl von Drittherstellern auf den Plan gerufen haben, teilweise sehr erfolgreiche und domänenspezifischere Erweiterungen für das Basissystem zu entwickeln (siehe z. B. die entsprechenden Hinweise in [Heim02]). Der Preis der Machbarkeit ist in diesem Fall also eine Einschränkung der Allgemeingültigkeit der Lösung sowohl durch eine starke Domänenfokussierung der Modelle als auch durch die Einschränkung der Standardisierung an den Produktgrenzen (also den Schnittstellen zu anderen Produkten).

Ein schönes Beispiel für die Auswirkungen eines domänenfokussierten Ansatzes gegenüber einem generischen und universellen Ansatz zeigt ein Vergleich der Beispiele aus [StVö05] und [Bigg02]: Ersteres Werk postuliert die Erzeugung domänenspezifischer Sprachen und kann mit kurzen und knappen Beispielen für Transformationen zeigen, wie effizient die Umsetzung entsprechender Systeme vonstatten gehen kann. Letzteres Werk verfolgt den Entwurf eines universellen Generators für die Übersetzung von DSLs in lauffähige Systeme und betreibt (in diesem, vorausgegangenen und weiteren Werken unter [Soft08]) einen sehr hohen Aufwand, um die aufgrund des universellen Ansatzes entstehenden komplexen Fragestellungen in den Griff zu bekommen. Dies ist nicht per se ein Problem, zeigt aber schon, welche drastisch höheren Ressourcenaufwände (intellektuell wie zeitlich) erforderlich sind, um mit zu allgemeinen Ansätzen zum Ziel zu kommen.

Ein weiterer Nutzen der Domänenfokussierung liegt in der Vereinfachung der Kommunikationswege – angefangen bei der Aufnahme einer Anforderung bis zu ihrer Umsetzung in eine Softwarelösung (siehe dazu die Ausführungen im vorhergehenden Abschnitt 2.2.1 sowie insbesondere die Erläuterungen zu Abbildung 2.6 auf Seite 29): Durch eine ausgewiesene Domänenexpertise können IT-Experten ohne zwischengeschaltete Analysten, Designer und Entwickler (deren Rolle sie im Extremfall in einer Person vereinigen) Kundenanforderungen aufnehmen und umsetzen. Natürlich besteht eine andere Gefahr von Qualitätsverlust, da mehrere verantwortungsvolle und komplexe Aufgaben potenziell in einer Person vereinigt sind (Analyse, Design und Umsetzung). Wir werden aber in den folgenden Teilen dieser Arbeit zeigen, dass die Fokussierung auf eine Problemdomäne die Entwicklung von Modellierungswerkzeugen erlaubt, die gegenüber generischeren Ansätzen wie z. B. der Model Driven Architecture (MDA, siehe [OMG06a]) und der Unified Modeling Language (UML, siehe [OMG06b]) während der Entwicklung tiefgehendere Modellprüfungen ermöglichen. Auf diese Weise können viele Fehlerquellen bereits durch die zugrunde liegenden Werkzeuge erkannt werden.

Auch Kunden sind sich der besonderen Bedeutung von Domänenexpertise bewusst und wünschen sich eine entsprechend kompetente und intensive Betreuung, wie zum Beispiel eine Studie der TechConsult GmbH zeigt ([Tech05], S. 8). In dieser Studie werden die in Abbildung 2.9 auf Seite 36 dargestellten Aussagen zu den wichtigsten heutigen Kriterien bei der Auswahl von IT-Dienstleistern mit der Erfüllung dieser Kriterien durch Dienstleister in Verbindung gebracht. Von besonderer Bedeutung sind an dieser Stelle die letzten vier Punkte: *Ansprechpartner vor Ort*, *Tiefe des Portfolios*, *einschlägige Referenzen* und *Branchenexpertise*. Alle vier Punkte (der erste eingeschränkt) betreffen die Domänenexpertise der Anbieter, die Vollständigkeit des Angebotssegments sowie das Vorhandensein entsprechender Referenzen. Genau diese Punkte stechen in der Studie dadurch hervor, dass sie die Einzigen sind, in denen die Zufriedenheit der Kunden hinter der Realität zurückbleibt. Genau diese vier Punkte genießen höchste Priorität. Interessant ist auch, dass Kunden hinsichtlich der technischen Unterstützung insgesamt sehr zufrieden sind und hier keine zusätzlichen Ansprüche stellen (Forderungen nach technischen Kenntnissen sortieren sich erst im Mittelfeld der Proritätenliste ein).

Wir fassen zusammen: Domänenfokussierung erlaubt eine Abstraktion technischer Details und vereinfacht zudem die Kommunikationswege mit dem Kunden von der Anforderungserhebung bis zu ihrer Umsetzung. Durch geringere Präzisionsverluste in der Kommunikation wird die Qualität der resultierenden Lösung verbessert. Zu-

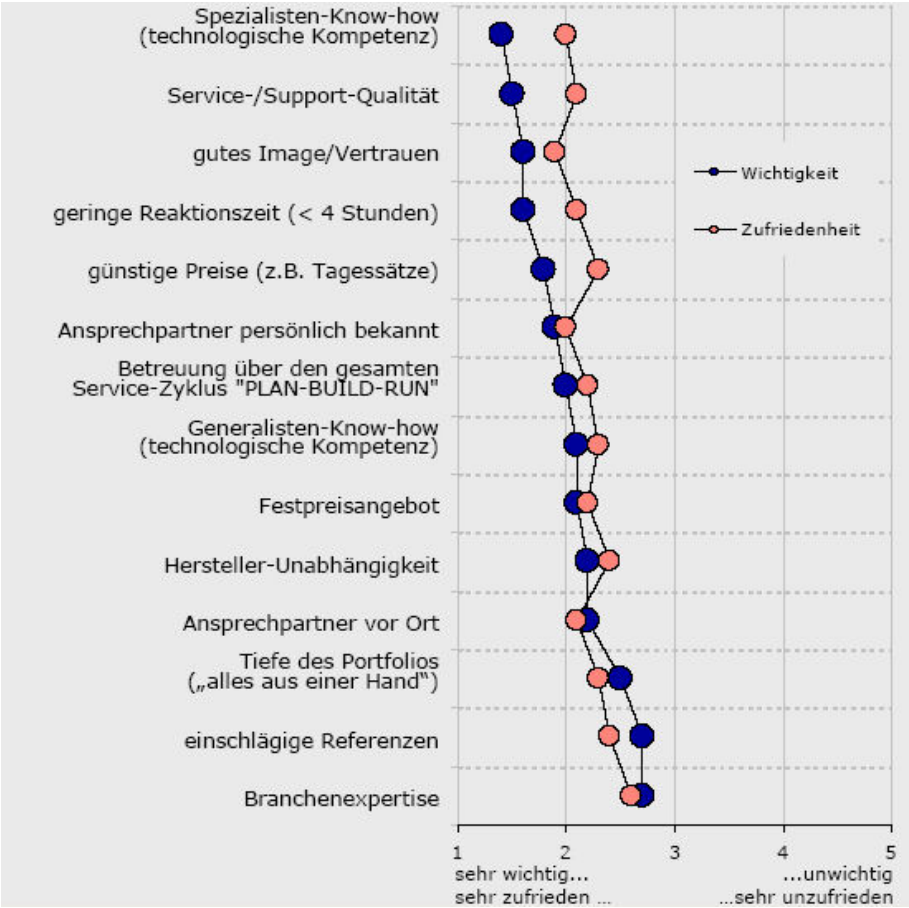


Abbildung 2.9: Wichtigkeit von und Zufriedenheit mit Kriterien bei der Auswahl von IT-Dienstleistern (nach [Tech05], S. 8)

dem ist eine direktere Kundenansprache möglich, die auch zu einem tiefergehenden Vertrauensverhältnis führt, das jeder Projektkultur zuträglich ist. Entsprechend werden wir in Kapitel 3 (insbesondere in Abschnitt 3.2) nach existierenden Methoden und Verfahren suchen, die uns bei einem domänenfokussierten Vorgehen unterstützen.

2.2.3 Automatisierung

Automatisierung ist in nahezu allen Branchen ein maßgeblicher Erfolgsfaktor zur Kostenreduktion und Qualitätssteigerung. Notwendige Voraussetzung für Automatisierung ist die Wiederverwendbarkeit von Komponenten, aus denen Systeme konstruiert werden. Die Wiederverwendbarkeit von Software ist einer der heiligen Grale der IT-Industrie, im Prinzip seitdem die erste Programmiersprache einen Unterroutinenzugriff erlaubt hat. Zunehmend systematisch wurde versucht Algorithmen,

Funktionbausteine und Teilsysteme so zu kapseln, dass eine einfache Wiederverwendung möglich ist. Die dazu benutzten Konzepte sind zunehmend leistungsfähiger geworden und erlauben immer komfortabler die Einbindung von Funktionsbausteinen Dritter, um mit steigender Geschwindigkeit hochkomplexe und leistungsfähige Systeme entwickeln zu können:

- Maschinen- und Assemblersprachen (Programmiersprachen der ersten und zweiten Generation) erlaubten den Aufruf von Subroutinen. Parameter wurden manuell zum Beispiel auf einem Stack übergeben und das Aufräumen des Systems war explizite Aufgabe des Entwicklers. Dennoch konnten so bereits komplexe Teilroutinen wiederverwendet werden.

- Programmiersprachen der dritten Generation (sogenannte Hochsprachen) formalisierten diesen Mechanismus zunächst durch die Möglichkeit der expliziten Definition von Prozeduren (Unterroutinen ohne Rückgabewerten) oder Funktionen (Unterroutinen mit Rückgabewerten).

- Weiterentwicklungen erlaubten Prozeduren und Funktionen in Modulen oder Units zu gruppieren (zum Beispiel in den Sprachen Pascal und Modula), um Systeme stärker zu strukturieren.

- Konsequente Fortentwicklung dieser Konzepte sind objektorientierte Sprachen, die Zustand (Daten) und Verhalten (Funktionen) in Form einer Klasse kapseln, um so einen stärkeren Zusammenhang innerhalb von Funktionsbausteinen und System-Entitäten zu formulieren.

- Komponentenbasierte Entwicklung gruppiert ganze Ansammlungen von Modulen oder Objekten, um so eine noch stärkere Zusammengehörigkeit von komplexen Teilsystemen zwecks hoher Wiederverwendbarkeit auszudrücken (siehe [HeCo01, Lau04]).

- Aspektorientierte Programmierung erlaubt die Ergänzung von Querschnitts- und Infrastrukturfunktionalitäten (z. B. Sicherheit, Logging und Transaktionen) bei konsequenter Separation der entsprechenden Implementierungen vom Code der Fachfunktionen (siehe [FCEl04]).

- Composite Oriented Programming (COP) erlaubt die flexible Komposition kleiner Fragmente zu komplexen Systemen und bietet eine Mischung aus objekt- und aspektorientierten Vorteilen (insbesondere bei Querschnittsfunktionalitäten und klassischen Anwendungsszenarien für Mehrfachvererbung) (für Details zu dieser sehr neuen Entwicklungsphilosophie siehe [Qi4j08]).

Doch hier stoppt die Entwicklung nicht: Neueste Forschungsbestrebungen gehen dazu über, in Form (halb)formaler und häufig auch grafischer Modelle technische und fachliche Zusammenhänge zu spezifizieren und dann aus diesen Modellen zumindest in Teilen automatisiert Code oder lauffähige Systeme zu generieren (sogenannte modellbasierte Verfahren, siehe z. B. [Eysh06]). Aktuelle Ansätze sind hier die sogenannte *Model Driven Architecture (MDA)* ([OMG06a]) mit der Modellierungssprache UML bzw. der zugrunde liegenden Meta Object Facility (MOF) der Object Management Group (OMG) basiert ([OMG06b, OMG06c, Wiki06f]). Grundidee ist, über eine allgemeine Modellierungssprache konkrete Systeme zu beschreiben und mittels definierter Transformationsregeln aus diesen Modellen lauffähigen Code für verschiedene Zielplattformen zu generieren (siehe Abschnitt 3.3.1 für Details). In der Praxis erweist sich dieser Ansatz als komplex und schwerfällig, da eine allgemeine Modellierungssprache nur unzureichend alle möglichen Szenarien der realen Welt abdecken kann und so immer Lücken in der Spezifikation bleiben, die anderweitig beschrieben werden müssen (siehe dazu unsere nachfolgenden ausführlicheren Anmerkungen zu Agile Modeling nach Scott Ambler in Abschnitt 3.3.9). Daher hat sich als Alternativbewegung die Philosophie des *Model Driven Software Development (MDSD)* (siehe Abschnitt 3.3.2) bzw. des *Domain-Specific Modeling (DSM)* (siehe Abschnitt 3.3.5) herausgebildet. Diese Philosophie setzt nicht auf eine einzelne Modellierungssprache auf, sondern versucht eine geeignete Modellierungsform je Problemraum zu finden, um so Modelle auf möglichst direkte und triviale Weise in ausführbare Systeme zu transformieren (siehe auch Abschnitt 3.3.2).

Interessant ist eine Betrachtung der zur Realisierung von Effizienzgewinnen bislang erfolgreichsten Evolutionsstufen: Der Wechsel von Assembler- auf Hochsprachen brachte einen Effizienzgewinn von ca. 400% (siehe [Iseg05]). Die nächsten evolutionären Entwicklungen bis hin zu modellgetriebenen Ansätzen haben die Leistungsfähigkeit von Entwicklungsteams nur noch moderat gesteigert, selbst für Modellierungsansätze wie die MDA weisen Studien auf „nur“ 35% Leistungssteigerung hin (siehe [RiVa03]). Das größte Potenzial scheinen neueste Ansätze in Form von modellgetriebener Generierung auf Basis domänenspezifischer Sprachen zu enthalten – hier weisen Ergebnisse auf bis zu weitere 500–1000% Effizienzsteigerung hin (siehe wiederum [Iseg05] in den Beispielen zu Nokia, Lucent und USAF).

Desweiteren sind modellgetriebene Systeme eine Investition in die Zukunft der generierten Architekturen und Systeme, da technische Evolution „unter der Haube“ (also am zugrunde liegenden Generator) stattfinden kann und Modelle aus technologischer Sicht statisch sind (siehe die Erfahrungswerte zur Transformation von

abstrakten Modellen und Sprachen aus [Ward94]). Technologien sind dagegen einem rasanten Wandel unterworfen. In immer kürzeren Zyklen werden neue Technologien, Protokolle und Datenformate eingeführt. Da heutige Softwareumgebungen relativ direkt mit den jeweiligen Technologien arbeiten und sich nur wenig Mühe geben, die technologischen Details zu verbergen, sind insbesondere Entwickler stetig damit beschäftigt, sich neues Detailwissen anzueignen, das schnell wieder veraltet. Neue Entwicklungswerkzeuge sollten sich daher darauf konzentrieren, möglichst stark von technologischen Details zu abstrahieren, damit Entwickler sich auf Architekturen, Prozesse und fachliche Algorithmen konzentrieren können (siehe dazu insbesondere auch [GSCK04], S. 60–61).

Zu beachten ist hierbei das besondere Verhältnis zwischen Verbundeffekten (englisch: economies of scope, siehe [Wiki06h, Wiki06i]) und Skaleneffekten (englisch: economies of scale, siehe [Wiki06g, Wiki06j]). Verbundeffekte werden erzielt, wenn sich trotz zunehmender Produktvielfalt Kostenvorteile für die Nutzer des Produkts ergeben. Dies geschieht überwiegend durch die qualitative Auswirkung einzelner Transaktionsaktivitäten auf die Nutzenfunktion der Marktteilnehmer (siehe [Wiki06h]). Im Gegensatz dazu bewirkt der Skaleneffekt Einsparungen durch die Erhöhung der produzierten Menge, üblicherweise durch die Verteilung der Fixkosten auf eine erheblich größere Zahl von Kunden. Beide Effekte beeinflussen auch die Entwicklung von IT-Produkten und Projekten, letztlich insbesondere durch die Kosten und Qualität der zugrunde liegenden Werkzeuge und Plattformen. Wichtig ist aber die Erkenntnis, dass IT-Projekte maßgeblicher durch Verbundeffekte denn durch Skaleneffekte betroffen sind (siehe dazu insbesondere [GSCK04], S. 158–160 und S. 348–350). Dies bedingt, dass sich sowohl die Qualitätssteigerungen als auch Kosteneinsparungen im Projektgeschäft nur durch eine konsequente Optimierung der eigentlichen Entwicklungsprozesse erreichen lassen. Skaleneffekte dagegen wirken sich lediglich auf Standardprodukte aus, die in großer Zahl verkauft werden (z. B. Office-Lösungen, Sicherheitssoftware, Betriebssysteme).

In Verbindung mit den bereits in Abschnitt 2.2.1 vorgestellten Erfolgsfaktoren von IT-Projekten ergibt sich damit die Erkenntnis, dass nicht so sehr die weitere Detailoptimierung technischer Fragestellungen von Bedeutung ist, sondern dass insbesondere in der Kommunikation und im Anforderungsmanagement Potenziale verborgen liegen. Daher liegt es nahe, die automatisierte Erzeugung von IT-Lösungen so aufzustellen, dass die zugrunde liegenden Konzepte und Modelle insbesondere auf eine Verbesserung der allgemeinen Kommunikationssituation sowie eine Annäherung an die Projektpartner (letztlich die Domänenexperten, siehe dazu auch Abschnitt 2.2.2

sowie die Ausführungen zur Notwendigkeit von *Model Driven Requirements Engineering (MDRE)* in [Trat06]) ausgerichtet sind.

Im Rahmen des noch zu beschreibenden CP-Ansatzes (siehe Teil II dieser Arbeit) wird sich diese Erkenntnis wie folgt niederschlagen:

- Die modellgetriebene Entwicklung von Software spezialisiert sich auf die Verwendung von einfachen Fachmodellen, um den besonderen Anforderungen von Domänenexperten gerecht zu werden und ein domänenspezifisches Vokabular zur Spezifikation und Umsetzung von Anforderungen bereitzustellen (siehe Abschnitt 4.2.2.2).
- Conceptual Programming wird unterteilt in CMDSD (Kapitel 5) als Grundlage für Werkzeuge, auf deren Basis effektive Projektprozesse aufgesetzt werden können und CDPM (Kapitel 6), um CMDSD-basierte Projekte effizient durchzuführen.

Wir fassen zusammen: Automatisierung wird als mögliche Methode um so greifbarer, je stärker es gelingt, die zu verbindenen Systeme bezüglich ihrer Schnittstellen zu abstrahieren. Existierende Ansätze zur modellgetriebenen Generierung von Software-Lösungen bieten hierfür Ansätze. Indem wir konsequent von technischen Details abstrahieren, erhöhen wir zum einen die Wiederverwendbarkeit der Modelle und schaffen zum anderen die Möglichkeit, die Transformationsregeln für Modelle zu verändern, ohne dass Änderungen im Modellierungswerkzeug oder an den Modellen selbst erforderlich wären. So reduzieren wir die Entwicklungszeit in Projekten, indem wir den Grad der Automatisierung durch die Generierung lauffähiger Lösungen erhöhen. Zudem sichern wir Investitionen sowohl in Modelle als auch in Technologien, da durch die hohe angestrebte Abstraktion Implementierungen „unter der Haube“ ausgetauscht bzw. erneuert und existierende Systeme durch einmalige Neugenerierung überarbeitet werden können. Den exemplarischen Beweis dafür treten wir in Abschnitt 9.1.2.3 auf S. 385 an. Die dafür existierenden Ansätze und Technologien betrachten wir in Kapitel 3 (insbesondere in Abschnitt 3.3), um dann einen Best-of-Breed-Ansatz für die Herleitung unserer auf den mittelständischen Kontext fokussierten Lösung zu selektieren.

2.2.4 Agilität

Agile Softwareentwicklung stellt nach den Definitionen der *Agile Alliance* ([AgAl06]) die folgenden Werte ([AAGI06]) in den Mittelpunkt des Denkens und Handels agiler Entwickler (siehe dazu das bereits Gesagte in Abschnitt 2.2.1 ab S. 24):

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Sind diese Forderungen zunächst eher ideell zu sehen, gibt es mittlerweile eine Vielzahl von Untersuchungen, Forschungsarbeiten und Studien, die die Effektivität agiler Softwareentwicklung belegen. Hier sind insbesondere [BoTu03b] und die dortige recht umfangreiche Auflistung verschiedenster Studien sowie der sehr interessante Aufsatz [ErFa02] zu nennen. Letzterer überträgt Erkenntnisse basierend auf der finanzmathematischen Optionspreistheorie und hier speziell zu Realoptionen (engl.: *real options*) auf die Rahmenforderungen agilen Projektmanagements am Beispiel des Extreme-Programming-Ansatzes (siehe [BeAn05]) und weist so nach, dass insbesondere die möglichst späte endgültige Entscheidungsfindung zu einer Risikominimierung und Wertmaximierung in Projekten beiträgt.

Auffallend ist, dass die meisten heutigen Verfahren zur Verbesserung der Qualität im Projektmanagement von IT-Projekten einen äußerst präskriptiven bzw. planbasierten Charakter haben, so zum Beispiel der Rational Unified Process (siehe [Vers00, Kruc04]) oder das V-Modell bzw. V-Modell-97 (siehe [DrHM98, DrWi99], detaillierte Ausführungen hierzu folgen in Abschnitt 3.4.2 ab Seite 127). Insbesondere werden der formale Charakter eines Projekts sowie die Dokumentation der einzelnen Tätigkeiten und der erstellten Artefakte betont, um jederzeit eine Übergabe des Projektes an neue bzw. andere Mitarbeiter zu ermöglichen und Ergebnisse reproduzierbar und nachvollziehbar zu machen. Zudem stellen die Artikel die technische Modellierung z. B. in Form von UML-Modellen in den Mittelpunkt der Aufmerksamkeit. Erfahrungsberichte zeigen, dass die starken Strukturierungsvorgaben dieser Prozesse häufig selbst Auslöser für Probleme in Projekten sind, da sowohl die Kommunikation zwischen den Projektmitgliedern als auch die Tätigkeiten zur aktiven Qualitätsverbesserung von IT-Systemen (z. B. das Schreiben von Tests) häufig zu kurz kommen. Letztlich weil ein sehr hoher Dokumentationsaufwand gefordert oder zumindest seitens der Projektmitglieder angenommen wird (siehe dazu z. B. die Erfahrungsberichte in [Ambl04a, Shal01] sowie die Fehleruntersuchung in [LeTu93]).

Erst in der jüngeren Vergangenheit erfreuen sich agile Prozesse und Methoden einer erhöhten Aufmerksamkeit (siehe [Beed02, Ecks04, Mart03, Popp03]). Im Mittelpunkt steht der Wunsch nach Vorgehensmodellen, die die Kommunikation im Team und mit dem Kunden fördern und die daraus resultierenden höheren Erwartungen

an ein effektives Anforderungsmanagement erfüllen. Wechselnde Anforderungen werden dabei nicht als die Ausnahme sondern als Regelfall begriffen. Entsprechend sind agile Methoden und Verfahren darauf ausgerichtet, wechselnde Anforderungen zu akzeptieren, ohne Einschränkungen in der Qualität des entstehenden Systems hinzunehmen und sie sogar effizient in den Projektverlauf zu integrieren.

Interessanterweise nähern sich präskriptive Vorgehensweisen zunehmend agilen Prozessen an und übernehmen Eigenschaften der agilen Softwareentwicklung, um Kommunikation, Flexibilität und den effektiveren Umgang mit sich stetig wechselnden Anforderungen zu betonen (siehe dazu z. B. [Hirs02, Ambl04a, Mart01] für Anpassungen des RUP sowie [RaBr06a, RaBr06b] für die Veränderungen des V-Modells zum V-Modell XT). Zudem verfolgen die stark präskriptiven Verfahren ebenso wie die inhärent agilen Vorgehensmodelle einen weiteren wichtigen Trend: Modellbasierte Generierung von Software gerät zunehmend in den Aktivitätsfokus der Entwickler. Für beide Vorgehensweisen ist die konsequente Verfolgung modellorientierter Vorgehensweisen naheliegend:

1. Stark präskriptive Verfahren legen inhärent großen Wert auf ausführliche und präzise Dokumentation. Insofern sind Modelle seit jeher ein natürlicher Bestandteil dieser Verfahren. Die Idee, der Schwergewichtigkeit dieser Verfahren entgegenzuwirken, indem Modelle direkt zur Generierung von Software eingesetzt werden oder im Extremfall sogar unmittelbar ausgeführt werden, liegt also nahe. Derartige Ansätze werden daher zunehmend verfolgt (siehe u. a. [BaTZ05] für das V-Modell XT und im Fall von RUP per Definiton, da die MDA-Philosophie maßgeblich aus UML hervorgegangen ist und UML wiederum die standardisierte Modellierungssprache für den RUP ist).

2. Die modellbasierte Generierung von Software ist auch eine wichtige Ergänzung für agile Verfahren, da die Modellbildung u. a. die Kommunikation im Rahmen der Anforderungsdefinition zwischen Entwicklern und Kunden fördert und zudem die Flexibilität im Umgang mit wechselnden technischen Anforderungen maßgeblich verbessert (siehe [Ambl02a, Ambl04b, Ambl06a] für ausführlichere Ausführungen hierzu).

Allerdings gibt es auch Szenarien, unter denen sich agile Softwareentwicklung noch beweisen muss (zitiert nach [Wiki06k]) und zu denen es bislang mehr Hinweise daraufhin gibt, dass präskriptive und planbasierte Verfahren (wie die in Abschnitt 3.4.2 vorgestellten Vorgehensmodelle) leistungsfähiger sind (siehe auch [BoTu03b]):

- *Large scale development efforts (> 20 developers). Scaling strategies have been*

described.[8]

- *Distributed development efforts (non-co-located teams). Strategies have been described in* Bridging the Distance[9] *and* Using an Agile Software Process with Offshore Development[10]*.*
- *Mission- and life-critical efforts*
- *Command-and-control company cultures*

Im Hinblick auf das Ziel dieser Arbeit werden wir diese Punkte nicht weiter vertiefen, da sie im mittelständischen Kontext aufgrund der bereits diskutierten Strukturen und Budgetbeschränkungen die Ausnahme darstellen. Einzige potenzielle Ausnahme sind „command and control“ Firmenkulturen, die bei inhabergeführten mittelständischen Unternehmen durchaus zu finden sind. Hier werden wir im Zusammenhang mit CDPM zeigen, wie die Vorzüge von CMDSD-Ansätzen dazu genutzt werden können, solche Strukturen aufzubrechen und agiler zu gestalten (siehe Kapitel 6 ab S. 241).

Wir fassen zusammen: Agile Vorgehensmodelle unterstützen in optimaler Weise Umgebungen, in denen sich Anforderungen häufig ändern, Systeme kontinuierlich weiterentwickelt werden müssen und Kosten durch Projektinfrastrukturen auf das notwendige Minimum reduziert werden sollen. Der Preis dafür ist erhöhte Disziplin, die bei modellbasierten Ansätzen aber ohnehin von großer Bedeutung ist. In Kombinationen mit den Grundwerten agiler Softwareentwicklung stellen agile Entwicklungsverfahren somit das ideale methodische Werkzeug zur Optimierung von IT-Projekten unter den bestehenden Rahmenbedingungen dar. In Kapitel 3 werden wir deshalb (insbesondere in Abschnitt 3.4 und 3.4.3) verschiedene (agile) Vorgehensmodelle betrachten, um daraus Schlüsse für unseren Lösungsansatz abzuleiten.

2.3 Zusammenfassung

Wir haben in diesem Kapitel zunächst die besonderen Stärken und Schwächen des deutschen Mittelstands herausgearbeitet. Ein Schwerpunkt lag dabei auf den Aktivitäten und Potenzialen mittelständischer Unternehmen hinsichtlich der Durchführung von IT-Projekten mit individuellen Entwicklungen und Lösungen. Spezielle Stärken

[8] Siehe [Ambl06b].
[9] Siehe [Ambl02b].
[10] Siehe [Fowl06a].

liegen in kurzen Entscheidungswegen, sehr hoher fachlicher Expertise und der schnellen Reaktionsfähigkeit auf sich verändernde Rahmenbedingungen. Dem gegenüber stehen als besondere Schwächen eine generell eher geringe IT-Expertise und äußerst beschränkte Budgets und Personalressourcen für Projekte. Aus den Ergebnissen verschiedener Studien und Analysen haben wir vier Ansatzpunkte identifiziert, die es uns erlauben werden, durch geeignete Kombination und Adaption die Qualität mittelständischer IT-Projekte bei gleichzeitiger Senkung von Kosten zu steigern. Diese Ansatzpunkte haben wir mit den Begriffen *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität* zusammengefasst.

Wir haben dann gezeigt, wie wir durch die Kombination dieser Lösungsansätze durch entstehende synergetische Effekte einen Paradigmenwechsel in der Software-Entwicklung gerade für kleine und mittlere Unternehmen herbeiführen werden. Somit sehen wir das auf Seite 8 als Kernthese formulierte Ziel unserer Arbeit als zumindest theoretisch erreichbar an und werden die nachfolgenden Kapitel der Ausarbeitung und Verifikation der konkreten Methode widmen. Die zuvor genannten Ansatzpunkte werden sich dabei gegenseitig befruchten, so dass das resultierende Verfahren unter Beachtung gewisser Randbedingungen (insbesondere Effektivitätsgewinne durch die Begrenzung auf eine eingeschränkte Problemdomäne), die von uns in Kapitel 1 formulierten Ziele strukturiert erreicht:

- Kundenorientierung sichert uns auf grundlegende Weise eine für alle Seiten verständliche und auf effektive Kommunikation ausgerichtete Vorgehensweise.
- Domänenfokussierung garantiert die Qualität des Vorgehensmodells, indem zum Einen das Anforderungsmanagement auf Basis einer für Domänenexperten verständlichen Sprache strukturiert und zum Anderen die Qualität der Lösungen durch eine technologische Konzentration auf die Abbildung des Problemraums sichergestellt werden. Im Zusammenhang mit dem kundenorientierten Vorgehen ist gewährleistet, dass die resultierenden Modellierungsverfahren auch für Fachexperten einfach zu erfassen sind.
- Automatisierung gewährleistet kosteneffiziente Umsetzungen modellierter Systeme, indem auf Basis modellbasierter Verfahren direkt ausführbare Systeme erzeugt werden. Modellbasierte Generierung schafft die Voraussetzungen für Rapid Protoyping, was wiederum aufgrund domänenfokussierter Sprachen und Modelle eine enge Zusammenarbeit mit dem Kunden – im Extremfall in Form gemeinsamer Modellierung und Entwicklung durch ein gemischtes Team aus Domänen- und IT-Experten – ermöglicht. So werden Kommunikationswege ebenso wie das Risiko für Missverständnisse in der Anforderungsdefinition mi-

nimiert, was wiederum die zu befürchtenden Kosten für Fehlerbehebungen drastisch reduziert.

- Agilität garantiert, dass die zuvor beschriebene enge Form der Zusammenarbeit so effektiv wie möglich durchgeführt werden kann. Wechselnde Anforderungen sind kein Risiko, sondern ein im Vorgehensmodell berücksichtigtes Ereignis, auf das planvoll, kosteneffizient und flexibel reagiert werden kann. Rapid Prototyping erlaubt dabei das „schnelle Ausprobieren“ neuer Ideen, Domänenfokussierung sichert zu, das Anforderungen sich tatsächlich im Wesentlichen dann ändern, wenn neue fachliche Ideen oder Herangehensweisen ins Auge gefasst werden.

In Summe entsteht so eine Vorgehensweise, die einen Paradigmenwechsel von klassischen Technologie- und Controlling-Verfahren (und häufig auch stark präskriptiven Vorgehensweise des Projektmanagements) hin zu einer kunden-, domänen- und anforderungsorientierten Projektsteuerung und -realisierung fokussiert.

Abbildung 2.10 fasst diese Synergieeffekte auf Grundlage von Abbildung 2.2 auf Seite 21 noch mal zusammen.

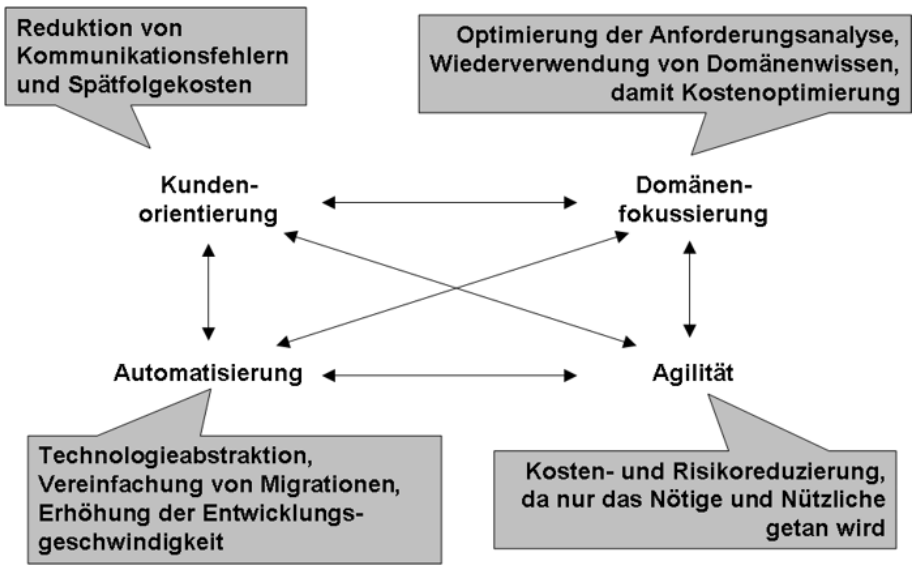


Abbildung 2.10: Integrierter Optimierungsansatz durch Conceptual Programming (Zusammenfassung)

Demzufolge werden wir im folgenden Kapitel 3 zunächst einen Überblick über existierende kundenorientierte, domänengetriebene und modellgetriebene Ansätze und Verfahren zur Software-Entwicklung geben. Zudem überprüfen wir existierende Vorgehensmodelle zur Abwicklung von IT-Projekten dahingehend, wie weit sie diese

Ansätze unterstützen. Die Ergebnisse dieser Informationssammlung schließen unsere Anforderungsanalyse und Spezifikation für einen neuen Ansatz zur Optimierung mittelständischer IT-Projekte und leiten Teil II dieser Arbeit ein, der diesen neuen Ansatz in Form des Conceptual Programming (CP) beschreibt.

Kapitel 3

Technische und methodische Grundlagen

„Aller Anfang ist leicht – wenn man ihn mit dem Ende vergleicht."
— Gerhard Uhenbrock (*1929), Deutscher Aphoristiker

In diesem Kapitel werden wir bereits existierende Ansätze ausführlich untersuchen und mit den in Kapitel 2 identifizierten Optimierungspotenzialen bei der Umsetzung von Software-Lösungen in kleinen und mittleren Unternehmen vergleichen. Im Zentrum unserer Betrachtungen stehen also die in Abschnitt 2.2 formulierten Forderungen nach *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität*. Dazu befassen wir uns in Abschnitt 3.1 mit kundenorientierten Disziplinen der Software-Entwicklung, die sowohl die Bedürfnisse des Anwenders in den Mittelpunkt der Projektarbeit rücken. Im folgenden Abschnitt 3.2 stellen wir domänenfokussierte Ansätze der Software-Entwicklung vor. Anschließend gehen wir in Abschnitt 3.3 auf den Stand der Forschung und Entwicklung zu modellgetriebenen Verfahren ein, mit denen eine Automatisierung des Software-Entwicklungsprozesses angestrebt wird. Letztlich beschäftigen wir uns in Abschnitt 3.4 mit existierenden Vorgehensmodellen zur Umsetzung von IT-Projekten, wobei wir dem gegenwärtigen Trend zu agilen Ansätzen besondere Aufmerksamkeit widmen.

Wir werden jeweils auf die Stärken und Schwächen der verschiedenen Ansätze eingehen und diese unter besonderer Berücksichtigung der von uns in Kapitel 2 formulierten Anforderungen analysieren. In Abschnitt 3.5 fassen wir unsere Beobachtungen und Erkenntnisse zusammen und leiten über zu unserer Definition des Conceptual Programming (CP) in Teil II dieser Arbeit. Dabei werden wir eine Reihe von Konzepten identifizieren, die in den jeweils betrachteten Ansätzen zu Leistungs- und

Qualitätsverbesserungen geführt haben. Allgemein versteht man unter einem Konzept (zitiert aus [Wiki09r])

- *einen Plan, ein Programm für ein Vorhaben, siehe Konzeption*
- *einen ersten Entwurf*
- *eine Vorstufe einer Theorie*
- *eine gedankliche Zusammenfassung (Vorstellung) von Gegenständen und Sachverhalten, die sich durch gemeinsame Merkmale auszeichnen [...]*

[Wikt09] bezeichnet ein Konzept als *„formuliertes Gedankengerüst zur Realisierung von etwas“*. Wir definieren für dieses Kapitel den Begriff *Konzept* wie folgt:

Definition (Konzept)**.** *Ein Konzept fasst die Stärken, Best Practices oder bewährten methodischen Vorgehensweisen existierender Ansätze zusammen und beinhaltet direkt oder indirekt eine Handlungsanweisung hinsichtlich des in dieser Arbeit zu formulierenden Lösungsansatzes.*

Die in den analysierten Ansätzen identifizierten Konzepte greifen wir bei der Definition unseres Lösungsansatzes wieder auf und zeigen, wie diese Einzelkonzepte synergetisch zu einem leistungsfähigeren Ansatz verbunden werden können. Konzepte werden wir wie folgt hervorheben:

> **Konzept 3.1** (Identifikation von Konzepten)**.** *Wir werden besondere Ansatzpunkte und Leistungsmerkmale existierender Vorgehensmodelle zur späteren einfachen Referenzierung in Form von Konzepten festhalten. Wenn wir Konzepte referenzieren, geschieht dies durch Verweise der folgenden Form: 3.1 ▷ 48. Dieser Verweis bezieht sich auf das Konzept 3.1 auf Seite 48.*

Dazu zeigen wir in Abschnitt 3.6, wie sich unser Modell aus den verschiedenen Konzepten, die wir im Verlauf dieses Kapitels identifizieren werden, herleitet.

Nach der Definition von CP in Teil II wird dann in Kapitel 7 eine kritische Abgrenzung zu den hier besprochenen Ansätzen erfolgen. In Teil III werden wir dann belegen, wie sich die synergetische Kombination der identifizierten Konzepte in deutlichen Effizienzsteigerungen bei der Entwicklung von Systemen niederschlägt.

3.1 Kundenorientierte Disziplinen der Softwareentwicklung

Zunächst betrachten wir verschiedene Teildisziplinen des Software Engineering, die sich mit der Frage beschäftigen, wie Technologie und Anwender stärker in Einklang gebracht werden. Wir legen unseren Fokus dabei auf Disziplinen, die sich mit einer Veränderung der Kommunikation zwischen Domänenexperten und IT-Experten beschäftigen und dadurch versuchen, den Prozess der Anforderungsanalyse und der Umsetzungsvalidierung effektiver zu gestalten. Dabei beschränken wir uns im Schwerpunkt auf Ansätze, die eine signifikante Sichtbarkeit oder Anhängerschaft gewonnen haben und skizzieren weniger bekannte oder erfolgreiche Ansätze nur kurz. Wir betrachten in chronologischer Entstehungsreihenfolge das Convergent Engineering (Abschnitt 3.1.1), Intentional Programming (Abschnitt 3.1.2), Concept Programming (Abschnitt 3.1.3), Convergent Architecture (Abschnitt 3.1.4), Fluent Interfaces (Abschnitt 3.1.5) und Language Workbenches (Abschnitt 3.1.6). Abschnitt 3.1.7 fasst zudem einige weniger bekannte und verbreitete Ansätze zusammen.

3.1.1 Convergent Engineering (CE)

Die Idee des *Convergent Engineering* geht zurück auf ein Werk von David A. Taylor [Tayl95]. Taylor besitzt langjährige Erfahrungen in der Geschäftsprozess- und Systemanalyse und ist ein Verfechter objektorientierter Technologien als Mittel, um Geschäftsprozessanalyse und -design mit Systemanalyse und -entwurf zu kombinieren. Dazu verbindet er objektorientierte Analyse- und Designmethoden mit Methoden des Business Engineering. Die resultierende Methodologie bezeichnet er als *Convergent Engineering* (siehe [Tayl95], Vorwort). Die Methode zielt darauf ab, Geschäftsprozesse in Unternehmen zu modellieren und zu optimieren. Trotz des breit gefächerten Anspruchs wird die Methodenicht durch eine Fokussierung auf domänenspezifische Probleme verfeinert.

Grundmotivation seiner Methodologie sind die folgenden Erkenntnisse (kurz zusammengefasst nach [Tayl95], S. 1–11):

- Unternehmen genießen nur noch dann einen Wettbewerbsvorteil, wenn es Ihnen gelingt, ihre Produkte im Vergleich zu Mitbewerbern schneller, günstiger und in höherer Qualität zu produzieren. Führerschaft in einem oder zwei dieser

Aspekte reicht heute – außer bei einer dominanten Position in einer Marktnische – nicht mehr aus.

- *Business Engineering* ist die Methode, die als Grundlage zur kontinuierlichen Qualitätssteigerung bei gleichzeitiger Senkung von Kosten und Zeitverbrauch Anwendung findet.
- Es gibt Interessens- und Flexibilitätskonflikte zwischen den taktischen Veränderungen in Organisationsstrukturen und Unternehmensprozessen sowie den Technologien, die strategisch zur Unterstützung des Unternehmens eingesetzt werden.
- Studien zeigen, dass nur die Unternehmen überleben, die sich diesem Zwang zur Veränderung aktiv stellen und ihn erfolgreich bewältigen.
- Wichtig ist dabei eine kontinuierliche Evolution, da auch die Zwänge zur Veränderung kontinuierlich wirken. Punktuelle Änderungen alleine reichen nicht aus.
- Analysen zeigen, dass Software Engineering dann die effektivste Unterstützung in diesem Veränderungsprozess leistet, wenn Business Engineering und Software Engineering eng miteinander verzahnt sind und sich gegenseitig befruchten. Fach- und IT-Domänen müssen also enger zusammenrücken und ihre jeweiligen Stärken synergetisch bündeln.

  Konzept 3.2 (Enge Kommunikation)**.** *Die Verkürzung der Kommunikationskette zwischen Domänen- und IT-Experten verbessert die Projektqualität signifikant. Daher ist eine gemeinsame Sprache anzustreben, die eine solche Kommunikation ermöglicht.*

- *Convergent Engineering* strebt daher an, die Analyse und Umsetzung von Geschäftsstrategien so eng wie möglich mit den Prozessen zur Implementierung von Softwaresystemen zu koppeln. Die notwendigen Übersetzungs- oder Reformulierungsschritte werden dabei minimiert. So soll die Flexibilität und Unmittelbarkeit des Vorgehens gewahrt bleiben.
- Wesentliche Vorteile von Convergent Engineering sind laut Taylor:
  - eine Vereinfachung des Software-Entwicklungsprozesses, weil nur ein System entworfen und umgesetzt wird (denn die Übersetzung von Ergebnissen des Business Engineering auf die Welt des Software Engineering entfällt),
  - die Beseitigung der Kluft zwischen Geschäftsprozessen und IT-Prozessen aufgrund der engen Verbindung,

- die Förderung von Veränderungen, da die Anforderungen an die Koordination von Geschäfts- und Softwareanforderungen minimiert werden.

Taylor will den Ist-Zustand durch die Nutzung modellbasierter Systeme verbessern. Die Disziplin der Anwendungsentwicklung sieht er als Methode einer Zeit, in der Anwendungen eine überschaubare Größe hatten und das dazugehörige Marktumfeld im Vergleich zu heute eher statisch ausgelegt war. Resultat der klassischen Anwendungsentwicklung und ihrer Methoden sind daher auch statische und insbesondere monolithische Applikationen (von ihm auch als Software Silos bezeichnet), die sich Änderungen tendenziell widersetzen. Beobachtbar wird dies an der langen Vorlaufzeit zwischen der Planung und Umsetzung komplexerer Projekte (bis zu fünf Jahren, siehe [Tayl95], S. 15). Zudem weist er darauf hin, dass die Beschleunigung der Entwicklungsprozesse zwar vordergründig von essentieller Bedeutung zu sein scheint, das Unausweichliche aber nur hinauszögert: Ab einem gewissen Punkt sind die Veränderungen und Anpassungen an komplexen Systemen so aufwändig, dass Geschwindigkeitsverbesserungen auf Seiten der Implementierer nicht mehr ausreichen, um Anwendungen schnell genug an veränderte Geschäftsanforderungen anzupassen. Rückblickend wird die Richtigkeit dieser 1995 aufgestellten These nur zu klar, betrachtet man die gegenwärtige Konzentration auf agile und modellgetriebene Prozesse, die nur zu konsequent die Ideen von Taylor gezielt zu optimieren suchen.

Taylor strebt eine Vereinfachung mittels modellbasierter Entwicklung an. Dabei stützt er sich auf die folgende Argumentation:

- Modellierung hat sich sowohl für Geschäftsprozesse als auch IT-Systeme seit vielen Jahren bewährt. Ein nachhaltiges Problem ist aber die Vielzahl verfügbarer Modelle. Exemplarisch greift Taylor dazu Daten-, Prozess-, Workflow-, Finanz- und Simulationsmodelle heraus. Schon diese Modelle werden nur selten durch ein integriertes IT-System repräsentiert, sondern spiegeln sich in einer Vielzahl von Systemen wider. Die physikalische Trennung von logisch eng zusammengehörenden Modellen reduziert ihre Aussagekraft und beschränkt die Effizienz in der Umsetzung von neuen Anforderungen durch die Vielzahl zu ändernder Systeme und Schnittstellen.
- Daraus resultiert laut Taylor die essentielle Notwendigkeit, alle Modelle miteinander zu verbinden und eine Gesamtmodellsicht auf Unternehmen zu erstellen. Dieses Gesamtmodell stellt dann auch die Grundlage für eine integrierte IT-Systemlandschaft, da mittels seiner Convergent Engineering Methode Fachmodell und Systemausprägung eine direkte Entsprechung besitzen ([Tayl95], S. 22). Alle existierenden Modelle werden somit Facetten eines integrierten

Gesamtmodells.

- Aus der Integration ergeben sich drei maßgebliche Nutzungsmöglichkeiten:
  1. Zur Repräsentation der Prozesse und Strukturen eines Unternehmens, so dass es einfacher wird, das Unternehmen zu verstehen und Optimierungsmöglichkeiten zu finden.
  2. Zur Simulation. So können Projektionen bezüglich zukünftiger Operationen ausgearbeitet werden, um Unternehmensstrategien zielgerichteter festlegen zu können.
  3. Als Ausführungsplattform. Modelle sollen direkt ausgeführt werden, um zum einen den Automatisierungsgrad der Softwareerstellung zu erhöhen und zum anderen direkter die Korrektheit von Modellen zu verifizieren.
- Objektorientierte Technologien und objektorientierte Modellierung sind die technologische Grundlage für die Definition des Convergent Engineering. Objekte werden als Modellierungselemente verwendet.
- Insbesondere weist Taylor darauf hin, dass das Denken in Objekten der natürlichen und anschaulichen Denkweise von Menschen entspricht (siehe [Tayl95], S. 47). Physikalische Objekte aus der realen Welt werden dabei in Form von Gedankenmodellen ausgeprägt.

  Wir sehen darin ein weiteres Indiz für die Nutzung domänenspezifischer Modelle als Grundlage für ein effizientes Verfahren zur automatisierten Erzeugung von Software. Domänenobjekte kommen den gedanklichen Modellen von Entitäten der realen Welt am Nächsten und helfen Domänenexperten dabei, ihr Wissen in einem formalen System zu formulieren.
- Um den eigentlichen Modellierungsprozess optimal zu unterstützen, wird auf die Bedeutung der richtigen Teamzusammensetzung hingewiesen.
- Ein optimales Team sollte sich aus Domänenexperten der zu modellierenden Fachdomäne, ausgebildeten Objektmodellierern, einem Mediator sowie einem Schriftführer zusammensetzen, um den Modellierungsvorgang optimal zu unterstützen (siehe [Tayl95], S. 53).
- Zudem sind die Teilnehmer eines Modellierungsworkshops intensiv über die Potenziale und Möglichkeiten des Convergent Engineering aufzuklären, um ihre Kreativität so anzuregen, dass die zu modellierenden Prozesse nicht nur in einem formalen Modell abgebildet sondern auch verbessert werden.
- Die Workshops beginnen mit der Erstellung eines Grobmodells der zu behandelnden Fragestellung. Dieses Grobmodell wird im Verlauf des Workshops

schrittweise verfeinert. Als optimale Workshopdauer werden drei bis fünf Tage angesehen. Es sollten nicht mehr als zehn Personen an einem Workshop teilnehmen.

- Die Erstellung des Modells folgt den Prinzipien objektorientierter Modellierung, wobei in allen Phasen großer Wert darauf gelegt wird, domänenspezifische Begrifflichkeiten zu verwenden und von technischen Details wie z. B. Persistenz zu abstrahieren.
- In einem abschließenden Schritt werden die Modelle zunehmend verfeinert, so dass sie an die Implementierer übergeben werden können. Insbesondere werden die definierten Anforderungen in Methoden übersetzt, die in der Lage sind, die Anforderungen zu erfüllen.
- Taylor formuliert zudem ein Objektmodell für die Modellierung von Geschäftsobjekten und Geschäftsprozessen. Das Objektmodell vereinfacht die eigentliche Modellierung von Unternehmensstrukturen.
- Um die Modellierung zu unterstützen, definiert Taylor eine Modifikation des Wasserfallmodells, indem er die Analysephase aufgibt und das Wasserfallmodell auf jedes einzelne Geschäftsobjekt des modellierten Systems anwendet. Dabei wird eine pragmatische Vorgehensweise postuliert, bei der zwar die Phasen konsequent eingehalten, die Menge an zu produzierender Dokumentation aber insbesondere für überschaubare Geschäftsobjekte drastisch reduziert wird. Auf diese Weise will Taylor die Kosten des Wasserfallmodells bei Erhaltung seines Nutzens beherrschbar halten (siehe [Tayl95], S. 148–150.).

Zusammenfassend argumentiert Taylor in seinem Ansatz basierend auf seinen Erfahrungswerten sehr deutlich für ein Zusammenwachsen von Fach- und IT-Modellierung, so dass die resultierenden Systeme leichter anpassbar werden. Er bekräftigt so unsere These, dass nur eine engere Zusammenarbeit zwischen Domänen- und IT-Experten, unterstützt durch effiziente Modellierungserkzeuge, zu einer nachhaltigen Kostenreduktion sowie notwendigen Flexibilisierung von IT-Systemen führen kann.

Wir folgen allerdings nicht seiner These von der Notwendigkeit eines integrierten Gesamtmodells für Fach- und IT-Systeme. Vielfältige Projekterfahrungen seit dem Erscheinen von Taylors Werk zeigen, dass die Komplexität eines solchen Gesamtmodells so groß ist, dass entsprechende Projekte nur mit gewaltigen finanziellen und organisatorischen Aufwand erfolgreich bewältigt werden können. Bekanntestes Beispiel dafür ist sicherlich die Firma SAP mit ihrer gleichnamigen Standard-ERP-Software. SAP deckt mit seiner Lösung und dem darunterliegenden Modell vielfältige Unternehmensprozesse ab – Branchenspezialisierungen sind aber nur ansatzweise

berücksichtigt und die Komplexität des Modells ist mit mehr als 10.000 zugrunde liegenden Datenbanktabellen, deutlich mehr Datenbanktriggern und Millionen von Codezeilen so hoch, dass typische SAP-Projekte viele Mannjahre umfassen, wobei 70% des Projektaufwands in die Anpassung der Standardlösung gehen. Derartige Beispiele zeigen, dass ein integriertes Gesamtmodell zwar eine gute theoretische Idee ist, aber aufgrund der Komplexität realer Prozesse nur in hochstandardisierten Bereichen machbar erscheint. Insbesondere in dem von uns betonten mittelständischen Kontext und bei der Fokussierung auf branchenspezifische differenzierende Software scheint dieser Ansatz daher nicht weiter relevant zu sein.

Implizit bestätigt Taylor damit das von uns identifizierte Optimierungspotenzial der Domänenfokussierung, das wir in Abschnitt 3.2 detaillierter untersuchen.

Rückblickend hat sich gezeigt, dass Objektorientierung nicht das Allheilmittel für Modellierungsprobleme ist. Taylor hat darin 1995 den zentralen Mechanismus zur Annäherung von Domänen- und IT-Experten und für die einfache Modellierung gesehen. Mittlerweile hat sich gezeigt, dass Objektorientierung alleine nicht ausreichend ist, da sie in vielen Fällen entweder zu grob- oder feingranular ist und Domänenexperten mit vielfältigen technischen Fragen belastet, die effizienter abstrahiert werden müssen (siehe dazu neue technische Ansätze wie z. B. AOP, COP, etc. in [FCEl04, Qi4j08]).

Zudem sehen wir eine Schwäche darin, dass Convergent Engineering ein sehr breit gefächertes Thema (allgemeine Geschäftsprozessanalyse und -optimierung) angeht und keine direkten Ansatzpunkte für weitere Optimierungen anbietet, wenn eine Domänenspezialisierung (zum Beispiel vertikal für ein Thema wie Kundenbeziehungsmanagement oder horizontal für eine bestimmte Branche) vorgenommen wird. Es werden nur sehr allgemeine Hinweise für die Modellierung gegeben.

Zur Diskussion des interessanten Widerspruchs zwischen Software Factories (Abschnitt 3.3.3) und dem hier vorgestellten Convergent Engineering sei auf Seite 108 verwiesen.

3.1.2 Intentional Programming (IP)

Intentional Programming (IP) zielt darauf ab, eine erweiterbare Plattform für Metaprogrammierung bereitzustellen. Czarnecki definiert IP als *„an extendible programming and metaprogramming environment based on* active source*, that is, a source that may provide its own editing, rendering, compiling, debugging, and versioning behavior“* (siehe [CzEi00], S. 503). Erklärte Ziele sind:

- die optimale Kombination von natürlichen Notationen für Algorithmen mit hoher Flexibilität und ausgezeichneter Performance,
- die Bereitstellung domänenspezifischer Unterstützung für Programmieraufgaben aller Art,
- eine erhöhte Abstraktion durch Formulierung von Intentionen,
- der engen Integration von Quelltexten und Entwicklungsplattform durch Quelltexte und Erweiterungsbibliotheken mit durch Metaprogrammierung angereicherten Informationen z. B. über die Visualisierung der Code-Artefakte, Debugging-Möglichkeiten, usw.

Zudem kann eine IP-Entwicklungsplattform um sogenannte *Intentionen* erweitert werden. Intentionen sind Erweiterungen der Programmiersprache, die dynamisch in die Plattform eingebunden werden und sowohl Hinweise auf die Eingabe, Visualisierung und das Debugging, aber auch auf die Ausführung, usw. der mit diesen Intentionen beschriebenen Algorithmen enthalten. Es handelt sich also um die Idee einer hochdynamisch erweiterbaren Entwicklungsplattform, die wir auch in ihren Grundsätzen festhalten:

Konzept 3.3 (Erweiterbarkeit)**.** *Plane der Domäne angemessene Erweiterungsmöglichkeiten. Diese unterstützen die Evolutionsfähigkeit einer Domäne und erlauben die graduelle Weiterentwicklung der domänenspezifischen Sprache.*

Besonderheit ist zudem, dass Quelltexte nicht als Textformat angesehen werden sondern als Graphstrukturen repräsentiert werden (Abstract Syntax Trees, AST, siehe z. B. [Jone03]), was einen viel flexibleren Umgang mit den Quelltexten erlaubt. Diese sind besonders geeignet, um die Vermischung unterschiedlichster Darstellungsformen für die Programmlogik im klassischen IP-Ansatz zu realisieren. Dort werden z. B. formelbasierte Darstellungen mit Schaltplänen, Logikausdrücken und anderen Darstellungsformen kombiniert, um so die für das jeweils betrachtete Problem angemessendste Darstellung zu nutzen. Wir vermerken entsprechend:

Konzept 3.4 (Höherwertige Repräsentationsformen)**.** *Visualisiere Beschreibungen in der dem jeweiligen Kontext angemessensten Form.*

Ausführliche Erläuterungen zu diesem Ansatz finden sich in Kapitel 11 von [CzEi00] sowie in den vom Urheber des Ansatzes Charles Simonyi veröffentlichten Ausführungen unter [Simo95, Simo96, Simo97, InSo09]. Die theoretischen Ausführungen zu IP beschäftigen sich schwerpunktmäßig mit Fragen der Quelltextrepräsentation als AST, der Transformation, Interpretation und Visualisierung solcher Graphen sowie den Anforderungen an Metaprogrammierung, um dynamische Erweiterungsbibliotheken zu realisieren.

[Pont07] benennt drei Hauptkritikpunkte an Intentional Programming:

1. Das theoretische Argument, dass menschliche Intention praktisch nicht festgehalten werden kann (also dass Computerbenutzer nicht wirklich wissen, was Anwender wollen). Diesem Argument können wir nur zweierlei begegnen:
   a) Wir können anhand praktischer Beispiele zeigen, wie Intention formuliert werden kann. Insbesondere die vielfältigen neuen Veröffentlichungen zur Verwendung domänenspezifischer Sprachen seit 2005 zeigen, dass dies sehr wohl möglich ist (siehe Abschnitt 3.2.8 für detailliertere Ausführungen).
   b) Wären Computernutzer nicht in der Lage, Anforderungen von Anwendern zu verstehen, gäbe es keine funktionierenden Lösungen für Anwenderprobleme. Diese Erkenntnis lässt den vorliegenden Kritikpunkt eher metaphysisch denn real erscheinen.

   Somit halten wir diesen Kritikpunkt für nicht haltbar.
2. Das praktische Argument ist, dass sich Entwickler sehr schwer damit tun, mit Abstraktionen zu leben und gerne in die Details der Programmierung einsteigen, um zu verstehen, ob Probleme aus ihrer Perspektive „richtig“ gelöst werden. Dieses Argument werden wir aufgreifen, wenn wir als grundlegendes Prinzip für die in unserem Ansatz zu entwickelnden CMDSD-Werkzeuge das Prinzip der Abstraktion als Folge von Konzept 3.23 ▷ 73 formulieren (siehe Abschnitt 5.4.2.6 ab Seite 181). Prinzipiell deckt sich dieser Kritikpunkt mit unseren Erfahrungswerten – ebenso prinzipiell muss aus unserer Sicht allerdings gelten, dass die Kundenbedürfnisse im Vordergrund stehen und daher

die optimalen Werkzeuge zur Lösung von Problemen zu verwenden sind. Die individuellen Bedürfnisse von Entwicklern können und dürfen somit nicht einer Verbesserung des Entwicklungsprozesses entgegenstehen.

3. Das (etwas) zynische Argument aus dem Jahr 2005 ist, dass es noch nicht viele greifbare Beispiele für den Erfolg von Intentional Programming gibt. Diesem Argument können wir nur mit einem Verweis auf den beginnenden Siegeszug domänenspezifischer Sprachen aus den letzten Jahren begegnen – detaillierter befassen wir uns mit diesem Thema in Abschnitt 3.2.8 ab Seite 94.

Intentional Programming ist aus unserer Sicht ein wichtiger Weg, um zu zeigen, wie eine effizientere Form der Spezifikation und Programmierung aussehen kann. Wir werden diesen Ansatz mit unserer Forderung nach einer hohen Domänenfokussierung kombiniert mit einer abstrakten und intentionalen Vorgehensweise wieder aufgreifen, um den Werkzeugen für unseren Lösungsansatz eine konzeptionelle Grundstruktur zu geben (siehe Kapitel 5, hier insbesondere Abschnitt 5.4.3.11). Wir begegnen der Kritik an IP[1] dadurch, dass wir unseren Lösungsansatz bewusst immer auf genau eine Domäne fokussieren und im Gegensatz zu IP nicht versuchen werden, eine generische Plattform für IP anzugeben.

Konzept 3.5 (Fokussierung zwecks Realisierbarkeit)**.** *Fokussiere mittelstandsorientierte Lösungen auf eine eng eingegrenzte Domäne, um mit einem zu mittelständischen Budgets passenden Aufwand zu realisieren.*

Die Ausführungen in [CzEi00] liefern viele Ansätze, um die Ideen von IP für unseren Ansatz gewinnbringend zu nutzen: Wir können die Komplexität unserer Plattform nachhaltig reduzieren, indem wir die dynamischen Aspekte von IP ausschließen: Wir brauchen nicht zwingend die Möglichkeit, zur Laufzeit dynamisch Bibliotheken mit Spracherweiterungen integrieren zu können (siehe [CzEi00], S. 504), da wir nur eine

[1] Insbesondere reagieren wir auf den oben erwähnten Kritikpunkt Nummer 3, hinter dem sich eigentlich folgende Aussage bzw. Anklage verbirgt: „Wie kann es sein, dass ein durch privates Vermögen finanziertes Unternehmen wie [InSo09] mit exzellenten Zugängen zur Forschungsgemeinde nach über 13 Jahren (seit 1995) immer noch keine greifbaren Ergebnisse vorweisen kann?“ Wir wollen uns hier nicht an Vermutungen beteiligen, sondern merken nur an, dass – unabhängig von der Qualität der Ergebnisse – eine 13-jährige Entwicklungszeit für die Basisplattform in jedem Fall nicht zu dem von uns betrachteten mittelständischen Kontext passt und wir daher den generellen IP-Ansatz nur insoweit in unsere Überlegungen einbeziehen werden und können, wie er Anregungen für die von uns in Kapitel 2 identifizierten Optimierungspotenziale liefert.

abgegrenzte Domäne betrachten wollen. Wir brauchen im Gegensatz zu IP keinen generischen Visualisierungsansatz für eine Vielzahl theoretisch denkbarer Modelle (siehe [CzEi00], S. 505–506), sondern können uns auf die für uns relevanten und initial festgelegten Modelle beschränken. Refactoring-Operationen (siehe [CzEi00], S. 507) haben für uns eine geringere Bedeutung, da wir weniger code- und mehr modellorientiert arbeiten wollen[2].

Sobald IP-Plattformen verfügbar werden, sind diese aber in jedem Fall als hochinteressante Plattformen anzusehen, auf denen die CMDSD-Werkzeuge basieren könnten, die wir in Kapitel 5 beschreiben werden.

3.1.3 Concept Programming

Der Ansatz des *Concept Programming* geht auf Arbeiten von Christophe de Dinechin beginnend im Jahr 2000 zurück (siehe [Manc08]). Ziel ist, die mentalen Modelle und Konzepte, die bei der Entwicklung von Software entstehen, möglichst direkt und nachvollziehbar im Code abzubilden. Zur Bewertung verbesserter Konzepte hat Dinechin eine Reihe von Pseudo-Metriken definiert, die auf sublime Probleme im Programmcode hinsichtlich seiner Zugänglichkeit und Verständlichkeit aufmerksam machen (siehe [Dine07], S. 34–40):

1. *Syntactic Noise* untersucht inwieweit Zusammenhänge zwischen erzwungener Form einer Sprache und dem betrachteten Problemraum erkennbar sind – scheinen Konstrukte aus der Luft gegriffen zu sein, wirkt sich dies negativ aus (siehe [Dine07], S. 36).
2. *Semantic Noise* betrachtet die Differenz zwischen intuitiver Betrachtung einer Implementierung bzw. Modellierung und der tatsächlichen Interpretation bzw. Bedeutung derselben – Abweichungen erzeugen „semantischen Lärm“ (siehe [Dine07], S. 37).
3. *Bandwidth* bewertet, inwieweit eine Sprache betrachtete Probleme vollständig abdeckt – eine geringe Abdeckung führt zu Problemen, da erweiterte Konzepte von allen Nutzern der Sprache eingeführt werden müssen (siehe [Dine07], S. 38).

[2]Modell-Refactorings sind natürlich durchaus wichtig, aber abstrakte und fachlich orientierte Modelle benötigen – insbesondere im mittelständischen Umfeld und der dadurch begrenzten Projektgröße – nach unseren Erfahrungen vor allem einfache technische Refactorings (z. B. Umbenennungen). Weiterführende Refactorings sind sicher eine interessante Optimierungsmöglichkeit, wenn sich ein Ansatz etabliert hat und detaillierte Erfahrungswerte in der Verwendung des Ansatzes vorliegen, übersteigen als Thema aber den Rahmen dieser Arbeit.

4. *Signal/Noise Ratio* befasst sich mit der Frage, wie viel Syntax durch eine Modellierungs- oder Codierungssprache erzwungen wird, ohne dass diese einen direkten Problembezug hat – es wird also das Verhältnis zwischen lösungsrelevanten Sprachkonstrukten und erzwungenen Infrastrukturkonstrukten bewertet (siehe [Dine07], S. 39).

Dinechin weist explizit daraufhin, dass diese Metriken subjektiv und sehr schwer messbar sind. Er sieht hier Analogien zur Musik – auch dort sollte Lärm soweit möglich reduziert werden, was aber nicht einfach ist, da Lärm in den Ohren des Einen auch Kunst in den Ohren eines Anderen sein kann (siehe [Dine07], S. 40). Die Überlegungen von Dinechin zur „korrekten“ oder „besseren“ Abstraktion mentaler Konzepte, die einer Implementierung zugrunde liegen, führen ihn zur Implementierung von XLR, einer textbasierten Metasprache, die es erlaubt, nahezu beliebige neue (text-basierte) Konstrukte in die Sprache aufzunehmen und für diese auch die Syntax festzulegen (siehe [Dine09]). Insofern ähnelt dieser Ansatz stark den Ideen des *Intentional Programming* (siehe Abschnitt 3.1.2). Allerdings geht Dinechin aber empirisch dadurch weiter, dass er sich intensiver mit der Frage auseinandersetzt, welche Faktoren für die „konzeptionelle Relevanz“ einer Sprache von Bedeutung sind. Resultat sind die oben kurz skizzierten Pseudo-Metriken. Diese werden wir bei der Definition unseres Vorgehensmodells in Kapitel 5 (zusammen mit weiteren Forschungsergebnissen und empirischen Erfahrungswerten) aufgreifen, um unser integriertes Vorgehensmodell zu detaillieren.

Zudem geht Dinechin auf die Umsetzung von Konzepten in Codierungs- und Modellierungssprachen ein. Im Wesentlichen definiert er die folgenden Schritte (siehe [Dine07], S. 41ff.):

- *Define the problem space*
- *Identify individual concepts*
- *Document concept behaviors*
- *Choose notation for each concept*
- *Select or invent representation*

Auch diese Vorgehensweise werden wir in Kapitel 5 aufgreifen, hier insbesondere in unseren Ausführungen zur Modellbildung (siehe Abschnitt 5.4.3.8 ab Seite 213, aber auch den ab Seite 207 vorgestellten Ansatz in Abschnitt 5.4.3.5.2). Wir formulieren als Vorbereitung:

Konzept 3.6 (Systematische Modellbildung)**.** *Der Vorgang der Modellbildung ist konsequent zu systematisieren, indem zunächst der Kontext des Modells präzise abgegrenzt wird, dann enthaltene Detailkonzepte identifiziert und beschrieben, Notationen zur Darstellung der Konzepte ausgewählt und Repräsentationen festgelegt werden.*

3.1.4 Convergent Architecture (CA)

In [Hube02] liefert Hubert zwei wichtige Beiträge in seiner Weiterentwicklung des Convergent Engineering (siehe dazu Abschnitt 3.1.1). Zunächst definiert er den Metabegriff des *Architekturstils*, um klarer zu beschreiben, wie mehr Struktur in IT-Prozesse gebracht werden kann und wie durch diese zusätzlichen Strukturen Effizienz- und Qualitätsgewinne ermöglicht werden. Dann beschreibt er mit seiner *Convergent Architecture* eine spezifische Ausprägung eines Architekturstils.

Hubert strebt an, die Kommunikation auf allen Ebenen eines IT-Projekts zu verbessern und sieht in Architekturstilen („architectural styles“) ein maßgebliches Mittel dazu ([Hube02], S. 5): *„An architectural style expresses the language and design culture that helps stakeholders at all levels to communicate at a higher, more effective level. [...] The style is the next higher level of design expression. [...] this translates to, for example, the use of accurate terms for component structures and their relationships to express something the architect considers to be of higher value.“* Er sieht einen unmittelbaren Nutzen darin, dass ein Architekturstil es erlaubt, innerhalb einer Organisation auch außerhalb der Entwicklungsabteilung klarer mit den Geschäftseinheiten zu kommunizieren ([Hube02], S. 6): *„Once an organization has agreed on an architectural style as its language of IT architecture, it can move beyond improved communication in the development organization to improved communication between all levels of business.“* Durch Einführung dieser zusätzlichen Ebene differenziert Hubert zwischen drei verschiedenen Entwicklungsebenen ([Hube02], S. 7):

1. Der Entwicklung des Architekturstils.
2. Der Entwicklung einer konkreten Architektur als Instanz eines Architekturstils.
3. Der Entwicklung des eigentlichen Systems.

Er merkt dazu an, dass sich die Qualität von Lösungen insbesondere durch die Wiederholbarkeit der Entwurfsentscheidungen verbessert ([Hube02], S. 8): *„One way the architectural style ensures an increasing level of repeatable quality is by prescribing properties of design.“* Einen weiteren Vorteil sieht er darin, dass sich der Architekturstil einer stetigen Evolution unterzieht, die ihre Energie aus Projekterfahrungen bezieht und dort entdeckte Verbesserungen kontinuierlich in den Stil einfließen lässt, bis dieser selbst durch technische Neuerungen überholt und durch einen neuen Stil abgelöst wird ([Hube02], S. 9–10).

Wir halten fest:

Konzept 3.7 (Architekturstil)**.** *Ein Architekturstil führt eine höhere Kommunikationsebene zwischen Entwicklungs- und Geschäftsabteilungen ein, die die Qualität der Lösungen ebenso wie den Informationsaustausch durch kontinuierliche Verwendung einer konsolidierten und systematisierten Sprache verbessert.*

Besondere Bedeutung bei der Ausprägung von Architekturstilen misst Hubert dabei einer adäquaten Werkzeugunterstützung zu, da es zum einen erst die richtigen Werkzeuge erlauben, Architekturmerkmale in größerem Stil wahrzunehmen ([Hube02], S. 11): *„You cannot see the 90 percent of the design (or lack thereof) of IT systems without tools.“* Zum anderen sind IT-Systeme entwurfsintensive Systeme, so dass passende Modelle und Werkzeuge besonderes Augenmerk verdienen ([Hube02], S. 11): *„The fact that IT systems are design-intensive virtual worlds means that models and tools take on a particular importance.“*

Hubert definiert einen Architekturstil schließlich wie folgt ([Hube02], S. 13): *„An architectural style conveys the principles and the means to most effectively achieve a design vision. [...] An architectural style is a family of architectures related by common principles and attributes.“* Wir leiten daraus die nachfolgende konzeptionelle Vorgehensweise ab:

Konzept 3.8 (Prinzipien- und methodenbasierte Definition)**.** *Ein Architekturstil wird durch die Prinzipien und Methoden beschrieben, die ihm zugrunde liegen und dabei helfen, die mit ihm verbundene Vision zu realisieren.*

Insbesondere sieht Hubert einen großen Nutzen und eine wichtige Aufgabe von Architekturstilen darin, ein ganzheitliches Bild von IT-Systemen zu liefern ([Hube02], S. 13): *„An architectural style [...] is holistic in that it covers the entire software lifecycle, including project design and tool design aspects. It is specific in that it consolidates and integrates the many structural, procedural, and descriptive aspects that have been adressed as separate entities in traditional methodologies.“* Architekturstile gehen also sowohl in die Breite als auch in die Tiefe, um ein Zielthema geschlossen zu behandeln. Wir halten entsprechend folgendes Konzept fest:

Konzept 3.9 (Ganzheitliche Betrachtungsweise)**.** *Architekturstile sind besonders leistungsfähig, da sie eine ganzheitliche Sichtweise einnehmen und organisatorische, technische, methodische und werkzeugbezogene Fragen im Verbund betrachten und lösen.*

Bei der Betrachtung eines Problems sowohl in seiner Breite als auch Tiefe weist Hubert darauf hin, dass man alleine schon aufgrund der Komplexität der damit verbundenen Betrachtungen keinen Anspruch auf Vollständigkeit erheben könne ([Hube02], S. 14): *„More formally, it can be said, that [an IT-architectural style provides a useful set of reasonable alternatives – not all alternatives – and coordinates them to work well together].“* Wir halten fest:

Konzept 3.10 (Geschlossene Menge sinnvoller Alternativen)**.** *Ein Architekturstil sollte eine sinnvolle Menge von Alternativen anbieten. Diese beruhen auf Best Practices aus dem Anwendungsgebiet, müssen aber nicht vollständig sein, sondern sollten die meistverbreiteten Möglichkeiten abdecken können.*

Hubert gibt einen weiteren Hinweis darauf, dass Architekturstile die Komplexität von IT-Architekturen reduzieren, insbesondere wenn sie projektübergreifend und damit wiederholt angewendet werden ([Hube02], S. 15): *„The IT-architectural style explicitly targets the problem of unnecessary complexity in IT architecture. [...] IT-architectural style is most valuable when used as the basis for entire IT infrastructures. Although it can be used for individual projects, the most compelling reason to introduce an IT-architectural style is its ability to support architectural integrity across many projects.“* Dies passt exzellent zu dem von uns identifizierten Optimierungspotenzial der Domänenfokussierung (siehe Abschnitt 2.2.2), da sich mindestens aus Sicht des Dienstleisters eine zusammenhängende Folge von Projekten zu einem gleichartigen Thema ergibt und somit hier durch Verwendung eines einheitlichen Architekturstils Nutzen geschaffen wird. Wir halten demgemäß fest:

Konzept 3.11 (Wiederverwendung)**.** *Architekturstile haben einen besonders hohen Nutzen, wenn sie projektübergreifend wiederverwendet werden. Domänenfokussierende Vorgehensmodelle führen naturgemäß mindestens auf Seiten des IT-Dienstleisters zu einem projektübergreifenden und sich wiederholenden Projektfokus.*

Hubert definiert vier Eigenschaften, die ein Architekturstil aufweisen muss:

1. *Ein architektonisches Metamodell* ([Hube02], S. 16–18): Das Metamodell beschreibt die Prinzipien, die einer Architektur zugrunde liegen. Es liefert die strategische Grundlage für die (Weiter)Entwicklung einer Architektur und fördert so auch die Kommunikation auf allen Ebenen einer Organisation, da es eine strategische Klammer für die operativen Tätigkeiten bereitstellt. Wir haben das Metamodell bereits in Konzept 3.8 ▷ 62 festgehalten.
2. *Ein Entwicklungsmodell für den gesamten Lebenszyklus eines Systems* ([Hube02], S. 18–20): Das Entwicklungsmodell beschreibt, wie die Vision des architektonischen Metamodells umzusetzen ist. Dabei geht es speziell auf die Aspekte ein, die den jeweils behandelten Architekturstil ausmachen. Um dies in der richtigen Breite und Tiefe zu ermöglichen, unterscheidet Hubert drei fundamentale Ebenen ([Hube02], S. 19):
   a) *Entwicklungsstrukturen.* Auf dieser Ebene werden die konkreten Ressourcen beschrieben, die für den Entwurf, die Implementierung und die Auslieferung eines Systems erforderlich sind.

> **Konzept 3.12** (Entwicklungsstrukturen)**.** *Beschreibe, aus welchen Bestandteilen sich ein System zusammensetzt. Dies beinhaltet die Entwurfs-, Implementierungs- und Betriebsphasen.*

b) *Entwicklungsprozess.* Diese Ebene beschreibt den Prozess zur Erzeugung der Entwicklungsstrukturen.

> **Konzept 3.13** (Entwicklungsprozess)**.** *Beschreibe, wie die Bestandteile des Systems zu entwickeln sind.*

c) *IT-Organisationsstruktur.* Mit dieser Ebene wird das Zusammenspiel zwischen Verantwortlichkeiten, Rollen und Personen auf der einen Seite und dem spezifischen Entwicklungsprozess auf der anderen Seite dargestellt.

> **Konzept 3.14** (Organisationsstruktur)**.** *Definiere die Rollen und Verantwortlichkeiten im Rahmen des Entwicklungsprozesses.*

3. *Eine vollständige Entwicklungsumgebung* ([Hube02], S. 20–21): Hubert fordert eine konsequente Werkzeugunterstützung, da die präzise Beschreibung des Architekturstils eine automatische Werkzeugunterstützung ermöglicht. Wie bereits ausgeführt erscheint diese essentiell, da es sich bei IT-Systemen um schwer visualisierbare und hochkomplexe Systeme handelt, so dass maschinelle Unterstützung zur besseren Kommunikation von Architekturen besonders bedeutsam ist.

> **Konzept 3.15** (Werkzeugunterstützung)**.** *Die Komplexität von IT-Lösungen erfordert eine konsequente Werkzeugunterstützung bei dem Entwurf und der Umsetzung eines Systems.*

4. *Technologieprojektionen* ([Hube02], S. 21–24): Hubert sieht ein großes Problem der heutigen IT darin, dass technologische Veränderungen weniger strukturell als vertrieblich motiviert sind – zu oft wird eine Technologie nur durch eine vergleichbare Technologie ersetzt, um neue Verkaufsargumente zu erzeugen und „Neues“ anzubieten. Dies setzt IT-Lösungen einem großen Druck aus, da diese schnell an Aktualität verlieren und entwickelte Architekturen häufig nur für eine oder sehr wenige Systemversionen Bestand haben, bevor sie durch neue Architekturen abgelöst werden. Daher empfiehlt Hubert eine konsequentere modellbasierte Vorgehensweise, die insbesondere möglichst viele technologische Implementierungsdetails automatisiert aus den Modellen generiert[3].

[3] Wir werden diese Grundidee in Abschnitt 3.3 zu modellbasierten Vorgehensmodellen genauer untersuchen.

Konzept 3.16 (Technologieprojektionen)**.** *Technologische Implementierungsdetails sind so weit wie möglich automatisiert aus Modellen zu generieren.*

Hubert weist auf weitere Aspekte hin, die einen Architekturstil beeinflussen: Er benennt dazu *Spezifizität*, *Entropie*, *das Entwurfsparadoxon*, *organische Ordnung* und *Organisationsevolution* ([Hube02], S. 24–29).

Spezifizität sieht er als Mittel zur Steigerung von Abstraktionsebenen im Gegensatz zu den weitverbreiteten eher generischen Ansätzen. Wir werden uns mit diesem Aspekt detaillierter in Abschnitt 3.2 zu domänenfokussierten Vorgehensmodellen beschäftigen. Wir halten aber bereits fest:

Konzept 3.17 (Spezifizität)**.** *Je spezifischer ein betrachtetes Problemgebiet ist, um so abstrakter und ausdrucksstärker können Architekturen und Modelle gestaltet werden.*

Entropie ist die entscheidende Kraft, die ganz ohne expliziten Aufwand auf alle Aspekte eines Systems einwirkt[4]. Hubert sieht eine wesentliche Leistung von Architekturstilen darin, dass sie übergeordnete Kriterien und Muster für eine strukturierte Entwicklung von Systemen liefern und somit entropischen Kräften implizit wie explizit entgegenwirken.

Das Entwurfsparadoxon besteht laut Hubert darin, dass wir besonders stark von Modellen und Methoden abhängig sind, um Unabhängigkeit für unsere Systeme zu erlangen. Unsere Entwürfe müssen sich insbesondere auf die Dinge konzentrieren, die wir flexibel verändern wollen. Dies bedeutet, dass zur Flexibilisierung von Fachthemen diese fachlichen Aspekte wesentlicher Bestandteil unserer Entwürfe und auch Modelle sein müssen. Umgekehrt sollten wir Aspekte, die wir nur implizit verändern wollen (z. B. technologische Implementierungen) nach Möglichkeit nicht zu expliziten Bestandteilen unseres Entwurfs machen, da wir sonst auch die Modelle anpassen müssen, wenn die Grundlagen sich ändern (siehe dazu auch unsere vorausgegangenen Hinweise zu Technologieprojektionen). Wir halten fest:

[4]Salopp formuliert: „Unordnung entsteht aus eigener Kraft.“

Konzept 3.18 (Flexibilisierung von Fachlichkeit)**.** *Die Aspekte eines Systems, die wir bewusst und willentlich verändern wollen, müssen expliziter und zentraler Bestandteil unserer Entwürfe sein. Implizite Veränderungsmöglichkeiten sollten auch nur implizit in den Entwürfen berücksichtigt werden, z. B. durch geeignete Technologieprojektionen.*

Organische Ordnung bezeichnet laut Hubert die Erkenntnis, dass eine rigide und möglichst umfassende Vorplanung von Entwicklungsschritten nur selten zum gewünschten Ergebnis führt, insbesondere in hochgradig veränderlichen Umgebungen. Entsprechend sieht er das Wasserfallmodell als eher ungeeignete Methode zur Abwicklung von Projekten und fordert iterative und inkrementelle Prozesse zur Entwicklung von Lösungen. Wir werden diesen Punkt eingehender in Abschnitt 3.4 zu agilen Vorgehensmodellen betrachten. Dennoch halten wir bereits hier fest:

Konzept 3.19 (Iteratives und inkrementelles Vorgehen)**.** *Iterative und inkrementelle Vorgehensweisen sind besonders geeignet, um in hochveränderlichen Umgebungen zielführende Lösungen zu entwickeln. Insbesondere unterstützen sie auch den kontinuierlichen Wissensaufbau und kommen so der natürlichen Dynamik von Organisationsprozessen sehr nahe.*

Unter organisatorischer Evolution versteht Hubert schließlich, dass Architekturstile auch bei der Organisationsentwicklung helfen sollten. Er ist unsicher bzg. ihrer tatsächlichen Effektivität, da ihm Ansatzpunkte fehlen, um aufzuzeigen, wie Kreativität anstelle von Reaktivität durch Architekturstile gefördert werden kann. Er schließt daher mit dem Hinweis, dass der Eigner eines Architekturstils diesen zumindest dahingegend ausrichten kann, dass der Stil Kreativität nicht im Wege steht. Wir sehen die Förderung von Kreativität in Organisationen basierend auf unseren Erfahrungswerten als einen sehr wesentlichen Wert an, den es zu unterstützen gilt – denn so können die Alleinstellungsmerkmale und Stärken von Organisationen gefördert und betont werden und dienen letztlich häufig dazu, Systeme in Hinblick auf den tatsächlich wichtigen Funktionsumfang zu konfektionieren. Daher fordern wir:

Konzept 3.20 (Förderung kreativer Organisationen)**.** *Moderne Vorgehensmodelle müssen die Kreativität von Organisationen fördern, um so die Stärken einer Organisation explizit zu machen und weiterzuentwickeln.*

Im weiteren Verlauf stellt [Hube02] einen konkreten von ihm entwickelten Architekturstil in Form der *Convergent Architecture* vor. Dieser basiert auf den von uns bereits vorgestellten Prinzipien des *Convergent Engineering* (siehe Abschnitt 3.1.1) und setzt insbesondere auf UML-basierte Modellierung, eine an den Rational Unified Process angelehnte Vorgehensweise sowie auf die Generierung von Systemen basierend auf der Java-Enterprise-Plattform. Wir wollen diese Ausführungen hier nicht weiter betrachten, da sie uns zu weit von unserem mittelstandsorientierten Fokus entfernen würden[5].

Zusammenfassend stellen wir fest, dass Hubert uns wichtige Anhaltspunkte für die Definition unseres eigenen Lösungsansatzes geliefert hat. Neben einer Reihe von Detailkonzepten hat er mit der Definition von Architekturstilen ein Muster zur Definition spezifischer Vorgehensmodelle geliefert, auf das wir nachfolgend zurückgreifen werden. Zudem hat er uns in der Bedeutung der von uns in Kapitel 2 identifizierten Optimierungspotenziale bestärkt, indem er auf die Bedeutsamkeit von Modellen, der Spezifizität von Entwürfen und einer iterativen und inkrementellen Vorgehensweise hingewiesen hat.

3.1.5 Fluent Interfaces

Mit dem Begriff *Fluent Interfaces* beschreibt Martin Fowler eine bestimmte Art und Weise, APIs und Schnittstellen zu programmieren (siehe [Fowl05a]). Grundidee des Ansatzes ist es, Programmcode so formulieren zu können, dass er die Intention des Schreibers möglichst direkt ausdrückt und auch Nicht-IT-Experten erlaubt, die Intention des Codes zu verstehen. Dies geschieht durch zweierlei Maßnahmen: Zum einen werden Schnittstellen so konzipiert, dass Methodenaufrufe leicht mitein-

[5] Damit greifen wir den späteren Ergebnissen des Kapitels vor, da wir sehen werden, dass die spezifischen Ansatzpunkte von Hubert keinen nachhaltigen Erfolg für typische mittelständische Projekte bringen werden. Wir verweisen dazu insbesondere auf unsere Ergebnisse insbesondere in den Abschnitten 3.3.1 und 3.4.2.

ander verkettet werden können[6] und zum anderen werden die Methodenaufrufe so entworfen, dass sich durch ihre sinnvolle Verkettung Formulierungen ergeben, die der natürlichen Sprache möglichst nahekommen. Folgendes Beispiel (aus [Marg08]) zeigt ein schönes Beispiel für ein Fluent Interface, bei dem man trotz Unkenntnis der Programmiersprache und der genauen Semantik der verwendeten Methoden sofort ein intuitives Verständnis von der Bedeutung des Codes erhält:

```
Repeat.Call<string>(Save).WithParameters("myfile.txt").UntilSucceeds.Start(10);
Repeat.Call(Ping).PauseBetweenCalls(2000).Start(100);
Repeat.Call(Open).InBackgroundThread.At(ThreadPriority.Lowest).
        OnSuccessCall(Opened).OnExceptionCall(Failed).Start();
```

Mittlerweile finden sich viele Programmbibliotheken und Frameworks, die diese Idee aufgegriffen haben (siehe z. B. [Cawl05]).

Hauptkritikpunkt an Fluent Interfaces ist, dass ihr Entwurf sehr aufwändig ist, da die orthogonale Abbildung von Konzepten auf verkettete Methodenaufrufe durch die Notwendigkeit, intern einen sinnvollen Kontext beizubehalten, eine komplexe und anspruchsvolle Aufgabe ist (siehe z. B. [Misc07]).

Für unsere Betrachtungen stellen Fluent Interfaces also lediglich eine Form zur Umsetzung domänenspezifischer Sprachen (siehe Abschnitt 3.2.8) dar, die sich aber eher für sehr technische und IT-nahe Domänen eignet, da das Fluent Interface immer in den Kontext der jeweiligen Programmiersprache eingebettet ist und somit auch den potenziellen „Ballast“ der Programmiersprache mit sich führen muss (siehe dazu die Pseudo-Metriken von Dinechin aus Abschnitt 3.1.3).

3.1.6 Language Workbenches

Der Begriff *Language Workbench* wurde von Martin Fowler geprägt, um die Werkzeuge zu charakterisieren, die das *Language Oriented Programming* unterstützen (siehe [Fowl06b] für den ursprünglichen Artikel zu Language Workbenches sowie unsere Ausführungen in Abschnitt 3.2.3 ab Seite 79 zu Language Oriented Programming). Fowler sieht hier sehr großes Potenzial für die Zukunft und erläutert in seinem Artikel neben der reinen Begriffsbildung auch die zugrunde liegende Historie

[6] Üblicherweise dadurch, dass *alle* Methoden Rückgabewerte haben, auch wenn sie eigentlich nur Werte setzen. In diesem Fall würden sie z. B. eine Referenz auf die Instanz, zu der sie gehören, zurückliefern – eine Strategie, die sich an Smalltalk-Mechanismen anlehnt.

sowie Chancen und Risiken des Ansatzes. Wir nehmen diese Ausführung zusammen mit den bereits diskutierten Hinweisen in anderen Vorgehensmodellen als Indikator dafür, eine stringente Werkzeugunterstützung als essentiellen Bestandteil unseres Ansatzes zu definieren.

Die grundsätzliche Struktur, die Fowler für Language Workbenches beschreibt, ähnelt sehr stark den Ideen des Intentional Programmings bzgl. der Repräsentation und Manipulation von Quelltexten als Graphenstrukturen (siehe auch Abschnitt 3.1.2).

Fowler weist insbesondere auf eine Reihe von Problemen und Risiken bei der Verwendung komplexerer DSLs hin, die den Nutzen des breiteren Einsatzes von DSLs in Frage stellen. Wir stellen diese Nachteile – *symbolic integration*, *language cacophony*, *symbolic barrier*, *beschränkte Syntax bei internen DSLs*, *lay programmer*, *COBOL inference*, *Entwicklungskosten für Werkzeuge* sowie *vendor lock-in* – erst in Abschnitt 7.1.8 ab Seite 299 vor, da wir dort auch gleich unseren Ansatz mit diesen Problemen abgleichen und kritisch dazu Stellung nehmen.

Insgesamt kommt Fowler trotz der von ihm angeführten Nachteile zu dem Schluss, dass Language Workbenches aufgrund ihrer Vorteile (der drastischen Erhöhung der Produktivität in der Entwicklung sowie der Möglichkeit zur engeren Zusammenarbeit zwischen Domänen- und IT-Experten) eine der interessantesten gegenwärtigen Entwicklungen sind. Wir werden dieses Konzept daher für unseren Lösungsansatz in Teil II der Arbeit wieder aufgreifen und verweisen dazu bereits auf das formulierte Konzept 3.15 ▷ 64.

3.1.7 Weitere Ansätze

Neben den zuvor geschilderten spezifischen Vorgehensmodellen und Methoden, die konkrete Verfahren oder Handlungsanweisungen in den Mittelpunkt stellen, gibt es eine Reihe allgemeinerer Überlegungen dazu, wie Ansätze prinzipiell strukturiert sein müssen, um produktiver Software entwickeln zu können. Hier zeichnen sich zwei Denkrichtungen ab:

- Ansätze, die sich mit einer Veränderung der Arbeits*weise* von Programmierern und Entwicklern beschäftigen und
- Ansätze, die sich mit einer Veränderung der Arbeits*werkzeuge* von Programmierern und Entwicklern beschäftigen.

Außerhalb von Projektmanagementthemen beschäftigt sich erstaunlicherweise sehr wenig Literatur mit den psychologischen Aspekten von Software-Entwicklung. Ins-

besondere sind hier die folgenden Arbeiten für unsere Ziele von Relevanz:

- die Arbeiten von Gerald M. Weinberg in [Wein98]. Zu beachten ist, dass die Originalausgabe dieser Neuauflage weitgehend unverändert 1971 publiziert wurde und aufgrund ihrer Diskussion der Zusammenhänge zwischen der menschlichen Psyche und der Tätigkeiten von Programmierern weiterhin äußerst aktuell ist, da sie sich mit den grundlegenden Prinzipien der Erstellung leicht verständlicher und zugänglicher Softwaresysteme beschäftigt. Insbesondere befasst sich Weinberg in den Kapiteln 11 und 12 mit der Frage, wie Programmiersprachen aus psychologischer und linguistischer Sicht konstruiert sein sollten, um Entwickler optimal zu unterstützen[7]. Weinberg baut dabei u. a. auf Arbeiten von [Zipf49, Mill56, Wein65, Min68, Samm69] auf, die sich als exzellente Startpunkte für über diese Arbeit hinausgehende Forschungen zur psychologisch-motivierten Analyse der Struktur von Entwicklungsumgebungen anbietet. Weinbergs Prinzipien für den Bau von menschlich gut zugänglichen Programmiersprachen werden wir in der Definition der Prinzipien für unseren CMDSD-Ansatz wieder aufgreifen, da sie von direkter Relevanz sind (siehe Abschnitt 5.4.2 ab Seite 172). Dort finden sich auch ausführliche Erläuterungen der Prinzipien im Kontext unseres Ansatzes. Wir halten also zunächst fest:

  Konzept 3.21 (Psychologisch vorteilhafte Strukturbildung)**.** *Die Prinzipien von Uniformität, Kompaktheit, Lokalität und Linearität sowie von Tradition und Innovation liefern nach [Wein98] durch psychologische und linguistische Studien untermauerte Prinzipien für den Entwurf zugänglicherer Programmiersysteme.*

- die Ausführungen von James Martin aus [Mart81], der bereits sehr früh die folgende Auffassung vertreten hat: *„This book is dedicated to the view that a higher level of automation is needed in developing data processing applications.“* Der Zeitpunkt des Erscheinens lässt bereits erahnen, dass viele spezifische Aspekte des Buches heute nicht mehr relevant sind (so beschäftigt er sich z. B. intensiv mit der Frage, wie Datenbankabfragesprachen das Generieren von Reports drastisch vereinfachen können im Vergleich zu klassischer

[7] Es sei uns folgende Anekdote erlaubt: Als Weinberg auf die Unterschiede zwischen realen Sprachen und Programmiersprachen eingeht und anmerkt, dass Programmiersprachen im Gegensatz zu realen Sprachen aus einer Reihe vernünftiger Gründe nicht verbalisiert (gesprochen) werden, kam uns einen Moment lang in den Sinn, dass unser Ziel sich zu realen Sprachen ähnlich verhält wie Höhlenmalerei – denn auch Höhlenmalerei beschäftigt sich häufig mit genau einer Domäne (Jagd, Tiere, Fruchtbarkeit, Religion) und verwendet sehr einprägsame grafische Modelle. Damit sei die anekdotische Abschweifung beendet.

COBOL-Programmierung). Wesentlich interessanter und auch heute noch relevant sind allerdings seine Überlegungen zu der Frage, ob Entwicklungsarbeiten vorspezifiziert oder anwendergetrieben sein sollten. Seine Thesen dazu (ab S. 54 im oben erwähnten Buch) weisen viele Ideen und Denkansätze auf, die lange vor dem Stichwort „Agilität“ bereits genau die Grundideen von agilen Vorgehensmodellen und Ansätzen formulierten. Auf S. 55 trifft Martin im Grunde die gleiche Unterscheidung zwischen präskriptiven bzw. planbasierten Vorgehensmodellen und (wie er es nennt) kundengetriebenen Vorgehensmodellen („user-driven computing“) , die wir auch in [BoTu03b] finden und den nachfolgenden Abschnitten 3.4.2 und 3.4.3 aufgreifen werden, um präskriptive bzw. planbasierte Vorgehensmodelle von agilen Vorgehensmodellen zu differenzieren. Wir wollen die wichtigsten Punkte von Martin zitieren (Auszug von S. 55 aus [Mart81], unter der Überschrift *User-Driven Computing*), die wir später aufgreifen werden, um sowohl unseren CMDSD-Ansatz (Kapitel 5) als auch unseren CDPM-Ansatz (Kapitel 6) zu formulieren:

- *Users do not know in detail what they want until they use a version of it, and then they modify it quickly and often frequently. [. . .]*
- *Users may create their own applications, but more often with an analyst who does this in cooperation with them. [. . .]*
- *Applications are created with a generator [. . .] more quickly than the time to write specifications.*

  [. . .]
- *Maintenance is continuous. Incremental changes are made constantly to the applications by the users or the analysts who assists them.*
- *The system is self-documenting, or interactive documentation is created when the application is created.*

  [. . .]

Letztlich propagiert Martin also bereits die Idee eines modellgetriebenen (und damit zumindest teilweise selbstdokumentierenden) generierenden und iterativ evolutionären Ansatzes. Diese Idee werden wir mit unserem CP-Ansatz aufgreifen und durch Verfahren und Methoden unterfüttern, die es zu Martins Zeit noch nicht (in hinreichend praxisrelevanter Form) gab und die erst durch die heutige Leistungsfähigkeit von Computersystemen praktisch bedeutsam werden und formulieren daher:

> **Konzept 3.22** (Kooperative, inkrementelle Vorgehensmodelle)**.** *Vorgehensmodelle sollten eine kooperative und inkrementelle Vorgehensweise unterstützen, da auf diese Weise der Prozess des Wissensaufbaus zwischen Fachexperten und IT-Experten optimal unterstützt wird.*

Interessant sind auch neuere Experimente zur grafischen Visualisierung von Sachverhalten: Aus der jüngeren Vergangenheit sticht unserer Ansicht nach insbesondere *Subtext* von Jonathan Edwards hervor (siehe [Edwa09a]). Edwards schlägt eine neue Form der grafischen Visualisierung von Programmlogik vor, die sich einer zweidimensionalen Darstellung bedient (siehe [Edwa07]): Logik wird zweidimensional modelliert und dargestellt, in der Horizontalen werden basierend auf Boolescher Algebra über Entscheidungstabellen die Alternativen des Programmablaufs visualisiert. Partitionierte Spalten veranschaulichen die Überdeckung aller Entscheidungsfälle äußerst effektiv. In der Vertikalen wird die Dynamik des Programms, also das Ablaufverhalten, dargestellt durch zeilenweise segmentierte Funktionsgraphen, die über Spannbäume geordnet werden (siehe [Edwa07b] für eine sehr anschauliche Präsentation dieser Vorgehensweise). Interessant daran ist der hohe Grad an visueller wie kontextsensitiver Unterstützung, die der Anwender bei der Erstellung der Ablauflogik erfährt. In [Edwa08] überträgt Edwards diese Ideen auch auf die Modellierung grafischer Oberflächen am Beispiel von Webformularen. Edwards vertritt allgemein die These, dass die Zeit reif ist, um die historisch auf Basis der sequentiellen Natur von Texten gewachsene übliche Repräsentation von Programmen (als Quelltexte) genauestens zu prüfen und ggf. zu überarbeiten, da die Rechenleistung heutiger Rechner wesentlich ausgefeiltere IDE-Konzepte[8] ermöglicht, die einen natürlicheren Zugang zu komplexen Algorithmen und ihrer Beschreibung erlauben. Wir sehen darin unterstützende Argumente für unseren Anspruch, eine modellbasierte Automatisierung von domänenspezifischen Problemen erreichen zu können, da Edwards für eine sehr klassische Domäne illustriert, wie diese wesentlich ansprechender, intuitiver und informationshaltiger visualisiert werden kann. Zu Bedenken ist hier natürlich wieder der zu investierende Aufwand – hier unterstützt uns aber die zunehmende Verfügbarkeit sehr leistungsfähiger Open-Source-Lösungen, deren Existenz wir in Abschnitt 5.4.2.6 auf Seite 185 aufgreifen werden.

Wassermann hat sich ebenfalls mit grundlegenden Konzepten beschäftigt, die bei der Erstellung von Software immer wieder Anwendung finden und dabei unterstützen, den Software-Erstellungsprozess leistungsfähiger zu gestalten (siehe [Wass96]). Er

[8] IDE steht für „Integrated Development Environment“.

identifiziert die folgenden grundlegenden Konzepte:

- *Abstraktion* ([Wass96], S. 24). Abstraktion findet sich ausgeprägt im Prinzip des *Information Hidings*. Abstraktion zielt darauf ab, Entwicklern oder Modellierern den Fokus auf die richtige Detaillierungsebene zu gewähren, indem ablenkende Details verborgen werden. Es ähnelt dem bereits von uns identifizierten Konzept 3.17 ▷ 65, wobei Spezifizität ein Mittel zur Abstraktion ist. Daher führen wir auch dieses Konzept auf:

  Konzept 3.23 (Abstraktion)**.** *Wähle für die Modellierung von Lösungen die Abstraktionsebene, die dem Problemraum am Natürlichsten entspricht. Verberge alle anderen Details.*

- *Analyse- und Designmethoden und -notationen* ([Wass96], S. 24–25). Notationen definieren den Werkzeugkasten, dessen man sich bei der Modellierung bedient. Methoden beschreiben die Anwendung dieser Werkzeuge. Beide zusammen sind ein unabdingbares Paar für den Entwurf moderner und komplexer Lösungen. Wir halten fest:

  Konzept 3.24 (Strukturierte Werkzeugnutzung)**.** *Der Entwurf komplexer Systeme erfordert einen Werkzeugkasten mit konkreten Werkzeugen, Modellen und anderen Hilfsmitteln. Eine Methodik muss die korrekte Anwendung des Werkzeugkastens beschreiben.*

- *Prototypisierung von Nutzerschnittstellen* ([Wass96], S. 25–26). Die Allgegenwärtigkeit von Software-Lösungen bedingt eine ebensolche Allgegenwärtigkeit von Nutzerschnittstellen aller Art. Somit wird der Entwurf angemessener Nutzerschnittstellen immer bedeutsamer. Erfahrungswerte zeigen, dass sich Nutzerschnittstellen als einer der kritischen Erfolgsfaktoren von Lösungen am Besten dadurch konzipieren lassen, dass sie durch gemeinsame Prototypisierung und Evaluierung mit den späteren Anwendern entworfen werden. Wassermann weist warnend darauf hin, dass Prototypisierung in sehr großen Projekten als Entwurfsmethode gedacht ist, sich aber nicht als Abkürzung zum Rapid System Deployment verstehen sollte. Hieraus ergeben sich für unseren Kontext aber keine Konsequenzen, da das mittelständische Umfeld wie bereits dargestellt von mittleren bis kleinen Projekten geprägt ist. Wir halten fest:

  Konzept 3.25 (Rapid Prototyping von Nutzerschnittstellen)**.** *Nutzerschnittstellen sind einer der zentralen Erfolgsfaktoren von Software-Lösungen. Erfahrungswerte zeigen, dass diese am Erfolgreichsten durch Rapid Prototyping gestaltet werden können.*

- *Modularität und Architektur* ([Wass96], S. 26–27). Architektur ist ein bedeutsamer Erfolgsfaktor, um ein System in der erwarteten Qualität zu realisieren. Wassermann gibt verschiedene hinlänglich bekannte Vorgehensweisen zum Aufbau einer wirkungsvollen Architektur an, seine Punkte sind aber bereits durch das von uns identifizierte Konzept 3.7 ▷ 61 abgedeckt.
- *Lebenszyklus und Prozess* ([Wass96], S. 27–28). Jedes Projekt benötigt einen definierten Entwicklungsprozess. Dieser darf durchaus sehr pragmatisch sein und seine Angemessenheit sollte sich immer aus der betrachteten Domäne ergeben. Wir werden dies bei der Analyse von Vorgehensmodellen für die Entwicklung von Software in Abschnitt 3.4 berücksichtigen und formulieren:

  Konzept 3.26 (Angemessenheit des Entwicklungsprozesses)**.** *Entwicklung muss nach einem definierten Prozess erfolgen. Der Prozess sollte sich maßgeblich an der betrachteten Domäne orientieren.*

- *Wiederverwendung* ([Wass96], S. 28–29). Wiederverwendbarkeit kann Kosten sparen, insofern ist sie in einem sinnvollen Maß anzustreben. Diesen Punkt haben wir bereits in Konzept 3.11 ▷ 63 berücksichtigt.
- *Metriken* ([Wass96], S. 29). Metriken sind essentiell für die Verbesserung von Prozessen. Insofern sollten Metriken erhoben und ausgewertet werden, die eine angemessene Prozessqualität sichern können. Anhand der sich verändernden Metriken sollten Maßnahmen zur Verbesserung der Entwicklungsprozesse bewertet werden.

  Konzept 3.27 (Metrische Bewertung)**.** *Prozesse sollten in ihrer Leistungsfähigkeit anhand klar definierter Metriken bewertet werden.*

- *Werkzeuge und integrierte Entwicklungsumgebung* ([Wass96], S. 29–30). Hier greift Wassermann die bereits mehrfache angeführte Bedeutung einer integrierten Werkzeugunterstützung für Entwurf und Umsetzung auf. Wir haben diesen Punkt bereits in Konzept 3.15 ▷ 64 berücksichtigt.

Weiterhin beschäftigen sich [KMAu04] mit der Frage, mit welchen Hürden Einsteiger in Entwicklungsumgebungen zu kämpfen haben. Sie identifizieren eine Reihe von Barrieren, die das Erlernen von Systemen teilweise deutlich erschweren und schlagen anschließend eine Metapher vor, um diese Hürde zu überwinden. Die Hürden sind:

- *Designhürden*, die das Erkennen des eigentlichen Problems und der Lösungsmöglichkeiten behindern,

- *Auswahlhürden*, die das Auffinden der möglichen und sinnvollen Lösungsbestandteile erschweren,
- *Koordinationshürden*, die das Zusammenfügen einzelner Lösungsbausteine zu komplexen Gesamtlösungen erschweren,
- *Anwendungshürden*, die die Anwendung bestimmter Komponenten durch einen obskuren oder komplexen Zugang erschweren,
- *Verständnishürden*, die Probleme im Verständnis der Ausführungsergebnisse eines Lösungsansatzes beschreiben und
- *Informationshürden*, die Schwierigkeiten im Auf- und Wiederfinden von wichtigen Informationen zur Lösung eines Problems zusammenfassen.

Die Emfehlungen, die sie aussprechen, haben wir bereits mit den Konzepten 3.9 ▷ 62, 3.10 ▷ 62, 3.17 ▷ 65, 3.23 ▷ 73 und 3.24 ▷ 73 berücksichtigt.

Der Vollständigkeit halber wollen wir auch den Begriff des *Semantic-oriented programming* (siehe [Wiki08i]) erwähnen, der sich von den Ideen und Ansätzen aber mit Intentional Programming (siehe Abschnitt 3.1.2) und Concept Programming (siehe Abschnitt 3.1.3) sowie Language Workbenches (siehe Abschnitt 3.1.6) überschneidet, so dass wir ihn nicht weiter vertiefen wollen. Es existiert auch noch der Oberbegriff des *Visual Programming* (siehe [Wiki09d]), der aber keine neuen Erkenntnisse liefert außer bezüglich der reinen Oberbegriffsbildung für alle Programmiersprachen, die auf Basis grafischer Manipulationen und Operationen die Beschreibung von Programmverhalten erlauben[9]. Allerdings liefert [Wiki09d] einen guten Überblick über existierende grafische Programmiersprachen, die wiederum als Inspiration für die Realisierung eigener domänenspezifischer Ansätze dienen könnten.

3.2 Domänenfokussierte Verfahren zur Softwareentwicklung

Domänengetriebene Verfahren der Softwareentwicklung konzentrieren sich auf eingeschränkte Problemkreise und versuchen, die Entwicklungsprozess für Lösungen in diesem Bereich durch spezialisierte Entwicklungsumgebungen und Programmiersprachen zu vereinfachen. Zu den Verfahren dieses Typs gehört das Rapid Application Development (RAD, Abschnitt 3.2.4) ebenso wie Architectured Rapid Application

[9] Sie enthalten somit z. B. grafische Modellierungssprachen, die sich in ausführbare Programme übersetzen lassen.

Development (ARAD, Abschnitt 3.2.5), Computer Aided Software Engineering (CASE, Abschnitt 3.2.2) sowie Software Produktlinien (Abschnitt 3.2.6) als ganzheitlichem Ansatz für entwicklungsumgebungsgetriebene Vorgehensweisen. Programmiersprachen der vierten Generation (4GL, Abschnitt 3.2.1), Language Oriented Programming (LOP, Abschnitt 3.2.3), Domain Driven Design (DDD, Abschnitt 3.2.7) sowie Domain Specific Languages (DSL, Abschnitt 3.2.8) gehen dagegen von einer programmiersprachenfokussierten Sichtweise aus, bei der Konzepte der jeweiligen Programmiersprachen äußerst domänenorientiert ausgerichtet sind, so dass auch IT-unerfahrene Personen mit diesen Sprachen arbeiten können.

3.2.1 Programmiersprachen der vierten Generation (4GL)

Programmiersprachen der vierten Generation wurden begrifflich von [Mart81] eingeführt. James Martin versteht darunter nicht-prozedurale Programmiersprachen zur hoch-abstrakten Spezifikation von Lösungen. In den 80er Jahren wurde der Begriff auch sehr intensiv im Marketing von IT-Unternehmen eingesetzt, um RAD-Systeme (siehe Abschnitt 3.2.4) zu vermarkten (siehe auch [Wiki08j]). Gemeinsames Ziel aller 4GL-Sprachen ist es, klassische IT-Probleme mit wesentlich weniger Code durch höherwertige Abstraktionen zu lösen, weshalb sie häufig auch als Untermenge von domänenspezifischen Sprachen (siehe Abschnitt 3.2.8 bzw. [HeMe02]) angesehen werden.

Häufigste Anwendungsbereiche für 4GL-Sprachen sind datenbanknahe Werkzeuge wie Berichts- oder Eingabeformulargeneratoren bzw. Werkzeuge, die Informationen aus CASE-Systemen (siehe Abschnitt 3.2.2) übernehmen, um daraus komplexere Anwendungen zu generieren. Erfahrungswerte mit 4GL-Systemen weisen auf deutliche Effizienzsteigerungen hin ähnlich zu denen, die wir für Concept Programming (Abschnitt 3.1.3) identifiziert haben und noch für domänenspezifische Sprachen (Abschnitt 3.2.8) und Language Oriented Programming (Abschnitt 3.2.3) identifizieren werden.

4GL-Sprachen haben sich letztlich nur in Nischen durchgesetzt. Erfahrungswerte weisen auf folgende Gründe für das prinzipielle Scheitern des Ansatzes hin:

- Viele 4GL-Sprachen waren als „general purpose“ Sprachen konzipiert, die für Probleme aller Art geeignet sein sollten.
- Die Komplexität der Wartung und Weiterentwicklung einer solch umfangreichen Lösung hat die meisten 4GL-Anbieter (insbesondere Unternehmen, die

dem Mittelstand zuzuordnen sind) überfordert, so dass nach einer initialen Phase die Weiterentwicklung und Wartung des Systems zunehmend kostspieliger wurde und viele 4GL-Systeme aufgrund wachsender Stabilitäts- und Leistungsprobleme zunehmend den Erwartungen nicht mehr gerecht werden konnten (siehe auch [Ward94, Dmit06]).

- Zudem waren die meisten dieser Systeme nicht formal spezifiziert sondern „pragmatisch“ entworfen, was zu allerlei Detailproblemen in ihrer Verwendung führte.

Erfahrungswerte zeigen aber auch, dass die Beschränkung auf eine begrenzte Domäne durchaus zur Etablierung erfolgreicher 4GL-Systeme führte. Wir sehen darin sowohl unsere Forderung nach Domänenfokussierung (siehe Konzept 3.36 ▷ 89) als auch in unserer Forderung nach Automatisierung bestätigt, da es viele 4GL-Beispiele gibt, die zeigen, dass große Produktivitätsgewinne möglich sind. Um dem Problem der unzureichend strukturierten Entwicklungsweise von 4GL-Systemen Rechnung zu tragen, werden wir in unserem Lösungsansatz entsprechende Vorschläge für ein integriertes Vorgehensmodell machen (siehe Kapitel 5).

3.2.2 Computer Aided Software Engineering (CASE)

Computer Aided Software Engineering (CASE) charakterisiert den intensiven Einsatz von Werkzeugen zur Umsetzung von Software-Lösungen (siehe [Wiki09e]). CASE-Werkzeuge betrachten entsprechend weite oder alle Teile des Software-Erstellungsprozesses von der Analyse über das Design bis zur Umsetzung und Wartung, je nach Werkzeug aber in sehr unterschiedlicher Detaillierungstiefe. CASE-Werkzeuge hatten ihren stärksten Marketeinfluss zwischen Anfang der 80er und 90er Jahre des letzten Jahrhunderts (siehe [Wiki08k]). Mitterweile sind sie weitgehend durch speziellere Ansätze wie domänenspezifische Sprachen (siehe Abschnitt 3.2.8) oder allgemeinere Ansätze auf Basis von MDA und UML (siehe Abschnitt 3.3.1) ersetzt worden.

CASE-Werkzeuge sind technisch eng mit Programmiersprachen der vierten Generation (siehe Abschnitt 3.2.1) verwandt, die ebenso auf einer höheren Abstraktionsebene die Implementierung komplexer technischer Architekturen ermöglichen sollen. Zudem werden vielfältige Modelle verwendet, um Lösungen zu beschreiben (z. B. Datenfluss- oder ER-Diagrame, siehe wiederum [Wiki08k]).

Als problematisch erwiesen sich die fehlende Standardisierung der verwendeten Notationen und Technologien sowie ihre schleppende Weiterentwicklung aufgrund der

hohen technischen Komplexität durch den sehr generischen Ansatz (siehe [Schm06]).

Ein wesentlicher Kritikpunkt an CASE-Werkzeugen hat sich auch in Erfahrungen mit dem *Convergent Engineering* (siehe Abschnitt 3.1.1 bzw. [Tayl95], S. 71) gezeigt – CASE-Tools arbeiten häufig zu datenzentriert:

- Klassen bzw. Entitäten werden häufig über die Variablen definiert, die sie enthalten.
- Für diese Vorgehensweise werden vielfältige Hilfsmittel bereitgestellt (z. B. automatisierte Erzeugen von *gettern* und *settern* für die Variablen).
- Dies verführt dazu, Objekte datenzentriert zu betrachten und den Verhaltensaspekt in einer Art und Weise zu externalisieren, die nicht mit dem Philosophien objektorientierten Designs übereinstimmt (es werden äußerst passive Objekte definiert, die sich im Wesentlichen über ihren Zustand definieren; zudem verstößt die Fokussierung auf die Modellierung des internen Zustands gegen das Prinzip der Kapselung).
- Dies wiederum erschwert *Domain Driven Design* (siehe Abschnitt 3.2.7).

Des Weiteren offenbarte sich, dass CASE-Werkzeuge überwiegend zu generisch entworfen waren und daher für spezifische Domänen eine zu geringe Ausdrucksstärke besaßen (siehe [Schm06] sowie unsere detaillierten Ausführungen in Abschnitt 3.3.6). Dies erzeugte ein zusätzliches operatives Problem dadurch, dass die parallele und kollaborative Arbeit mit CASE-Werkzeugen häufig nur schwer möglich war, da es keine klare Trennung von themenrelevanten Artefakten gab und so häufig nur eine sehr begrenzte Zahl von Entwicklern gleichzeitig mit einem Werkzeug arbeiten konnten.

Konzept 3.28 (Nebenläufigkeit)**.** *Erlaube nebenläufiges Arbeiten, um die Kollaboration innerhalb eines Teams zu vereinfachen. Dazu müssen insbesondere in den zugrunde liegenden Werkzeugen auch entsprechende konzeptionelle Vorarbeiten erfolgen. Berücksichtige, dass aufgrund der typischen mittelständischen Projektgröße nebenläufiges Arbeiten nur in beschränktem Umfang möglich sein muss, da die Entwicklungsteams üblicherweise sehr klein sein werden.*

3.2.3 Language Oriented Programming (LOP)

Der Begriff *Language Oriented Programming* geht auf Martin Ward zurück, der in [Ward94] die Grundlagen für die Methoden und Technologien legt, die sich aktuell unter dem Begriff *Domain Specific Languages* bündeln (siehe den nachfolgenden Abschnitt 3.2.8).

Ward schlägt zur Lösung komplexer Aufgaben ein zweistufiges Vorgehen vor:

1. Entwickle eine möglichst abstrakte Sprache zur Beschreibung des speziellen Problems und seiner Lösung.
2. Implementiere die abstrakte Sprache mit einer Sprache niedriger Abstraktion, um die Lösung ausführbar zu machen.

Er sieht darin einen vielversprechenden Ansatz zu einer Reihe von Problemen in der Software-Entwicklung, die wir ebenfalls ausführlich in Kapitel 2 thematisiert haben: Hohe Wartungskosten und daraus resultierend zu geringe Budgets für die Neu- und Weiterentwicklung von Systemen, nur geringes Domänenwissen bei den Entwicklern, schnell wechselnde Anforderungen sowie Kommunikationsprobleme sowohl in der Anforderungsdefinition also auch in der Umsetzung von Systemen. Nicht zuletzt deswegen, weil auch die zukünftigen Anwender nur selten in der Lage sind, eine vollständige Spezifikation des zu realisierenden Systems zu liefern. Diesen Problemen will er durch die Definition einer domänenbezogenen Sprache begegnen, die es Anwendern erleichtert, in der Problemdomäne zu kommunizieren und Programmierern erlaubt, Domänenwissen abstrakt auszudrücken.

Seine Erfahrungen zeigen Folgendes:

- Die Einführung einer domänenspezifischen Abstraktion führt zu einer sauberen Trennung von fachlichen und technischen Verantwortlichkeiten, was insbesondere die Produktevolution fördert.
- Die Beschränkung auf eine Domäne ermöglicht es tatsächlich, gleichzeitig ausdrucksstarke und abstrakte Sprachen zu entwickeln.
- Die hohe Abstraktion bringt deutliche Produktivitätsgewinne, da einerseits Algorithmen sehr kompakt formuliert und andererseits technische (und insbesondere wiederkehrende) Detailprobleme unter einer Abstraktionsschicht verborgen werden können.
- Die Portierbarkeit erhöht sich drastisch. Ward sieht die Ausführung von domänenspezifischen Sprachen durch vorgeschaltete Transformatoren, die Sprache

höherer Abstraktionsebenen in Sprachen niedrigerer Abstraktionsebenen übersetzen. Um ein solches Gesamtsystem zu portieren, muss lediglich die niedrigste Abstraktionssprache für die Transformatoren und das Laufzeitsystem portiert werden – die höherwertigen Abstraktionen können unverändert erhalten bleiben. Wir halten fest:

> **Konzept 3.29** (Portierbarkeit durch Abstraktion)**.** *Achte darauf, dass sich domänenspezifische Sprachen durch Veränderung der zugrunde liegenden Transformationen auf neue Technologien portieren lassen.*

- Anwender werden in die Lage versetzt, Programmerweiterungen aufgrund der hochabstrakten Domänensprachen vorzunehmen, da diese eben aufgrund der hohen Abstraktion sehr kompakt und damit leicht zu erlernen sind.
- Mit dem *Chief Programmer's Approach* (siehe [MaBu87]) lassen sich eindrucksvolle Lösungen in kurzer Zeit realisieren, wenn man die Schwächen des Ansatzes (insbesondere die Verfügbarkeit eines entsprechend kompetenten Entwicklungsleiters) bewältigt. Wir vermerken:

> **Konzept 3.30** (Chief Programmer)**.** *Ein für die Rolle des Chief Programmers geeigneter Entwicklungsleiter ist von hoher Bedeutung für mittelständische IT-Dienstleister, um mit begrenzten Ressourcen die Projektkomplexität eines domänenspezifischen und produktlinienartigen Ansatzes zum Erfolg zu führen.*

Zusammenfassend zeigt sich, dass Ward wichtige Hinweise darauf gibt, dass wir durch das von uns identifizierte Potenzial der Domänenfokussierung im Zusammenspiel mit der automatisieren Generierung tatsächlich signifikante Einsparpotenziale realisieren können. Dazu muss es uns aber gelingen, diese Vorgehensweisen auch in ein Projektvorgehensmodell einzubetten, das die besonderen Rahmenbedingungen von mittelständischen Projekten beachtet[10] und zudem einen Weg findet, der eine wirtschaftlich machbare Entwicklung solcher Werkzeuge und Sprachen erlaubt[11].

3.2.4 Rapid Application Development (RAD)

Rapid Application Development (RAD) ist eine Methode zur Entwicklung von Anwendungen, die folgende Faktoren betont (siehe [Uocd97]):

[10] Dazu werden wir insbesondere agile Vorgehensmodelle in Abschnitt 3.4.3 betrachten.
[11] Hierauf kommen wir insbesondere bei Software-Produktlinien in Abschnitt 3.2.6 zurück.

- die kontinuierliche und enge Beteiligung von Anwendern am Entwicklungsprozess,
- die stringente Verwendung von CASE-Werkzeugen (siehe Abschnitt 3.2.2) zur Dokumentation und Erstellung von Systemen,
- die Prototypisierung von Systemen zwecks schnellen und frühen Feedbacks.

Die Hauptgefahr des RAD-Ansatzes liegt darin, dass die meisten Werkzeuge so konzipiert sind, dass sie die Entwicklung domänenunspezifischer Datenpflegewerkzeuge besonders schnell ermöglichen. Dies verleitet in einem Unternehmen häufig dazu, eine Vielzahl von individuellen Lösungen für Einzelprobleme zu entwickeln, die sich dann durch pragmatischen Gebrauch als Standard etablieren und somit zu Datensilos mit großen Integrationsproblemen führen.

Trotzdem sind RAD-Ansätze für die Entwicklung komplexer Lösungen heutzutage nicht mehr wegzudenken, so dass wir die prinzipielle Vorgehensweise aufgrund der hohen Überdeckung mit den von uns identifizierten Optimierungspotenzialen aus Kapitel 2 als wichtigen Bestandteil eines Lösungsansatzes ansehen. Daher werden wir ihn in unser Vorgehensmodell in Teil II der Arbeit aufnehmen. Der Gefahr der Datensilobildung begegnen wir durch eine stringente Domänenfokussierung der von uns entwickelten Lösungswerkzeuge, die verhindert (oder zumindest erschwert), dass das Werkzeug auch zur Behandlung anderer Probleme eingesetzt wird und somit zu Silobildung beitragen kann[12].

Wir halten als Ergänzung zu Konzept 3.25 ▷ 73 fest:

Konzept 3.31 (Rapid Application Development)**.** *Die Prinzipien und Konzepte des Rapid Application Development haben trotz ihres sehr allgemeinen Ansatzes und ihrer technischen Fokussierung wichtige Beiträge zur Beschleunigung der Anwendungsentwicklung geliefert. Daher sind moderne Modellierungsumgebungen nach RAD-Prinzipien zu gestalten.*

[12] Da wir umgekehrt in Kapitel 5 fordern werden, unseren Ansatz nur für etablierte Domänen unter der Verwendung ebenso etablierter Standards einzusetzen, besteht auch eine natürliche Notwendigkeit, standardisierte Im- und Exportformate anzubieten, um so ebenfalls der Silobildung entgegenzuwirken.

3.2.5 Architectured Rapid Application Development (ARAD)

Architectured Rapid Application Development (ARAD) ist eine Weiterentwicklung von RAD-Werkzeugen (siehe Abschnitt 3.2.4), die es Architekten erlaubt, neue Erweiterungen von Werkzeugen zu definieren. Diese können von Entwicklern dazu genutzt werden, um passenden Code zu dem beschriebenen Problem zu generieren (siehe [Lind05]): *„[ARAD] software uses patterns that can serve as building blocks to generate common parts of an application – a user interface or a set of business rules, for example. Architects and senior designers can build their own templates, which can then be reused by programmers to automatically generate large chunks of code. Because the tools often generate as much as 80% of native code, developers can add business logic and complete applications faster than they can with traditional development methods, according to industry analysts and users.“* Der Schwerpunkt dieses optimierten Ansatzes liegt wie bei RAD darauf, den Großteil der repetitiven und standardisierten Aufgabenstellungen in möglichst kurzer Zeit realisieren zu können, so dass in einem Projekt mehr Ressourcen verfügbar sind, um sich mit der eigentlichen Geschäftslogik zu beschäftigen. ARAD versucht also nicht, domänenspezifische Probleme zu optimieren, sondern die nicht die Domänenschicht betreffenden Fragestellungen so nicht-invasiv wie möglich zu behandeln. Erfahrungswerte zeigen, dass zwischen 65 und 80% des typischen Infrastrukturcodes von Anwendungen generiert werden können und Entwickler im Vergleich zu manueller Programmierung das 15fache an Produktivität gewinnen können (siehe [Have05], erste Seite).

Wir leiten als Erkenntnis aus Ansätzen wie ARAD aber auch der langen Geschichte von RAD ab, dass unser Lösungsansatz die existierenden Erfahrungswerte zu diesen Ansätzen aufgreifen muss, um die nicht-domänenspezifischen Fragestellungen so effizient wie möglich zu behandeln – wir müssen allerdings Konzepte und Vorgehensmodelle finden und definieren, die einen ähnlichen Produktivitätsgewinn zur Lösung der domänenspezifischen Probleme erlauben. Entsprechend halten wir fest:

Konzept 3.32 (Automatisierung von Infrastruktur)**.** *In Form von ARAD (und verwandten Ansätzen) gibt es bewährte Erfahrungen zur automatisierten Generierung von Infrastrukturcode – diese sind auch bei zukünftigen optimierenden Vorgehensmodellen zu berücksichtigen. Infrastrukturaspekte sollten bei der Beschreibung von Problemen und Lösungen also tatsächlich nur dann eine Rolle spielen, wenn es einen engen Domänenbezug gibt.*

3.2.6 Software-Produktlinien

Software-Produktlinien positionieren sich als Antwort auf stetig steigende Ansprüche an Softwaresysteme und softwareintensive Produkte, die einen hohen Funktionsumfang bei geringen Kosten fordern (siehe [BKPS04], S. 1 sowie [ClNo07]). Eine Software-Produktlinie ist in der gängigsten Definition nach Clements und Northrop wie folgt definiert: *„A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“* (siehe [ClNo02], S. 5) Die Wiederverwendung vieler standardisierter Komponenten, die Aufgaben einer bestimmten Domäne lösen, ermöglicht also Kosten- und Zeitersparnisse. Die Entwicklung einer solchen Software-Produktlinie erfolgt dabei auf eine strukturierte und organisatorisch vorgeschriebene Art und Weise. Weiterer Vorteil einer Software-Produktlinie ist die höhere Qualität durch eine größere Verbreitung der Basisplattform und damit eine intensivere Prüfung ihrer Leistungsfähigkeit. Durch die Verteilung von Entwicklungskosten auf viele Produkte können zudem mehr Funktionen geliefert werden als dies bei einem Einzelprodukt möglich wäre.

Der Ansatz der Software-Produktlinienentwicklung findet sich bereits in der Fertigungsindustrie (speziell der Automobilindustrie) in den 1980er Jahren, in denen die Entwicklung gemeinsamer Plattformen für die Automobilfertigung einen Ausweg aus der damals herrschenden Absatzkrise geboten hat (siehe [BKPS04], S. 4 und [CuNo98]). Organisationen, die sich aus unterschiedlichsten Gründen für den Einsatz von Software-Produktlinien entschieden haben (siehe z. B. die Fallstudien auf den Seiten 417–512 von [ClNo02]) berichten u. a. von den folgenden Vorteilen,

die sich nach der Einführung ergeben haben (siehe [ClNo02], S. 17 unten für eine gute Übersicht):

- flächendeckende Produktivitätssteigerungen
- verkürzte Realisierungszeiten („time-to-market“)
- gesteigerte Produktqualität
- verbesserte Kundenzufriedenheit

Zudem liefern Software-Produktlinien die Möglichkeit, die *economies of scope* (siehe unsere Ausführungen auf Seite 39) nutzen zu können (siehe dazu auch [ClNo02], S. 9).

Insgesamt bieten Software-Produktlinien somit eine Reihe wichtiger Verbesserungen, so dass wir sie nachfolgend noch detaillierter vorstellen wollen:

- Software-Produktlinien strukturieren sich in drei essentielle Aktivitäten (siehe [ClNo02], Kapitel 3):
  - *core asset development*[13] zur Entwicklung der für die Software-Produktlinie erforderlichen Basiskomponenten,
  - *product development*[14] zur Ausprägung von Produkten auf Grundlage der Basiskomponenten (üblicherweise durch Konfiguration, Komposition, Parametrisierung und Erweiterung)
  - *Management* zur technischen und organisatorischen Überwachung und Steuerung der anderen beiden Aktivitäten.

  Abbildung 3.1 stellt diese drei Aktivitäten als eng miteinander verzahnte Regelkreise dar[15]. Wir halten fest:

[13] Wir präferieren den alternativen Begriff *domain engineering*, der deutlicher zu den von uns identifizierten Optimierungspotenzialen passt.

[14] Auch hier favorisieren wir den alternativen Begriff *application engineering*, da er strukturell besser zum Begriff *domain engineering* passt.

[15] Diese Abbildung wurde entnommen aus [ClNo07], genauer: `http://www.sei.cmu.edu/productlines/frame_report/PL.essential.act.htm`.

Konzept 3.33 (Domain Engineering und Application Engineering)**.** *Der Entwurf und die Weiterentwicklung der Domäne ist organisatorisch vom Entwurf und der Weiterentwicklung durch Einrichtung separater Regelkreise zu trennen. Um den Informationsfluss zwischen diesen beiden Regelkreisen gezielt zu steuern, wird ein dritter Managementregelkreis installiert. Er sorgt dafür, dass neue Leistungsmerkmale der Domänenschicht auch in die ausgeprägten Lösungen einfließen, aber keine schwerwiegenden Inkompatibilitäten erzeugen. Zugleich profitieren sie von den Erfahrungen und Anforderungen bei der Lösungsausprägung.*

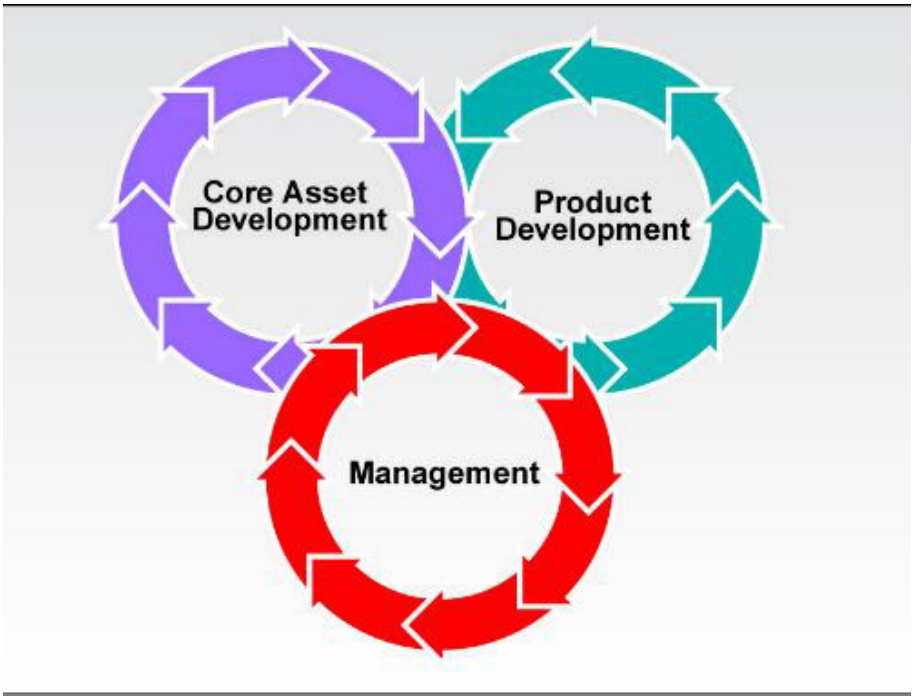


Abbildung 3.1: Essentielle Aktivitäten für Software-Produktlinien

- Ausgehend von diesen drei Hauptaktivitäten werden 29 Methodengebiete definiert, die bestimmte Fragestellungen aus Sicht des Software Engineerings, des technischen oder des organisatorischen Managements adressieren. Die Bandbreite der behandelten Themen reicht dabei von der Architekturdefinition und -evaluierung über Testverfahren bis hin zur Marktevaluierung, Risikobehandlung und Auslieferung von bzw. Training in Produktinstanzen. Wie sich an der Bandbreite und Zahl von Methodengebieten erkennen lässt, sind die theoretisch erforderlichen Grundlagen recht umfangreich. Es handelt sich bei der Entwicklung von Softwareproduktlinien um einen relativ dokumentations-

intensiven Prozess. Die verfügbare Expertenliteratur enthält aber auch eine Reihe von Fallstudien, die zeigen, dass Software-Produktlinien sehr gut im mittelständischen Umfeld eingesetzt werden können (siehe [ClNo02], S. 485–512, [BKPS04], S. 207–246 sowie [SEI09]).

- Es werden 22 Entwurfsmuster zur Lösung typischer Probleme und Fragestellungen, die bei der Annäherung an Software-Produktlinien entstehen, vorgestellt (siehe [ClNo02], Kapitel 7).

Zudem definieren Software-Produktlinien eine Reihe von Variabilitäten, die bei Berücksichtigung in der strategischen Entwicklungsplanung der Linie die Ausprägung von Lösungsinstanzen vereinfachen und zudem sicherstellen, dass eine graduelle Evolution möglich ist. Die in [BKPS04], S. 14–19, beschriebenen Variabilitäten sind:

- Variabilität in Features,
- Variabilität in Abläufen,
- Variabilität im Datenumfang,
- Variabilität im Datenformat,
- Variabilität im Systemzugang,
- Variabilität in Benutzerschnittstellen,
- Variabilität über Systemschnittstellen und
- Variabilität in der Qualität.

Insgesamt stellt die Theorie zu Software-Produktlinien einen bewährten und sehr umfassenden Rahmen bereit, um die Entwicklung von Lösungsfamilien strategisch zu verwalten und Einzellösungen basierend auf der Familienplattform auszuprägen. Die oben kurz angedeutete Menge von Methoden und Regeln zeigt bereits, dass die Entwicklung einer Software-Produktlinie kein einfaches Unterfangen und eigentlich eher eine im Umfeld von Großunternehmen angesiedelte Tätigkeit ist. Dennoch finden sich auch in der Literatur zu Software-Produktlinien eine Reihe erfolgreicher Beispiele aus dem mittelständischen IT-Umfeld, in denen IT-Dienstleister spezialisierte Software-Produktlinien erfolgreich umgesetzt haben, um Marktnischen zu besetzen. Der formale Rahmen erscheint uns sehr nützlich, um eine die *economies of scope* (siehe Seite 39) nutzende Plattform aufzubauen, die sich der Lösung von Aufgaben einer spezifischen Domäne widmet. Durch die Beschränkung auf eine stark eingegrenzte Domäne und die Fokussierung auf die notwendigen Features des mittelständischen Kontextes sehen wir eine Grundlage für unseren Lösungsansatz und werden Software-Produktlinien daher als zentralen Bestandteil unseres Ansatzes in

Kapitel 5 wieder aufgreifen. Bereits an dieser Stelle wollen wir darauf hinweisen, dass ein Down-Sizing des Gesamtansatzes auf eine Menge von Bausteinen, die für den mittelständischen Kontext als ausreichend anzusehen ist, erforderlich sein wird. Hier verweisen wir auch auf Seite 131 und unsere Ausführungen zu Feature Driven Development, das hier einen wichtigen Beitrag liefern wird.

Wir vermerken für die weitere Ausgestaltung unseres Ansatzes:

Konzept 3.34 (Eng fokussierte Software-Produktlinien)**.** *Die Theorie der Software-Produktlinien stellt ein bewährtes und erprobtes Konzept zur systematischen Erstellung und Evolution domänenfokussierter Lösungen bereit. Für den mittelständischen Kontext müssen allerdings reduzierte Software-Produktlinien in Betracht gezogen werden, die die Prozesskomplexität auf ein möglichst pragmatisches Niveau beschränken und den Entwicklungsprozess maximal agil und kundenorientiert gestalten.*

3.2.7 Domain Driven Design (DDD)

Wie wir bereits in Abschnitt 2.2.2 ausgeführt haben, ist ein gutes Domänenmodell von essentieller Bedeutung bei der Erstellung komplexer und leistungsfähiger Software. Entsprechend wichtig ist es auch, in einem Vorgehensmodell für den Entwicklungsprozess entsprechende Aktivitäten und Phasen vorzusehen. Dennoch hat es sich lange Zeit als extrem schwierig erwiesen, gute Domänenmodelle systematisch aufzubauen (siehe [Evan04a], S. xvii: *„A good domain model can be incredible valuable, but it's not something that's easy to make.“*). Eric Evans ist mit [Evan04a] angetreten, um in Form von Mustern und Best Practices ein Vorgehensmodell und eine Entwurfsphilosophie zu formulieren, die existierende Erfahrungswerte in Form des von ihm so bezeichneten *Domain Driven Design (DDD)* verbindet, sodass Anwender und Entwickler einem Handlungsleitfaden folgen können, um zu einem guten Domänenmodell zu gelangen.

DDD setzt einige Grundannahmen voraus, die gut zu unseren Überlegungen bezüglich der Notwendigkeit von Domänenfokussierung, Agilität und Automatisierung aus Abschnitt 2.2 passen:

- *Entwicklung ist iterativ* ([Evan04a], S. xxii).
- *Zwischen Entwicklern und Domänenexperten gibt es eine enge Beziehung*[16] ([Evan04a], S. xxii-xxiii).

Ein agiles Vorgehensmodell wird von DDD entsprechend vorausgesetzt, um hohen Aufwänden im Anfangsstadium durch ein möglichst iteratives und evolutionäres sowie auf das jeweils nächste zu erreichende Ziel ausgerichtete Vorgehen zu begegnen. Hier ist die bekannte Kritik an agilen Prozessen in sehr großen Projekten zu erwähnen[17]. Da wir im mittelständischen Umfeld nur in den seltensten Fällen von Projekten der angesprochenen Größenordnungen ausgehen können, hat diese Kritik für unsere weiteren Betrachtungen keine Auswirkungen. Daher halten wir fest:

Konzept 3.35 (Agilität)**.** *Agile Prozesse unterstützen die Destillation von Domänenwissen nachhaltig und verstärken das Band zwischen Entwicklern und Fachexperten.*

DDD spielt seine Stärken nach Aussage von Evans insbesondere dann aus, wenn sich nicht nur ein einzelner Entwickler sondern ein ganzes Team seine Prinzipien zunutze macht, da das Team dann über eine gemeinsame domänenspezifische Sprache hohe Effizienzgewinne realisieren kann und zudem Missverständnisse in der Anforderungsdefinition deutlich reduziert werden (siehe [Evan04a], S. xxvii).

Analog zu unseren Ausführungen in Abschnitt 2.2.2 sieht Evans Modelle als Abstraktion eines bestimmten abgeschlossenen Problemkreises. Ein Domänenmodell definiert er als eine rigoros organisierte, selektive und abstrakte Repräsentation des erforderlichen Domänenwissens[18]. Insbesondere weist er dabei darauf hin, dass es bei Domänenmodellierung auf keinen Fall darum geht, die Realität so präzise wie möglich nachzuempfinden (dafür ist sie einfach zu komplex), sondern eine geeignete

[16] Nebenbemerkung: Da der Entwicklungsprozess iterativ ist, arbeiten beide Gruppen logischerweise während der gesamten Projektdauer miteinander.

[17] Es wird beobachtet, dass man im Durchschnitt besser qualifizierte Entwickler braucht, um in agilen Projekten erfolgreich zu sein, da sehr viel Wissen in den Köpfen der Teammitglieder steckt und diese entsprechend effektiv damit umgehen müssen. Zudem gibt es nur sehr begrenzte Erfahrungswerte und Beispiele für den erfolgreichen Einsatz agiler Methoden in großen Projekten (siehe dazu unsere Ausführungen in Abschnitt 3.4.3 ab Seite 129 sowie z. B. [BoTu03b], S. 20ff.). In größeren Projekten wird es aber zunehmend schwerer, dieses Leistungsniveau zu halten.

[18] [Evan04a], S. 3: *„A domain model is not a particular diagram; it is the idea that the diagram is intended to convey. . . . it is a rigorously organized and selective abstraction of that knowledge.“*

Abstraktion zur Lösung der domänenspezifischen Probleme zu finden. Modelle kommen demgemäß bei der Domänenmodellierung drei zentrale Aufgaben zu ([Evan04a], S. 3–4):

- *Das Modell und der Kern eines Entwurfs formen sich gegenseitig.* Hiermit meint Evans die enge Verbindung zwischen einer Analyse, der daraus resultierenden Modellierung und der Notwendigkeit, diese konsequent auch in der Implementierung zu repräsentieren, damit insbesondere auch spätere Wartungsarbeiten effizient behandelt werden können. Denn die Implementierung soll nach Möglichkeit ein (dann leichter verständliches) Spiegelbild der Domänenmodellierung sein.
- *Das Modell ist das Rückgrat der Sprache, die das Team verwendet.* Aufgrund der engen Beziehung zwischen Modell und Implementierung, können Entwickler sich über ihr Programm unterhalten, indem sie Begriffe aus dem Domänenmodell verwenden.
- *Das Modell repräsentiert das destillierte Domänenwissen.* Das Domänenmodell formt die Art und Weise, wie man die Domäne repräsentiert, über sie nachdenkt und spricht. Es enthält die essentiellen Begrifflichkeiten, Strukturen, Konzepte und Relationen zwischen den Bausteinen der Domäne. Aufgrund der engen Verbindung zur Implementierung können sich Entwickler und Domänenexperten gemeinsam über das Problem unterhalten und neue Erkenntnisse zur Funktionsweise eines Systems können auch wieder direkt in das Domänenmodell zurückfließen. Wir halten aufgrund der Ähnlichkeiten zu unseren Vorüberlegungen fest:

  Konzept 3.36 (Durchgängiges Domänenmodell)**.** *Die betrachtete Domäne ist durchgängig zu modellieren. Alle relevanten Aspekte werden Bestandteile des Modells und Veränderungen des Modells bedeuten ebenso Veränderungen der betrachteten Domäne wie sich Veränderungen in der Domäne bei korrekter Modellierung zwangsweise als Veränderungen im Modell widerspiegeln müssen.*

Evans sieht die folgenden essentiellen Zutaten, die Domain Driven Design erst zu voller Wirkung bringen:

1. *Das Sammeln der Informationen ist ein stetiger intensiver Prozess.* Informationen über die Domäne werden während der gesamten Projektdauer weitergesammelt und können auch jederzeit Auswirkungen auf das Domänenmodell haben. Dies liegt in der Komplexität der meisten Domänen begründet – tieferschürfen-

de Erkenntnisse erschließen sich erst nach hinreichend intensiver Beschäftigung mit der Domäne.

2. *Es wird eine universelle Sprache verwendet*[19]*.* Diese dient zur einheitlichen Kommunikation zwischen Domänen- und IT-Experten und hilft, damit verbundene Übersetzungsfehler zu vermeiden. Das Domänenmodell dient üblicherweise als Rückgrat und stellt aufgrund der bereits erwähnten engen Bindung zwischen Modell und Implementierung eine Verbindung zwischen der Domäne und den Implementierungstechnologien her. Veränderungen an der Sprache kennzeichnen zudem Veränderungen am Modell: Werden neue Begriffe eingeführt, impliziert das üblicherweise neue Konzepte. Werden Begriffe gestrichen, fallen damit auch Konzepte weg. Werden Begriffe ergänzt, haben sich neue Erkenntnisse ergeben. Insoweit ist eine universelle Sprache ein wichtiger Bestandteil für den Erkenntnisprozess bei der Domänenanalyse – durch konsequente Verwendung der universellen Sprache werden Erkenntnisse augenscheinlicher. Zugleich vereinfacht sich die prinzipielle Kommunikation, da es eine gemeinsame Sprache gibt, die einerseits von den Domänenexperten verstanden wird, von den IT-Experten aber auch benutzt werden kann, um Widersprüche und implizites Wissen (erkennbar an neuem Vokabular) aufzudecken. Zudem ist diese Sprache das Handwerkszeug für Modellierungsphasen, da sie stetig zur Kommunikation über die Domäne verwendet wird. Evans geht dabei von einem iterativ-evolutionären (und im Grunde auch agilen) Prozess aus. Deshalb fügt sich dieses Vorgehensmodell gut in die von uns postulierten Optimierungspotenziale für mittelständische Projekte ein (siehe insbesondere unsere Ausführungen in Abschnitt 2.2.4). Es folgt:

> **Konzept 3.37** (Domänenmodell als Handwerkszeug)**.** *Das Domänenmodell ist das zentrale Werkzeug in den iterativ-evolutionären Modellierungsphasen, in denen die Lösung entworfen und umgesetzt wird.*

3. *Modellierung und Codierung muss eine gemeinsame Tätigkeit sein.* Modellierer müssen verstehen, welche Grenzen Implementierungen mit sich bringen und welche Auswirkungen Modelländerungen haben. Codierer müssen verstehen, dass Code-Veränderungen auch das beschriebene Modell verändern und ggf. Anpassungen erfordern (siehe [Evan04a], S. 60–62). Wir nehmen dies als Aufforderung, Codierung und Modellierung in dem von uns vorgeschlagenen Vorgehensmodell so eng wie möglich zusammenzubringen – optimalerweise wer-

[19] Evans bezeichnet diese als *„ubiquitous language“*, siehe [Evan04a], S. 23ff.

den wir einen Weg finden, bei den besonderen mittelständischen Rahmenbedingungen Modellierung der Codierung gleichzusetzen. Das thematisieren wir insbesondere in Kapitel 5 detailliert. Wir formulieren:

> **Konzept 3.38** (Zusammenarbeit von Fach- und IT-Experten)**.** *Fachexperten und Codierer entwickeln gemeinsam, um gegenseitiges Verständnis für die jeweilige durch die Person vertretene Domäne zu entwickeln. Fachexperten erlangen so ein Verständnis für die Grenzen, Möglichkeiten und Schwierigkeiten der eingesetzten Methoden und Technologien. Codierer erhalten so einen Einblick in die tatsächlichen Probleme auf Fachseite.*

Zudem gibt Evans eine Vielzahl grundlegender Objekttypen an, die dabei helfen, Modelle in Software umzusetzen. Dies sind in [Evan04a] zunächst für Objekte und ihre Beziehungen untereinander definiert:

- *entities* (S. 89–96) als Objekte mit einer physikalischen und logischen Identität, die üblicherweise eine längere Lebenslinie besitzen – dazu zählen die meisten Geschäftsobjekte (z. B. Unternehmen, Guthaben, Bestellungen, Kunden, usw.),
- *value objects* (S. 97–103) als Objekte, die lediglich Charakteristika komplexerer Objekte definieren und keine eigene Identität haben (z. B. ein bestimmtes Gewicht, eine bestimme Menge Geld, usw.),
- *services* (S. 104–108) als zustandslose Einheiten von Geschäftslogik, die nicht eindeutig einer Entität oder einem Werteobjekt zuordbar sind und womöglich sogar mehrere Objekte zusammen verwenden, um Ergebnisse zu liefern und
- *modules* bzw. *packages* (S. 109–115) als eng zusammenhängende Einheiten von Objekten und Diensten, die nur lose an andere Module gekoppelt sind.

Zudem gibt er mit dem Muster *Layered Architecture* (S. 68–75) einen Lösungsansatz vor, um Domänenmodelle möglichst weitreichend von den zugrunde liegenden technologischen Implementierungen zu abstrahieren[20]. Zugleich illustriert er mit dem Anti-Muster *Smart UI*, wie eine Architektur aufgebaut sein kann (und heute auch häufig vorzufinden ist), die alle Vorteile von DDD zunichte macht (siehe S. 76–78)[21].

[20] Grundidee ist, dass eine Anwendung in saubere Schichten mit klaren Verantwortlichkeiten unterteilt wird, bei denen eine Schicht immer nur auf Elemente aus der eigenen oder tieferen Schicht zugreifen darf. Es gibt dazu ausführliche Grundlagenarbeiten, z. B. auch die von Evans zitierten [BMRS96]. Evans differenziert als typische Schichten die Nutzerschnittstelle, die Anwendungslogik, die Domänenschicht und die Integrationsschicht.

[21] Hier wird im Prinzip die gesamte Geschäftslogik in der Nutzerschnittstelle untergebracht. Dazu

Weiterhin stellt er Muster vor, um den Lebenszyklus von Objekten stringenter zu behandeln und dadurch die Integrität von Objekten und die Grenzen zwischen Objekten klarer zu definieren:

1. *aggregates* (S. 125–135), um sinnvolle Grenzen für komplexe Objekte und die Verwaltung von Konsistenzbedingungen auf diesen Objekten zu finden,
2. *factories* (S. 136–146), um die Erzeugung komplexer Objekte mit einem wohldefinierten Vorgehen und so gut gekapselt wie möglich zu vollziehen,
3. *repositories* (S. 137–161), um den restlichen Lebenszyklus ab der Geburt für eine Objektmenge gleichen Typs bereitzustellen, hier inbesondere Lokalisierungs- und Lebenszyklusfunktionen.

Diese Muster bilden eine universelle Sprache zur Beschreibung der DDD-Vorgehensweise, die Evans anhand vieler Beispiele veranschaulicht. Anschließend gibt er weiterführende Hinweise sowohl für das Refactoring von Anwendungen als auch für das strategische Design von Architekturen und Domänen. Insbesondere weist Evans daraufhin, dass sich die tiefgreifendsten und für den Erfolg wichtigsten Erkenntnisse erst durch kontinuierliches Refactoring ergeben und nur so systematisch Durchbrüche in der Domänenanalyse erzielt werden können (S. 188–192). Er betont insbesondere, wie wichtig es ist, Domänenkonzepte explizit zu machen (S. 205–241), da nur so die nötige Kommunikation zwischen Domänenexperten und IT-Experten geschaffen werden kann, die für derartige Durchbrüche zwingend erforderlich ist. Um kontinuierliches Refactoring zu ermöglichen, sind möglichst einfache Entwürfe essentiell. Entsprechend liefert Evans eine Reihe von Entwurfsmustern, um Entwürfe so einfach und veränderbar wie möglich zu gestalten (S. 243–292) und beschreibt, wie man diese Einzelbestandteile seines Ansatzes dann systematisch kombinieren kann, um tiefergehendes Domänenwissen zu erwerben und auch strukturiert aufzubereiten (S. 293–326).

Anschließend wendet sich Evans dem Thema *Strategic Design* zu. Darunter versteht er Entwurfsmuster und -methoden, um auch äußerst komplexe und umfangreiche Domänen über einen möglichst langen Zeitraum in einer evolutionsfähigen Software-

wird die Anwendung in möglichst viele kleine Funktionen unterteilt, die jeweils inklusive ihrer Benutzungsschnittstelle als Modul realisiert wird. Eine relationale Datenbank dient dann häufig als gemeinsamer Speicher für die so aufgeteilten Funktionen. Dafür werden dann möglichst automatische Entwicklungswerkzeuge für die Oberfläche sowie für visuelle Programmierung eingesetzt. Evans merkt an, dass dies besonders häufig in Teams passiert, die frische Erfahrungen mit modellgetriebener Entwicklung sammeln. Es ist also wichtig, Modelle möglichst ausdrucksstark zu gestalten, um die beschriebene unglückliche Partitionierung der Anwendung von vorneherein zu verhindern.

lösung abbilden zu können. Die drei dazu wichtigsten Prinzipien, die gleichermaßen für unseren Problemkontext gelten, sind (S. 328–329):

1. *Context.* Die betrachtete Domäne muss einerseits klar abgegrenzt sein, andererseits in sich aber auch widerspruchsfrei beschrieben werden.
2. *Destillation.* Um die widerspruchsfreie Beschreibung einer Domäne in endlichem Aufwand realisierbar zu machen, erachtet Evans es als essentiell, sich dabei auf den Kern der Domäne zu fokussieren und diesen besonders gut zu lösen. Randprobleme können abstrahiert oder in initialen Entwürfen außen vor gelassen werden, um so eine möglichst gute Modellierung der essentiellen Domänenthemen zu erreichen.
3. *Large-scale structure.* Um den Überblick über ein sehr komplexes System zu bewahren, ist es erforderlich dieses System nach Verantwortlichkeiten zu modularisieren. Hierzu führt Evans eine Reihe sinnvoller Entwurfsmuster an, insbesondere z. B. *Responsibility Layers* zur Erstellung sauberer Schichtenmodelle.

Anschließend führt Evans diese drei Prinzipien auf den Seiten 331–483 mit einer Vielzahl spezifischer Entwurfsmuster und praktischen Beispielen detailliert aus.

Evans schließt seine Betrachtungen mit Ausführungen zur Kombination der einzelnen von ihm beschriebenen Lösungsbausteine und gibt zudem eine Reihe von Erfahrungswerten aus länger laufenden und auf DDD basierenden Projekten, die sowohl Chancen als auch Schwierigkeiten im Einsatz dieses Vorgehensmodells illustrieren (S. 485–506).

Den starken Zusammenhang zwischen domänengetriebenen und modellgetriebenen Ansätzen betont Evans auch noch mal in [Evan04b], entsprechend werden wir noch ausführlicher in Abschnitt 3.3 auf modellgetriebene Verfahren eingehen und in Abschnitt 3.5 schließlich aufzeigen, welche dieser Ansätze wir wie kombinieren werden.

Zusammenfassend formulieren wir:

Konzept 3.39 (Domain Driven Design als Konstruktionsschema)**.** *Mit Domain Driven Design existiert eine hinreichend strukturierte und durch viele Erfahrungswerte ausgereifte Entwicklungsmethode für den Entwurf und die Umsetzung domänenspezifischer Lösungen, die zudem aufgrund einer überschaubaren Menge von Entwurfsmustern leicht zugänglich und somit auch explizit für mittelständische Projekte geeignet ist.*

3.2.8 Domain Specific Languages (DSL)

Domänenspezifische Sprachen (DSLs) werden als Oberbegriff dafür verwendet, Entwicklungssysteme durch Bereitstellung von Abstraktionsmechanismen für die Lösung sehr spezifischer Probleme leistungsfähiger zu gestalten. Die Modellierung der Lösung wird also vereinfacht, indem man sie auf einen eingeschränkten Problemkreis fokussiert. Domänenspezifische Sprachen müssen die folgenden Anforderungen erfüllen (siehe [GSCK04], S. 142–143):

- Die Domäne muss präzise definiert sein.
- Die Konzepte in der Modellierungssprache müssen für Anwender aus der Domäne klar verständlich sein. Ebenso sollten die Konzepte so benannt sein, dass Domänenexperten sie sofort zuordnen können.
- Die Notation der Modellierungssprache sollte für den Verwendungszweck so leicht wie möglich verwendbar sein.
- Die Modellierungsprache sollte eine klare Grammatik zur zweifelsfreien Definition von Ausdrücken besitzen.
- Die Bedeutung von Ausdrücken sollte klar abgegrenzt sein, so dass Domänenexperten Ausdrücke auch direkt verstehen können.

Domänenspezifische Sprachen können sowohl grafisch als auch textuell sein. [Fowl05b] gibt interessante Hinweise dazu, wie aus domänenspezifischen Sprachen Code generiert bzw. diese Sprachen direkt ausgeführt werden können.

Wie bereits mehrfach zuvor erläutert, bieten sich durch diesen Ansatz vielfältige Vorteile, die wir nicht noch einmal wiederholen wollen. Die breite Verwendung domänenspezifischer Lösungsansätze demonstriert einmal mehr, wie viel Potenzial der von uns identifizierte Baustein der Domänenfokussierung aus Abschnitt 2.2.2 doch tatsächlich besitzt – zieht er sich doch durch unterschiedlichste Methodiken und Ansätze, die wir in den vorausgegangenen Abschnitten vorgestellt haben.

3.3 Modellgetriebene Verfahren der Softwareentwicklung

Modellgetriebene Verfahren der Software-Entwicklung fokussieren die teilweise oder vollständige Generierung von Lösungen aus formalen Modellen. Prominentester Vertreter dieses Ansatzes ist die Model Driven Architecture (MDA, Abschnitt 3.3.1).

Zudem gibt es eine Vielzahl von Bestrebungen, modellbasierte Verfahren durch das wesentlich weniger formale Model Driven Software Development (MDSD, Abschnitt 3.3.2) zum Erfolg zu führen: Indem man sich stärker an der spezifischen Domäne orientiert und die Modelle darauf ausrichtet. Zudem betrachten wir verschiedene Teildisziplinen und historische Ansätze, die ähnliche Ziele verfolgen: Software Factories (Abschnitt 3.3.3), Generative Programmierung (Abschnitt 3.3.4), Domain-Specific Modeling (DSM, Abschnitt 3.3.5), Model Driven Engineering (MDE, Abschnitt 3.3.6), Business Process Modeling (BPM, Abschnitt 3.3.7), ARIS (Abschnitt 3.3.8) und Agile Modeling (Abschnitt 3.3.9).

Aus diesen Betrachtungen werden wir wertvolle Erkenntnisse gewinnen für die von uns zu definierenden Transformationen von Modellen in ausführbare Systeme, die wir zur Beschreibung des Conceptual Model Driven Software Development (CMDSD, Kapitel 5) nutzen. Zudem sammeln wir Erfahrungswerte zu bewährten Anwendungsweisen dieser Methoden in konkreten Projekten sowie zu ihrer Nutzung bei der Zusammenarbeit mit Fachexperten.

3.3.1 Model Driven Architecure (MDA)

Die Model Driven Architecture (MDA, siehe [Wiki06b]) ist ein von der Object Management Group (OMG, siehe [OMG06a]) entwickelter Ansatz zur modellbasierten Spezifikation von Software-Systemen. Er setzt sich aus einer Reihe von Spezifikationen zusammen, u. a. der *Unified Modeling Language (UML)*, der *MetaObject Facility (MOF)*, *XML Metadata Interchange (XMI)*, und dem *Common Warehouse Metamodel (CWM)* (siehe [OMG09] für die aktuellsten Versionen der Spezifikationen). Das Ziel dieses Ansatzes wird in [OMG01], S. 1 wie folgt beschrieben: *„The OMG's mission is to help computer users solve integration problems by supplying open, vendor-neutral interoperability specifications. The Model Driven Architecture (MDA) is OMG's next step in solving integration problems.“* Auf S. 3, Abschnitt 2.1 des Dokuments wird der Ansatz detailliert: *„The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. [...] The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go.“*

Hier wird erstmalig auf eine wesentliche Charakteristik des MDA-Ansatzes eingegangen: Man unterscheidet zwischen plattformunabhängigen und plattformspezifischen Modellen, die durch Abbildungen ineinander überführt werden. Dadurch soll eine plattformunabhängige Form der Spezifikation ermöglicht werden.[22] Prominentestes Produkt des MDA-Ansatzes ist die Unified Modeling Language (UML, siehe [OMG06b, Wiki06c, Wiki06d]) als grundlegende Modellierungssprache, die wiederum formal über die Meta Object Facility (MOF, siehe [OMG06c]) beschrieben wird. Aus den Modellen werden (manuell, halbautomatisch mit Software-Unterstützung und in manchen Fällen auch automatisiert) plattformspezfische Architekturen und letztlich Programmcode generiert, der dann um Geschäftslogik erweitert werden kann und muss, um das fertige System zu realisieren. Ein weiteres Ziel der OMG ist es, auch das Reverse Engineering von Systemen zu ermöglichen, indem aus existierenden Programmcodes veränderbare Modelle generiert werden, um so durch Roundtrip Engineering eine Programm- und Architekturevolution zu ermöglichen.

Zentrales Merkmal der Model Driven Architecture ist die Auftrennung der Modelle in drei Stufen:

1. *CIM: Computation Independent Model.* Diese Modellform ist die abstrakteste Ebene, die das vorliegende Problem ohne Berücksichtigung technischer Aspekte aus reiner Fachsicht beschreibt.

2. *PIM: Platform Independent Model.* Diese Modellebene enthält zudem Beschreibungen der Zielarchitektur ohne Berücksichtigung der konkreten Implementierungsplattform, die gewählt wird.

3. *PSM: Platform Specific Model.* Diese Modellebene beschreibt die konkrete Umsetzung für eine Zielplattform mit konkreten Technologien, Bibliotheken und Frameworks sowie einer technologiespezifischen Architektur.

Großer Vorteil dieser Aufteilung ist die Möglichkeit zur sauberen Trennung von fachspezifischen, technischen und strukturbedingten Informationen, die vor der MDA häufig innerhalb eines UML-Modells vermischt wurden, was die Wiederverwendbarkeit massiv reduzierte bzw. erschwerte. Die Model Driven Architecture bewährt sich mittlerweile in vielen Projekten und es gibt eine Reihe von Studien und Artikeln, die anhand konkreter Projektbeispiele die prinzipielle Praktikabilität des Ansatzes beweisen (siehe [Gruh05]).

[22] Wir weisen darauf hin, dass wir diesen Teil der strukturellen Kurzübersicht über die MDA an [Kent02] angelehnt haben.

Zur Erhöhung der Abstraktionsfähigkeit von UML bzw. zur Verbesserung der Verständlichkeit für reine Fachanwender zeichnet sich ab, dass es abstraktere Modellierungssprachen und Weiterentwicklungen von UML geben wird (siehe [Ther05] für allgemeine perspektivische Betrachtungen oder [Sysm09, Zoll05] für Anmerkungen zu SysML).

Als nachteilig erweist sich aber in vielen Fällen, dass eine Mehrzahl der MDA-Projekte technologisch sehr spezifisch ausgerichtet ist. In der Mehrzahl der Fälle wird gegenwärtig eine JavaEE-Implementierung des Systems vorausgesetzt (siehe auch [BeBG05]), was mitunter dazu führt, dass die modellspezifischen Ergänzungen wiederum sehr eng mit einer spezifischen technologischen Plattform verheiratet sind. Dies verringert die Übertragbarkeit der Modelle und rückt zudem verstärkt technologische Feinheiten in den Blickpunkt. Dadurch wird im Umkehrschluss der Zugriff auf das enthaltene fachliche Domänenwissen erschwert: Fachanwender müssen sich anhand technologischer Details einen Überblick über die fachliche Modellierung verschaffen, was im Einzelfall schwer bis unmöglich werden kann.

Zudem zeigt sich, dass heutige MDA-Projekte zumeist zunächst umfangreiche Festlegungen bezüglich der Struktur des Projektes und der Art der MDA-Verwendung erfordern. Erstaunlicherweise konzentrieren sich daher auch viele bisherig existierende Best-Practice-Hinweise darauf, wie ein MDA-basiertes Projekt und gerade nicht ein MDA-basiertes System zu strukurieren ist[23] (siehe z. B. [WeWe05]).

Praktische Beispiele zur Ausführbarkeit von UML belegen die Stärken und Schwächen des MDA-Ansatzes. Im in [Bühl05] beschriebenen Fall wird die Funktionsweise eines Bahnstellwerkes mit UML modelliert und die Modelle werden ausgeführt, um sie zu testen. Als Stärken im Vergleich zu einem direkten MDSD-Ansatz zeigen sich die folgenden Punkte:

- Auch sehr spezielle Aufgabenstellungen können ohne Zusatzaufwand direkt mit UML modelliert werden. Bei einem MDSD-basierten Ansatz wäre es zunächst erforderlich, geeignete Modelle zu definieren und diese über Transformationen in einen ausführbaren Zustand zu bringen.
- Die UML-Darstellung (Sequenzdiagramm auf einer DIN A4-Seite) ist wesentlich kompakter als die existierende Spezifikation in Textform (ca. 50 Seiten).

[23] Der Rational Unified Process (RUP) ist natürlich eine Strukturierungsmöglichkeit, wird aber aufgrund seiner Komplexität nur in geringem Maße in Projekten angewendet. Insbesondere findet RUP bei kleineren Projekten in wenig sicherheitsrelevantem Umfeld kaum Verwendung. Gerade hier liegt aber der Fokus unserer Betrachtungen, siehe dazu auch unsere Ausführungen in Kapitel 2.

Umgekehrt zeigen sich aber auch Schwächen dieses Ansatzes:

- Nur die dynamische Semantik des Bahnstellwerkes kann modelliert werden. Um das Gesamtsystem inklusive Benutzeroberfläche, etc. zu modellieren, mangelt es in der UML an Ausdrucksmitteln.
- In diesem speziellen Fall kann zwar mit der UML-Modellierung die Dynamik des Systems modelliert und auch testbar gemacht werden. In allgemeineren Fällen mit komplexen dynamischen Interaktionen mit Anwendern und Fremdsystemen fällt dies ungleich schwerer (falls es aufgrund der steigenden Komplexität überhaupt effektiv möglich ist), da erheblich mehr Schnittstellen und komplexere Dateneinflüsse an der Tagesordnung sind.
- Es wird zwar in dem Artikel darauf hingewiesen, dass die UML-Darstellung kompakter als die textuelle Beschreibung ist. Dennoch weist der Autor auch darauf hin, dass es sich hier um ein recht einfaches Beispiel handelt, und bereits hier seltenere (und damit weniger vertraute) UML-Elemente verwendet werden mussten, um die fachlichen Anforderungen abzubilden. Dies relativiert den Vorteil der leichten Verständlichkeit.

Auch Eric Evans weist in seinen Ausführungen zur Entwicklung von Domänenmodellen auf die Schwächen der UML hin (siehe [Evan04a], S. 35–36): Modellierungssprachen wie die UML haben zwei gravierende Nachteile:

- Sie sind einerseits zu vollständig in dem Sinne, dass sie Anwendern das Gefühl geben, jedes Objekt, dass sie jemals codieren werden, auch modellieren zu müssen.
- Sie sind andererseits nicht vollständig genug, da sie sich vornehmlich mit Attributen und Relationen beschäftigen und Verhalten relativ schwer zu visualisieren ist. Wichtige Punkte zum dynamischen Verhalten lassen sich zwar in einzelnen Interaktionsdiagrammen visualisieren, aber komplexeres Domänenverhalten ist wiederum nur durch textuelle Ergänzungen abbildbar. Auf diese Weise werden Nutzer der UML immer wieder dazu verleitet, wichtige Aspekte aus den Diagrammen rauszulassen, wodurch diese nicht vollständig genug sind.

Die textuellen Erweiterungen lassen sich nicht auf einfache Weise in Generierungswerkzeugen berücksichtigend, was die Verwendung der UML erschwert, da neue Wege gesucht werden müssen, Nutzen aus der bereits geleisteten Modellierungsarbeit zu ziehen – sei es durch die vergleichsweise komplexe Entwicklung speziell optimierter Generatoren oder durch die manuelle Übertragung des enthaltenen Wissens (mit allen damit verbundenen Problemen).

Allerdings gibt es auch zunehmende Kritik am UML-Ansatz insgesamt und spätestens seit 2005 gibt es eine Reihe divergierender Expertenmeinungen darüber, ob die UML überhaupt ihren eigenen Zielen gerecht werden kann (siehe z. B. [Fisc05b], Zitat: *„UML ist gut für Skizzen“*). Die UML-Kritiker bezweifeln zunehmend, dass das primäre Ziel der UML-Befürworter – die Modellprüfung, ob Kundenwünsche korrekt umgesetzt wurden – überhaupt von der UML geleistet werden kann. Schließlich erfordern komplexe fachliche Modelle ein tiefgehendes Verständnis von den fachlichen Zusammenhängen, das so erst einmal weder in der UML vorhanden noch leicht integrierbar ist.

Auch in ausführlichen Spezifikationsphasen mit technischen und fachlichen Experten zeigen sich vielfältige Schwächen von UML (siehe z. B. [Ecke05] für eine pragmatische, wenn auch polemische Beschreibung dessen, was auch dem Author und vielen anderen Kollegen immer wieder in Projekten aufgefallen ist):

> *„[. . .] The third big issue was about the UML. Everywhere I went, people were struggling with it, and I think the repetition of that confusion finally made me take a hard look at it. The UML is not only too complex, it seems to be incomplete (or at the very least, it takes far too many kinds of diagrams to express the design).*
>
> *The reason for the complexity of the UML is, ironically, inappropriate mixing of abstraction levels. [. . .]“*

Wir halten fest:

Konzept 3.40 (Gleichförmige Abstraktion)**.** *Die Abstraktionen, die in einer domänenspezifischen (Modellierungs)sprache Verwendung finden, sollten gleichförmig sein, d. h. sie sollten sich auf einer gemeinsamen Abstraktionsebene befinden.*

Eckel fährt in [Ecke05] fort:

> *„Because of all these misdirections, it's no wonder people get confused about what to do with the UML. Is it for modeling, or is it for detailed design? My guess is that forces within Rational wanted it to be everything to everyone, but this goes back to my original point: a method must be really, really simple in order for people to use it without getting lost.*

> *Two things became clear about the UML [. . .] First, when walking through use cases/user stories, the static structure diagram is what appears. This is not a revelation; CRC cards or simply hunting for nouns and verbs produce the same results. However, we found that a very simple form of structure diagram would suffice, and this seemed to be very helpful for people because they could see that it was only intended to capture the domain model, and not express every possible detail about the code. Without this guidance, people can waste a lot of time filling in all the methods, arguments and fields and lose sight of the big picture.*
>
> *The second thing that became clear was that a UML structure chart is a very helpful step in expressing the domain model, but it lacks an essential ingredient: mechanism. The structure chart only expresses the relationships between classes, not how they interact. Sometimes this relationship is enough, along with some words and hand-waving, to understand how the system will work. But, I began to see, that is only in the case where the mechanism is so obvious that everyone can see it. (I am aware that the UML has a various number of different diagrams that show different aspects of mechanism, but none of them are able to sum it up.)“*

Interessante Schwächen offenbaren sich bei MDA-basierten Systemen in der Modellierung nichtfunktionaler Anforderungen: UML (und andere von der MOF abgeleitete Verfahren) bieten keine expliziten Ausdrucksmittel für nichtfunktionale Anforderungen wie Mindestdatendurchsatz, maximale Berechnungsdauer bei Algorithmen, ergonomische Anforderungen, etc. Dies kann die MDA aufgrund des überaus generischen und sehr umfassenden Ansatzes auch nicht leisten. Zwar gibt es die Möglichkeit, durch Tagging oder spezielle Profile nichtfunktionale Anforderungen auszudrücken, doch ist hier die Werkzeuguntestützung nur rudimentär und komplexere Sachverhalten lassen sich nicht ohne Weiteres beschreiben. Gegenwärtig sind Modellierungswerkzeuge nach unserem Kenntnisstand im Wesentlichen darauf beschränkt, funktionale Anforderungen optisch abzubilden. Ein wichtiger Ansatzpunkt also für neue Modellierungsverfahren, die domänenspezifischer ausgerichtet sind und somit auch domänenspezifischere nichtfunktionale Anforderungen als inhärenten Modellbestandteil ansehen.

Außerdem verstößt die Unified Modeling Language (UML) gegen elementare Prinzipien der kognitiven und experimentellen Psychologie: Insbesondere die experimentelle Psychologie hat in Experimenten nachgewiesen, dass Menschen im Mittel lediglich sieben bis neun nicht miteinander verwandte Konzepte integrieren können (sie-

he [Tayl95], S. 25 bzw. [Tayl82])[24]. Die UML dagegen offeriert gegenwärtig nicht nur 13 Diagrammtypen, sondern auch je Diagramm üblicherweise eine erhebliche größere Zahl von Elementen – insbesondere bei den am häufigsten verwendeten Diagrammen wie Klassen-, Aktivitäts- und Sequenzdiagrammen (siehe [Kech06]). Diese erwähnten kognitiven Regeln betreffen alle Bereiche der menschlichen Wahrnehmung, so dass das grundlegende Problem stetig präsent ist. Wir werden eine effektivere Form der Modellbildung und -komposition in Abschnitt 5.4.3.8.4 ab Seite 223 vorstellen und halten daher fest:

Konzept 3.41 (Kognitive Zugänglichkeit)**.** *Modelle sollten in Hinsicht auf kognitive Zugänglichkeit entworfen werden, d. h. sie sollten eine überschaubare Menge möglichst widerspruchsfreier und klarer semantischer Konzepte mit einem direkten Domänenbezug enthalten.*

Zudem hat sich gezeigt, dass das Denken in Objekten der menschlichen mentalen Modellrepräsentation von realen Entitäten sehr nahe kommt (siehe [Tayl95], S. 47). Die UML macht hier aber einen Schritt rückwärts, da hier zum Beispiel Entitäten erster Klasse wie „Class“ verwendet werden (im Gegensatz zu realen Entitäten erster Klasse wie „Auto“). Dies entspricht dem Unterschied in der Wahrnehmung zwischen „das ist ein Konzept mit den Eigenschaften eines Autos“ und „das ist ein Auto“. Die erste Formulierung ist formal korrekter, intuitiv aber komplizierter. Effektiv erschwert der generische und domänenunspezifische Charakter von UML auf diese Weise den Modellierungsprozess für Domänenexperten.

Weiterhin wird die MDA dafür kritisiert, eine zu fokussierte Sicht auf Modelle zu haben (siehe [Kent02]): Die MDA betrachtet lediglich plattformunabhängige und plattformspezifische Modelle, es gibt aber eine Vielzahl weiterer Dimensionen, die unsere Art, zu modellieren, erheblich beeinflussen (siehe den nachfolgenden Abschnitt 3.3.6 zu MDE für ausführlichere Details hierzu).

[24] Wir müssen aber auf die Kernkritik an diesen Experimenten hinweisen, dass menschliches Erinnerungsvermögen nicht nur von Kurz- und Langzeitgedächtnis geprägt wird, sondern viele andere Kontextfaktoren (wie Kultur, Alter, Erfahrung) eine wichtige Rolle spielen. Dennoch erscheint es uns sinnvoll, darauf hinzuweisen, dass Darstellungen geringer Komplexität leichter zu verstehen sind als komplexe Darstellungen – was zunächst wie eine Trivialität erscheinen mag, im Angesicht der möglichst einfachen Darstellung komplexer Fragestellungen und Kontexte aber durchaus nicht trivial ist. Die damit verwandten Experimente und Forschungen gehen auf [Mill56] zurück, die relevante Kritik findet sich u. a. in [Jone02, Tuft03, Reyn08].

Iseger fasst in seinen Ausführungen zum Ansatz des Domain-Specific Modeling (DSM, siehe [Iseg05] bzw. Abschnitt 3.3.5) wesentliche Kritikpunkte am MDA-Ansatz wie folgt zusammen: *„The Object Management Group (OMG) promotes Model-Driven Architecture (MDA), a generative method that comes down to transforming one UML model into another, possibly several times and possibly automatically. At each stage you edit the models in more detail and in the end you generate substantial code from the final model [...]. These translations are largely in the hands of the tool vendor and will therefore not be optimal for each specific case. The main difference however, concerns when to apply which method. DSM requires domain-expertise, a capability a company can achieve only when focusing on working in the same application domain. These are typically product or system development houses more than project houses. Devices with embedded software in automotive and medical segments come to mind immediately but also configuration of CRM, ETL, Business Process Management and workflow systems are, like many others, good candidate domains. In these situations, companies get significantly more benefit by adopting DSM instead of MDA. Industrial experiences consistently show productivity improvements of between 500% and 1000% [for DSM]. Some reports from early implementations of MDA quote 35% at most [...]. MDA requires a profound knowledge of its methodology, something which is external to the company and has to be gained by experience. There where domain-expertise is usually already available and applied in an organization, MDA-expertise usually has to be found outside the organization.*

Studien weisen ungeachtet aller Kritik auf ca. 35% Produktivitätsgewinn bei der Verwendung von MDA gegenüber klassischen Ansätzen hin (siehe [RiVa03]).

Zusammenfassend liefert die Betrachtung der MDA für unseren Lösungsansatz also keine wesentlichen neuen Erkenntnisse: Modellbasiertes Vorgehen prinzipiell verspricht Effizienzgewinne und eine Abstraktion von konkreten Technologien, die für die Systemevolution hilfreich ist. MDA und UML im Speziellen werden gemeinhin als zu komplex und schwerfällig angesehen – insbesondere für kleinere Projekte. Sie können also im Normalfall keinen Beitrag zur Steigerung der Effizienz in mittelständischen IT-Projekten liefern. Allerdings stellt UML eine Vielzahl von bewährten Diagrammtypen bereit, die sowohl mittelbar als auch unmittelbar in domänenspezifischeren Ansätzen verwendet werden können, wenn sich dies anbietet.

3.3.2 Model Driven Software Development (MDSD)

Ziel von MDSD ist es laut [StVö05], S. 4 *„domänenspezifische Abstraktionen zu finden und diese einer formalen Modellierung zugänglich zu machen.“* Dazu werden die folgenden drei Dinge benötigt (wiederum zitiert aus [StVö05], S. 4):

- Domänenspezifische Sprachen, um die Modelle überhaupt formulieren zu können.
- Sprachen, in denen die notwendigen Modell-zu-Code-Transformationen ausgedrückt werden können.
- Compiler, Generatoren oder Transformatoren, die in der Lage sind, die Transformationsdefinitionen anzuwenden, um aus domänenspezifischen Modellen Programme zu erzeugen, die auf vorhandenen Plattformen ausgeführt werden können.

Als wesentliches Abgrenzungsmerkmal zwischen MDSD und MDA gilt, dass MDA primär die Interoperabilität von Werkzeugen und Standardisierung von Modellen anstrebt während MDSD auf praktisch einsetzbare Bausteine für den Softwareentwicklungsprozess abzielt, die auf heute verfügbaren und anwendbaren Technologien beruhen (siehe [StVö05], S. 5).

Zur Modellierung werden beim MDSD-Ansatz *Domain Specific Languages (DSLs)* (siehe Abschnitt 3.2.8) verwendet, die textuell oder grafisch das jeweilige Domänenproblem auf einer möglichst hohen Abstraktionsebene beschreiben. DSLs müssen im Gegensatz zum MDA-Ansatz nicht zwingend auf der Meta Object Facility (MOF) beruhen, auch wenn dies gegenwärtig häufig der Fall ist. Prinzipiell kann aber jede Form abstrahierender Darstellung verwendet werden, die angemessen erscheint. Somit handelt es sich bei den DSLs nicht um ein neues Konzept, sondern lediglich um die explizite Ausformulierung einer pragmatischen Vorgehensweise, die seit Einführung der strukturierten Programmierung in der Informatik vorzufinden ist: Durch bestimmte Mechanismen wird von Details abstrahiert. Unter diesem Gesichtspunkt können selbst Sprachen wie *awk* (siehe [AhKW88]) oder *HTML* (siehe z. B. [Wiki06n]) als DSLs angesehen werden – auch wenn hier eine sehr große Nähe zu Programmiersprachen existiert und in diesen Fällen somit die technische Abstraktion vor der fachlichen Abstraktion steht. Für Einzelfälle mag es gar sinnvoll sein, eine DSL direkt in einer Programmiersprache zu formulieren (siehe z. B. die Beispiele zu Ruby/Rake in [Schm05]). Da es hier aber vornehmlich um die abstraktere Modellierung und Formulierung sehr technischer Anwendungsfelder geht, in denen auch eine hohe technische Affinität (wenn nicht sogar personelle Äquivalenz) zwischen

fachlichen und technischen Projektmitarbeitern vorliegt, soll diese Thematik hier nicht weiter vertieft werden, da die eigentliche Zielgruppe unserer Arbeit hiervon nicht berührt wird[25].

Der Schwerpunkt bei der heutigen Verwendung von MDSD-Ansätzen liegt immer noch darauf, *projektspezifische* DSLs zu definieren, die repetitive oder fehleranfällige Bestandteile eines Systems durch bessere Abstraktionen ersetzen und es so ermöglichen, mit höherer Qualität zu entwickeln (siehe z. B. [Schm05]). Dadurch nutzt man unserer Ansicht nach aber nur einen Teil des tatsächlichen Potenzials von MDSD – insbesondere werden hier die „komplexen Ausnahmen“ in Projekten zu Lasten der viel häufiger auftretenden „hinlänglich bekannten“ Standardsituationen überbetont. Die Standardprobleme werden dann üblicherweise über MDA-Ansätze abgebildet, wobei aber wiederum die fachliche Fokussierung zugunsten der bereits im Abschnitt über MDA erläuterten technischen Ausrichtung verloren geht (siehe Abschnitt 3.3.1). In diesem Zusammenhang wird auch von architekturzentriertem MDSD (AC-MDSD) gesprochen (siehe [StVö05]). Auf die Konstruktion von DSLs soll hier zunächst nicht weiter eingegangen werden – allgemeine Hinweise finden sich z. B. in [Evan04a]. Letztlich führt diese Aufteilung auch dazu, dass es aufgrund eines latenten Mangels an Werkzeugunterstützung für diese speziellen DSLs (siehe z. B. [Dmit06]) immer wieder repetitiver Aufwand in die Entwicklung entsprechender Werkzeuge investiert wird. Unterstützung hierzu kristallisiert sich zunehmend z. B. in Form von Frameworks wie der *openArchitectureWare* heraus (siehe z. B. [OAW, StVö05]). Letztlich wenden sich aber auch diese Ansätze nur an die Symptome des Problems („zu starke Fokussierung auf das einzelne Problem“) und nicht auf die eigentliche Ursache des Problems („zu starke technische Fokussierung“).

Zum Nutzen von MDSD gibt es inzwischen eine Reihe von Studien und Artikeln – häufig direkt aus dem MDA-Umfeld, wobei sich in einigen Artikeln interessanterweise deutlich erkennen lässt, dass nicht MDA sondern eigentlich MDSD der entscheidende Erfolgsfaktor war (so zum Beispiel in [Gruh05]). Auch zeigt sich bei Softwarehäusern mit einer eigenen Produktpalette, dass im Verlauf der praktischen Entwicklung eines Frameworks der Schwerpunkt von MDA-Werkzeugen sich häufig in Richtung von MDSD-Plattformen verschiebt, bei denen eine eingeschränkte und gegebenenfalls sogar angepasste Menge von UML-Diagrammen mit einer selbst entwickelten und auf die Zielsysteme optimierte Ausführungsplattform verbunden und

[25] Man könnte auf „Literate Programming“ (siehe [Knut92]) oder „Fluent Interfaces“ (siehe [Fowl05a, Cawl05] bzw. Abschnitt 3.3.4) als Methoden zurückgreifen. Da wir aber einen Weg suchen, direkt mit Fachexperten zusammenzuarbeiten, scheiden solche programmiernahen Methoden aus (siehe dazu unsere Ausführungen in Abschnitt 3.1.5).

sogar direkt ausgeführt werden können (siehe z. B. [Krau05]). Praktische Projekte zeigen, dass dieser Ansatz sehr vielversprechend ist – ein hoher Bezug zum konkreten Problem, eine relativ einfache Möglichkeit zur Definition der Transformationen und eine Beschränkung auf das Wesentliche (den Weg vom Modell zum ausführbaren System) können auch kurzfristig und ohne übermäßige Investitionen in vollkommen neue Technologien und Methoden zu Erfolgen führen (siehe [Krau05, Ther05]).

Zusammenfassend lässt sich also sagen, dass der MDSD-Ansatz durch eine pragmatischere Anwendung der in Abschnitt 3.3.1 geschilderten Prinzipien, durch Verzicht auf die zwar formal vollständigeren aber auch erheblich aufwändigeren Anforderungen an eine durchgängig formal fundamentierte Modellbildung und durch den Verzicht auf Reverse Engineering eine deutlich schlankere Alternative zum klassischen MDA-Ansatz bietet. Diese Optimierungsideen werden wir direkt aufgreifen und in unsere Forderungen an CMDSD-Systeme (siehe Kapitel 5) einarbeiten:

Konzept 3.42 (Modellgetriebene Vorgehensweise)**.** *Modelle sind der Ausgangspunkt für alle Arbeiten, Systeme werden aus Modellen generiert. Der umgekehrte Weg hat für unsere weiteren Betrachtungen gemessen am erforderlichen Aufwand keine praktische Relevanz.*

3.3.3 Software Factories

Software Factories haben in der jüngeren Vergangenheit insbesondere aufgrund von [GSCK04] viel Aufmerksamkeit erhalten. Sie beschreiben einen Ansatz, um Anwendungen schneller zur Marktreife zu bringen und basieren auf der Erkenntnis, dass sich das folgende wiederkehrende Muster in den meisten Ansätzen zur Automatisierung von Software finden lässt:

1. Ein Framework wird entwickelt, auf dessen Grundlage Produkte mit einer gemeinsamen Architektur ausgeprägt bzw. konfiguriert werden.
2. Eine Sprache wird definiert, um die Konstruktion von Anwendungen zu unterstützen.
3. Der Konstruktionsprozess wird so weit wie möglich durch Werkzeuge automatisiert.

4. Die Werkzeuge werden von Entwicklern und Anwendern verwendet, um durch inkrementelle Entwicklung auf Anforderungen zu reagieren und so passende Lösungen zu generieren.

Wir finden diesen Ansatz z. B. auch bei Language Workbenches (Abschnitt 3.1.6) und Software-Produktlinien (Abschnitt 3.2.6). Die Autoren stellen fest, dass gegenwärtig diese Form der Automatisierung aber immer noch sehr kostenintensiv ist und daher neue Wege erforderlich sind, um diesen Ansatz nicht nur für sehr breite horizontale Domänen wirtschaftlich zu machen, sondern beliebige Domänen abdecken zu können (siehe [GSCK04], S. xvii). Insbesondere erwarten die Autoren analog zu unseren Ausführungen sehr große Produktivitätsgewinne durch die Fokussierung auf engere vertikale Domänen (siehe [GSCK04], S. xviii) und bestätigen dadurch das von uns identifizierte Optimierungspotenzial der Domänenfokussierung aus Abschnitt 2.2.2. Sie stellen zudem fest, dass der schnell wachsenden Komplexität von Systemen eine relativ konstante (und sehr kleine) Menge extrem talentierter Entwickler gegenübersteht und die große Masse an Entwicklern eben „nur durchschnittlich ist“. Um die Produktivität also nachhaltig zu erhöhen, gilt es für diese breite Masse an Entwicklern wesentlich leistungsfähigere Werkzeuge zu erschaffen, die sie besser und auf höherem Abstraktionsniveau unterstützen (siehe [GSCK04], S. 5). Wir vermerken dies:

Konzept 3.43 (Massentaugliche Werkzeuge)**.** *Werkzeuge müssen anstreben, die Produktivität der breiten Masse, also der Mehrzahl der durchschnittlichen IT-Fachkräfte, zu erhöhen, da sich nur so weitere großflächige Einspar- und Optimierungspotenziale realisieren lassen.*

Die größte Herausforderung sehen Greenfield und Short in der dezentralen Implementierung von Features (d. h. eine fachliche Anforderung wird erst durch das Zusammenspiel einer Vielzahl unterschiedlicher und komplexer Technologien realisiert). Sie erhöht die Hürden für die Verständlichkeit einer Lösung stark (also die Frage, welche Komponenten in Summe für ein Feature notwendig sind) und erschwert somit die Rekonstruktion von Entwürfen deutlich ([GSCK04], S. 38–41). Greenfield und Short kommen zu dem Schluss, dass Software-Entwicklung in weiten Teilen immer noch auf der falschen Abstraktionsebene arbeitet. Sie beschreiben Wege, um den Grad der Abstraktion zu erhöhen, stellen aber auch fest, dass es immer wieder Probleme geben wird, die sich nicht nur durch die vorgegebenen Abstraktionen lösen

lassen sondern auch gezielte Erweiterungen erfordern ([GSCK04], S. 54). Vor allem stellen sie auch fest, dass visuelles Modellieren mit abstrakten Modellen zu einer nachhaltigen Effizienzsteigerung führt und unterstützen so unsere These vom Wert modellbasierter Generierung von Systemen (siehe [GSCK04], S. 75–79 sowie unsere früheren Ausführungen in Abschnitt 2.2.3):

Konzept 3.44 (Visuelle Modellierung)**.** *Insbesondere klare visuelle Modellierung führen zu einer deutlichen Produktivitätssteigerung.*

Nach einer ausführlichen Detailproblemanalyse (siehe [GSCK04], S. 35–139) schlagen sie schließlich modellbasierte und domänenspezifische Sprachen als den zentralen Mechanismus zur Steigerung der Entwicklungsefizienz vor. So generieren sie fertige Lösungen, bei denen es nicht möglich ist, den Code zu verändern. Ein Reverse Engineering der Modelle wird auf diese Weise vermieden (siehe [GSCK04], S. 139–154). Das deckt sich mit den von uns bereits identifizierten Synergien zwischen Domänenfokussierung und Automatisierung aus Kapitel 2.

Darauf aufbauend definieren sie Software Factories wie folgt ([GSCK04], S. 163): *„A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components.“* Letztlich sollen Software Factories also den Weg zum breitflächigen und einfacheren Aufbau und Einsatz von Werkzeugen für die domänenspezifische Modellierung von Lösungen ebnen – ähnlich wie beim Language-Workbenches-Ansatz (siehe Abschnitt 3.1.6). Greenfield und Short sehen hier einen bedeutsamen Wachstumsmarkt für große IT-Unternehmen, die die Ressourcen besitzen, derart grundlegende und generische Werkzeuge zu entwickeln, sich aber schwer damit tun, in einer Vielzahl von Spezialmärkten siginifikante Marktanteile zu besetzen. Sie werben daher dafür, dass entsprechend große und leistungsstarke IT-Anbieter daher die Entwicklung von Software Factories fokussieren sollten, um ihren Partnernetzwerken darauf basierend die Ausprägung domänenspezifischer Lösungen zu erleichtern und so sowohl ihre Marktpositionen auszubauen als auch ihre Partner deutlich zu stärken[26].

[26] Berücksichtigt man, dass die Arbeiten zu Software Factories massiv aus dem Microsoft-Umfeld vorangetrieben werden, ergeben sich große Potenziale für den Mittelstand, der traditionell Microsoft-basierte Lösungen aufgrund ihrer vergleichsweise günstigen Preisstruktur und guten initialen Zugänglichkeit in großem Maße nutzt.

Im Folgenden geht [GSCK04] dann ausführlich auf die Technologien, Konzepte und Formalismen ein, mit denen dieses Ziel erreicht werden soll (siehe S. 164ff.). Diese Themen wollen wir hier aber nicht weiter vertiefen, da sie uns zu sehr von unserem Problem entfernen.

Wir halten fest:

- Software Factories werden zukünftig den Rahmen zur Ausprägung domänenspezifischer Lösungsansätze bieten.
- Sie steigern die Entwicklungseffizienz durch eine nachhaltige Erhöhung des Abstraktionsgrades.
- Sie erlauben die Umsetzung von Lösungen durch die konsequent modellbasierte (und weitgehend visuelle) Modellierung domänenspezifischer Lösungen, die dann zur Generierung des eigentlichen Codes verwendet werden.
- Sie setzen statt auf MDA auf MDSD, d. h. Reverse Engineering ist kein erstrebenswertes Ziel, sondern die Modelle werden als die zentralen Entwicklungsartefakte behandelt.

Wir finden also aus der Argumentationssicht von großen IT-Anbietern gestützt weitere Argumente für die von uns aus mittelständischer Sicht identifizierten Optimierungspotenziale für IT-Projekte.

Interessanterweise gibt es einen gewissen Widerspruch in der zuvor vorgestellten Idee von Software Factories und den Ideen des *Convergent Engineering* (siehe [Tayl95] sowie unsere Betrachtungen dazu in Abschnitt 3.1.1): Sind Software Factories vornehmlich darauf ausgerichtet, die Entwicklungsleistung von Implementierern durch eine Domänenspezialisierung der Werkzeuge zu optimieren, hält Taylor diesen Ansatz für grundlegend problematisch. Eine Beschleunigung der Entwicklungszeiten für Systeme durch optimierte Werkzeuge für Implementierer zögert seiner Ansicht nach das Unausweichliche nur heraus: Ab einem gewissen Komplexitätsgrad wird es Entwicklern unmöglich, sich wandelnde Geschäftsanforderungen noch schnell genug durch Anpassungen der zugrunde liegenden IT-Prozesse abzubilden, so dass die Leistungsfähigkeit von Unternehmen geschwächt wird (siehe [Tayl95], S. 16). Interessant ist für unsere nachfolgende Definition von *Conceptual Programming* die Beobachtung, dass beide Lager darum bemüht sind, sowohl den Abstraktionsgrad der Entwicklungsprozesse zu erhöhen (Software Factories durch Domänenspezialisierung, Convergent Engineering durch eine Annäherung von Business Engineering und (objektorientiertem) Software Engineering) als auch die Kommunikationsschritte zwischen Domänen- und IT-Experten zu reduzieren. Diese Grundideen werden

wir in unserer Methodologie aufgrund ihrer Nachhaltigkeit als Kernfundamente aufnehmen. Ersteres haben wir bereits durch Konzept 3.23 ▷ 73 getan, letzteren Aspekt formulieren wir wie folgt:

Konzept 3.45 (Kurze Kommunikationswege)**.** *Die Kommunikationswege zwischen Umsetzern und Fachexperten müssen so kurz wie möglich sein, um alle Arten von Kommunikationsfehlern zu vermeiden und schnelles Feedback zwecks effizienter Evolution eines Lösungsansatzes zu ermöglichen.*

3.3.4 Generative Programmierung (GP)

Generative Programmierung (GP) definiert ihr Ziel wie folgt (siehe [CzEi00], S. 1): *„Generative Programming is about manufacturing software products out of components in an automated way, that is, the way other industries have been producing mechanical, electric, and other goods for decades.“* Auf dem Weg dahin sind nach Ansicht von Czarnecki und Eisenecker zwei Schritte besonders bedeutsam (siehe wiederum [CzEi00], S. 1): *„First, we need to move our focus from engineering single systems to engineering families of systems – this will allow us to come up with the 'right' implementation components. Second, we need to automate the assembly of the implementation components using generators.“* Bereits in der Einleitung ihres Standardwerks werden also auch unsere Thesen zu Automatisierung und Domänenfokussierung aus Kapitel 2 als zentrale Bausteine zukünftig effizienterer Verfahren identifiziert.

Generative Programmierung unterscheidet zwei Entwicklungszyklen ([CzEi00], S. 7): *„one for designing and implementing a generative domain model (development for* reuse*), and another for using the generative model to produce concrete systems (development* with reuse*)“*. Hier werden sofort Analogien zu Software-Produktlinien (siehe Abschnitt 3.2.6 sowie Konzept 3.33 ▷ 85) erkennbar. Der wesentliche Unterschied liegt darin, dass Generative Programmierung sich stärker darauf fokussiert, wie Variabilität, Flexibilität und Konfigurierbarkeit technisch realisiert werden können (z. B. durch Metaprogrammierung, der Verwendung aspektorientierter Ansätze, durch Literate Programming, usw.) während sich Software-Produktlinien stärker mit

der Bereitstellung eines organisatorischen Vorgehensmodells zur Definition, Realisierung, Pflege von Software-Produktlinien sowie der Erzeugung von Produktinstanzen befassen. Ein wesentlicher Vorteil generativer Programmierung ist es, die *economies of scope* (siehe dazu unsere Ausführungen ab Seite 39) auf Software-Entwicklung zu übertragen, weil durch die generative Vorgehensweise weniger Zeit und Aufwand erforderlich sind, um mehr Produktvarianten auszuprägen.

Generative Programmierung untereilt sich dann weiter in eine Betrachtung von Analyse- und Designmethoden zur Domänenbildung (siehe [CzEi00], S. 60–164) sowie in die Beschreibung von Technologien zur Implementierung generativer Ansätze (siehe [CzEi00], S. 165–568)[27].

Bei den Methoden zur Domänenanalyse ist insbesondere *Feature Modeling* hervorzuheben, das die Beschreibung der Variabilitäten von Systemfamilien sowie die Abbildung der Variabilitäten auf Modelle ermöglicht (siehe [CzEi00], S. 82–130). Wir werden darauf im Zusammenhang mit dem auf Feature Modellierung optimal angepassten Vorgehensmodell *Feature Driven Development* auf Seite 131 zurückkommen. Unter einem Feature versteht man eine wichtige Eigenschaft einer Konzeptinstanz, die es uns erlaubt, Gemeinsamkeiten oder Unterschiede differenziert zu betrachten (siehe [CzEi00], S. 83). Features werden durch einen aussagekräftigen und präzisen Namen identifiziert und über sogenannte Feature-Diagramme modelliert. Diese dienen zur Taxonimiebildung und helfen im Rahmen der Domänenanalyse, um wiederverwendbare oder konfigurierbare Anforderungen, die von Bedeutung für die möglichen Stakeholder einer Lösung sind, zu repräsentieren. Features werden auf allen Systemebenen (also sowohl bei fachlichen Anforderungen, in der Architektur, in Subsystemen und Komponenten und auf der Implementierungsebene) identifiziert. Feature-Modelle erweitern Feature-Diagramme um textuelle Bestandteile, die die Motivation von Features, ihre detaillierte Bedeutung, die wichtigsten Stakeholder für bestimmte Features, etc. identifizieren. Feature-Diagramme wiederum sind Bäume, die optionale und zwingende Features in einer Hierarchie beschreiben und es dadurch ermöglichen, die verschiedenen Dimensionen und Erweiterungspunkte zu charakterisieren. Es handelt sich um einen sehr leichtgewichtigen und gut verständlichen Ansatz, um Konzepte detaillierter zu beschreiben und so die Kernpunkte eines Systems zu identifizieren. [CzEi00] liefert auch sehr schöne Beispiele dafür, um wie viel ausdrucksstärker Feature-Diagramme für bestimmte Szenarien sein können als UML-Diagramme (siehe z. B. Seite 103–104). Wir halten fest:

[27] Schon das Seitenverhältnis belegt die Differenzierung zu Software-Produktlinien: Technologische Aspekte stehen im Vordergrund.

Konzept 3.46 (Feature-Modellierung zur Domänenabgrenzung)**.** *Feature-Modellierung sowie Feature Driven Development haben sich vielfach bewährt, um mit ihnen domänenspezifische Eigenschaften in ihren Grenzen und Möglichkeiten strukturiert zu beschreiben.*

Auf der technischen Seite wird generisches Programmieren, template-basiertes Programmieren, aspektorientiertes Programmieren, Generatoren, statische Metaprogrammierung in C++ sowie intentionale Programmierung (siehe auch Abschnitt 3.1.2) vorgestellt.

Zusammenfassend lässt sich sagen, dass generative Programmierung für unser spezifisches Anliegen eine Reihe wichtiger Beiträge liefert:

- Feature Modeling liefert eine wichtige Methode zur konzeptionellen Modellierung von Variabilitäten, um diese dann planvoll und strukturiert in eine Systemfamilie aufzunehmen. Insbesondere werden damit die Grenzen und Möglichkeiten der Systemfamilie klar beschrieben.
- [CzEi00] stellt eine Vielzahl von Technologien und Verfahren bereit, um generative Systeme effizient zu realisieren. Somit ist [CzEi00] eine exzellente Grundlage für mittelständische IT-Dienstleister, die sich mit diesem Thema im Rahmen der Umsetzung unseres in Teil II beschriebenen Vorgehensmodells beschäftigen wollen.
- Zudem liefern Czarnecki und Eisenecker in Anhang A von [CzEi00] (S. 721–736) einen guten Überblick über psychologische Erkenntnisse über Konzepte und interne Modelle, die Menschen verwenden, um Konzepte zu internalisieren und wieder abzurufen. Diese sind bedeutsam für die Art und Weise, auf die wir Domänen in Modelle überführen sollten, um eine möglichst hohe Zugänglichkeit zu erreichen.

Analog zu Konzept 3.39 ▷ 93 halten wir fest:

Konzept 3.47 (Generative Programmierung)**.** *Generative Programmierung liefert einen umfangreichen, gut zugänglichen, praktisch erprobten und als Standard etablierten Werkzeugkasten, um unterschiedlichste Formen der generativen Programmierung mit technischen Ansätzen zu unterstützen. Bei der Planung eines entsprechenden Systems sollten daher die dort beschriebenen Erkenntnisse zu Modellierungsansätzen, Generierungsverfahren und der Psychologie der Domänenmodellierung herangezogen und dediziert auf ihre Eignung geprüft werden.*

3.3.5 Domain-Specific Modeling (DSM)

Domain-Specific Modeling (DSM) adressiert viele der Themen, die wir auch in Kapitel 2 als Optimierungspotenziale für mittelständische IT-Projekte identifiziert haben. [Iseg05] gibt einen guten Überblick über DSM, den wir hier kurz zusammenfassen wollen.

DSM postuliert, die Abstraktionsebene modellbasierter Verfahren durch die Abkehr von Programmkonstrukten weiter zu steigern und stattdessen ausschließlich Konzepte aus der Domäne zu beschreiben. Die darunterliegende Implementierung und insbesondere die Auswirkungen auf die Codestruktur werden auf Modellebene gar nicht mehr betrachtet. Abbildung 3.2 auf Seite 113 zeigt ein Beispiel für ein solches domänenspezifisches Modell, in dem tatsächlich keine programmnahen Elemente mehr zu finden sind, weitere finden sich in [Tolv05]. Wesentliche Vorteile sind laut Iseger zum einen, dass Entwickler daran gehindert werden, Programmerweiterungen vorzunehmen, die für die Domäne irrelevant sind (da die entsprechenden Sprachmittel gar nicht zur Verfügung stehen) und andererseits im Gegensatz zur UML eine vollständige Codegenerierung möglich wird.

Die notwendigen Modellierungswerkzeuge entstehen auf Grundlage von Metamodellierungswerkzeugen – häufig wird in diesem Zusammenhang Metacase (siehe [Meta09]) erwähnt, was nicht verwundert, da auch viele der Artikel zu DSM von Angestellten des Unternehmens stammen. DSM-Anwender entwickeln zunächst auf Basis solcher Werkzeuge eine Modellierungssprache und realisieren dann plattformspezifische Lösungen auf Grundlage dieser Modelle. Auch hier sind natürlich

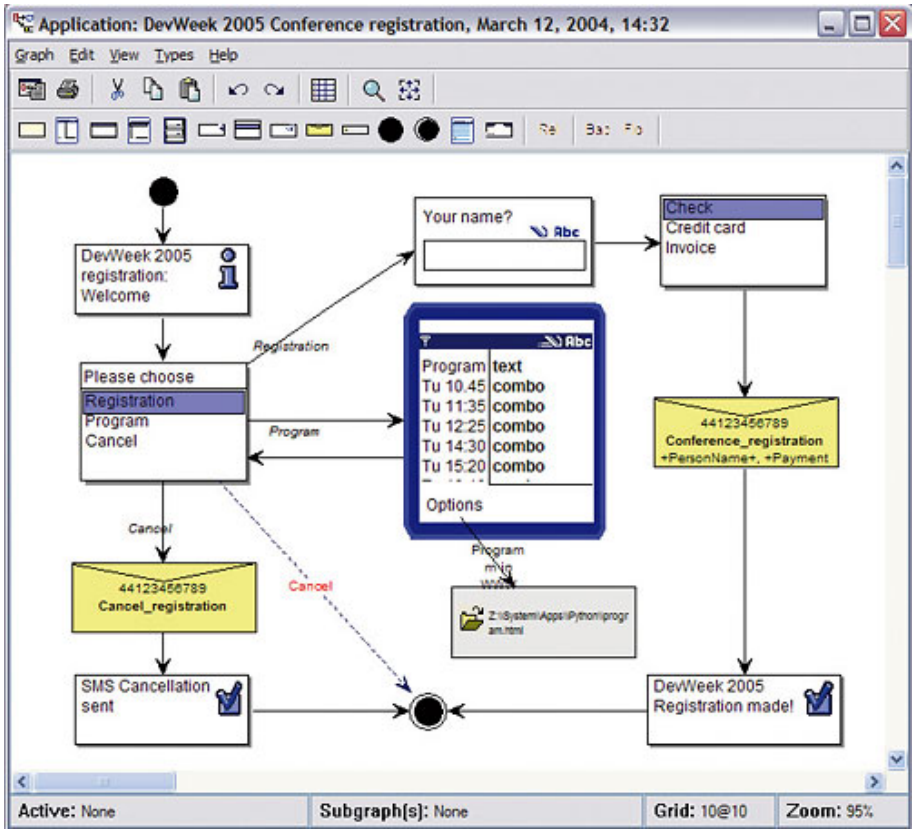


Abbildung 3.2: Beispiel für ein domänenspezifischen Modell (entnommen aus [Iseg05])

deutliche Ähnlichkeiten zu den bereits vorgestellten Language Workbenches aus Abschnitt 3.1.6 zu erkennen, was die Synergie-Effekte zwischen domänenfokussierten und modellbasierten Ansätzen (insbesondere auch im Zusammenhang mit dem von uns formulierten Potenzial von Kundenorientierung) noch einmal betont.

Ein wesentlicher Unterschied zu MDA-basierten Ansätzen ist, dass es weder die Möglichkeit (noch Notwendigkeit) zum Reverse Engineering gibt (bzw. geben muss) und die Erzeugnisse für Zwischenebenen nicht weiter manuell angereichert werden (müssen), um schließlich ein ausführbares System zu erhalten: Das gesamte notwendige technische Wissen über Implementierungsdetails und die Architektur ist in den abstrakten Modellen gekapselt. Ein großer Vorteil des DSM-Ansatzes gegenüber MDA ist, dass die verfügbaren Generatoren deutliche stärkere Ausdrucksmittel als die MDA-inhärenten Stereotypen bereitstellen, um domänenspezifische Modelle auszuprägen. Im Gegensatz zu MDA-Werkzeugen ist man von der Notwendigkeit befreit, Teile der Sprache zu deaktivieren, um nicht domänenbezogene Konzepte aus dem Modellierungswerkzeug zu entfernen (siehe [VMF09]).

Iseger zitiert Studien zu DSM, die auf Effizienzgewinne in der Entwicklung von 500–1000% hindeuten und zitiert dafür Beispiele von Nokia, Lucent und USAF.

Wir halten fest:

Konzept 3.48 (Exklusive Domänenmodellierung)**.** *Die Domäne ist so exklusiv wie möglich zu modellieren, d. h. Randaspekte und Infrastrukturthemen sind – wenn irgend möglich – nicht explizit in der Modellierung abzubilden. Geht die modellbasierte Generierung exklusiv von den Modellen aus, ist auch das Reverse Engineering von Anwendungen nicht von Interesse zur Ausprägung effektiver und evolutionsfähiger Lösungen.*

3.3.6 Model Driven Engineering (MDE)

Model Driven Engineering (MDE) ist ein von Stuart Kent geprägter Begriff (siehe [Kent02], der die Begriffe *Analyse*, *Prozess* und *Architektur* enger miteinander verbindet, als dies z. B. die MDA tut (siehe Abschnitt 3.3.1). Kent stellt zunächst fest, dass es mehr Dimensionen gibt, die für einen Modellierungsansatz relevant sind als dies z. B. die MDA vorgibt[28] – vor allem sieht er die MDA-Differenzierung nach PIM und PSM lediglich als eine mögliche Dimension (im Speziellen geht es um den Abstraktionsgrad der Modelle). Weitere denkbare Dimensionen sind das Thema (*subject*), also z. B. Kunden, Bestellungen, Administration, usw., betrachtete Aspekte (z. B. Sicherheit, Datenhaltung, Präsentation oder Workflows) sowie die betroffenen Zielpersonen (z. B. Entwickler oder Domänenexperten). Alle diese Dimensionen haben Einfluss auf die Gestaltung und Weiterverarbeitung von Modellen und können zu sehr unterschiedlichen Modellierungsansätzen führen, weswegen Kent die MDA auch aufgrund ihrer extrem auf PIM und PSM fokussierten Sicht kritisiert.

Außerdem ergänzt Kent den Prozessaspekt in der Modellierung. Modellierungsvorgänge setzen sich aus Makro- und Mikroprozessen zusammen (siehe [Kent02], S. 292ff.). Der Makroprozess beschreibt, in welcher Reihenfolge Modelle zu erstellen sind, der Mikroprozess legt fest, wie einzelne Modelle erstellt werden. Wir werden diesen Aspekt in Kapitel 6 aufgreifen, um für unseren Lösungsansatz ein leicht umzusetzendes Vorgehensmodell vorzugeben und halten fest:

[28] Diese unterscheidet im Wesentlichen zwischen plattformunabhängigen und plattformspezifischen Modellen, siehe Abschnitt 3.3.1.

Konzept 3.49 (Makro- und Mikroprozesse der Modellbildung)**.** *Der Makroprozess der Modellbildung beschreibt, in welcher Reihenfolge Modelle zu erstellen sind, der Mikroprozess legt fest, wie einzelne Modelle erstellt werden.*

Des Weiteren betont Kent die Bedeutung guter Werkzeugunterstützung, damit modellbasierte Verfahren ihre volle Wirkung entfalten können (siehe [Kent02], S. 292–294). Er stellt dabei fest, dass Entwickler kleine und gut zu integrierende Werkzeuge zu bevorzugen scheinen, was wir als Hinweis darauf nehmen, dass wir die Realisierung solcher Werkzeuge auch in einem mittelständischen Kontext erreichen können, wenn wir nur hinreichend aufgabenfokussiert vorgehen. Wir halten entsprechend fest:

Konzept 3.50 (Enger Rahmen)**.** *Entwickler bevorzugen überschaubare und möglichst gut auf ein Problem ausgerichtete Werkzeuge. Entsprechend sollten mittelständische Probleme unterstützende Werkzeuge versuchen, einen engen Problemkreis zu lösen, was sich auch positiv auf den Werkzeugentwicklungsaufwand selbst auswirkt.*

Kent schließt seine Überlegungen damit, dass domänenspezifische Sprachen und Modelle wiederum in einer Metasprache beschrieben werden müssen, damit es möglich wird, Werkzeuge auf Grundlage der Metamodelle automatisiert zu erzeugen (um z. B. die Validität der Modelle zu prüfen, etc.). Er gesteht aber gleichzeitig ein, dass es hier noch fundamentalen Forschungsbedarf gibt, um diesen Denkansatz zu praktischer Relevanz zu führen.

Wir werden den letzten Aspekt daher für unsere mittelstandsorientierten Betrachtungen außen vor lassen und stattdessen die Separation von Makro- und Mikroprozessen sowie das wiederkehrende Thema der Werkzeugunterstützung (das uns auch in den vorausgegangenen Betrachtungen immer wieder begegnet ist) aufgreifen, wenn wir unseren Lösungsansatz vorstellen.

Schmidt weist ergänzend in [Schm06] daraufhin, dass insbesondere CASE-Werkzeuge die Probleme eines generischen Modellierungsansatzes aufgezeigt haben: *„one size*

fits it all“ führt zu sehr schwer wartbaren Generatoren, ineffizienten Plattformen für eine große Zahl von Domänen und erschwert zudem die kollaborative Arbeit an Lösungen, da es keine klaren Abgrenzungen zwischen domänenspezifischen Aspekten gibt, die eine Parallelisierung von Arbeiten erlauben würden. Auch er sieht die Lösung in MDE und betont zwei der vier Optimierungspotenziale, die wir in Kapitel 2 auch als relevant identifiziert haben: Domänenfokussierung und Automatisierung durch Generierung. Er unterstützt unsere These, indem er darauf hinweist, dass die Realisierbarkeit von Generatoren drastisch zugenommen hat, weil die zugrunde liegenden Frameworks und APIs zur Lösung komplexer technischer Probleme erheblich ausdrucksstärker geworden sind als zur Blütezeit von CASE-Werkzeugen (siehe [Schm06], S. 27).

3.3.7 Business Process Modeling (BPM)

Business Process Modeling (BPM) befasst sich damit, Geschäftsprozesse zu analysieren, optimieren und formal zu beschreiben (siehe [Wiki09f]). Dazu existieren eine Reihe von Standards (siehe [KLLe09]), die alle Themenaspekte von der Analyse über die Beschreibung, Optimierung, das Reengineering u. v. a. m. abdecken. Eine Betrachtung zeigt, dass sich hier keine generellen Erkenntnisse für unseren Ansatz finden, es aber immer wieder wichtig ist, sich bewusst zu machen, wie umfangreich und erschöpfend existierende Standards nicht nur spezifische Domänen bereits behandeln (es also unnötig ist, das Rad neu zu erfinden), sondern dass es zu diesen Domänen auch umfassende (positive wie negative) Erfahrungswerte gibt, die bei der Entwicklung neuer Modelle und Ansätze genutzt werden sollten. Damit wollen wir bereits die Betrachtung von BPM schließen, da sie aufgrund mangelnder Verbindung zu unserem speziellen Ziel den Rahmen der Arbeit sprengen würden.

3.3.8 ARIS

Die von der IDS Scheer AG entwickelte Methode ARIS ist eins der weltweit führenden Systematiken zur Modellierung von Geschäftsprozessen (siehe z. B. [Andr05, ScJo02]) und steht heute für die „Architektur integrierter Informationssysteme“. ARIS wird häufig mit der gleichnamigen Produktfamilie der Firma IDS Scheer AG gleichgesetzt – auf das Werkzeug wollen wir aber nicht weiter eingehen. Die Methode ARIS bietet über 100 Modelltypen, um Geschäftsprozesse und ihre Zusammenhänge zu beschreiben (siehe [Lehm08]). Trotz ihrer sehr ausgereiften und umfassenden

Möglichkeiten ist ARIS für den mittelständischen Kontext aus folgenden Gründen kaum von Bedeutung:

- Die Lizenzkosten sind vergleichsweise hoch. So liegt die Arbeitsplatzlizenz für das grundlegende ARIS Toolset bei ca. 6200 EUR, weitere Kosten fallen für diverse Zusatzmodule an, so dass sich schnell Gesamtkosten von 8000–10.000 EUR pro Arbeitsplatz ergeben. Dazu kommen dann Schulungskosten, usw., so dass die von uns identifizierten typischen mittelständischen IT-Budgets (siehe Seite 17) meistens nicht reichen werden, um diese Kosten für die Modellierung aufzubringen.
- In der Praxis werden häufig die Unübersichtlichkeit und Unzugänglichkeit des Werkzeugs sowie die damit verbundenen Schulungs- und Akzeptanzprobleme bemängelt (siehe [Lehm08], S. 178).
- Es fehlt eine Möglichkeit, aus den ARIS-Modellen auch direkt Lösungen zu generieren. Die ARIS-Modelle müssen zunächst in andere Formen überführt werden und von dort weiterverarbeitet werden. Zum Beispiel bietet das ARIS Toolset Werkzeuge an, um aus den ARIS Geschäftsprozessmodellen UML-Diagramme und aus diesen wiederum Code zu generieren (siehe z. B. [Andr05]). Auch hier zeigt sich aber wieder die Schwäche des indirekten MDA-basierten Ansatzes in Kombination mit klassischer Geschäftsprozessmodellierung: Es sind zusätzliche technische Modellierungsebenen erforderlich, um aus einem von Fachanwendern verständlichen Modell ausführbare Systeme zu erzeugen. Einfacher ist die direkte Übersetzung, da der Abstraktionsgrad sowohl der Geschäftsprozessmodelle als auch der UML-Modelle keine wirkliche (automatische) Detailanreicherung erlaubt und stattdessen bewährte Standardtransformationen direkt aus dem Geschäftsprozessmodell ausführbare Systeme generieren könnten. Untersuchungen zeigen aber, dass dieser Weg sehr schwerfällig ist und selbst ARIS-Experten häufig dazu übergehen, die ARIS-Modelle als Vorlage zur Neuerstellung passender UML-Modelle zu verwenden (siehe [Eysh06]). Dies führt insgesamt zu Aufwänden und Irritationen, die nicht zu den von uns erhobenen Rahmenbedingungen mittelständischer IT-Projekte passen.

Zusammenfassend lässt sich daher sagen, dass ARIS ein Ansatz ist, der exzellent geeignet ist, um Geschäftsprozesse in Großunternehmen zu modellieren, für den mittelständischen Kontext in seiner heutigen Form aufgrund der damit verbundenen Aufwände weitgehend ungeeignet ist. Allerdings bietet ARIS eine wertvolle Grundlage für die Entwicklung spezialisierter domänenspezifischer Sprachen: Die über 100 Modelltypen (insbesondere z. B. eEPKs) bieten vielfältige Vorlagen für

reduzierte und neue Modelltypen zur Beschreibung ausgewählter Domänenaspekte und sollten somit durchaus zur Inspiration im Rahmen der Entwicklung von DSLs herangezogen werden. Auch wir werden im Rahmen der prototypischen Evaluierung unseres Lösungsansatzes auf ARIS als Inspirationsquelle zurückgreifen (siehe Abschnitt 8.1.8.1.2 ab Seite 342).

3.3.9 Agile Modeling (AM)

Agile Modeling ist kein vollständiges Vorgehensmodell, liefert aber vielfältige Hinweise zu leichtgewichtigen Modellierungsansätzen, die für unseren Lösungsansatz relevant sind. Agile Modeling geht auf Arbeiten von Scott Ambler zurück – er beschäftigt sich mit der Frage, ob und wie sich agile Ansätze und Methoden auf die modellgetriebene Entwicklung übertragen lassen (siehe [Ambl02a, Ambl06a, Ambl04b]). Ambler definiert Agile Modeling wie folgt: *„Agile Modeling (AM) is a chaordic, practice-based methodology for effective modeling and documentation of software-based systems. The AM methodology is a collection of practices, guided by principles and values, for software professionals to apply on a day-to-day basis. AM is not a prescriptive process. In other words, it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeler. AM is chaordic [...], in that it blends the 'chaos' of simple modeling practices and blends it with the order inherent in software modeling artifacts.“* ([Ambl02a], S. 8–9). Ambler verfolgt mit seinem Ansatz drei Hauptziele ([Ambl02a], S. 9–10):

1. Agile Modeling verbessert den Umgang mit Modellierungsansätzen, indem es eine Sammlung von Werten, Prinzipien und Methoden bereitstellt, die den Modellierer im effizienteren Umgang mit der Modellierung unterstützen. Durch diese Systematik wird eine schwer präzise zu fassende Methode sehr anschaulich und praxisorientiert dargestellt. Da wir dieses Muster auch bei anderen Vorgehensmodellen gefunden haben und bei weiteren finden werden, halten wir hier fest:

   > **Konzept 3.51** (Werte, Prinzipien und Methoden)**.** *Aufeinander aufbauende Werte, Prinzipien und Methoden sind eine sehr geeignete Systematik, um formal schwer zu fassende Vorgehensmodelle und Methoden mit ihren zentralen Zielen, charakteristischen Eigenschaften und wegweisenden Arbeitsmitteln umfassend zu beschreiben.*

2. Agile Modeling macht agile Vorgehensmodelle effektiver durch den gezielten und dosierten Einsatz von Modellierungstechniken.

3. Agile Modeling macht Modellierungsaktivitäten durch eine agilere Vorgehensweise effektiver.

Agile Modelle charakterisiert Ambler als ([Ambl02a], S. 12–14) zielgerichtet, verständlich, hinreichend präzise, konsistent, detailliert und so einfach wie möglich einen positiven Wert vermittelnd.

In Analogie zum Extreme Programming ([BeAn05]) greift Ambler dazu zunächst die vier Werte *Kommunikation*, *Einfachheit*, *Feedback* und *Mut* auf und ergänzt diese um den fünften Wert *Bescheidenheit*, um so ein Fundament für seine weiteren Ausführungen zu Agile Modeling und die dafür relevanten Prinzipien und Methoden zu schaffen ([Ambl02a], S. 19–26).

Anschließend definiert er 11 Kernprinzipien, die essentiell für agiles Modellieren sind ([Ambl02a], S. 27–37). Für unsere Betrachtungen sind dabei besonders die folgenden Prinzipien relevant:

- *Software Is Your Primary Goal.* Es geht darum, qualitativ hochwertige Software zu erzeugen. Es geht nicht darum, Dokumentation um der Dokumentation willen zu schreiben. Diese Gewichtung führt zu einer starken Fokussierung auf die Löungsentwicklung und vermeidet Ablenkungen durch „dokumentarische Pflichten“.
- *Travel Light.* Es sollte nur wirklich relevante Modelle und Dokumentation geben. Dadurch müssen nur die tatsächlich relevanten Artefakte verändert werden (mehr gibt es ja gar nicht).
- *Assume Simplicity.* Man sollte jederzeit davon ausgehen, dass die einfachste Lösung auch die beste Lösung ist. Das wird nicht immer richtig sein, aber in den Fällen, in denen komplexere Lösungen erforderlich sind, lohnt sich ein entsprechendes Investment auch. Die Implementierung komplexer Lösungen „in vorauseilendem Gehorsam“ führt nur zu einer Steigerung der Gesamkomplexität, die es wiederum erschwert, Änderungen einzubringen.
- *Embrace Change.* Änderungen sind ein alltägliches Fakt. Darum sollte man auf Veränderungen vorbereitet sein und Anforderungen so präzise verstehen, dass man ihre Relevanz und ihre Auswirkungen auf existierende Modelle auch bewerten kann.
- *Incremental Change.* Inkrementelle Änderungen erlauben eine allmähliche Evolution des Systems. Die so gewonnene Zeit hilft Domänenexperten und IT-Experten bei der präzisen Formulierung ihrer tatsächlichen Anforderungen und beim Aufbau des erforderlichen Problemverständnisses.

Wir sehen in diesen ausgewählten Prinzipien besonders wichtige Bausteine zur Formulierung eines mittelstandsorientierten Ansatzes, da sie helfen, Kosten zu reduzieren und die Kommunikation zu verbessern. Analog zu Konzept 3.39 ▷ 93 und 3.47 ▷ 112 halten wir fest:

Konzept 3.52 (Agile Modeling)**.** *Agile Modeling liefert das notwendige Handwerkszeug und ein generelles Vorgehensmodel, um die in Konzept 3.49 ▷ 115 beschriebenen Makro- und Mikroprozesse der Modellierung effektiv in der Praxis zu unterstützen.*

Ambler fährt fort, indem er unterstützende Prinzipien definiert ([Ambl02a], S. 38–43) und anschließend eine ganze Reihe von Methoden beschreibt, die das agile Modellieren ausmachen (S. 44-72). Uns erscheinen insbesondere die Kategorien seiner Kernmethoden von besonderer Relevanz. Diese sind unterteilt in:

- Iterative und inkrementelle Modellierung
- Teamarbeit
- Einfachheit
- Validierung

Insgesamt geht es darum, Lösungen schrittweise in enger Zusammenarbeit mit den Domänenexperten, basierend auf möglichst einfachen und klar verständlichen Modellen zu erarbeiten und sie dann so schnell wie möglich durch Codierung zu verifizieren und durch Tests abzusichern. Laut Ambler hilft es, existierende Standards wiederzuverwenden, nur das Wichtige zu dokumentieren und Modelle vor allem zur Förderung von Kommunikation und Verständnis zu verwenden, um die Produktivität zu steigern.

Im weiteren Verlauf von [Ambl02a] detailliert er diese Methoden und zeigt ihre Stärken durch Praxisbeispiele auf. Außerdem geht Ambler darauf ein, wie sich diese Methoden auf Extreme Programming und den Unified Process übertragen lassen und schließt letztlich mit Ausführungen, wie man typische Hindernisse bei der Einführung und Anwendung von Agile Modeling überkommen kann.

Zusammenfassend lässt sich sagen, dass Ambler einen Rahmen für eine agile und aufwandsminimierende Vorgehensweise in der Verwendung moderner Modellierungs-

techniken beschreibt. Er beruht auf langjährigen Erfahrungen und vielen Projekten.Wir werden bei der Formulierung unseres eigenen Lösungsansatzes insbesondere auf die oben formulierten Ideen zurückkommen.

3.4 Vorgehensmodelle

Die Beschreibung systematischer Vorgehensweisen zur Entwicklung von Softwareanwendungen wird in der Informatik mit dem Begriff „Vorgehensmodell“ (englisch: „Software Process Model“) bezeichnet ([BuKn02], S. 1). Insgesamt gibt es nur eine begrenzte Zahl von Vorgehensmodellklassen (auch Lebenszyklusmodelle genannt), von denen die drei für diese Arbeit zentralen Modelle in Abschnitt 3.4.1 vorgestellt werden (in enger Anlehnung an [BuKn02], S. 3ff.):

- Phasen-, Wasserfall- und Schleifenmodelle (Abschnitt 3.4.1.1)
- Prototypische Vorgehensmodelle (Abschnitt 3.4.1.2)
- Inkrementelle, evolutionäre, rekursive und iterative Modelle (Abschnitt 3.4.1.3)

Neben der Unterscheidung nach dem Lebenszyklus können Vorgehensmodelle dahingehend klassifiziert werden, ob sie planbasiert bzw. präskriptiv aufgebaut sind (siehe Abschnitt 3.4.2) oder ob sie agile Ansätze verfolgen (siehe Abschnitt 3.4.3). Da wir Agilität bereits als essentielles Optimierungspotenzial für unseren Ansatz identifiziert haben (siehe Abschnitt 2.2.4) sowie Konzept 3.35 ▷ 88), werden wir die planbasierten bzw. präskriptiven Vorgehensmodelle in Abschnitt 3.4.2 nur kurz benennen und ausschließlich die agilen Vorgehensmodelle in Abschnitt 3.4.3 eingehender betrachten.

3.4.1 Familien von Vorgehensmodellen

Jede der nachfolgenden Familien weist charakteristische Eigenschaften, Vor- und Nachteile auf. Für eine ausführlichere allgemeine Beschreibung sei auf [BuKn02], S. 3–16 verwiesen, auf das die Beschreibungen in den folgenden Abschnitten auch stark Bezug nehmen. Die einzelnen Familien werden nur sehr kurz vorgestellt, um auf die jeweils wichtigen Vorteile einzugehen, die auch für die Ausformulierung des *Conceptual Programming* Ansatzes von Bedeutung sind und dessen Struktur motivieren.

3.4.1.1 Phasen-, Wasserfall- und Schleifenmodelle

Phasen-, Wasserfall- und Schleifenmodelle zeichnen sich durch eine Einordnung aller Aktivitäten in Phasen aus. Diese Phasen werden in sequentieller Reihenfolge durchgeführt. Erst bei den späteren Modellen (Wasserfall / Schleifen) ist eine Rückkopplung auf frühere Phasen möglich, um neue Erkenntnisse, auftretende Probleme, etc. in den Entwicklungsprozess einzubeziehen. Das am weitesten bekannte Wasserfallmodell wurde in [Royc70] erstmals vorgestellt.

Typische Phasen sind Analyse, Entwurf, Implementierung, Integration, Installation und Einsatz (siehe [BuKn02], S. 3–7 für eine ausführlichere Beschreibung). Übergänge werden durch die Fertigstellung von vorgegebenen Artefakten (häufig Dokumente, am Ende das fertige System) in Form von Meilensteinen dokumentiert (siehe Abbildung 3.3).

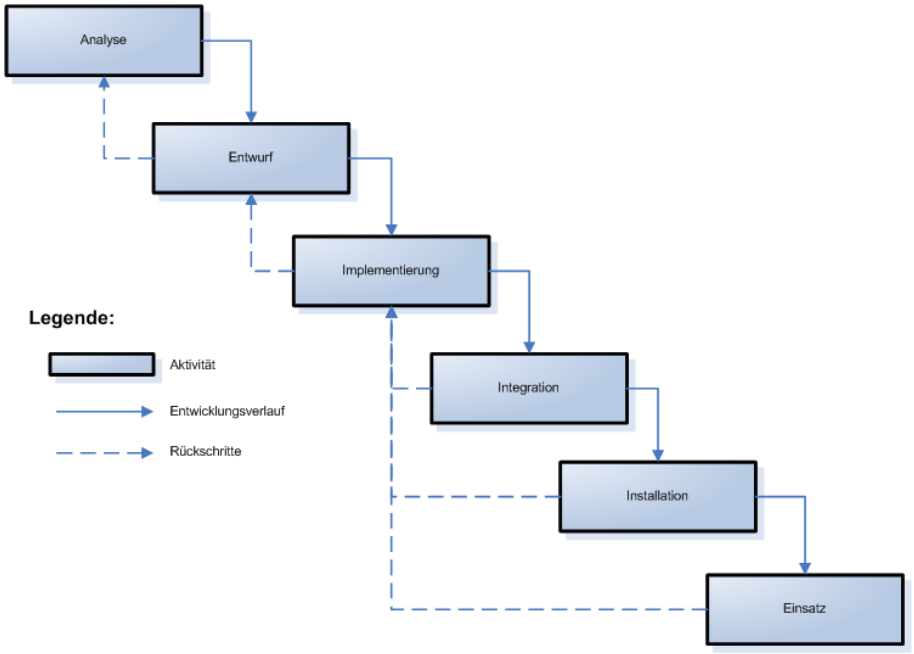


Abbildung 3.3: Wasserfallmodell (nach [BuKn02], S. 4)

Wesentliche Vorteile dieser Modellfamilie sind:

- die einfache Durchführung der Integration aufgrund der gründlichen Vorbereitung in den vorausgegangenen Phasen,
- ein einfaches Konfigurations- und Versionsmanagement aufgrund der geringen Zahl von Versionen und Konfigurationen.

Als nachteilig erweist sich, dass

- die Anforderungen für Projekte klar sein müssen und kaum Schwankungen unterliegen dürfen, da sonst die Analysephase nicht verlassen wird,
- sehr gute Kenntnisse von den verwendeten Methoden und Technologien vorausgesetzt werden, da es kaum Gelegenheit gibt, Erfahrungen aus späteren Phasen im Rahmen eines Projektes noch einmal gewinnbringend einzusetzen (siehe dazu [Tayl95], S. 145),
- feste Endtermine nur schwer eingehalten werden können, da Phasen vollständig abgeschlossen sein müssen, bevor mit der nächsten Phase begonnen werden darf. Kommt es zu unerwarteten Verzögerungen, wird dies zwangsweise dazu führen, dass bei Projektende kein fertig ausführbares System vorliegt.
- nur geringe Einsparpotenziale bei Termin- und Ressourcenproblemen existieren. Das Weglassen von Funktionalitäten ist nur bedingt effektiv, da die entsprechenden Funktionalitäten bereits in allen abgeschlossenen Phasen berücksichtigt wurden und somit bereits Aufwände erzeugt haben.

Insgesamt zeigt sich somit, dass dieser Ansatz sich nur gering für das typische mittelständische IT-Umfeld eignet. Es gibt weder ist die notwendige Flexibilität gegenüber Änderungen noch eine besondere Unterstützung für die Kommunikation zwischen Entwickler und Kunden. Einzig die klare Vorgehensweise als „Meta-Eigenschaft“ dieser Modellfamilie kann als positiv angesehen werden.

3.4.1.2 Prototypische Vorgehensmodelle

Prototypische Vorgehensmodelle zeichnen sich durch die wiederholte Bearbeitung einzelner Entwicklungsphasen aus (siehe [BuKn02], S. 7–11). Der bekannteste Vertreter dieser Familie ist das Spiralmodell von Boehm (siehe [Boeh88]). Dieses sieht neben der prototypischen Vorgehensweise auch noch einen iterativen Entwicklungszyklus vor, bei dem das entstehende Produkt schrittweise verfeinert wird. Damit gehört dieses Modell auch zur Familie der nachfolgend dargestellten inkrementellen, evolutionären, rekursiven und iterativen Vorgehensmodelle (siehe Abbildung 3.4 für ein Beispiel).

Interessant an den prototypischen Vorgehensmodellen sind die unterschiedlichen Formen von Prototypen, die erstellt werden können. Diese reichen von Papierskizzen für Eingabemasken bis hin zu ausführbaren Systemen mit eingeschränkter Funktionalität. Ziel von Prototypen ist, mit Anwendern möglichst realitätsnah und konkret die existierenden Anforderungen festzulegen, auf ihre Sinnhaftigkeit und Vollständigkeit zu überprüfen und letztlich auch die Machbarkeit abzuwägen.

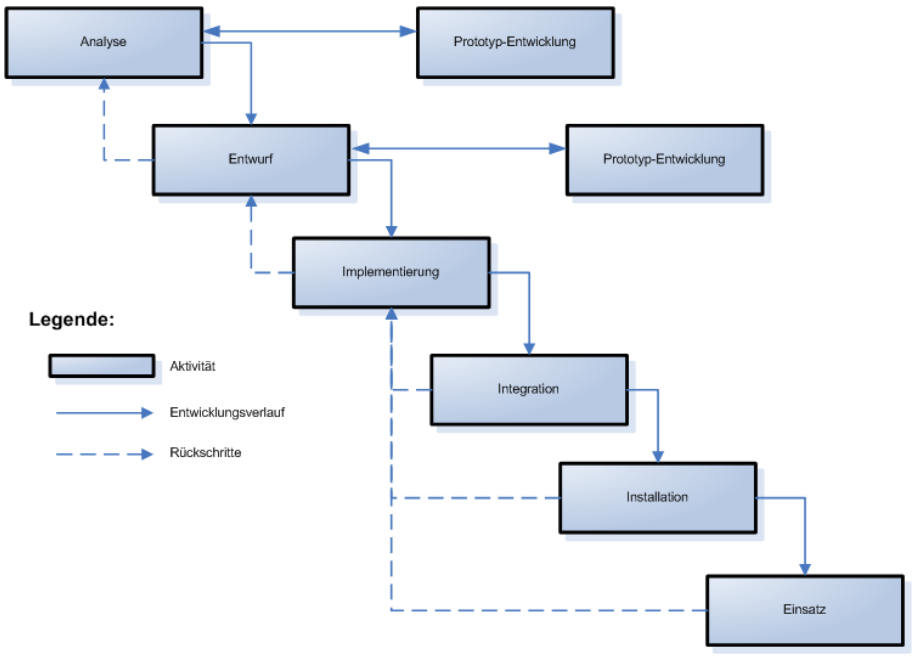


Abbildung 3.4: Prototypisches Vorgehensmodell (nach [BuKn02], S. 7)

Allgemeine Vorteile prototypischer Vorgehensmodelle sind:

- In der Integrationsphase sind kaum Probleme zu erwarten, da alle Anforderungen in vorangehenden Phasen erhoben und umgesetzt wurden. Prototypen sichern zudem, dass die nicht-funktionalen Anforderungen korrekt abgebildet werden.
- Es ist weder ein umfangreiches Versions- noch Konfigurationsmanagement erforderlich, da sich die Zahl der Versionen im Vergleich zur Familie der Phsen-, Wasserfall- und Schleifenmodelle (siehe den vorausgehenden Abschnitt) nur unwesentlich erhöht.
- Bei unklaren Anforderungen ist die Prototyperstellung sehr effektiv, um Kunden- und Benutzerwünsche zu konkretisieren und zu überprüfen. Zudem können instabile Anforderungen reduziert werden, da dem Kunden visuelle Repräsentationen seiner Anforderungen angeboten werden können und so die Entscheidung forciert wird.
- Entwurfs- und Implementierungsprototypen minimieren das Risiko für ein Scheitern eines Projektes, da so kritische technische Fragestellungen frühzeitig evaluiert werden können.

Die Familie hat die folgenden Nachteile:

- Wiederum sind fundierte Sachkenntnisse in den verwendeten Basistechnologien und Werkzeugen erforderlich, da Erfahrungen aus späteren Phasen wiederum nur schwer im Rahmen eines Projekts wiederverwendet werden können.
- Die technologischen Möglichkeiten zur Erstellung von Prototypen müssen geschaffen werden, erst recht, wenn diese ausführbar sein sollen.
- Feste Auslieferungstermine sind nur schwer einzuhalten, da zwar früh Prototypen entstehen, das endgültige System aber erst nach der vollständigen Abarbeitung aller Phasen bereit steht und dem Kunden durch das Vorhandensein von Prototypen möglicherweise sogar eine falsche Sicherheit vorgegaukelt wird.
- Es gibt analog zu den Phasen-, Wasserfall- und Schleifenmodellen nur geringe Einsparungspotenziale in den späteren Projektphasen.

Zentraler Vorteil ist die Idee ausführbarer Prototypen als effektivste aber auch aufwändigste Variante der Prototypenerstellung (siehe [BuKn02], S. 8). Sie wird im weiteren Verlauf der Arbeit Anwendung finden. Wir formulieren daher:

Konzept 3.53 (Ausführbare Prototypen)**.** *Generiere aus allen Modellen ausführbare Prototypen für das letztlich angestrebte System. Entscheide einzelfallbasiert, ob diese Prototypen weiterverwendet werden oder nur als Vorlage für nachfolgende Neuentwicklungen dienen.*

3.4.1.3 Inkrementelle, evolutionäre, rekursive und iterative Modelle

Inkrementelle, evolutionäre, rekursive und iterative Modelle zeichnen sich insbesondere durch die inkrementelle Vorgehensweise bei der Softwareentwicklung aus: Ausgehend von einer Teilmenge von Anforderungen werden Ergebnisse in sequentieller Reihenfolge erstellt (siehe auch [BuKn02], S. 11–15) ähnlich zu den Schleifen-, Phasen- und Wasserfallmodellen (siehe auch Abbildung 3.5).

Evolutionäre Vorgehensmodelle erweitern diese Systematik um Risikoanalysen, die eine effektivere Teilmengenbildung in den Folgeinkrementen erlauben. Rekursive Ansätze zeichnen sich durch einen besonderen Systembegriff aus: Sie verstehen darunter nicht nur das Gesamtsystem, sondern auch Teilsysteme, Teilsysteme von Teilsystemen, etc. Für all diese Systeme werden die einzelnen Entwicklungsaktivitäten durchlaufen.

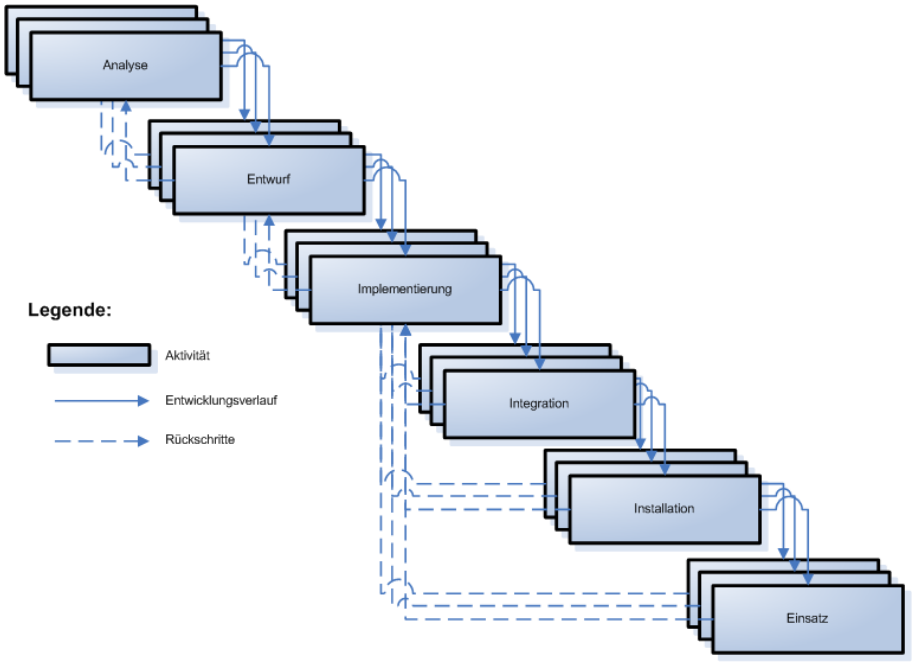


Abbildung 3.5: Inkrementelles Vorgehensmodell (nach [BuKn02], S. 12)

Die folgenden Vorteile zeichnen Mitglieder dieser Modellfamilie aus:

- Projekte können effizient begonnen werden, selbst wenn nicht alle Anforderungen stabil oder geklärt sind, indem man mit der Bearbeitung der stabilen Anforderungen startet.
- Erkenntnisse aus den ersten Inkrementen können in nachfolgenden Inkrementen nutzbringend eingesetzt werden, was insbesondere auch die Nutzung von Werkzeugen und Technologien ermöglicht, zu denen noch keine tiefgreifenden Erfahrungen existieren.
- Integrationstests werden sehr effektiv unterstützt, da die großen Schnittstellen im Rahmen der Tests immer wieder geprüft werden.
- Benutzer können schrittweise an das neue System herangeführt werden und müssen nicht sofort alle Bestandteile verwenden.
- Erfahrungen aus der Evaluierung und Nutzung früh fertig gestellter Inkremente können direkt in die Formulierung und Umsetzung späterer Anforderungen einfließen.
- Feste Endtermine können gut gehalten werden, da die fertig gestellten Inkremente bereits auslieferbar sind, während noch nicht realisierte Anforderungen mit geringer Priorität einfach weggelassen werden können. Sie sind dann tat-

sächlich aufwandsneutral.

Nachteilig für Vorgehensmodelle dieser Familie ist, das

- ein effektiveres Versions- und Konfigurationsmanagement benötigt wird, da sehr viele Systemversionen erzeugt und ggf. aufgrund später Erkenntnisse frühe Versionen ganz oder teilweise wieder hergestellt werden müssen.
- die Integration von Inkrementen bei schlecht partionierbaren Anforderungen sehr aufwändig sein kann.
- eine ausführliche Systemdokumentation benötigt wird, um in späteren Inkrementen konsistent auf den Entwurfsprinzipien der Vorgänger aufbauen zu können.

Für das nachfolgend zu definierende Konzept des *Conceptual Programming* erweist sich als besonders wichtig, dass bei Mitgliedern dieser Modellfamilie Anforderungen sehr flexibel modifiziert, konkretisiert und bewertet werden können, zum Ende eines jeden Inkrements bereits ein lauffähiges System existiert und in nahezu allen Phasen der Entwicklung Erfahrungen sowohl der Entwickler als auch der Anwender in die weiteren Aktivitäten einbezogen werden können. Wir formulieren also:

Konzept 3.54 (Inkrementelles, evolutionäres, rekursives und iteratives Vorgehen)**.** *Inkrementelle, evolutionäre, rekursive und iterative Vorgehensmodelle besitzen besondere Vorteile in der Anforderungsdefinition und schrittweisen Umsetzung. Daher sind diese Arten von Vorgehensmodellen für den von uns betrachteten mittelständischen Kontext besonders geeignet.*

3.4.2 Präskriptive oder plangetriebene Vorgehensmodelle

Präskriptive oder plangetriebene Vorgehensmodelle zeichnen sich insbesondere dadurch aus, dass sie aus der Perspektive eines Ingenieurs Software entwickeln (siehe [BoTu03b], S. 10ff.), was bedeutet, dass sie

- Software in definierten Prozessschritten entwickeln,
- die Prozessschritte dazu verwenden, um durch eine Serie von Repräsentationen vom Entwurf zum ausführbaren System überzuleiten,

- eine möglichst vollständige Dokumentation der einzelnen Prozessschritte erfordern, um jeden Einzelschritt verifizieren zu können.

Originär handelte es sich dabei häufig um wasserfallartige Vorgehensweisen, die in der jüngeren Vergangenheit aber immer mehr zu inkrementellen und evolutionären Prozessen adaptiert wurden, wobei selbst hier immer noch der Fokus darauf liegt, jeden Schritt präzise hinsichtlich seiner Auswirkungen auf Anforderungen, Design und Codierung zu dokumentieren, um so eine Nachvollziehbarkeit bzw. Nachverfolgbarkeit (*traceability*) des Prozesses zu gewährleisten.

Besonders wichtige Komponenten präskriptiver bzw. plan-getriebener Vorgehensmodelle sind

- Möglichkeiten für *Prozessverbesserungen*, indem es definierte Vorgehensweisen gibt,
- Beschreibungen der *Prozessleistungsfähigkeit*, so dass der Einsatz spezifischer Prozesse zu planbaren (d.h. vorhersagbaren und messbaren) Ergebnissen führt und die Steigerung der Prozessleistungsfähigkeit auch zu einer vorhersehbaren Steigerung der Ergebnisqualität führt,
- *Prozessgruppen*, die die Verantwortung für die Definition, Wartung und Verbesserung spezifischer Prozesse tragen,
- ein *Risikomanagement*, das auf planvolle und analytische Weise eine Identifikation und Quantifizierung von Projektrisiken ermöglicht und die Nutzung von Risikovermeidungsplänen ermöglicht,
- *Verifikationsmöglichkeiten*, um sicherzustellen, dass Arbeitsergebnisse die an sie gestellten Spezifikationen erfüllen (das Produkt also auf die richtige Weise gebaut wird),
- *Vaidationsmöglichkeiten*, die belegen, dass der Wert eines Produkts den Erwartungen entspricht (also das richtige Produkt gebaut wurde) und
- eine *Software-Systemarchitektur*, die eine Verbindung zwischen den Anforderungen der Stakeholder und den Lösungsschritten definiert, um die angemessene Erfüllung dieser Anforderungen sicherzustellen.

Die wichtigsten präskriptiven bzw. plangetriebenen Vorgehensmodelle sind das V-Modell [DHMi98, DrWi99, Wiki08a], das Capability Maturity Model Integration (CMMI), ([BoTu03b], S. 186), das Capability Maturity Model for Software (SW-CMM) ([BoTu03b], S. 188), PRINCE2 [LiRa08], Cleanroom ([BoTu03b], S. 192), Personal Software Process (PSP) ([BoTu03b], S. 190), Team Software Process (TSP)

([BoTu03b], S. 181), windexRational Unified Process (RUP) ([BoTu03b]m, S. 179)[29]. Aufgrund der mit dem Einsatz dieser Modelle verbundenen Aufwände müssen wir diese Vorgehensmodelle für den mittelständischen Kontext ausschließen und werden sie nicht weiter betrachten. Wir verweisen aber auf den Ausblick mit weiteren Gedanken zu diesem Themenfeld (siehe Abschnitt 10.3.6).

3.4.3 Agile Vorgehensmodelle

Agile Vorgehensmodelle sind eine Weiterentwicklung von Rapid Prototyping Ansätzen, die auf der Philosophie basieren, Software-Entwicklung mehr als Handwerk denn als industriellen Prozess anzusehen (siehe [BoTu03b], S. 16ff.). Enstanden sind diese Vorgehensmodelle aus den Anforderungen der Internet-Öknomie an eine besonders hohe Entwicklungsgeschwindigkeit und -flexibilität – Rahmenbedingungen, die nur schwer zu den klassischen planbasierten und präskiptiven Modellen (siehe Abschnitt 3.4.2) passen.

Generell handelt es sich bei agilen Prozessen um sehr leichtgewichtige Ansätze mit kurzen iterativen Entwicklungszyklen. Anwender werden aktiv und kontinuierlich in den Entwicklungsprozess einbezogen, um Anforderungen zu definieren, priorisieren und verifizieren. Das (stillschweigend) gesammelte Teamwissen ersetzt in weiten Teilen formale Dokumentation.

Boehm und Turner fordern in [BoTu03b], S. 17 für agile Prozesse die folgenden Eigenschaften:

1. Sie müssen über mehrere Zyklen iterativ ablaufen.
2. Sie müssen inkrementell arbeiten, dürfen also nicht das Endprodukt in einem Schritt liefern.
3. Sie müssen selbstorganisierend sein. Die Teammitglieder müssen also die Möglichkeit haben, die beste Vorgehensweise zur Erreichung der nächsten Ziele selbst festzulegen.
4. Sie müssen die Ausprägung neuer Prinzipien, Prozesse und Arbeitsstrukturen während des Projektes erlauben.

[29] Letzterer subsummiert mittlerweile eine ganze Reihe klassischer objektorientierter Vorgehensmodelle, die mit der Einführung und Etablierung des RUP an Bedeutung verloren haben bzw. durch diesen abgelöst wurden. In der Praxis sind die folgenden Modelle zuweilen aber immer noch zu finden und auch in der gängigen Literatur tauchen sie immer wieder auf: OMT, Booch, Objectory, Fusion, Catalysis und KobrA.

Je nach Vorgehensmodell werden konkrete Methoden definiert, die die angestrebten Werte des jeweiligen Modells fördern (siehe dazu auch Konzept 3.51 ▷ 118). Wichtige Konzepte agiler Ansätze sind u.a. (wiederum nach [BoTu03b], S. 18–19):

- *stetige Veränderungen als etwas Natürliches und zu Erwartendes zu begreifen* und diese entsprechend als positive Kraft zu begreifen, die die Kreativität der Entwickler stimuliert, den Produktwert für den Kunden steigert und somit durch das Modell inhärent berücksichtigt werden muss.
- *kurze Entwicklungszyklen und stetige Auslieferungen* als grundlegendes Paradigma anzusehen. Dadurch wird die Emergenz von Anforderungen forciert und sichergestellt, dass immer nur die nächstwichtigsten Funktionen für den Kunden realisiert werden, um so den Wert des Produkts zu maximieren.
- *einfaches Design*, indem nur die gerade in Entwicklung befindlichen Anforderungen berücksichtigt werden[30]. Da Anforderungen sich immer wieder ändern werden, macht es keinen Sinn, für eine ungewisse Zukunft zu planen.
- *Refactoring* als Mittel, um Software immer wieder so zu restrukturieren, dass sie die aktuellen Anforderungen hinreichend unterstützt. Dabei werden Code-Duplikate entfernt, Kommunikationsschichten optimiert, flexible Erweiterungspunkte ergänzt, etc. ohne dass sich das grundlegende Verhalten der Software ändert. Wichtig für die fortwährende Korrektheit des Systems ist dabei das Vorhandensein automatisch ausführbarer Testfälle.
- *Pair Programming*, um die Zusammenarbeit von Entwicklern zwecks Wissensaustausch und -aufbau zu optimieren.
- *Retrospektiven*, um nach einer Iteration Erfahrungswerte aufgreifen und zur Optimierung der nächsten Iterationen nutzen zu können (zuweilen auch als Reflexion bezeichnet).
- *Stillschweigendes oder implizites Wissen*, indem das Projektwissen in den Köpfen der Teammitglieder und nicht in expliziter Dokumentation stets auf den neuesten Stand gebracht wird.
- *testgetriebene Entwicklung*, um kurze Iterationszyklen sowie die Software-Evolution zu unterstützen und überhaupt erst zu ermöglichen (siehe dazu auch die vorausgegangenen Ausführungen zu Refactoring).

Zu den wichtigsten agilen Vorgehensmodellen gehören das V-Modell XT [RaBr06a, RaBr06b], eXtreme Programming (XP) [BeAn05, ErFa02], Adaptive Software Development (ASD) ([BoTu03b], S. 21), Crystal ([BoTu03b], S. 21), Feature Driven

[30] Das maßgebliche englische Akronym ist hier: *„YAGNI – You Ain't Gonna Need It"*.

Development (FDD) ([BoTu03b], S. 22 bzw. [PaFe02]), Scrum ([BoTu03b], S. 21) und Dynamic Systems Development Method (DSDM) ([Stap02]). Auch auf diese Verfahren wollen wir nicht im Detail eingehen, sondern verweisen lediglich auf die Literatur. Unsere prinzipiellen Betrachtungen zur Natur agiler Vorgehensmodellen reichen aus, um zusammen mit den grundlegenden Erkenntnissen zu kundenorientierten, domänenfokussierten und modellbasierten Verfahren zu einem integrativen Ansatz zu gelangen. Einzige Ausnahme soll das Feature Driven Development sein, dem wir kurz etwas stärkere Aufmerksamkeit schenken wollen, da man es als extrem schlanke und agile Variante des *Domain Engineering* von Software-Produktlinien betrachten könnte und wir Software-Produktlinien bereits als vielversprechenden Rahmen für einen neuen Lösungsansatz identifiziert haben.

Feature Driven Development (FDD) ist 1997 entstanden und ist ein agiles Projektmanagementmodell zur kundennutzenorientierten Realisierung von Projekten: *„Die Entwicklung wird anhand eines Feature-Plans organisiert. Eine wichtige Rolle spielt der Chefarchitekt (engl. Chief Architect), der ständig den Überblick über die Gesamtarchitektur und die fachlichen Kernmodelle behält. Bei größeren Teams werden einzelne Entwicklerteams von Chefprogrammierern (engl. Chief Programmer) geführt. FDD definiert ein Prozess- und ein Rollenmodell, die gut mit existierenden klassischen Projektstrukturen harmonieren. Daher fällt es vielen Unternehmen leichter, FDD einzuführen als XP oder Scrum. Außerdem ist FDD ganz im Sinne der agilen Methoden sehr kompakt. Es lässt sich auf 10 Seiten komplett beschreiben.“* (zitiert aus [Wiki08l], hier taucht auch wieder das von uns identifizierte Konzept 3.30 ▷ 80 des Chief Programmers auf). [Nebu09] präsentiert eine entsprechende 10-seitige Übersicht. FDD besteht im Wesentlichen aus fünf Schritten:

1. Entwickle ein Gesamtmodell.
2. Erstelle eine Feature-Liste.
3. Plane je Feature.
4. Entwirf je Feature.
5. Konstruiere je Feature.

Dieses einfache Modell greift viele der Eigenschaften klassischer Vorgehensmodelle auf, maximiert aber gleichzeitig den Kundennutzen, da jedes Feature bezüglich seiner Relevanz für die Anforderungen des Kunden bewertet und entsprechend priorisiert wird. Somit erhalten wir hier ein sehr einfaches und schlankes Vorgehensmodell, das wir in Kapitel 5 aufgreifen werden, um den Software-Produktlinienansatz auf

den mittelständischen Kontext zuzuschneiden. Für ausführlichere Beschreibungen zu FDD verweisen wir auf [PaFe02].

3.5 Zusammenfassung

Wir haben in diesem Kapitel den Stand der Technik zu den von uns in Kapitel 2 identifizierten vier Optimierungspotenzialen für mittelständische IT-Projekte (*Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität*) betrachtet und Stärken und Schwächen existierender Verfahren aufgezeigt. Insbesondere hat sich dabei gezeigt, dass es zwar eine Vielzahl sehr leistungsfähiger Ansätze gibt, ein tatsächlich integrierender Ansatz aber noch für den mittelständischen Kontext fehlt. Die existierenden Verfahren beschäftigen sich überwiegend mit Großprojekten von einer Dauer von mehr als zwei Jahren und mehr als 50 Teammitgliedern.

Nachfolgend werden daher die Stärken der zuvor beschriebenen Ansätze noch einmal unter Berücksichtigung der besonderen Anforderungen mittelständischer IT-Projekte zusammengefasst. Aus den Stärken leiten sich Konzepte ab, die bewährte Verfahren und erfolgreiche Ansätze konsequent weiterverfolgen. Diese Konzepte haben wir gesammelt und werden sie im folgenden Teil der Arbeit in Form konkreterer Werte, Prinzipien und Methoden ausarbeiten.

Modellbasierte und generative Ansätze weisen eine Vielzahl von (mindestens theoretischen) Vorteilen für Werkzeugentwürfe auf, von denen die wichtigsten hier noch mal genannt werden sollen:

- Die Qualität der Software steigt aufgrund konsequenter Wiederverwendung.
- Kosten für Software sinken, da Wiederverwendung schnellere Realisierungszeiten bedingt.
- Software-Ingenieure kommen in den Genuss anspruchsvollerer Aufgaben, da sie sich verstärkt auf die fachliche Modellierung konzentrieren können.
- Technologischen Veränderungen kann – zumindest in den neueren Ansätzen, die von der konkreten technischen Umsetzung abstrahieren – leichter begegnet werden, da sich zwar die Transformationen aber nicht zwingend die Modelle ändern.
- Aufgrund der Abstraktion von technischen Details sinken die Anforderungen an die technischen Fähigkeiten von Entwicklern. Allerdings steigen mindestens in gleichem Maße die Anforderungen an die Modellierungsfähigkeiten des Entwicklers.

Dennoch konnten sich diese Ansätze bisher bestenfalls in Nischen durchsetzen. Die Gründe liegen einer Reihe von Faktoren, von denen die wichtigsten hier noch einmal aufgeführt werden sollen:

- Die Fokussierung auf fachliche und kommunikative Fragestellungen erfordert eine andere „Gattung“ von Software-Entwicklern. Viele klassische Entwickler sind weder vorbereitet noch daran interessiert, sich weg von den technischen Fragestellungen hin zu den fachlichen Themen zu bewegen, was häufig zu einer lustlosen und unmotivierten Nutzung der entsprechenden Werkzeuge führt. Dadurch werden die Werkzeugpotenziale nicht vollständig ausgeschöpft und das Nutzenpotential kann sich nicht vollständig einstellen.
- Es gibt für viele der theoretischeren Ansätze nur eine unzureichende Werkzeugunterstützung. Mangelhafte Werkzeuge in Kombination mit komplexen neuen Anforderungen an das Verständnis der Entwickler führt dazu, dass neue Methoden unzureichend oder gar nicht in den Entwicklungsprozess aufgenommen werden.
- Höhere Aufwände in den ersten Prozessen halten Entscheider von der Auswahl neuer Entwicklungsverfahren ab. Der Druck zur Kostensenkung lässt nur selten Luft für kurzfristig höhere Investitionen in neue Technologien.
- Neue Ansätze wollen zu viel auf einmal. Sie werden häufig (zum wiederholten Male) als Wundermittel gegen alle möglichen Probleme verkauft. Da potenzielle Anwender aber bereits (zumeist leidvolle) Erfahrungen mit der Qualität früherer „Wundermittel“ gemacht haben, stellt sich eine grundlegend pessimistische Abwehrhaltung ein, die zu einer „Ablehnung aus Prinzip“ führen kann.
- Proprietäre Lösungen lassen Zweifel über die Zukunftssicherheit von Produkten aufkommen, so dass diese mglw. keine ausreichende Marktpräsenz erreichen können.

MDSD scheint im Vergleich zu MDA Kosten- und Geschwindigkeitsvorteile in der Lösungsumsetzung zu besitzen (siehe unsere Ausführungen in den Abschnitten 3.3.1 und 3.3.2), da dass wir MDSD-Entwurfsprinzipien eine wichtige Rolle beim Entwurf unseres Vorgehensmodells beimessen werden, Domain-Specific Modeling kann die uns höchsten bekannten Effiziensteigerungen vorweisen (siehe Abschnitt 3.3.5).

Überlegungen aus dem Bereich domänenfokussierter Vorgehensmodelle (insbesondere aus den grundlegenden Erkenntnissen von Evans zum Thema „Domain Driven Design“ ([Evan04a], Abschnitt 3.2.7) weisen auf Vereinfachungspotenziale bei der grundlegend modellbasierten Vorgehensweise hin. Die Modelle in Form domänen-

spezifischer Sprachen konzentrieren sich dabei auf eingeschränkte Problembereiche und erhalten hier eine höhere Ausdruckskraft und gleichzeitig aufgrund des eingeschränkten Problemkreises eine einfachere Struktur (siehe die Abschnitte 3.2.1, 3.2.3 und 3.2.8). Entsprechend werden sie eine wichtige Rolle beim Entwurf unseres Lösungsansatzes spielen. Außerdem haben Software-Produktlinien (siehe Abschnitt 3.2.6) einen Weg aufgezeigt, die von uns auf Seite 39 als wünschenswert identifizierten *economies of scope* über ein industriell bewährtes Vorgehensmodell zu realisieren, so dass wir sie ebenfalls in unserem Lösungsansatz aufgreifen werden. Feature Driven Development (FDD, siehe Seite 131) wird uns dabei helfen, die Methodik der Software-Produktlinien auf ein für den Mittelstand passendes Maß zuzuschneiden.

Mit analogen Überlegungen und Erfahrungswerten zur effektiveren Integration der Kunden in den Entwicklungsprozess beschäftigen sich die Arbeiten von Martin aus [Mart81] und die in Abschnitt 3.4.3 vorgestellten agilen Vorgehensmodellen. Wir haben die Notwendigkeit der engen Zusammenarbeit zwischen Kunden und Entwicklern herausgearbeitet, die bei der frühzeitigen Fehlererkennung und daraus resultierenden Kostenvermeidung von großem Wert ist. Agile Vorgehensmodelle werden demgemäß einen wichtigen Bestandteil unserer Lösung ausmachen.

Zudem zeigen Erfahrungen aus dem Domain Driven Design (siehe wiederum unsere Ausführungen in Abschnitt 3.2.7), dass großes Potenzial darin liegt, Modellierung und Implementierung zu eng verzahnten Tätigkeiten zu machen, insbesondere auch um den Erkenntnisgewinn zwischen Modellierern, Domänenexperten und Entwicklern zu beschleunigen. Diese Erfahrungen dienen uns als Grundlage, um nach einem vereinheitlichen Entwicklungskonzept zu suchen, das Modelle zum zentralen Entwicklungsartefakt macht.

Für einen optimierenden Ansatz ist es sinnvoll, Werkzeuge und Methoden als Teile eines Gesamtökosystems für die Software-Entwicklung zu betrachten. Für das Vorgehen, einerseits die eingesetzten Werkzeuge und andererseits die verwendeten Methoden zu betrachten, hatten bereits Martin und Weinberg ([Wein98, Mart81], siehe Abschnitt 3.1.7) entsprechende Ideen, die sich in ähnlicher Form immer wieder finden, z. B. auch im Convergent Engineering und der Convergent Architecure (Abschnitte 3.1.1 und 3.1.4).

Als Nächstes werden wir kurz umreißen, wie sich unser Lösungsansatz aus den identifizierten Konzepten ergibt. In Teil II der Arbeit wird unser Lösungsansatz dann detailliert vorgestellt.

3.6 Herleitung des CP-Lösungsansatzes

Basierend auf den Konzepten, die wir in den vorausgegangenen Abschnitten dieses Kapitels als wertvoll identifiziert haben, werden wir nun die prinzipielle Struktur unseres Lösungsansatzes herleiten. Im nachfolgenden Hauptteil II dieser Arbeit stellen wir dann die einzelnen Bestandteile detailliert vor. Dabei werden wir uns wieder auf die in diesem Kapitel identifizierten Konzepte berufen. Um redundante Erläuterungen zu vermeiden, werden wir diese Referenzen am Seitenrand andeuten. Beispielhaft weist die nebenstehende Randmarkierung daraufhin, dass wir das in Konzept 3.1 ▷ 48 beschriebenen Mittel der Konzepte verwendet haben, um die in diesem Abschnitt beschriebene Vorgehensweise zu motivieren.

K-3.1 ▷ 48

Zunächst einmal liefert Konzept 3.33 ▷ 85 die Motivation für die Grundstruktur unseres Lösungsmodells: Wir werden unter dem Oberbegriff des *Conceptual Programming (CP)* in Form des *Conceptual Model Driven Software Development (CMDSD)* das Domain Engineering zusammenfassen und analog dazu durch das *Concept Driven Project Management (CDPM)* ein Vorgehensmodell für das Application Engineering beschreiben. Beide Methoden werden in Kombination mit den allgemeinen CP-Prinzipien Steuermechanismen in Form eines Produktgremiums enthalten, um das synergetische Zusammenspiel sicherzustellen. Konzept 3.49 ▷ 115 legt dabei nahe, dass im Rahmen des CMDSD der Makroprozess der Modellbildung betrachtet wird (also die Frage nach den benötigten Modelltypen und ihrer aufeinander aufbauenden Struktur, aus der sich eine Modellierungsreihenfolge für den operativen Prozess ergibt). Der Makroprozess geht unmittelbar in die Makroebene des CDPM ein und wird dort um den Mikroprozess der Erstellung der einzelnen Modelle ergänzt.

Konzept 3.7 ▷ 61 indiziert die Notwendigkeit des CP als Metaebene für die Einzelbestandteile unseres Lösungsansatzes, die Konzepte 3.8 ▷ 62 und 3.51 ▷ 118 liefern uns eine Methodik, um unseren Lösungsansatz strukturiert durch eine aufeinander aufbauende und sich gegenseitig stützende und ergänzende Sammlung von zielgerichteten Werten, Prinzipien und Methoden zu beschreiben. Da dieses Vorgehen essentiell für das weitere Verständnis der Arbeit ist, werden wir ihm in Abschnitt 4.2 besondere Aufmerksamkeit widmen. Die Leistungsfähigkeit des prinzipiellen Vorgehens wird durch Konzept 3.9 ▷ 62 belegt, das auf Erfahrungswerten mit entsprechenden ganzheitlichen Ansätzen fußt. Konzept 3.10 ▷ 62 schließlich belegt die prinzipielle Machbarkeit dieser Vorgehensweise, da wir angehalten werden aus wirtschaftlichen Aspekten lediglich die meistverbreitetsten Fragestellungen zu betrachten und Sonderfälle durch sinnvolle Erweiterungsmöglichkeiten abzuhandeln. Zudem stützt Kon-

zept 3.11 ▷ 63 den strategischen Wert der initialen Investitionen, da eine hinreichend marktrelevante Domäne durch den potenziellen hohen Wiederverwendungsgrad mittelfristig zu drastischen Kostenersparnissen führen wird. Konzept 3.13 ▷ 64 legt fest, dass sowohl für CMDSD als auch CDPM ein strukturiertes Vorgehensmodell für den Entwicklungsprozess festzulegen ist.

Für das CMDSD-Vorgehensmodell motiviert Konzept 3.34 ▷ 87 einen produktlinienorientierten Ansatz, der gemäß Konzept 3.36 ▷ 89 eine einzelne Domäne fokussieren soll und diese in der praktischen Ausprägung von Lösungen gemäß Konzept 3.37 ▷ 90 durch Bereitstellung nicht nur eines Frameworks und eines Metamodells sondern eines Werkzeugs unterstützen wird (siehe auch Konzept 3.15 ▷ 64). Konzept 3.43 ▷ 106 motiviert dabei die Entwicklung eines praxisorientierten und pragmatischen Werkzeugs, was zusammen mit Konzept 3.50 ▷ 115 dazu führt, dass die Erstellung eines solchen Werkzeugs auch machbar erscheint. Konzept 3.5 ▷ 57 untermauert diese These.

Die Konzepte 3.30 ▷ 80, 3.39 ▷ 93, 3.46 ▷ 111 und 3.47 ▷ 112 zeigen uns einen Weg auf, wie ein hinreichend ausgebildeter und talentierter Entwicklungsleiter auch in einem mittelständischen Projektkontext unter Verwendung bewährter und entsprechend geeigneter Methoden (Domain Driven Design, Feature Driven Development und generativer Programmierung) die Entwicklung eines CMDSD-Werkzeugs steuern und realisieren kann. Bei einer systematischen Kombination dieser Einzelaspekte unterstützt ihn Konzept 3.6 ▷ 60 ebenso wie Konzept 3.34 ▷ 87 durch Definition eines Vorgehensmodells für den Gesamtprozess. Um zu bewerten, welche spezifischen Bestandtele der beschriebenen Methoden für die vorliegende Domäne von besonderer Bedeutung sind, muss sich der Chief Programmer mit der Metrik-basierten Bewertung (d. h. insbesondere einer Kosten-/Nutzen- und einer Risikoanalyse, siehe Konzept 3.27 ▷ 74) der einzelnen Bestandteile dieser Ansätze beschäftigen. Aufgrund der prinzipiellen agilen Vorgehensweise im Rahmen der Entwicklung empfehlen wir bereits jetzt eine pragmatische Herangehensweise sowie die spätere Feinjustierung oder Revision initial getroffener Entscheidungen, wenn mehr Erfahrungen im Rahmen der Werkzeugentwicklung zu den berücksichtigen Anforderungen gesammelt sind. Die Komplexität und die Kosten des resultierenden Entwicklungsprozesses (sowohl im Rahmen des CMDSD als auch des CDPM) orientieren sich dabei maßgeblich an der betrachteten Domäne, da je nach Risikoklasse und Sicherheitsaspekten hochgradig unterschiedliche Anforderungen für den Entwicklungsprozess resultieren können (siehe Konzept 3.26 ▷ 74).

Bei der Erstellung des CMDSD-Werkzeugs hat der Chief Programmer eine Reihe

von Rahmenbedingungen zu beachten, um einen langfristigen Erfolg sicherzustellen:

- Die Ausprägung konkreter Lösungen mit dem CMDSD-Werkzeug erfolgt durch einen modellgetriebenen Ansatz (Konzept 3.42 ▷ 105), bei dem soweit irgend möglich nur fachliche Aspekte in den Modellen abgebildet werden (Konzept 3.18 ▷ 66). Dafür muss das Werkzeug aber auch kein Reverse Engineering unterstützen (siehe Konzept 3.48 ▷ 114).
- Die Modelle müssen
  - erweiterbar sein, um zukünftige Anforderungen abzudecken (Konzept 3.3 ▷ 55),
  - in einer möglichst angemessenen, dennoch abstrakten und – soweit möglich – visuell sehr gut zugänglichen Form dargestellt werden (Konzepte 3.4 ▷ 56, 3.17 ▷ 65, 3.21 ▷ 70, 3.23 ▷ 73, 3.40 ▷ 99, 3.41 ▷ 101, 3.44 ▷ 107),
  - eine Projektion auf zukünftige Technologien unterstützen (Konzept 3.16 ▷ 65). Dies lässt sich am Besten durch entsprechende Ausprägung der grundlegenden Modelltransformationen erreichen (siehe Konzept 3.29 ▷ 80). Grundidee ist dabei, Infrastrukturkomponenten automatisiert und ohne domänenunabhängige Eingriffsmöglichkeiten zu generieren (siehe Konzept 3.32 ▷ 83)[31].

Zudem muss das Werkzeug RAD-Aspekte berücksichtigen und somit insbesondere auch die Prototypisierung von Lösungen unterstützen (siehe die Konzepte 3.31 ▷ 81, 3.25 ▷ 73 und 3.53 ▷ 125), denn damit legt es die Grundlage für besonders effektive Projekte auf Basis von CDPM zu legen. Wichtig ist dabei auch, eine nebenläufige Bearbeitung von Modellen und Systemkomponenten in dem zu entwickelnden Werkzeug zu berücksichtigen, auch wenn diese Anforderung aufgrund der kleinen Teamgrößen weniger komplex ist als man zunächst erwarten könnte (siehe Konzept 3.28 ▷ 78).

Ergebnis des domänenspezifischen ausgeprägten CMDSD-Vorgehensmodells ist eine strukturierte Entwicklungssystematik insbesondere auch für das Projektvorgehensmodell, das die Instanziierung konkreter Lösungen steuert (siehe Konzept 3.12 ▷ 64).

Für das CDPM-Vorgehensmodell haben wir ein agiles Vorgehensmodell mit den dafür charakteristischen inkrementellen, evolutionären, rekursiven und iterativen

[31] Implizit bedeutet dies, dass PIM und PSM quasi fest für die spezifische Domäne vorgegeben werden und nur an den Stellen parametrisierbar sind, an denen sich domänenrelevante Auswirkungen ergeben (siehe auch Abschnitt 3.3.1 zu Erläuterungen zu PIM und PSM).

Bestandteilen als leistungsfähigste Vorgehensmodellfamilie identifiziert (siehe die Konzepte 3.35 ▷ 88, 3.19 ▷ 66 und 3.54 ▷ 127), wobei wir bislang kein spezifisches agiles Vorgehensmodell ausgewählt haben. Durch diesen Ansatz verstärken wir die Möglichkeit für eine enge Zusammenarbeit von Fach- und IT-Experten (siehe Konzept 3.38 ▷ 91 sowie 3.22 ▷ 72), die in existierenden Ansätzen die allgemeine Projektqualität nachweislich verbessert (siehe Konzept 3.2 ▷ 50). Wir werden zusätzlich weitere Methoden angeben, um die Kommunikation zwischen allen Projektbeteiligten zu unterstützen, um insbesondere auch möglichst kurze Kommunikationswege zu installieren, die zur Fehlervermeidung in und effizienten Evolution von Lösungen beitragen (siehe Konzept 3.45 ▷ 109). Agile Modeling wird dabei nachhaltig die enge Zusammenarbeit von Domänen- und IT-Experten unterstützen (siehe Konzept 3.52 ▷ 120). Durch diese Maßnahmen wird ein stärkeres Verständnis der beteiligten Parteien füreinander und für die Grenzen und Möglichkeiten der eingesetzten Modelle und Werkzeuge erzeugt. Das führt im Umkehrschluss dazu, dass insbesondere auch Fachexperten dazu angeregt werden, aufgrund dieses erweiterten Technologie- und Methodenverständnisses neue Lösungsansätze zu entwickeln (siehe auch Konzept 3.20 ▷ 67).

Konzept 3.24 ▷ 73 belegt wiederum die Sinnhaftigkeit des zuvor beschriebenen prinzipiellen Zusammenspiels und der aufeinander aufbauenden Strukturen von CMDSD und CDPM.

Auf diese Weise ergibt sich aus den Ergebnissen des vorliegenden Kapitels der Rahmen unseres nachfolgenden Lösungsansatzes.

Teil II

Conceptual Programming (CP)

Kapitel 4

Conceptual Programming (CP)

„Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction. “
— Albert Einstein (1879–1955), Physiker

In diesem Kapitel erläutern wir die fundamentale Herleitungsweise zur Definition unseres Ansatzes über ein allgemeines System aus Werten, Prinzipien und Methoden (Abschnitt 4.1). Diese konkretisieren wir für das *Conceptual Programming* in Abschnitt 4.2 und begründen ihre Notwendigkeit auf Basis der vorausgegangenen Analyseergebnisse. In Abschnitt 4.3.2 erklären wir eine Validationsmöglichkeit für CP-basierte Entwicklungen, in Abschnitt 4.3.3 schließlich gehen wir darauf ein, was zu bedenken ist, um CP-basierte Modelle für konkrete Domänen auszuprägen. Die Zusammenfassung der dabei erzielten Erkenntnisse in Abschnitt 4.4 leitet über in die konkrete Beschreibung des *Conceptual Model Driven Software Development (CMDSD)* in Kapitel 5 sowie des *Concept Driven Project Management (CDPM)* in Kapitel 6. Wir definieren CP wie folgt:

K-3.38 ▷ 91
K-3.48 ▷ 114
K-3.37 ▷ 90
K-3.34 ▷ 87
K-3.33 ▷ 85

Conceptual Programming beschreibt ein Paradigma zur Umsetzung komplexer IT-Lösungen im mittelständischen Umfeld, das auf folgenden Grundideen beruht:

1. *Domänenexperten und IT-Fachleute spezifizieren und realisieren Lösungen gemeinsam.*
2. *Domänenspezifische Werkzeuge unterstützen diesen Prozess durch Abstraktion technischer Details und automatisierter Generierung lauffähiger Lösungen.*
3. *Domänenspezifische Vorgehensmodelle fördern die kooperative und effiziente Spezifikation und Umsetzung komplexer Lösungen.*
4. *Conceptual Programming setzt sich zusammen aus den Hauptmethoden des* Conceptual Model Driven Software Development *(CMDSD, die fachmodell-*

getriebene Software-Entwicklung), des Concept Driven Project Management *(CDPM, das fachmodellgetriebene Projektmanagement) und des* Produktgremiums*, die synergetisch die Evolution und Ausprägung domänenspezifischer Lösungen fördern.*

Da *Conceptual Programming* den philosophischen Überbau für zwei konkrete, methodische Ausprägungen beschreibt, ist dieses Kapitel entsprechend knapp und informell gehalten. Wie bereits in Abschnitt 3.6 erklärt, werden wir die von uns hergeleiteten Konzepte als Randmarkierung referenzieren, wenn Sie für unsere Ausführungen bedeutsam sind.

Im nächsten Schritt führen wir eine Systematik ein, die eine Top-Down-Vorgehensweise bei der Definition unseres Paradigmas und seiner Hauptmethoden erlaubt.

4.1 Werte, Prinzipien und Methoden als Systematik

Dem Vorbild agiler Methoden folgend werden wir sowohl für *Conceptual Programming*, *Conceptual Model Driven Software Development* als auch *Concept Driven Project Management* ein System von Werten, Prinzipien und Methoden definieren (siehe [BeAn05], S. 13ff.). Auf diese Weise vereinfacht sich die tatsächliche Umsetzung des Paradigmas und es können sogar domänenspezifische Extrapolationen der hier definierten Methoden ausgeprägt werden. Zudem verdeutlichen wir durch die ursächliche Betrachtung von IT-Projekten, welche Wirkung wir durch unseren Ansatz erzielen. Dazu werden wir im Folgenden stets den direkten Bezug zwischen Problemursachen und den von uns favorisierten Lösungsansätzen herstellen.

K-3.8 ▷ 62
K-3.51 ▷ 118

Allerdings wollen wir in der vorliegenden Arbeit einen stringenteren Ansatz verfolgen als das aus unserer Sicht bei vielen agilen Methoden der Fall ist. So werden zum Beispiel *Kommunikation*, *Einfachheit*, *Feedback*, *Mut* und *Respekt* als Grundwerte des Extreme Programming definiert – doch erscheinen uns die Abstraktionsebenen dieser „Werte“ sehr unterschiedlich. Man kann Kommunikation und Feedback auch als Prinzipien ansehen, denen andere Werte zugrunde liegen. Wir streben eine klarere Struktur unserer Systematik an, damit sie leichter zu verinnerlichen und anzuwenden ist.

Um unsere Paradigmen kohärent vorstellen zu können, definieren wir zunächst unser Verständnis von Werten, Prinzipien und Methoden und zeigen die Zusammenhänge

zwischen den Begriffen auf. Die nachfolgenden Kapitel bauen auf diesen Definitionen auf und erläutern die Ziele von CMDSD und CDPM. So entwerfen wir ein geschlossenes, konsistentes und nachhaltiges Modell der Softwareentwicklung basierend auf speziellen Technologien und Methoden, optimiert für eine spezielle Zielgruppe.

Da *Conceptual Programming* ein domänenübergreifendes Paradigma ist, wird es je nach betrachteter Domäne andere Ausprägungen erfahren, besonders in den Details. Die Beschreibungen der beiden Hauptmethoden enthalten entsprechende Hinweise zu Adaptionsmöglichkeiten in den Abschnitten 5.5.4 und 6.5.6. Allgemeinere Hinweise finden sich in Abschnitt 4.3.3.

4.1.1 Werte

Das Herkunftswörterbuch des Duden ([WeKS01], S. 924) definiert einen Wert als *„positive Bedeutung, Gewichtigkeit, besondere Qualität; [Kauf-, Markt]preis; Ergebnis einer Messung“* und verweist zudem auf den Zusammenhang zu den Begriffen *Würde*, definiert als *„Achtung gebietender Wert, der einem Menschen innewohnt“* ([WeKS01], S. 935).

Im Rahmen des Extreme Programming werden Werte wie folgt definiert: „*Values are the root of the things we like and don't like in a situation*“ (siehe [BeAn05], S. 14). Zudem werden Werte mit den folgenden Eigenschaften verbunden: Sie sind universell, konsistent, zweckgerichtet („purposeful“) und ergänzen einander („complementary“), siehe dazu die Mindmap auf der Umschlaginnenseite von [BeAn05].

Aufbauend auf diesen Quellen definieren wir Werte für unser weiteres Vorgehen wie folgt:

> *Werte sind die abstrakten Ziele, die wir im Rahmen einer konkreten Tätigkeit anstreben, um Mehrwert zu schaffen. Werte eines Paradigmas müssen konsistent sein, sich gegenseitig ergänzen und stützen, um in ihrer Gesamtheit mehr zu sein als die Summe der Einzelteile. Man kann Werte bildlich gesprochen auch als Grenzen einer moralischen Landkarte begreifen. Sie definieren den Handlungsspielraum, innerhalb dessen wir uns bewegen können.*

4.1.2 Prinzipien

Werte allein sind nicht konkret genug, um eine Methode oder ein Paradigma in ausreichender Stärke zu prägen. Daher werden Prinzipien verwendet, um Werte zu konkretisieren. Das Herkunftswörterbuch des Duden ([WeKS01], S. 630) definiert ein Prinzip als „*Anfang, Ursprung, Grundlage; Grundsatz*“. Extreme Programming beschreibt Prinzipien wie folgt: „*Bridging the gap between values and practices are* principles *[...]. Principles are domain-specific guidelines for life*“ (siehe [BeAn05], S. 15). Außerdem werden Prinzipien mit den folgenden Eigenschaften verbunden: Sie sind domänenspezifisch, allgemein anwendbar („widely applicable“) und widersprüchlich („contradictory“)[1], siehe dazu wiederum die Mindmap auf der Umschlaginnenseite von [BeAn05].

Wir verstehen den Begriff „Prinzip“ wie folgt:

> *Prinzipien sind domänenspezifische Grundsätze, die uns helfen, unsere Werte zu leben. Sie sind das Bindeglied zwischen abstrakten Werten und konkreten Methoden. Bildlich gesprochen sind Prinzipien Wegweiser, die uns helfen, uns auf einer moralischen Landkarte zu bewegen und so unsere Ziele zu erreichen.*

Es sei angemerkt, dass es uns natürlich deswegen möglich ist, Prinzipien (die domänenspezifisch sind) für die zunächst domänenunabhängigen Verfahren von CP, CMDSD und CDPM anzugeben, da diese selbst innerhalb einer eigenen (technischen und organisatorischen) Problemdomäne definiert werden, die sich aus dem Thema dieser Arbeit ergibt: Unser Anliegen ist die Übertragung und Adaption von Methoden und Verfahren aus Großprojekten auf mittelständische Rahmenbedingungen. Auch KMUs sollen in der Lage sein, komplexe, leistungsfähige und anpassbare Systeme in Individualprojekten zu entwickeln. Für diese Domäne können wir präzise Prinzipien angeben, die es uns dabei unter Beachtung der von uns als sinnvoll identifizierten Werte ermöglichen, unsere soeben beschriebenen Ziele zu erreichen.

[1] Die Bedeutung dieser Eigenschaft entzieht sich uns. Weder haben wir eine sinnvolle Erläuterung in [BeAn05] gefunden noch in anderen Artikeln. Möglicherweise ist gemeint, dass Prinzipien widersprüchlich erscheinen können. Wir sind der Auffassung, dass Widersprüche bei der Definition von Prinzipien zu vermeiden sind. Scheinbare Widersprüche können existieren und müssen aufgelöst bzw. erläutert werden. Andernfalls wird der Zugang zu einem Vorgehensmodell unnötig erschwert und seine Akzeptanz ohne Not gefährdet.

4.1.3 Methoden

Um den Einstieg in ein Paradigma zu vereinfachen, werden aus den zugrunde liegenden Werten und Prinzipien konkrete Methoden abgeleitet, die Anwender des Paradigmas bei ihren täglichen Aufgaben wie einen Leitfaden verwenden können, um die Werte des Paradigmas zu bestätigen und seinen Prinzipien zu folgen. Extreme Programming beschreibt Methoden („practices“) wie folgt: „*Practices are things you do day-to-day*“ (siehe [BeAn05], S. 13). Die folgenden Eigenschaften werden mit Methoden verbunden: Sie sind objektiv und klar (siehe [BeAn05], S. 13) sowie konkret, nachvollziehbar und angemessen („situated“) (siehe wiederum die Umschlaginnenseite von [BeAn05]).

Wir definieren Methoden daher wie folgt:

> *Methoden sind konkrete und nachvollziehbare Handlungsanweisungen, denen wir folgen, um auf Basis unserer Prinzipien unsere Werte zu leben. Man kann Methoden bildlich gesprochen als die Wege auf einer moralischen Landkarte ansehen, die wir konkret beschreiten, um unsere Ziele zu erreichen.*

4.1.4 Der Zusammenhang zwischen Werten, Prinzipien und Methoden

Basierend auf den obigen Definitionen ergibt sich der Zusammenhang zwischen Werten, Prinzipien und Methoden auf anschauliche Weise:

- Werte sind abstrakte, nicht greifbare Ideale, die wir anstreben und die unsere Lebens- und Handlungsweise prägen.
- Methoden sind die aus Werten materialisierten greifbaren Mittel und Wege, die es ermöglichen unsere abstrakten Ziele im Alltag konkret zu leben.
- Prinzipien nehmen eine Position zwischen Werten auf der einen und Methoden auf der anderen Seite ein. Sie sind weniger abstrakt als Werte und sind beobachtbar, sie haben allerdings noch nicht die Konkretheit von Methoden, die bereits so scharf formuliert werden können, dass sie als Handlungsanweisungen fungieren.

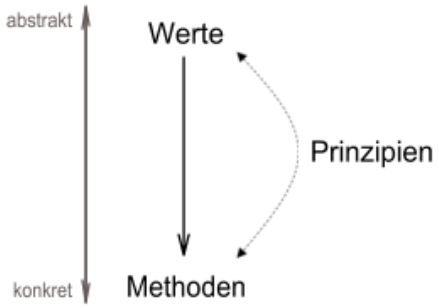


Abbildung 4.1: Zusammenhänge zwischen Werten, Prinzipien und Methoden

Dieser Zusammenhang ist in Abbildung 4.1 noch einmal grafisch dargestellt.

Da sich auf Basis dieser drei Bestandteile ein konsistentes und abgerundetes Paradigma definieren lässt, werden wir im Folgenden zunächst für CP, dann für CMDSD und CDPM die zugrunde liegenden Werte, daraus resultierende Prinzipien und die zu verfolgenden Methoden vorstellen.

4.2 CP als Paradigma

Conceptual Programming folgt einer definierten Menge von Werten, Prinzipien und Methoden, die wir bereits in der Einführung dieser Arbeit und in der Anforderungsanalyse ausführlich diskutiert haben, ohne sie allerdings im Vorfeld als solche zu bezeichnen. Der Vollständigkeit halber fassen wir die Systematik in diesem Abschnitt noch einmal unter Referenzierung der relevanten vorausgegangenen Abschnitte zusammen. Anschließend führen wir die beiden Methoden ein, die es uns erlauben werden, die auf Seite 8 in Abschnitt 1.2 formulierten Ziele zu erreichen.

4.2.1 Werte

Die grundlegenden Werte, auf denen *Conceptual Programming* aufbaut, ergeben sich aus den Rahmenbedingungen, die wir in der Einführung zu dieser Arbeit in Kapitel 1 sowie in der Analyse zu den besonderen Rahmenbedingungen des Mittelstands in Kapitel 2 identifiziert haben. Im Einzelnen sind das folgende Werte:

1. *Wirtschaftlichkeit* als direkte Konsequenz aus den in Abschnitt 2.1 erläuterten Ressourcenbeschränkungen, denen der Mittelstand unterliegt,

2. *Flexibilität* als notwendige Konsequenz aus der in Abschnitt 1.1 dargestellten zunehmenden Veränderungsgeschwindigkeit von Märkten und Marktbedingungen sowie der in Abschnitt 3.4.3 gewonnenen Erkenntnis, dass Wandel eine Tatsache und daher als Chance und nicht als Risiko zu begreifen ist sowie
3. *Zugänglichkeit*, um mit den verfügbaren Ressourcen eine möglichst schnelle und effektive Adaption der neuen Methoden und Verfahren zu erreichen. Den Hintergrund dafür haben wir ausführlich in Abschnitt 2.2.1 dargestellt und seine Bedeutung durch bereits diesen Wert forcierende Ansätze untermauert (siehe Abschnitt 3.1).

Diese Werte werden wir in den folgenden drei Abschnitten noch einmal detailliert vorstellen, motivieren und mit ihren Zielen klar herausarbeiten.

4.2.1.1 Wirtschaftlichkeit

Wir haben die Knappheit von Ressourcen (finanziell, organisatorisch, ggf. technologisch) als eine der dominanten Rahmenbedingungen und Restriktionen in Bezug auf IT-Projekte im mittelständischen Umfeld identifiziert (siehe Abschnitt 2.1). Daher muss der rationale Umgang mit Ressourcen und die Effizienz unseres Verfahrens oberste Priorität genießen. Wir verfolgen dabei das Maximalprinzip bzw. Ergiebigkeitsprinzip, indem wir mit den vorhanden Ressourcen das maximal erreichbare Ergebnis anstreben (siehe [Wiki06l]).

Wir wollen die nachhaltige Bedeutung von Wirtschaftlichkeit als oberstes Prinzip eindringlich verdeutlichen: Wenn das von uns definierte Paradigma wirtschaftlich nicht durch den Mittelstand umsetzbar ist, sind alle anderen Anstrengungen vergeblich und wir sind gescheitert. Die Verbesserungen, die wir anstreben, müssen finanzierbar und umsetzbar sein. Um dieses Ziel zu erreichen, ist ein konsequentes Vorgehen bei der Auswahl der Mittel und Wege (d. h. der Werte, Prinzipien und Methoden, die wir zu CMDSD und CDPM definieren werden) erforderlich. Im Einzelnen bedeutet dies Folgendes:

- Methoden und Werkzeuge werden soweit möglich auf übermäßig generische und universelle Ansprüche verzichten, wenn sie dadurch finanzierbar werden. Der nachfolgend definierte Wert der Zugänglichkeit (siehe Abschnitt 4.2.1.3) unterstützt dies in seinem Spannungsverhältnis zum ebenso definierten Wert der Flexibilität (siehe Abschnitt 4.2.1.2): Wir wählen bei der Entwicklung unserer Werkzeuge einen Weg, der selbige Werkzeuge so leicht wie möglich bedienbar und so effektiv wie möglich gestaltet, auch wenn wir dadurch auf Detail-

variationsmöglichkeiten verzichten. Das im weiteren Verlauf definierte Prinzip der Domänenfokussierung (siehe Abschnitt 4.2.2.2) hilft dabei im Konkreten ebenso wie das Prinzip der Automatisierung (siehe Abschnitt 4.2.2.3).

- In dieser Arbeit beschränken wir uns darauf, bewährte Methoden, Modelle und Vorgehensweisen auf neue Weise miteinander zu kombinieren anstatt gänzlich neue Verfahren von Grund auf zu entwerfen. Das sichert uns einen reichen Schatz von Erfahrungswerten aus der Vergangenheit sowie ein umfassendes Grundwissen über Stärken und Schwächen der existierenden Ansätze. Wir konzentrieren uns in dieser Arbeit darauf, die Schwächen der vorhandenen Ansätze durch eine neuartige Kombination von Modellen und Vorgehensweisen zu beheben oder zumindest soweit zu reduzieren, dass sie auch für den Mittelstand wirtschaftlich tragbar werden.
- Wir werden zu jeder Zeit die einfachste Lösung suchen (sowohl im Hinblick auf Effizienz in der Realisierung als auch auf Effizienz in der Anwendung daraus resultierender Werkzeuge und Methoden), um wirtschaftlich effiziente Verfahren zu garantieren.
- Gegebenenfalls müssen wir suboptimale Lösungen in Kauf nehmen, wenn die Entwicklung einer optimalen Lösung zu teuer ist. Dies erfordert auch einen offenen Umgang mit Kunden, denen die Vor- und Nachteile verschiedener Lösungsansätze offen, ehrlich, direkt und nachhaltig dargestellt werden müssen. Genau aus diesen Gründen ergeben sich die in Abschnitt 6.4.2 definierten Prinzipien für das *Concept Driven Project Management* (CDPM).
- Die Forderung nach Wirtschaftlichkeit wird in vielen Fällen eine Abkehr von präskriptiven und dokumentationsintensiven Prozessmodellen zwingend nach sich ziehen (siehe unsere Analysen in Abschnitt 2.2.4 sowie unsere Anmerkungen in Abschnitt 3.4.2 zu den verschiedenen etablierten präskriptiven Vorgehensweisen).
- Effektivität (also das Verhältnis von tatsächlich erreichtem zu definiertem Ziel) allein ist im weiteren Verlauf nicht das maßgebliche Kriterium für die Verwendung von Methoden und Modellen. Die Effizienz (also das Verhältnis von Aufwand zu Nutzen) bei der Nutzung der vorhandenen Ressourcen steht im Vordergrund.

Trotz dieser Einschränkungen werden wir im weiteren Verlauf der Arbeit belegen, dass gerade die Forderung, streng wirtschaftliche Maßstäbe bei der Entwicklung von Methoden und Werkzeugen zu verwenden, als Katalysator dienen wird, um äußerst

effizient arbeiten zu können und gleichzeitig Lösungen von hoher Komplexität zu produzieren.

Zum Abschluss dieses Abschnitts wollen wir die Potenziale wirtschaftlicheren Vorgehens mit (den wenigen verfügbaren) Zahlen aus Studien greifbarer machen: Es gilt heute als akzeptiert, dass die späte Änderung von Anforderungen (womöglich erst nach der ersten Auslieferung) in großen Projekten ca. 100-mal so teuer ist wie das frühe Einbringen solcher Änderungen (z. B. direkt während der Anforderungsspezifikation). Für kleinere Projekte im Bereich von 2000-5000 LOC[2] bewegt sich dieser Faktor immer noch bei ca. 5:1 (siehe dazu [BoTu03b], S. 218–219 sowie [Boeh81], [SBBB02], S. 249–258 und [McGi96]). Wenn wir davon ausgehen, dass typische mittelständische Projekte eher klein als groß sind (passend zu unseren Budgetausführungen in Abschnitt 2.1), also eher im Verhältnis 5:1 als 100:1 Kosten bei späten Änderungen erzeugen werden, besteht aber im Optimalfall eben genau dieses Einsparpotenzial oder in anderen Worten: Gelingt es, alle Änderungen in frühe Phasen zu verschieben, können bis zu fünfmal so viele Änderungen in das Projekt aufgenommen werden. Natürlich ist das ein in der Realität unwahrscheinlicher Idealwert, er hilft aber, das Potenzial durch Nutzung eines möglichst wirtschaftlichen Vorgehensmodells zu verdeutlichen.

4.2.1.2 Flexibilität

Wie bereits in den Abschnitten 1.1 und 2.2 dargestellt, ist der Mittelstand gezwungen, sich mit zunehmender Geschwindigkeit an die wandelnden Rahmenbedingungen einer globalen Wirtschaft anzupassen. Daraus resultieren ebenso häufige Anpassungen an Softwaresystemen, die zur Unterstützung der geschäftskritischen Prozesse eingesetzt werden. Um diesen externen Faktoren Rechnung zu tragen, fordern wir als höchsten Wert Flexibilität – sowohl in den von uns postulierten Methoden als auch bei den Werkzeugen. Sollten denkbare Konzepte der zukünftig einfachen Anpassbarkeit unserer Werkzeuge und Methoden im Wege stehen, werden wir die Stichhaltigkeit dieser Konzepte nachhaltig hinterfragen und sie gegebenenfalls nicht umsetzen. Nur so können wir die Zukunftssicherheit von Investitionen in die von uns definierte Klasse von Werkzeugen und Methoden maximieren.

Flexibilität ist sowohl bei der Anforderungsdefinition und in der Projektabwicklung als auch im Design der Werkzeugeigenschaften von großem Vorteil. Wir haben bereits

[2] LOC steht für „Lines of Code“, also Programmzeilen an Code, die geschrieben wurden, um eine Lösung zu entwickeln.

gezeigt, dass viele Probleme für das Scheitern von Projekten bereits bei der Anforderungsdefinition entstehen (siehe Abschnitt 2.2.1) – daher müssen unsere Methoden und Werkzeuge in der Lage sein, flexibel mit sich verändernden Anforderungen umzugehen. Sollen Projekte mit dem Anspruch von Wirtschaftlichkeit und maximaler Effizienz abgewickelt werden, ist es zudem erforderlich, dass das Projektmodell flexibel mit der Verfügbarkeit von Ansprechpartnern, der Organisation des Ablaufs und der Ausgestaltung der Details umzugehen weiß. Letztlich ist ein Mindestmaß dieser Flexibilität, um auch größere Abweichungen von einem Standardmodell abbilden zu können. Insgesamt entsteht die Notwendigkeit, ein Gleichgewicht zwischen den folgenden Forderungen zu finden:

- einfache Anpassbarkeit der Lösungen an neue Rahmenbedingungen,
- flexibel erweiterbare Werkzeuge und Vorgehensmodelle,
- flexible Variationsmöglichkeiten in den modellierten Systemen,
- wirtschaftliche Realisierung solcher Lösungen.

Der scheinbare Widerspruch zwischen hoher Flexibilität und Wirtschaftlichkeit wird sich durch die nachfolgend definierten Prinzipien der Domänenfokussierung (siehe Abschnitt 4.2.2.2) sowie der Automatisierung (siehe Abschnitt 4.2.2.3) auflösen lassen.

Flexibilität hat natürlich ihren Preis, wenn gleichzeitig die Werte Zugänglichkeit (Abschnitt 4.2.1.3) und Wirtschaftlichkeit (Abschnitt 4.2.1.1) angestrebt werden:

- Umfassende Flexibilität kann nur durch entsprechend komplexe Werkzeuge geboten werden. Diese würden aber den denkbaren Rahmen mittelständischer IT-Projekte bzw. des Leistungsvermögens mittelständischer IT-Dienstleister überschreiten. Zudem erfordert die Arbeit mit komplexen Werkzeugen mehr Zeit als die Arbeit mit einfachen Werkzeugen, so dass wirtschaftliche Restriktionen den Einsatz übermäßig komplexer Werkzeuge ausschließen, wenn nicht entsprechend geschulte Fachkräfte zur Verfügung stehen. Letzteres ist aber eine häufigere weitere typische Restriktion im mittelständischen Umfeld (siehe die Auswertungen in Abschnitt 2.1).
- Umfassende Flexibilität bedeutete bisher immer auch Komplexität, da „umfassende Flexibilität“ formal gesehen mit Turing-Berechenbarkeit bzw. Turing-Vollständigkeit gleichzusetzen ist (siehe [Wege99], S. 20). Turing-vollständige Sprachen sind aber entweder ähnlich komplex wie typische bekannte Programmiersprachen wie C, Java, C++, usw. oder sie erfordern sehr hohe Fachkenntnisse in der Formulierung von Algorithmen, da die Sprachen mit sehr weni-

gen Konstrukten auskommen und somit umfangreiche Programmierung für die Modellierung komplexer Algorithmen erforderlich wird[3]. Somit steht hohe Flexibilität zunächst im Widerspruch zur Zugänglichkeit.

Auf diesen Konflikt und potenzielle Lösungsansätze werden wir ausführlich in unseren Ausführungen zum *Conceptual Model Driven Software Development (CMDSD)* in Kapitel 5 eingehen. Der Kern der Lösung besteht darin, dass nachführend beschriebene Prinzip der *Domänenorientierung* immer als Leitmaxime zu verwenden – Flexibilität ist nur relevant für eine (ggf. eng gefasste) Anwendungsdomäne. So kann die Komplexität der Werkzeuge wieder auf das gerade notwendige Maß reduziert werden.

4.2.1.3 Zugänglichkeit

Zugänglichkeit als letzter elementarer Wert meint verschiedene Aspekte von Einfachheit:

- Vorgehensmodelle sollen leicht zu erlernen und intuitiv anzuwenden sein[4].
- Werkzeuge müssen in der Bedienung so einfach und ergonomisch wie möglich sein. Auch Domänenexperten soll es so erlaubt sein, gemeinsam mit IT-Experten an Lösungen zu arbeiten. Die gesamte CMDSD-Methodologie (siehe Kapitel 5) ist stark von diesem Wert geprägt während die CDPM-Methodologie die konkreten Methoden liefert, um CMDSD-Werkzeuge und -Modelle genau in dieser Weise gemeinsam mit Domänenexperten zu verwenden (siehe Abschnitt 6.4.3).
- Fachmodelle müssen leicht verständlich sein. Dadurch wird wiederum die zuvor bereits besprochene Zusammenarbeit zwischen IT- und Domänenexperten vereinfacht. Das wird sich auch im Prinzip der Kundenorientierung (siehe Abschnitt 4.2.2.1) niederschlagen.
- Die Implementierung von Anforderungen sollte so einfach wie möglich gehalten werden – so werden zukünftige Änderungen erleichtert (siehe auch [BeAn05], S. 18–19) und zudem verhindert, dass durch ein unnötiges „Big Design Upfront“

[3] Dies ist vergleichbar zur Programmierung in Maschinensprache oder zur Beschreibung von Algorithmen über Turingmaschinen.

[4] Wir wollen bereits an dieser Stelle darauf hinweisen, dass Einfachheit mit viel Verantwortung verbunden ist – je einfacher zugrunde liegende Modelle erscheinen, umso größer ist die Gefahr, dass nachlässig mit ihnen umgegangen wird, weil „man das ja später eben noch mal richtig fertig machen kann“. Genau das ist aber eine große Gefahr, so dass wir in konkreten Projekten grundsätzlich Disziplin fordern (siehe Abschnitt 6.4.2.7).

(siehe dazu [BDUF06]) Aufwände verursacht werden, die sich angesichts nicht vorhersehbarer zukünftiger Anforderungsveränderungen oder -ergänzungen als unnötig erweisen könnten.

Der Nutzen von einfachen Ansätzen ist ebenso vielfältig wie offenkundig:

- Methoden müssen einfach in der Anwendung sein, damit sie schnell erlernt und möglichst fehlerfrei benutzt werden können. Einfache Methoden erhöhen außerdem die Chance, dass sie „gerne“ (also tatsächlich) in der Praxis umgesetzt werden und ein Paradigma somit hinreichend konsequent angewendet wird. Das erhöht die Erfolgschancen.
- Einfache Vorgehensmodelle schrecken nicht durch Komplexität ab, so dass Entscheidungen für den Einsatz dieser Methoden auch auf Managementseite mit höherer Wahrscheinlichkeit gefällt werden.
- Einfache Fachmodelle erhöhen die Nachprüfbarkeit modellierter Sachverhalte. So können Fehler frühzeitiger entdeckt und zu möglichst geringen Kosten behoben werden (siehe auch Abschnitt 2.2.1).

Allerdings zieht auch der Wert der Zugägnlichkeit Konsequenzen nach sich, die im Vergleich zu klassischen Verfahren teilweise sehr kontrovers erscheinen:

- Fachmodelle werden eine Reihe impliziter Annahmen ohne explizite Formalisierung machen. Das liegt daran, dass viele explizite (insbesondere mathematische) Formalismen Domänenexperten nicht vertraut sind und sie somit die Verständlichkeit von Fachmodellen drastisch reduzieren würden. Fachmodelle werden nur sehr spezielle Domänen modellieren, da universelle Modellierungssprachen sehr komplex und meistens nur von ausgewiesenen Experten vollständig erfasst werden können (siehe dazu die Ausführungen in den Abschnitten 2.2.2 und 3.2).
- Werkzeuge werden nicht jedes denkbare Szenario abdecken können. Universelle Werkzeuge sind sehr komplex und damit auch teuer in der Entwicklung und zeitaufwändiger in der Anwendung.
- Methoden werden eine bestimmte Vorgehensweise fokussieren und sich daher nicht für beliebige Branchen eignen. Die Szenarien, für die Conceptual Programming tendenziell eher nicht in Frage kommt, haben wir bereits in Abschnitt 2.2.4 diskutiert und dabei festgestellt, dass sie nur eine sehr kleine Minderheit mittelständischer Unternehmen betreffen. Die prinzipielle Eignung des CP-Ansatzes steht also nicht in Frage.

Hilfreiche Hinweise zur Modellierung ergonomischer Lösungen finden sich z. B. in [Mart81], S. 115 oder wesentlicher ausführlicher und aktueller in [ISO9241]. Zur allgemeineren Darstellung leicht verständlicher Konzepte haben wir bereits im Rahmen der Betrachtung zur generativen Programmierung in Abschnitt 3.3.4 auf einen guten Überblick zum Thema „Konzeptverständnis“ hingewiesen (siehe [CzEi00], S. 721–736 bzw. die dort referenzierten Grundlagen in [SmMe81, Newe90, SWC&95, Navr96]).

4.2.2 Prinzipien

Die Prinzipien, die aus den zuvor formulierten angestrebten Werten resultieren, haben wir bereits im Rahmen unserer Mittelstandsanalyse in Abschnitt 2.2 ausführlich dargestellt. Sie werden in diesem Abschnitt noch einmal kurz zusammengefasst.

4.2.2.1 Kundenorientierung

> *Richte alle Werkzeuge und Vorgehensmodelle auf die Bedürfnisse deiner Kunden aus.*

IT-Lösungen entstehen, um Kundenanforderungen zu erfüllen und bestehende Prozesse zu optimieren. Da also die Anforderungen und Aufgabenstellungen von Kunden im Fokus einer IT-Lösung stehen, liegt es nahe, Methoden, Werkzeuge und Prozesse derart zu gestalten, dass die unmittelbaren Interessen, Kenntnisse und Bedürfnisse des Kunden den Entwurfsfokus bilden. Auf diese Weise können existierende Probleme verbessert und Optimierungspotenziale sowohl in der Anforderungsdefinition als auch in der eigentlichen Umsetzung nachhaltig erschlossen werden (siehe dazu die detaillierten Ausführungen in Abschnitt 2.2.1).

Das abstrakte Prinzip der Kundenorientierung wird in den folgenden drei Abschnitten durch die konkreten Prinzipien von Domänenfokussierung, Automatisierung und Agilität untermauert.

4.2.2.2 Domänenfokussierung

> *Konzentriere deine Aufmerksamkeit auf eine zusammenhängende Problemdomäne und abstrahiere zugunsten dieser Domäne.*

Wir streben Einfachheit und Wirtschaftlichkeit in unseren Lösungen an. Wie wir bereits in Abschnitt 2.2.2 sowie in Abschnittt 3.3.3 gesehen haben, werden Vereinfachungen insbesondere durch Abstraktionen möglich. Um also neben Wirtschaftlichkeit und Einfachheit die geforderte Anpassbarkeit und Flexibilität vorweisen zu können, müssen Werkzeuge und Vorgehensmodelle bezüglich einer Problemdomäne optimiert werden. Grundlegendes Prinzip des Conceptual Programming ist damit konsequenterweise die Fokussierung auf *eine* Problemdomäne. So können wir – wie wir insbesondere in Kapitel 5 zeigen werden – von vielen Details und technischen Fragestellungen abstrahieren und errichten gleichzeitig ein Fundament, das Domänenexperten in Vokabular und Modellierung vertraut ist. Dieser Umstand wird uns wiederum bei der Definition des *Concept Driven Project Management* in Kapitel 6 zu Gute kommen. Darüber hinaus liefert uns das Prinzip der Domänenfokussierung einen Ansatzpunkt, um unseren Gesamtprozess auf eine solide Grundlage zu bauen – wir werden dazu im weiteren Verlauf auf Ergebnisse aus der Software-Produktlinien-Entwicklung zurückgreifen (siehe dazu auch Abschnitt 3.2.6).

Die Idee, ein modellbasiertes Entwicklungssystem als Grundlage für domänenspezifische Projekte zu realisieren, wird erst dadurch für mittelständische Anbieter realisierbar, dass eine Einschränkung auf eine Domäne vorgenommen wird. Ansonsten würden sich die in den Kapiteln 1 und 2 erläuterten Kosten- und Komplexitätsprobleme einstellen, die aus mittelständischer Perspektive nicht beherrschbar sind.

Insbesondere leiten wir dieses Prinzip aus der Forderung nach einfachen Fachmodellen im CMDSD ab, die es Domänenexperten zusammen mit IT-Experten erlauben, effizient an Lösungen zu arbeiten (siehe dazu die Ausführungen in Abschnitt 2.2.3). Die Erfahrungswerte aus dem Domain Driven Design (siehe unsere Ausführungen in Abschnitt 3.2.7) bestätigen, dass eine starke Domänenfokussierung durch Nutzung einer universellen domänenspezifischen Sprache vielfältige Vorteile für die Projektkommunikation und Qualität der erstellten Systeme hat. Auf diesen Sachverhalt werden wir in Abschnitt 5.4.3.7 zurückkommen.

Evans formuliert diesen Sachverhalt treffend in dem Abschnitt *The Heart of Software* von [Evan04a], S. 4 als *„The heart of software is its ability to solve domain-related problems for its user. All other features, vital though they may be, support this basic purpose.“*

4.2.2.3 Automatisierung

> *Automatisiere die Erzeugung von Lösungen so weit wie möglich.*

Um Kundenorientierung in der Umsetzungsphase zu leben und Implementierungen gemeinsam mit Domänenexperten vornehmen zu können, ist es erforderlich, die Implementierung des Systems weitgehend zu automatisieren. Automatisierung kann um so umfassender erfolgen, je höher die Abstraktionsebene ist, auf der Anforderungen formuliert werden (siehe dazu unsere Ausführungen in Abschnitt 2.2.3 oder beispielsweise zu Software Factories in Abschnitt 3.3.3). Ansätze zur modellgetriebenen Software-Entwicklung weisen den prinzipiellen Lösungsweg: Generische Modellierungssprachen wie UML erreichen einen Automatisierungsgrad von 40-70%, domänenspezifischere Sprachen kommen auf bis zu 95–100% (siehe die Abschnitte 3.2.8, 3.3.1 und 3.3.2). Aufgrund dieser Erkenntnisse wird Automatisierung bzw. Generierung das grundlegende Prinzip sein, mit dem wir die Werte von Wirtschaftlichkeit, Anpassbarkeit und Einfachheit forcieren.

K-3.23 ▷ 73

K-3.32 ▷ 83
K-3.42 ▷ 105

4.2.2.4 Agilität

> *Arbeite mit agilen Vorgehensmodellen, um dich Veränderungen zu stellen und Anforderungen durch Rapid Prototyping und iteratives Vorgehen optimal zu erfassen.*

In den vorangegangenen Abschnitten haben wir die Prinzipien der Kundenorientierung, der Domänenfokussierung und der Automatisierung basierend auf den Werten Wirtschaftlichkeit, Einfachheit, Anpassbarkeit und Flexibilität definiert. Obwohl jeder dieser Werte bzw. jedes dieser Prinzipien bis zu einem gewissen Grad Hinweise auf die mögliche Gestaltungsweise eines organisatorischen Vorgehensmodells geliefert haben, fehlte bisher der umfassende Rahmen, der diese Einzelteile effektiv verbindet.

Wir finden ihn in den Erkenntnissen agiler Software-Entwicklung, die auf folgenden Werten basiert (siehe Abschnitt 2.2.4 sowie [AAGI06]):

(A) **Individuals and interactions** over processes and tools
(B) **Working software** over comprehensive documentation
(C) **Customer collaboration** over contract negotiation
(D) **Responding to change** over following a plan

(A), (C) und (D) unterstützen uns bei der kundenorientierten Ausrichtung unserer Werkzeuge und Prozesse, in denen der Mensch und nicht das Verfahren oder das Werkzeug betont werden. (B) unterstützt direkt die wirtschaftliche Vorgehensweise, da lauffähige Systeme fokussiert werden. Diese erlauben zum Einen einen frühzeitigen Return On Invest (ROI, siehe [Wiki06m]) und ermöglichen zum anderen eine kontinuierliche Verifikation umgesetzter Anforderungen.

Somit fordern wir als grundlegendes Prinzip für Conceptual Programming Agilität. Agilität steht hier sowohl für die Arbeitsweise, die von CMDSD-Werkzeugen unterstützt werden muss (siehe Kapitel 5) als auch für die Art, wie das CDPM-Vorgehensmodell ausgestaltet wird (siehe Kapitel 6).

K-3.35 ▷ 88

4.2.3 Methoden

Nach der Definition der grundlegenden Werte und Prinzipien des Conceptual Programmings werden wir nachfolgend kurz die beiden Hauptmethoden umreißen, die aus diesen Werten und Prinzipien resultieren. Beide Hauptmethoden setzen sich aus einer Vielzahl von Untermethoden zusammen, so dass jeder Hauptmethode ein eigenes Kapitel dieser Arbeit gewidmet ist. Neben den beiden Hauptmethoden führen wir zudem eine Nebenmethode ein, die sich der Koordination der beiden Hauptmethoden widmet.

Die grundlegende Methodenstruktur ergibt sich aus den Erkenntnissen über effektive Gestaltung von Software-Produktlinien (siehe auch Abschnitt 3.2.6) und aus den Vorteilen eines werkzeuggestützten Ansatzes, die wir insbesondere in unseren Ausführungen zu Intentional Programming (siehe Abschnitt 3.1.2) sowie zu Language Workbenches (siehe Abschnitt 3.1.6) herausgearbeitet haben. Insbesondere bei Software-Produktlinien wird der Entwicklungsprozess in zwei konzeptionelle Hauptstränge zerlegt:

K-3.34 ▷ 87
K-3.15 ▷ 64
K-3.43 ▷ 106

K-3.33 ▷ 85

1. Der erste Strang befasst sich mit der Konzeption und Umsetzung der Software-Produktlinie. Hier werden allgemeine Komponenten und Variationspunkte identifiziert, in eine konsistente Gesamtarchitektur integriert und implementiert. Dieser Strang spiegelt sich in unserer CMDSD-Methodologie wider.
2. Der zweite Strang beschäftigt sich mit der Ausprägung von Produkten auf Basis der eigentlichen Software-Produktlinie. In konkreten Projekten werden Kundenanforderungen mit den Variationsmöglichkeiten der Software-Produktlinie verglichen. Basierend auf den Ergebnissen des Vergleichs werden

Anforderungen direkt umgesetzt oder modifiziert, in manchen Fällen sogar die Software-Produktlinie erweitert. Dies repräsentieren wir durch unsere CDPM-Methodologie.

3. Zwischen den beiden Strängen erfolgt eine Abstimmung (siehe Abbildung 4.2):
   - Neue Funktionalitäten der Software-Produktlinie fließen in Projekte ein und können gegebenenfalls dazu beitragen, existierende Produkte zu verbessern.
   - Übersteigen Anforderungen in Projekten die Möglichkeiten der Software-Produktlinie, ist im Rahmen der Wartung und Weiterentwicklung der Software-Produktlinie darüber zu entscheiden, ob funktionale Anpassungen vorgenommen werden und auf welche Weise diese erfolgen.

   Die Steuerung dieses Abstimmungsvorgangs übernimmt das Produktgremium (siehe Abschnitt 4.2.3.3).

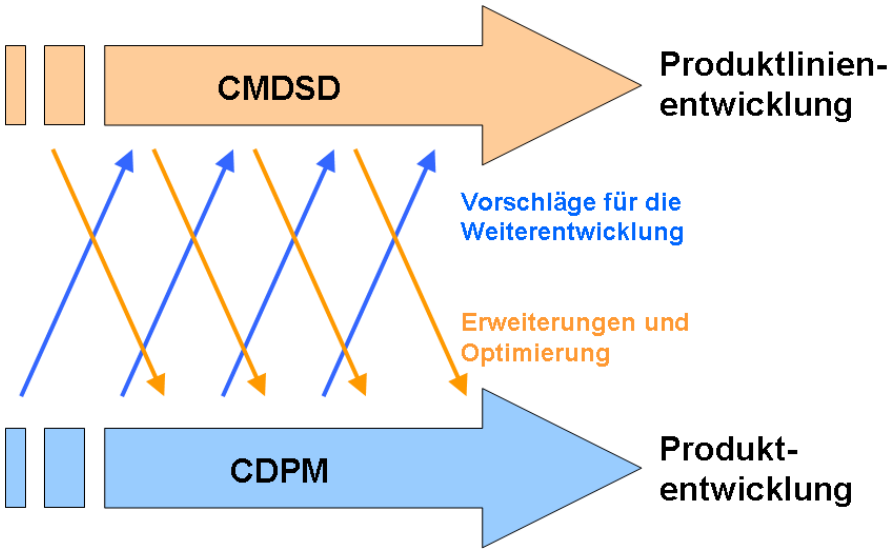


Abbildung 4.2: Wechselwirkungen zwischen CMDSD und CDPM

Um die konkrete Umsetzung des *Conceptual Programming (CP)* zu spezifizieren, stellen wir nachfolgend zunächst kurz die beiden Hauptmethoden des CP vor und führen sie dann in den beiden folgenden Kapiteln detailliert ein.

4.2.3.1 Conceptual Model Driven Software Development (CMDSD)

Conceptual Model Driven Software Development (CMDSD) steht für die agile Definition und Umsetzung mittelstandstauglicher Werkzeuge, die basierend auf den

grundlegenden Werten und Prinzipien aus den Abschnitten 4.2.1 und 4.2.2 die fachmodellgetriebene Entwicklung von Softwarelösungen ermöglichen.

In Kapitel 5 stellen wir das entsprechende Vorgehensmodell ausführlich vor.

4.2.3.2 Concept Driven Project Management (CDPM)

Concept Driven Project Management (CDPM) bezeichnet das Vorgehensmodell bei der Abwicklung von IT-Projekten zur individuellen Entwicklung komplexer IT-Lösungen auf Grundlage von CMDSD-Werkzeugen. Im Fokus steht dabei insbesondere die flexible Behandlung wechselnder Anforderungen, die zielgerichtete und wirtschaftliche Umsetzung dieser Anforderungen sowie die kontinuierliche Weiterentwicklung von Lösungen.

In Kapitel 6 beschreiben wir das CDPM-Vorgehensmodell detailliert.

4.2.3.3 Produktgremium

Um die strategische Entwicklung eines CP-basierten Ansatzes zu steuern, ist ein Gremium erforderlich, das mit folgenden Rollen besetzt ist:

1. Der *Chief Programmer* verantwortet die technische Evolution des CMDSD-Werkzeugs und bewertet Anforderungen aus Projekten hinsichtlich ihrer Verallgemeinerbarkeit (letztlich also der Frage, ob Anforderungen für Erweiterungen die Qualität besitzen, als allgemeine Features in die Produktlinie aufgenommen zu werden).
2. Der *Chief Consultant* koordiniert die Aktivitäten in Kundenprojekten und sammelt wiederkehrende Anforderungen aus Projekten, die Potenzial für eine Verallgemeinerung besitzen. Diese stimmt er mit dem Chief Programmer ab. Außerdem bewertet er neue Anforderungen und zukünftige Entwicklungsnotwendigkeiten, die sich durch strategische Anpassungen auf Seiten der Kunden ergeben. Er diskutiert diese Erweiterungen mit dem Chief Programmer, um sie in die strategische Produktlinienplanung einzubringen.
3. Der *Budgetverantwortliche* prüft kontinuierlich die Wirtschaftlichkeit des Ansatzes und überwacht den Ressourceneinsatz sowohl in den Projekten als auch in der Produktlinienentwicklung.

Mehrere Rollen können auch in Personalunion zusammengefasst werden – allerdings erhöht dies das Risiko ungünstiger Entscheidungen. Insbesondere sollten die Rollen

des Chief Programmers und des Chief Consultants nach Möglichkeit sauber getrennt bleiben, da eine Vermengung von Produkt- und Projektinteressen häufig zu Interessenskonflikten führt, die einer strategischen Entwicklung ebenso abträglich sind wie der Fähigkeit, taktischen Anforderungen in Projekten gerecht zu werden.

Insbesondere auf der strategischen Meta-Ebene bietet sich auch die Verwendung von Feature Driven Development an, um die strategischen Features zu identifizieren, bewerten, priorisieren und dann in die Entwicklung einzuplanen.

K-3.46 ▷ 111

4.3 Weitere Aspekte von CP

In den abschließenden Abschnitten dieses Kapitels erläutern wir, wie die Entwurfssystematik für die beiden Hauptmethoden CMDSD und CDPM unseres Vorgehensmodells strukturiert werden (Abschnitt 4.3.1). Dabei erklären wir auch, wie ein domänenspezifisches CP-basiertes Vorgehensmodell validiert werden kann (Abschnitt 4.3.2) und geben schließlich einige allgemeine Hinweise dazu, wie CP für spezifische Domänen adaptiert werden kann (Abschnitt 4.3.3).

4.3.1 Entwurfssystematik für CMDSD und CDPM

Für den weiteren Entwurf der beiden zentralen Methoden des Conceptual Programming werden wir unsere Systematik von Werten, Prinzipien und Methoden weiterverfolgen (siehe unsere Ausführungen zu dieser prinzipiellen Struktur in Abschnitt 4.1) und so quasi eine rekursive Systematik verwenden, indem wir wiederum Prinzipien und Methoden zur Detaillierung von CMDSD und CDPM angeben. Wir sehen dabei, dass sich die Werte sowohl für CMDSD und CDPM gar nicht von den Werten für CP unterscheiden können und dürfen, denn sonst würden wir versuchen, nicht zusammenpassende Bausteine zu kombinieren. Die Werte von CMDSD und CDPM sind also die gleichen drei Werte wie für CP: *Wirtschaftlichkeit*, *Flexibilität* und *Zugänglichkeit*. Die Prinzipien des CP müssen für CMDSD und CDPM allerdings konkretisiert werden: *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität* werden als Oberbegriffe natürlich ihre Gültigkeit behalten. Da CMDSD sich aber mit Werkzeugen und CDPM sich mit Vorgehensmodellen befasst, sind die Prinzipien so zu verfeinern, dass sie für die jeweils betrachtete Domäne (Werkzeuge vs. Methodik) eine optimale Umsetzbarkeit der allgemeingültigen Werte sicherstellen. Entsprechend werden neue Methoden zu definieren sein, um die verfeinerten

Prinzipien in ihrer jeweils betrachteten Domäne erfolgreich anzuwenden. Wir verweisen für die von uns in diesem Kapitel vorgenommene Unterteilung auf unsere Herleitung in Abschnitt 3.6.

Gestärkt durch diese Festlegungen werden wir in den folgenden beiden Kapiteln die vier Grundprinzipien für Werkzeuge und Vorgehensmodelle verfeinern und mit neuen Methoden unterfüttern, um so eine optimale Realisierung der von uns angestrebten Werte zu erreichen. Komplexere IT-Projekte können damit auf Grundlage von CP mittelstandsgerecht gestaltet werden.

4.3.2 Möglichkeiten zur Validierung eines CP-basierten Entwurfs

Hat sich ein mittelständischer Dienstleister zur Umsetzung einer CP-basierten Vorgehensweise für eine von ihm festgelegte Domäne entschieden und haben alle Kosten- und Risikoabwägungen ergeben, dass er die Entwicklung eines domänenspezifischen CP-Ansatzes durchführen sollte, stellt sich die interessante Frage, ab wann die Umsetzung als erfolgreich angesehen werden kann, wie also der (umgangssprachliche) Lackmustext für den Erfolg der CP-Umsetzung aussehen kann.

Mit dem „Zwei-Tages-Test“ greifen wir einen Vorschlag von Martin (aus [Mart81], S.106–107) auf[5], der die folgenden Erwartungen an ein Produkt stellt:

- Die *meisten* Endanwender sollten in der Lage sein, die zugrunde liegenden Werkzeuge und Produktkomponenten nach einer zweitägigen Einführung effektiv zu benutzen. Manche sollten das auch schneller lernen.
- Am Ende eines zweitägigen Kurses sollten sich die Anwender mit dem Produkt *wohlfühlen* und es selbständig verwenden können.
- Am Ende des Kurses sind die Anwender in der Lage, *erste nützliche* Arbeitsergebnisse mit dem Werkzeug zu erzielen. Die Betonung auf *nützlich* ist dabei sehr wichtig. Es macht keinen Sinn, Gimmicks zu erlernen, die keine echte Relevanz besitzen.
- Nach Beendigung des Kurses sollten die Teilnehmer nach einer Woche Pause immer noch in der Lage sein, das Werkzeug zu benutzen (die meisten Anwender

[5] Mit dem wir bei unserer täglichen Arbeit interessanterweise praktisch selbst gute Erfahrungen gemacht hatten, obwohl er bis dato formal nicht bekannt war. Die enthaltenen subjektiven Kriterien dienten bei vielen Tätigkeiten als Erkenntnisindikator für die Frage, wie gut ein System oder eine Anwendung tatsächlich funktioniert.

werden Tastaturkürzel und sonstige festen Eingabesequenzen nach einer Woche wieder vergessen haben).

- Endanwender müssen nicht zwingend weitere Schulungen machen, um das Produkt sinnvoll einsetzen zu können.
- Endanwender können ihr Wissen selbständig auffrischen oder erweitern, indem sie die Online-Hilfe verwenden.

Zudem weist Martin noch darauf hin, dass in manchen Fällen in einem zweitägigen Kurs nur eine Untermenge der Eigenschaften eines Produktes vermittelt werden kann und es dann durchaus legitim ist, in späteren Kursen fortgeschrittenes Wissen aufzubauen.

Aus unserer Sicht sind das gelungene und pragmatische Kriterien für die Zugänglichkeit eines Systems (siehe Abschnitt 4.2.1.3), zumal sie auch gleich die Auswirkungen der Zugänglichkeit für die beiden von uns formulierten Werte *Wirtschaftlichkeit* und *Flexibilität* indizieren: Es dürfen nicht zwingend weitere Schulungen erforderlich sein (es kann also Geld gespart werden) und es müssen produktive Ergebnisse nach einer solchen Schulung erzielbar sein (also ein früher *Return on Invest*)[6].

4.3.3 Adaption von CP

Bevor in den folgenden Kapiteln die Methoden des CP basierend auf den vorgestellten Werten, Prinzipien und Methoden im Detail eingeführt werden, müssen wir noch mal darauf hinweisen, dass es sich bei Conceptual Programming (inklusive seiner nachfolgend beschriebenen Methoden) um ein domänenunabhängiges Metamodell handelt. D. h. je nach Domäne kann es erforderlich werden, weitere Werte, Prinzipien oder Methoden auf Grundlage des vorhandenen Domänenwissens zu ergänzen und in die Systematik zu integrieren. Beispielsweise müsste der Wert *Rechtssicherheit* (sowohl für Kunden als auch Dienstleister) in einer strengen gesetzlichen Vorgaben unterliegenden Fachdomäne durch das Prinzip der *Rechtsverbindlichkeit* (hinsichtlich aller relevanten Absprachen) und durch Methoden wie *Notarielle Beglaubigung* o. ä. untermauert werden.
Die Ausprägung solcher domänenspezifischen Werte, Prinzipien und Methoden, die

[6] Seitenbemerkung: Es ist immer wieder amüsant, wie viele der heute verfügbaren IDE-Schulungen mindestens fünf Tage Umfang haben und dann noch drei weitere ähnliche Schulungen erfordern, bis überhaupt produktionsrelevante Ergebnisse erzielt werden können. Das die Anbieter der entsprechenden Produkte viele Vorteile aus dem Schulungsaufwand ziehen, ist klar, dass sich die Kunden nicht schon längst lautstark nach besseren Lösungen beklagen, verwundert doch immer wieder.

sowohl CP selbst als auch die nachfolgend vorgestellten Methoden des CMDSD und CDPM betreffen können, ist entsprechend nicht Bestandteil dieser Arbeit, da sie nur auf Grundlage der jeweiligen domänenspezifischen Anforderungen geschehen kann. Wichtig ist aber die Erkenntnis, dass eine solche Detaillierung von CP, CMDSD und CDPM für jede spezifische Fachdomäne erfolgen muss, um den besonderen Rahmenbedingungen gerecht zu werden. Letztlich führt dieser einmalige Aufwand aber wieder dazu, dass die operativen Prozesse und Werkzeuge optimaler auf den Domänenkontext eingestellt werden und somit der eigentliche Entwicklungsprozess für Lösungen wieder reibungsloser und professioneller umgesetzt wird. Die Beschreibungen der beiden Hauptmethoden enthalten entsprechende Hinweise zu Adaptionsmöglichkeiten in den Abschnitten 5.5.4 und 6.5.6.

4.4 Zusammenfassung

Wir haben in diesem Kapitel zunächst die grundlegende Aufteilung von *Conceptual Programming (CP)* in *Conceptual Model Driven Software Development (CMDSD)* und *Concept Driven Project Management (CDPM)* motiviert. Anschließend haben wir als Grundlage für die konkrete Ausprägung von CP, CMDSD und CDPM eine allgemeine Infrastruktur aus Werten, Prinzipien und Methoden definiert und deren Nachhaltigkeit für die ganzheitliche Entwicklung neuer Vorgehensmodelle gezeigt. Basierend auf dieser allgemeinen Systematik haben wir ausgehend von unseren Vorüberlegungen in den vorangegangenen Kapiteln 1 bis 3 die konkreten Werte, Prinzipien und Methoden des Conceptual Programming definiert. CP setzt sich zusammen aus agilen Methoden, Erkenntnissen aus den Bereichen modellgetriebener Software-Entwicklung, der Erzeugung von Software-Produktlinien sowie der Ausprägung von Software Factories. Daraus resultierte notwendigerweise die bereits in der Einleitung motivierte Aufspaltung von CP in CMDSD und CDPM. Diese beiden Hauptmethoden werden in den folgenden Kapiteln konkretisiert. Zudem haben wir auf die Notwendigkeit domänenspezifischer Instanziierungen von CP in Abschnitt 4.3.3 hingewiesen. Interessant ist in diesem Zusammenhang, dass es schon seit geraumer Zeit in der Informatik Überlegungen und Bestrebungen gibt, die CP sehr ähnlich sind (siehe z. B. [Mart81] auf S. 63 für die Aufforderung zu einem ähnlichen Ansatz aus dem Jahr 1981), sich aber nie ein entsprechender ganzheitlicher Ansatz entwickelt hat (der Rational Unified Process kommt dem noch am Nächsten, siehe unsere Hinweise in [Vers00, Kruc04]). Erst in den letzten Jahren sind alle methodischen Bausteine z. B. in Form von Agilität und Domain Driven Design zunehmend formal

erforscht worden, so dass sie nun zusammen mit den heute verfügbaren Werkzeugmöglichkeiten diese Vision wirklich machen können. Frühere Ansätze wie z. B. CASE konzentrierten sich immer nur auf Teilaspekte (bei CASE z. B. auf eine möglichst effiziente technische Beschreibung eines Systems).

Kapitel 5

Conceptual Model Driven Software Development (CMDSD)

„The eyes see only what the mind is prepared to comprehend.“
— Henri Bergson (1859–1941), Philosoph

In diesem Kapitel wird die grundlegende Methodik des *Conceptual Model Driven Software Development (CMDSD)* vorgestellt, die wir initial in [BiMa06] präsentiert haben. Ziel von CMDSD ist die Entwicklung spezieller auf einem Software-Produktlinien-Ansatz basierender Werkzeuge, die mittels eines modellorientierten Ansatzes die Grundlage für das im folgenden Kapitel 6 beschriebene *Concept Driven Project Management (CDPM)* schaffen. Durch Kombination beider Methoden erreichen wir wie bereits in Kapitel 4 ausgeführt das Ziel dieser Arbeit, auch komplexere IT-Individualprojekte im mittelständischen Kontext unter den in Kapitel 2 erarbeiteten, erschwerenden Rahmenbeschränkungen effektiv umsetzen zu können.

Abschnitt 5.1 geht zunächst auf die grundlegenden Ziele ein, die wir mit unserem CMDSD-Ansatz erreichen wollen. Abschnitt 5.2 motiviert den Ansatz durch eine kurze grundlegende Diskussion über die Motivation hinter der spezifischen Vorgehensweise. Dabei bauen wir auf den Ergebnissen der vorhergehenden Kapitel auf. Abschnitt 5.3 beschreibt CMDSD im Überblick und verweist dazu auf die Methoden, die bei der Umsetzung des Ansatzes angewendet werden müssen. Diese werden im Rahmen der detaillierten Beschreibung des CMDSD-Ansatzes aufbauend auf grundlegenden Werten und Prinzipien in Abschnitt 5.4 vorgestellt. Es folgt Abschnitt 5.5, der weitere bei der praktischen Umsetzung relevante Gesichtspunkte präsentiert. In Abschnitt 5.6 fassen wir die besonderen Leistungen dieses Kapitels noch einmal kurz zusammen.

Wie bereits in Abschnitt 3.6 auf Seite 135 erklärt, werden wir die von uns hergeleiteten Konzepte als Randmarkierung referenzieren, wenn Sie für unsere Ausführungen bedeutsam sind.

5.1 Ziele

„Du siehst Dinge und fragst 'Warum?',
doch ich träume von Dingen und sage 'Warum nicht?'."
— George Bernhard Shaw (1856–1950), Irischer Schriftsteller

> *Conceptual Model Driven Software Development (CMDSD, fachmodellgetriebene Softwareentwicklung) ist ein modellbasierter und generativer Ansatz auf Basis domänenspezifischer Modelle, die auf Grundlage einer Software-Produktlinie strategisch weiterentwickelt und in Lösungsinstanzen überführt werden.*

CMDSD beschreibt die Infrastruktursystematik für Werkzeuge, die im Rahmen des CP-Ansatzes zur Durchführung mittelständischer IT-Projekte eingesetzt werden sollen. Primäres Ziel ist es dabei, zu einer Infrastruktur zu gelangen, mit denen auch komplexere Projekte flexibel und zeitnah unter den besonderen in den vorausgegangenen Kapiteln erläuterten Einschränkungen von KMUs durchgeführt werden können. Zur Erreichung der in Abschnitt 4.2.1 definierten Werte werden wir die Prinzipien aus Abschnitt 4.2.2 verfeinern und Methoden angeben, die eine Realisierung von CMDSD-Werkzeugen unterstützen. Spezielle Ziele von CMDSD sind:

K-3.4 ▷ 56
K-3.41 ▷ 101

- die Entwicklung eines Vorgehensmodells, das es fachlichen und technischen Experten erlaubt, fachlich getriebene Modelle zu entwickeln, die fachliche Anforderungen fokussieren und auch für Domänenexperten ohne technische Kenntnisse weitestgehend intuitiv verständlich sind,

K-3.32 ▷ 83
K-3.42 ▷ 105

- die automatische Generierung von Programmbausteinen und vollständig ausführbaren Anwendungen aus diesen Modellen heraus (basierend auf MDSD bzw. generativer Programmierung) sowie
- die Erhaltung des Anwendungswissens in den Unternehmen, für die Anwendungen erstellt werden durch explizite und für Domänenexperten verständliche Dokumentation der entwickelten Systeme.

Zur Umsetzung unserer Ziele werden wir die folgenden konkreten Ansätze kombinieren:

- Model Driven Software Development (siehe Abschnitt 3.3.2) wird in Kombination mit Software-Produktlinien (siehe Abschnitt 3.2.6) das (bereits gut erforschte) Fundament für die Werkzeugentwicklung inklusive vielfältiger Erfahrungen zur Evolution und Ausprägung konkreter Projektergebnisse mit einer solchen Produktlinie. MDSD wird als Methodik für die konkrete technische Ausgestaltung der Software-Produktlinie verwendet. Abschnitt 5.4.3.3 beschreibt dieses Vorgehen detailliert.

  K-3.34 ▷ 87
  K-3.42 ▷ 105

  Software Factories (siehe Abschnitt 3.3.3) werden nach aktuellem Kenntnisstand in der Zukunft eine bedeutsame Rolle bei der Entwicklung von CMDSD-Werkzeugen spielen und entsprechend wichtig ist eine genaue Kenntnis der zugrunde liegenden Ideen und der konkreten Software Factories, die sich zunehmend am Markt positionieren.

- Domain Driven Design (siehe Abschnitt 3.2.7) zur Kapselung der spezifischen Domäne, auf die sich das Werkzeug fokussiert. Abschnitt 5.4.3.7 beschreibt die Vorgehensweise dazu genauer.

  K-3.36 ▷ 89
  K-3.37 ▷ 90

- Rapid Prototyping (siehe die verschiedenen Ansätze aus Abschnitt 3.4.3) für die essentiellste CMDSD-Werkzeugeigenschaft, die nachfolgend eine tragende Rolle in der Umsetzung der Projektmethodik CDPM (siehe Kapitel 6) spielen wird. Abschnitt 6.4.3.3 beschreibt die Umsetzung dieser Anforderung detaillierter.

  K-3.25 ▷ 73
  K-3.31 ▷ 81

- Erfahrungen und erfolgreiche Lösungsansätze aus den verschiedenen in Kapitel 3 beschriebenen existierenden Vorgehensmodellen werden gebündelt, um die von uns formulierten Werte bzw. Ziele effektiver zu erreichen. Die Abschnitte 5.4.3.9 bis 5.4.3.12 beschreiben diese Spezialisierung detailliert.

5.2 Das fachmodellgetriebene Manifest (Teil I)

„Everyone thinks of changing the world, but no one thinks of changing himself.“
— Leo Tolstoi (1828–1910), Russischer Schriftsteller

Bevor das Paradigma des *Conceptual Model Driven Software Development (CMDSD)* vorgestellt wird, wollen wir anhand eingängiger Thesen verdeutlichen, welche grundlegend andere Philosophie hinter der Entwicklung von Software mit

CMDSD steht[1]. Dabei handelt es sich um Ideen und Thesen, die einen ähnlichen Anspruch an die Veränderung von Software-Entwicklungsprozessen haben wie schon das *Cluetrain Manifest* in seiner Einfachheit die Beziehung zwischen Kunden und Unternehmen neu definiert hat (siehe [LLSW00]): Der Abbau jeglicher Distanz zwischen Kunden und Anbietern durch das Aufkommens des Internet wurde dort als Katalysator für die Notwendigkeit gesehen, zukünftig in Unternehmenskommunikation und Marketing eine wesentlich engere Form des gemeinsamen Dialogs zu finden. Wir sehen in der Notwendigkeit zur Bewältigung komplexer IT-Themen unter mittelständischen Rahmenbedingungen einen vergleichbaren Katalysator für ein Umdenken in der Software-Entwicklung. CMDSD-Werkzeuge sollen die Grundlage liefern, um in Projekten ein neues über die ohnehin bereits vorhandene Kundenbezogenheit herkömmlicher agiler Ansätze hinausgehendes Projektmanagementmodell zu definieren, dass die intensive Zusammenarbeit mit dem Kunden in den Mittelpunkt rückt[2].

Strukturell gesehen könnte man insbesondere auch im Sinne von [CoHa05], S. 29, diese Thesen als *Muster* oder *Entwurfsmuster* ansehen, die beschreiben, was den CMDSD-Prozess ausmacht. Da in diesem Kapitel aber die Idee eines gänzlich neuen Vorgehens bei der Erstellung von Software im Vordergrund steht, wird bewusst auf die Begrifflichkeit des Musters verzichtet, da dieser Begriff zu sehr technisch vorbelegt ist durch das Standardwerk [GHJV96]. Die fachmodellgetriebene Entwicklung von Software will sich aber gerade von einer technisch ausgeprägten Sicht der Welt distanzieren, so dass die Verwendung einer alternativen Begrifflichkeit sinnvoll erscheint. Dabei greifen wir erneut auf das bereits in Abschnitt 4.1 eingeführte Konstrukt aus Werten, Prinzipien und Methoden zurück.

Wie wollen wir aber geeignete Entwicklungswerkzeuge realisieren? Dazu müssen wir die Innovations- und Evolutionszyklen der Software-Industrie betrachten: Entwicklungs- und Architekturparadigmen sind dort einem ständigen Wechsel unterworfen. Waren beispielsweise noch vor wenigen Jahren Client-/Serverarchitekturen der Standard, sind heute Webportale und schichtenbasierte Webapplikationen eine dominierende Architekturform. Für jeden diesen Trends gilt aber: Nach einer initialen Adoptionsphase formulieren sich Standardverfahren, um typische Probleme und Herausforderungen der jeweiligen Architekturform zu lösen – häufig als Architekturmodelle (im Großen) oder Entwurfsmuster (im Kleinen) bezeichnet (siehe z. B. [Sche94, Bers96, GHJV96, OrHa98, Govo99, RoMo02, Fowl03, TKLM03]).

[1] Teil II des „fachmodellgetriebenen Manifests“ findet sich in Abschnitt 6.2 als Motivation unseres CDPM-Ansatzes.

[2] Insofern ist dieses Kapitel auch eine Einleitung zu unserem in Kapitel 6 beschriebenen Projektmanagementmodell, dass sich mit den Werten, Prinzipien und Methoden befasst, die bei der Ausprägung von Lösungsinstanzen nachhaltige Vorteile bringen.

Somit gilt es, den Zeitpunkt, an dem sich grundlegende Architekturen und Entwurfsmuster für eine technische Plattform in hinreichender Qualität und vor allem Stabilität herausgebildet haben, abzupassen und auf diesem technischen Wissen aufbauend eine entsprechende fachmodellbasierte Entwicklungsplattform für hinreichend umfassende Problemdomänen bereitzustellen. Auf einer solchen Standardplattform aufbauend können Entwickler dann schneller und leistungsfähiger ausgereifte Anwendungssysteme bereitstellen. Im Gegensatz zu aktuell vorherrschenden MDSD-Strömungen (siehe z. B. [VöSt05]) wird also im CMDSD-Ansatz davon ausgegangen, dass die führenden Hersteller technologischer Plattformen oder hinreichend spezialisierte Systemhäuser für ausgewählte Problemdomänen CMDSD-basierte Entwicklungsplattformen bereitstellen. Wird die Problemdomäne angemessen eingeschränkt, wird eine derartige Entwicklung auch für mittelständische IT-Dienstleister und Systemhäuser leistbar wie unsere Evaluierung in Kapitel 9 zeigen wird.

Praktische Erfahrungen (siehe z. B. [Ehle05]) dokumentieren den Wert dieses Ansatzes, denn es werden bereits Kosteneinsparungen durch die automatisierte Generierung in der Praxis bewährter Systemarchitekturen erreicht. Überlegungen sowohl von Wissenschaft als auch Industrie weisen durch die Arbeit an Software-Produktlinien und Software Factories (siehe die Abschnitte 3.1.2, 3.1.6, 3.2.6 und 3.3.3) ebenso in diese Richtung, wobei die Literatur zu beiden Themen im Gegensatz zu CMDSD bisher lediglich darum bemüht ist, *Entwicklern* die Arbeit zu erleichtern.

Die eigentliche Komplexität liegt also nicht in der technischen Modellierung und Umsetzung von Systemen, sondern in der leistungsfähigen Erfassung und Beschreibung der fachlichen Anforderungen. Was liegt also näher, als eine Modellierungstechnik vorzusehen, die alle technischen Rahmenbedingungen außen vorlässt und sich auf die relevanten fachlichen Schwerpunkte konstruiert, um so eine gemeinsame Modellierungssprache für Architekten, Fachanwender und Entwickler zu schaffen? Insbesondere regen wir an, dass Software-Entwicklung von Standardsystemen zukünftig nur noch durch sogenanntes *Conceptual Pair Programming* erfolgt, d. h. dass Fachanwender und Entwickler gemeinsam das fertige System entwickeln (siehe dazu später Abschnitt 6.4.3.5). Um dies zu ermöglichen, wird im Folgenden ein Modell vorgeschlagen, dass es für Standardlösungen erlaubt, das weitgehend vollständig lauffähige Systeme direkt aus der Fachspezifikation zu generieren[3]. So wird es möglich, dass

[3]Trotz der nachfolgend dargestellten fundierten Grundlage wird es in der Praxis wohl niemals zu einer 100%-igen Generierung kommen, da es in jedem Projekt so spezielle Anforderungen gibt, dass diese doch individuell ausprogrammiert werden müssen. Praktische Erfahrungen und Fallstudien zeigen aber, dass für ein hinreichend klar abgegrenztes Problemfeld eine automatisierte

Fachanwender und Entwickler gemeinsam die funktionalen Anforderungen des Systems spezifizieren und noch während der Spezifikation auf einen Prototypen zugreifen können, um die Korrektheit und Sinnhaftigkeit der Spezifikation am „lebenden“ Ergebnis zu überprüfen.

Die konsequente Verwendung von CMDSD impliziert auch, dass sich die grundlegende Natur von IT-Projekten verändern wird. Die Entwicklung von CMDSD-Umgebungen wird bei einzelnen Herstellern liegen und IT-Projekte selbst werden von einem stärkeren Beratungscharakter geprägt sein als dies heute der Fall ist: Zukünftige IT-Projektmitarbeiter werden sich darauf fokussieren (müssen), die fachlichen Zusammenhänge gemeinsam mit dem Fachexperten korrekt zu modellieren. Die Details der technischen Umsetzung werden in den meisten Fällen durch eine CMDSD-Umgebung übernommen, so dass die Technik in den Hintergrund tritt. Angesichts aktueller Offshoring-Bemühungen in der IT-Branche scheint dies auch bei Betrachtung der Parallelen zu anderen Wirtschaftszweigen und -branchen nur konsequent und natürlich: Die repetitiven und einfachen Tätigkeiten („Standardprogrammierung“ in der IT-Branche) werden zunächst in Billiglohnländer verlagert und anschließend in einem nächsten Schritt sogar weitgehend automatisiert. Dafür werden bisher eher elitäre und zahlenmäßig kleine Berufsgruppen zunehmend bedeutsamer (z. B. Architekten oder Domänenexperten auf IT-Seite).

5.3 CMDSD im Überblick

K-3.6 ▷ 60
K-3.13 ▷ 64

Auf Grundlage der Ergebnisse aus den vorausgegangenen Kapiteln setzen wir CMDSD wie nachfolgend beschrieben auf[4]:

1. *Wähle eine Domäne, für die Conceptual Programming als Lösungsansatz für IT-Individualprojekte verfolgt werden soll.* Hierbei kann es sich üblicherweise

Generierung von im Durchschnitt 90% der Anwendung erreicht werden kann (siehe dazu auch unsere Erfahrungen in Abschnitt 9.1.2.2).

[4]Wir weisen explizit darauf hin, dass viele der nachfolgenden vagen Formulierungen wie „vertretbarer Aufwand“, „hinreichende Erfahrung“, etc. bewusst gewählt sind, da mittelständische IT-Dienstleister in ihrer Struktur ebenso variabel wie mittelständische Unternehmen sind und sich so keine harten Zahlen vorgeben lassen. Für einen Dienstleister mit 10 Mitarbeitern gelten andere Voraussetzungen als für einen mit 100 Mitarbeitern, auch wenn beide mittelständisch ausgeprägt sind. Daher beschränken wir uns auf solche vagen Formulierungen und weisen darauf hin, dass insbesondere Machbarkeitsstudien (wenn auch häufig in einer mehr von Bauchgefühl als von harten Zahlen geprägten Form) im Mittelstand zum Tagesgeschäft gehören und mittelständische Unternehmen daher ein sehr gutes intuitives Verständnis dafür besitzen, was sie leisten können. Daher halten wir diese vagen Charakterisierungen auch für ausreichend. In Kapitel 9 entwickeln wir ein Formelsystem für eine zahlengetriebene Form der Risikoanalyse.

nur um eine Domäne handeln, zu der bereits extensives Domänenwissen vorhanden ist, da mittelständische IT-Dienstleister üblicherweise nicht über die Ressourcen verfügen werden, den Aufbau von Expertenwissen zu einer neuen Domäne in hinreichend kurzer Zeit durchzuführen (siehe dazu die vorausgegangenen Ausführungen in Abschnitt 4.2.2.2). Insbesondere muss ein *Chief Programmer* mit exzellentem Domänenwissen gefunden werden, der das Gesamtprojekt steuert (siehe Abschnitt 5.4.3.1). K-3.30 ▷ 80

2. *Identifiziere dabei einen Architekturstil als Grundlage für die zu entwickelnde Software-Produktlinie*. Dieser Schritt hilft einem IT-Dienstleister dabei, zu einer realistischen Einschätzung seiner eigenen Leistungsfähigkeit zu kommen (siehe Abschnitt 5.4.3.2). K-3.7 ▷ 61

3. *Setze die Entwicklung des CMDSD-Werkzeugs auf Basis der Theorie zu Software-Produktlinien auf.* Wir haben Software-Produktlinien bereits in Abschnitt 4.2.3 als adäquates Fundament zur Umsetzung von *Conceptual Programming* identifiziert. Zur prinzipiellen Vorgehensweise liegt vielfältige Literatur sowie eine Vielzahl von Erfahrungswerten vor (siehe dazu unsere Ausführungen in Abschnitt 3.2.6). Mit der im nachfolgenden Abschnitt 5.4.3.3 definierten CMDSD-Methode zur Kombination von MDSD mit Software-Produktlinien beschreiben wir detailliert, wie die Umsetzung von CMDSD-Werkzeugen auf dieser Grundlage erfolgen muss. Prinzipiell werden wir dabei mehrstufig vorgehen:

   a) Prüfe für jede Methode, die für die Entwicklung von Software-Produktlinien empfohlen wird auf domänen- und kontextabhängige Eignung. Dokumentiere alle Ergebnisse und unterscheide dabei zwischen K-3.26 ▷ 74

      i. Methoden, die unverändert übernommen werden,
      ii. Methoden, die modifiziert übernommen werden[5],
      iii. Methoden, die durch andere Methoden ersetzt werden und
      iv. Methoden, die gänzlich ignoriert werden.

      Insbesondere ist hier zu untersuchen, ob zu komplex erscheinende Eigenschaften von Software-Produktlinien ersetzt werden sollten (und können) durch Eigenschaften und Methoden des Feature Driven Development, das gerade für den Entwicklungsprozess ein wesentlich schlankeres Vorgehen erlaubt (siehe Abschnitt 5.4.3.3 für weitere Ausführungen). K-3.46 ▷ 111

[5] Hier sind sowohl die Modifikationen selbst als auch die Gründe für die Modifikationen zu dokumentieren, da dann im weiteren Verlauf regelmäßig geprüft werden kann, ob die modifizierten Methoden erneut anzupassen sind.

b) Konzipiere das CMDSD-Werkzeug. Dies ist ein domänenspezifisches Modellierungswerkzeug, dass die modellbasierte Generierung von Lösungen aufgrund spezifischer Fachmodelle ermöglicht. Insbesondere erlaubt es die Erstellung und Pflege aller relevanten Modelltypen (siehe wiederum Abschnitt 5.4.3.3).

c) Konzipiere basierend auf dem CMDSD-Modellierungswerkzeug eine Lösung für die gegebene Domäne. Diese dient als Grundlage einer kontinuierlichen Weiterentwicklung und Vermarktung basierend auf dem Software-Produktlinien-Ansatz (siehe erneut Abschnitt 5.4.3.3).

d) Definiere ein Projektmanagementmodell als Kombination aus Software-Produktlinienansätzen, Feature Driven Development und agilen Vorgehensmodellen, dass zu den bisherigen Methoden und zur Unternehmenskultur des Dienstleisters passt (siehe Abschnitt 5.4.3.4).

4. *Identifiziere den oder die Archetypen, die die Domäne aus Anwendersicht charakterisieren.* Abschnitt 5.4.3.5 beschreibt eine Herangehensweise zur Archetypisierung sowie die Nutzmöglichkeiten der so gewonnenen Archetypen.

5. *Identifiziere basierend auf den Archetypen die tatsächlichen Anforderungen, die sich aus der betrachteten Domäne ergeben.* Hier ist es wichtig, eine Abgrenzung gemäß dem Pareto-Prinzip vorzunehmen: Es ist allemal ausreichend für den mittelständischen Kontext, wenn ein CMDSD-Werkzeug die typischen 80% der Anforderungen abdecken kann, da sich für weitergehende Forderungen gut im Einzelfall (also im Rahmen eines Projektes) entscheiden lässt, ob die Umsetzung dieser Anforderungen dem wirtschaftlichen Rahmen entgegenkommt. Dies ist auch ein maßgeblicher Unterschied zu ähnlich gelagerten Plattformen, die bei Großunternehmen im Einsatz sind und die üblicherweise auch sehr umfangreiche (und entsprechend kostspielige und komplex zu erlernende) Möglichkeiten enthalten, um auch die 20% der Anforderungen abzudecken, die eher exotisch oder sehr speziell sind (siehe Abschnitt 5.4.3.6).

K-3.39 ▷ 93

6. *Verwende Domain Driven Design zur Domänenabgrenzung.* Dieser Schritt ist besonders wichtig, um sich den Umfang und die Komplexität der Domäne explizit klar zu machen. Auf dieser Basis kann dann eine Machbarkeitsentscheidung für das weitere CP-basierte Vorgehen getroffen werden (siehe Abschnitt 5.4.3.7).

7. *Identifiziere die zur Abbildung der Domäne erforderlichen Modelle und klassifiziere sie.* In diesem Schritt gilt es, die zuvor gefundenen Archetypen durch eine endliche (und möglichst überschaubare) Menge von Modellen formal be-

schreiben zu können. Abschnitt 5.4.3.8 liefert das notwendige Fundament für diesen Schritt, Abschnitt 5.4.3.8.1 beschreibt insbesondere eine Klassifikation von Modellen.

8. *Definiere die Modelle formal und reduziere die Modelle auf das Wesentliche.* Nachdem die prinzipiellen Modellformen im vorausgegangenen Schritt identifiziert wurden, sind diese nun präziser zu beschreiben und so kompakt wie möglich zu gestalten, um ihren Einsatz im Rahmen des im nächsten Kapitel beschriebenen CDPM sie effektiver einsetzen zu können. Abschnitt 5.4.3.9 beschreibt ein optimiertes Vorgehen zur Definition von Modellen und zur Reduktion der Modellkomplexität.

9. *Entwickle einen Generator, der aus den Modellen lauffähige Lösungen erzeugt.* Hier bietet sich insbesondere der Rückgriff auf generative Programmierung und modellgetriebene Entwicklung an als Grundlage für das benötigte Handwerkszeug zur Entwicklung solcher Generatoren.

K-3.42 ▷ 105
K-3.47 ▷ 112

10. *Entwickle das Werkzeug zur Modellpflege.* Ebenso wie die Modelle und der Generator muss das Modellierungswerkzeug die 80% der typischen Fälle gut ausdrücken können und für alle weiteren Anforderungen lediglich sinnvolle Erweiterungspunkte anbieten.

K-3.3 ▷ 55
K-3.43 ▷ 106
K-3.44 ▷ 107

11. *Entwickle den oder die zuvor identifizierten Archetypen zur Charakterisierung der Domänenlösung als Bestandteil der Software-Produktlinie.* Prinzipiell wird dazu bereits das CMDSD-Werkzeug verwendet, um so in sich geschlossene Archetypen (also solche, die durch die definierten Modelle vollständig abbildbar sind) zu erhalten. Dies sichert die maximale Erweiterbarkeit und Flexibilität im Rahmen projektspezifischer Ausprägungen und liefert zudem eine grundlegende Werkzeugmenge zur beschleunigten Ausprägung von Lösungsinstanzen.

Die weiteren hier definierten Prinzipien und Methoden sind Bestandteil jeder Tätigkeit, die Überwachung ihrer Anwendung obliegt dem Chief Programmer (siehe Abschnitt 5.4.3.1).

5.4 CMDSD als Paradigma

„Die Grenzen meiner Sprache sind die Grenzen meiner Welt.“
— Ludwig Wittgenstein (1889-1951), Österreichischer Philosoph

CMDSD ist in seiner Beschaffenheit vergleichbar mit dem Ansatz, der bei agiler Software-Entwicklung verfolgt wird. Allerdings liegt unser Fokus hier auf der Kunden- und Domänenorientierung. Zur Darstellung der Ideen von CMDSD verwenden wir das bewährte Konzept von Werten, Prinzipien und Methoden aus Kapitel 4.1., wobei die Thesen ähnlich strukturiert sind wie die des Agilen Manifests (siehe [BBBC01]).

Abschnitt 5.4.1 geht auf die grundlegenden Werte des CMDSD ein, Abschnitt 5.4.2 beschreibt die fundamentalen Prinzipien als Erweiterungen und Detaillierungen der Prinzipien des CP und Abschnitt 5.4.3 gibt schließlich konkrete Methoden vor, um CMDSD erfolgreich anzuwenden.

5.4.1 Werte

Wie bereits in Abschnitt 4.3.1 ausgeführt fußt CMDSD auf den gleichen Werten wie CP, also *Wirtschaftlichkeit* (Abschnitt 4.2.1.1), *Flexibilität* (Abschnitt 4.2.1.2) und *Zugänglichkeit* (Abschnitt 4.2.1.3) und entsprechend sind weder Änderungen noch Ergänzungen erforderlich. Allerdings werden die aus den Werten resultierenden Prinzipien nachfolgend eine Verfeinerung erfahren, da wir sie jetzt im besonderen Hinblick auf die Anforderungen betrachten, die sich bei der Entwicklung von Werkzeugen ergeben.

5.4.2 Prinzipien

Basierend auf den zuvor formulierten Grundwerten des CMDSD definieren wir als Nächstes in Anlehnung an unsere Ausführungen aus Abschnitt 4.1.2 die zentralen Prinzipien, denen wir bei der Umsetzung von CMDSD-Werkzeugen folgen. Im Einzelnen sind dies

- als Verfeinerungen des Prinzips der Kundenorientierung aus Abschnitt 4.2.2.1:
    - *Vereinfachung* (Abschnitt 5.4.2.1), um Werkzeuge optimal auf die kurzen Projektlaufzeiten und das häufig nur begrenzte IT-Know-How mittelständischer Ansprechpartner zuzuschneiden,
    - *Problemorientierung* (Abschnitt 5.4.2.2), um die Modellierung fachlicher Probleme durch IT- und Domänenexperten so direkt wie möglich umzusetzen und

  - *Uniformität* (Abschnitt 5.4.2.3), um Einarbeitungsaufwände in CMDSD-Werkzeuge so gering wie möglich zu halten und auch die Implementierungsaufwände in der Werkzeugentwicklung aus Dienstleistersicht zu minimieren.

- als Verfeinerungen des Prinzips der Domänenfokussierung aus Abschnitt 4.2.2.2:
  - *Standardkonformität* (Abschnitt 5.4.2.4), um sicherzustellen, dass auch nur solche Domänen durch CMDSD bzw. CP angegangen werden, die bereits hinreichend scharf umrissen sind und für die es zudem einsatzbare Standards gibt – seien diese technischer oder fachlicher Natur und
  - *Kompaktheit* (Abschnitt 5.4.2.5), um einen möglichst schnellen Überblick über die essentiellen Informationen zu einem Aspekt zu erhalten.
- als Verfeinerungen des Prinzips der Automatisierung aus Abschnitt 4.2.2.3:
  - *Abstraktion* (Abschnitt 5.4.2.6), um dem in Kapitel 1 und 2 erläuterten Ist-Problem äußerst komplexer Lösungen und Werkzeuge (aus dem Kontext von Großprojekten) insoweit Rechnung zu tragen, dass CMDSD-Werkzeuge eine Balance finden müssen zwischen Ausdrucksstärke und Vereinfachung und somit Richtlinien für „gute Abstraktionen“ zwingend zu berücksichtigen sind,
  - *Lokalität und Linearität* (Abschnitt 5.4.2.7), um Informationen so zusammenzuhalten, dass sie für die jeweils aktuelle Tätigkeit entweder möglichst kompakt verfügbar oder möglichst leicht referenzierbar sind und
  - *Tradition und Innovation* (Abschnitt 5.4.2.8), um sicherzustellen, dass Erfahrungswerte und intuitiv verständliche Lösungs- und Darstellungsmöglichkeiten in die Modellbasis einfließen.
- als Verfeinerungen des Prinzips der Agilität aus Abschnitt 4.2.2.4:
  - *Inkrementelle Evolution* (Abschnitt 5.4.2.9), um sicherzustellen, dass ein kontinuierlicher Anforderungserhebungsprozess durch schrittweise Spezifikation der Gesamtlösung durch das CMDSD-Werkzeug unterstützt wird und
  - *Rapid Protoyping* (Abschnitt 5.4.2.10), um zu gewährleisten, dass Prototypisierungsfunktionen inhärent mit dem CMDSD-Werkzeug umgesetzt werden und somit sichergestellt ist, dass frühes Feedback zur Vermeidung von Spätfolgekosten eingeholt werden kann, indem Domänen- und IT-Experten gemeinsam Prototypen entwickeln und evaluieren.

Abbildung 5.1 auf Seite 174 fasst die detailliertere Untergliederung der generelleren CP-Prinzipien noch einmal bildlich zusammen[6]

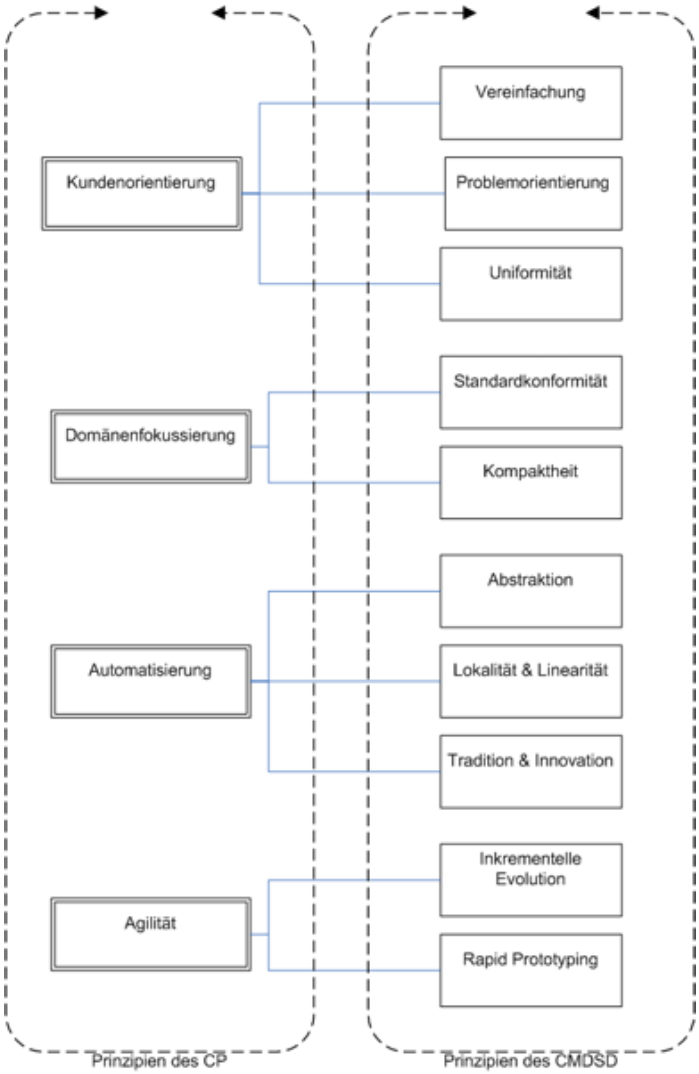


Abbildung 5.1: Prinzipien des CMDSD

5.4.2.1 Vereinfachung

Vereinfachung zielt darauf ab, eine möglichst intuitive Benutzung des Werkzeugs durch eine Redukion der Anzahl und Komplexität der abgebildeten Konzepte zu erreichen. Dazu gibt es mehrere Ansätze, die wir kurz ausführen:

K-3.44 ▷ 107

[6] In der Tat haben wir dafür die Symbolik von Organigrammen verwendet – letztlich auch, um zu zeigen, dass die Prinzipien hierarchisch organisiert sind und team-ähnlich zusammenspielen und sich ergänzen müssen.

> *„Ein Bild sagt mehr als 1000 Worte.“*

Fachmodelle sind in einem CMDSD-Werkzeug in einer möglichst übersichtlichen, einfachen und wenig formalen Notation grafisch zu modellieren. Insbesondere sollte sich diese Notation an gängigen Notationen aus der jeweiligen fachlichen Sicht der Anwender orientieren. Wie genau diese grafische Repräsentation aussehen kann, hängt von der jeweiligen Problemdomäne ab – und man muss sich auch immer vor Augen führen, dass nicht jedes Problem zwingend grafisch repräsentiert werden muss. Gerade in technischen Domänen können auch textbasierte domänenspezifische Sprachen sehr wirkungsvoll und leicht anwendbar an[7].

K-3.4 ▷ 56

> *Formalismen sind Mittel zum Zweck, nicht Selbstzweck.*

Formalismen sind dort zu verwenden, wo sie hingehören – insbesondere *nicht* in den Modellnotationen und -darstellungen, da Fachanwender in den seltensten Fällen einen ausreichenden formalen Hintergrund haben.

Die konkrete Anwendung dieses Prinzips spiegelt sich insbesondere in unseren nachfolgenden Ausführungen zu Fachmodellen in Abschnitt 5.4.3.7 wider.

5.4.2.2 Problemorientierung

> *Denke nicht lösungs- sondern problemorientiert.*

Da dieses Prinzip auf den ersten Blick gegenläufig zu sein scheint zu den häufiger formulierten Philosophie des „Wir liefern Lösungen und reden nicht über Probleme“, wollen wir das Prinzip ausführlicher erläutern: Die für ein Projekt relevante Lösung wird in der Regel ein technisches Artefakt – also ein ausführbares Softwaresystem – sein. Das Problem wird in der Regel ein fachliches Problem sein, dass durch das Domänenverständnis des Kunden bzw. Anwenders geprägt ist. Liegt der Fokus auf der Lösung (der technischen Komponente) wird häufig eine gründliche Analyse und

[7] Deswegen haben wir Vereinfachung als Prinzip definiert und geben spezifische Methoden an, die unabhängig von der Domäne dabei helfen, möglichst einfache grafische oder textuelle Modelle zu finden.

Bewertung der fachlichen Aspekte (des Problems) zugunsten technischer Details vernachlässigt. Daher sind technische Details bewusst und explizit zu vernachlässigen, um den notwendigen Raum und ausreichend Energie für die Fokussierung des Domänenproblems zu bewahren.

Eine natürliche Neigung des Menschen ist es, sich bevorzugt in ihm besonders vertrauten Bahnen zu bewegen (in denen er mglw. auch die größte Erfahrung besitzt), weil er dort die größte Sicherheit erfährt. Psychologisch gesehen ist dieses Verhalten spätestens seit [Jame50, Lebo82] Fokus intensiver Studien. Insbesondere für die IT-Branche stellt diese Neigung aber ein großes Problem dar: Nur in wenigen Branchen ist eine so enge Zusammenarbeit zwischen Experten aus unterschiedlichen Welten gefragt. Für IT-Spezialisten ist dieses Problem sogar in vielen Fällen noch prominenter: Insbesondere die technologischen Vorreiter (Berater, Evangelisten, usw.) kommen in häufiger wechselnden Kontakt mit variierenden Branchen, da sie nicht nur nach Branchen- sondern auch nach IT-Expertise eingesetzt werden. Dies führt zu einer Verstärkung des grundlegenden psychologischen Problems: Das eigene Fachwissen wird in Form eines Schutzwalls verwendet, um sich die Unsicherheit bezüglich der fachlichen Anforderungen nicht zu sehr anmerken zu lassen. Fachbegriffe aus der Informatik werden in Kombination mit komplexen Architekturen, umfangreichen Modellen und hochgradig technischen Spezifikationen genutzt, um Unsicherheiten bezüglich fachlicher Systemausprägungen zu überdecken. Umgekehrt zeigt sich ein ähnliches Verhalten. Viele Anwender sind vom Gefühl geprägt, dass IT aufgrund ihrer Omnipräsenz „ja nicht schwer sein kann“ und „man mit Excel ja auch selbst programmieren kann“ – immer wieder gehörte Zitate aus unserem Berufsalltag. Aufgrund dieser grundlegenden Vertrautheit stellt sich das Gefühl ein, bei IT-spezifischen Diskussionen mitreden zu können und vor allem auch zu müssen. Das sich dann häufig schnell zeigt, das moderne IT-Architekturen und Technologien eben nicht für jeden verständlich sind (insbesondere nicht bei ad hoc Diskussionen) führt wiederum dazu, Vorschlägen ohne tieferes Verständnis zuzustimmen oder sie abzulehnen. Dies wiederum führt zu einer häufig inkorrekten Einschätzung von Zusammenhängen, die – wenn sie auf budget-verantwortlicher Seite geschehen – dramatische Auswirkungen auf den Projekterfolg haben können.

Victoria Livschitz analysiert diesen Sachverhalt sehr treffend in [Livs05]:

> *„The problem is that modern software development is no longer mostly about programming. My wild guess is that something like 90 percent of software development happens outside of programming languages. Dozens of nonprogramming disciplines, from information security to release*

> *engineering to program management, define modern software development. Each discipline employs different principles, methods, and tools. Each requires specialized skills and educational backgrounds.“*

Im Grunde kann man dieser Problematik nur auf einer Weise begegnen – die IT muss von den technischen Zusammenhängen abstrahieren und Standardarchitekturen verstärkt als Basiswerkzeug einsetzen, um sich mehr Freiräume für den notwendigen Erwerb der jeweiligen Fachkenntnisse zu verschaffen. Der Fokus des Handelns muss also von der Lösungsorientierung (dem Aufbau einer komplexen IT-Infrastruktur) wechseln zur Problemorientierung (dem tiefergehenden Verständnis des Anwendungsproblems). Dieser Wandel erfordert Mut, Aufgewecktheit und Lernbereitschaft. Erfahrungen zeigen aber, dass sich bei erfolgreicher Umsetzung dieses Prinzips eine größere Arbeitszufriedenheit einstellt. Plötzlich findet man im Fachanwender statt eines „skeptischen Technikfeindes“ einen freundlichen Lehrer, der gerne sein Fachwissen kommuniziert und eine Erfahrungen in den Dienst des Projektes stellt. Und die Risiken des Projektes werden besser kalkulierbar, da der IT-Experte ein wachsendes und tiefergreifendes Verständnis der Fachdomäne erlangt.

K-3.7 ▷ 61
K-3.9 ▷ 62
K-3.2 ▷ 50
K-3.38 ▷ 91

Somit zielt dieses Prinzip auch unmittelbar darauf ab, das in Studien der Standish Group als signifikant erkannte Problem der mangelnden Nutzerbeteiligung besser in den Griff zu bekommen (siehe Punkt 2 aus Tabelle 2.4 auf Seite 26 sowie [Stan03b]). Zudem hilft es dabei, die Kommunikationsanlässe zwischen Domänen- und IT-Experten in eindeutige Bahnen zu lenken. Essentiell dafür ist eine entsprechende Vorbereitung der Werkzeuge.

K-3.22 ▷ 72
K-3.38 ▷ 91

5.4.2.3 Uniformität

> *Minimiere die Zahl der Ausnahmen im Werkzeug. Behandle gleiche Dinge immer gleich. Prüfe, ob scheinbar ähnliche Dinge nicht auch gleich sind.*

Weinberg hat in [Wein71, Wein98] (auf S. 218ff. der Ausgabe von 1998) das Prinzip der Uniformität für Programmiersprachen definiert. Da auch unsere Form der grafischen Modellierung eine Form der Programmierung ist, lässt sich dieses Prinzip nahtlos übernehmen. Es besagt, dass die Zahl der Ausnahmen darüber entscheidet, wie leicht sich ein Mensch etwas merken kann und ist durch experimentelle Ergebnisse aus der Psychologie belegt. In Programmiersprachen sollten ähnliche Konzepte also auch gleichförmig behandelt werden.

K-3.21 ▷ 70

K-3.11 ▷ 63
K-3.40 ▷ 99

Wir drehen diese Formulierung um und fordern für CMDSD-Werkzeuge, dass die Werkzeugentwickler die Zahl der Ausnahmen von der Regel für alle modellierten Domänenaspekte so gering wie möglich zu halten haben. Gleiche Probleme sollten also immer auf die gleiche Weise gelöst werden.

Unsere prototypische Entwicklung eines CMDSD-Werkzeugs gibt in Abschnitt 8.1.8.2.3 mit Filtermodellen ein Beispiel für eine solche uniforme Vorgehensweise: Es gibt in Portalsystemen verschiedene Stellen, an denen logische Bedingungen eine Rolle spielen, beispielsweise in Entscheidungszuständen[8] aber auch bei der Filterung von Daten[9]. Hier könnte man zwei Alternativen wählen: Entweder definiert man ein Modell zur Beschreibung logischer Bedingungen und eins für die Filterung von Daten oder man definiert ein einheitliches Modell logischer Ausdrücke. Wir haben uns zwecks einer uniformen Vorgehensweise für die zweite Alternative entschieden, die auch sofort Vorteile in der Entwicklung des Werkzeugs bringt: An allen Stellen, an denen logische Filter zur Beschreibung von Bedingungen benötigt werden, kann in der grafischen Oberfläche immer der gleiche Editor für diese logischen Filterausdrücke benutzt werden. Dies erspart dem Entwickler Zeit, da er nur einen solchen Editor implementieren muss und für den jeweiligen Aufrufort nur noch den korrekten Informationskontext für die Konstruktion der logischen Ausdrücke berücksichtigen muss. Auch dem Anwender erspart dies Zeit, da er nur erlernen muss, einen Editor zu bedienen, unabhängig davon ob er eine Entscheidung oder eine Datenfilterung modelliert[10].

Es sei erwähnt, dass Weinberg mehrere Arten von Uniformität herleitet – wir werden dieses Prinzip auch bei den Prinzipien zur Umsetzung von Projekten durch CDPM in Abschnitt 6.4.2.4 wiederfinden, dort dann aber aus operativer (also modellierender oder implementierender) Sicht während eines Projekts.

[8] Entscheidungszustände modellieren Sachverhalte wie „... wenn das Dokument akzeptiert ist, dann...“.

[9] Eine beispielhafte Form der Datenfilterung könnte „... zeige alle User, die den Status *aktiv* haben“ oder anders formuliert „... wenn der User aktiv ist, zeige ihn in der Tabelle...“ sein.

[10] Nebenbei bemerkt hat sich im Verlauf der weiteren Entwicklung von PIKAS gezeigt, dass es auch noch viele weitere Stellen gibt, an denen Filtermodelle relevant sind und nun alle gleichförmig über einen Mechanismus behandelt werden können, beispielsweise die folgenden:

- die bedingte Formatierung von Tabellenzellen (hier wird ein Filtermodell mit einer Formatierungsregel verbunden)
- die konditionale Gestaltung von Ansichten (hier werden sogenannte *Parts*, also logische Bildschirmbereiche, mit Filtermodellen verbunden, die angeben, wann der entsprechende Bereich abgezeigt werden soll.

5.4.2.4 Standardkonformität

> *Entwickle ein CMDSD-System erst dann, wenn die konzeptionelle Standardisierung abgeschlossen ist und die Standards verwendbar sind.*

Der Softwaremarkt ist in den letzten Jahren von einem zunehmenden Hype-Cycle (siehe [Wiki08m, Hons06, Lübe06]) geprägt, bei dem ein technologischer Aufhänger genutzt wird, um den Umsatz mit Software-Systemen anzukurbeln. War dies schon immer ein Kennzeichen des Software-Markts im Speziellen, so hat sich diese Tendenz in den letzten Jahren – nicht zuletzt durch den Hype der New Economy – massiv verstärkt. Insbesondere werden die Zyklen zunehmend kürzer, in denen eine neue glückselig machende Technologie zum Zentrum des Handelns erhoben wird. CMDSD fokussiert fachliche und nicht technische Anforderungen und Rahmenbedingungen und sollte daher nur dann als Möglichkeit angedacht werden, wenn zumindest auf konzeptioneller Ebene klar ist, welche Standards in einem vorherrschenden Themenfeld von Bedeutung sein werden. Dann ist immer noch zu prüfen, ob parallel existierende Alternativen inhaltlich hinreichend kompatibel sind, so dass ein Austausch die Gültigkeit der Modelle nicht in Frage stellt. Ist dies der Fall, kann erwogen werden, eine CMDSD-Lösung aufzubauen.

K-3.10 ▷ 62

Im Umkehrschluss ist CMDSD nicht anwendbar, wenn eine Zieldomäne noch nicht den notwendigen Reifegrad hinsichtlich ihrer Standardisierung erreicht hat. Dies ist – gerade bezogen auf KMUs – aber kein nachhaltiges Problem, da diese in der Adaption von Technologien und Prozessen häufig einige Jahre später als Großunternehmen aktiv werden.

Auf technischer Seite wird dieses Problem zudem dadurch abgefedert, dass eine hinreichende Abstraktion (siehe auch Abschnitt 4.2.2.2) den Austausch technologischer Implementierungen zu einem späteren Zeitpunkt einfach ermöglicht. Im günstigsten Fall wird ein erneuter Generierungslauf für das modellierte System reichen, um technische Funktionen nachzurüsten. Zudem ergibt sich hier das Potenzial, Wartungskosten für geschaffene Lösungen drastisch zu senken. Wartung hat viel damit zu tun, neben neuen Anforderungen insbesondere auch neuen technologischen Rahmenbedingungen gerecht zu werden – Martin geht in [Mart81] auf S. 83 davon aus, dass bis zu 80% existierender IT-Budgets für Wartung verbraucht werden. Gelingt es nun, die Kosten für technologische Upgrades drastisch dadurch zu reduzieren, dass im Generierungsschritt neue Technologien in das resultierende System für den

K-3.16 ▷ 65

K-3.29 ▷ 80

Nutzer transparent eingewoben werden, sinken auch die Wartungskosten, da der Aufwand für manuelle Migrationen entfällt[11].

Standardkonformität ist auch besonders wichtig für Dienstleister im KMU-Umfeld. Die Entwicklung einer CMDSD-basierten Lösung wird normalerweise erst dann Sinn machen, wenn die zugrunde liegenden Technologien und Werkzeuge einen sinnvollen Reifegrad haben. Andernfalls werden die Investitionen in den meisten Fällen für einen KMU-Dienstleister nicht tragbar sein – hier zeigt sich also, dass ein Dienstleister auch fundierte Investitionsabschätzungen und Chancen-/Risikenanalysen durchführen muss, bevor er sich für oder gegen die Entwicklung eines eigenen CMDSD-Werkzeugs für einen CP-Ansatz entscheidet.

5.4.2.5 Kompaktheit

> *Minimiere die Zahl der Informationseinheiten, die benötigt wird, um ein Konzept, eine Idee oder einen Algorithmus darzustellen.*

Dieses Prinzip stammt analog zur Uniformität (siehe Abschnitt 5.4.2.3) aus [Wein71, Wein98]. Weinberg geht in seinem Werk mehr auf die kompakte Schreibweise von Anweisungen und die möglichst kompakte Definition komplexer Operationen auf Datenstrukturen ein, beschreibt aber auch das dahinter liegende Prinzip des *(Information) Chunkings* (aus [Mill56]): Es gibt nur eine begrenzte Zahl von Informationseinheiten (üblicherweise zwischen fünf und neun), sie sich Menschen leicht merken können. Um komplexere Informationen zu memorieren, zerteilt man diese daher in kleinere Einheiten.

K-3.17 ▷ 65
K-3.21 ▷ 70

Entsprechend kann das Prinzip der Kundenorientierung auf Werkzeugseite dadurch gezielter umgesetzt werden, dass das Werkzeug versucht, möglichst wenige Informationseinheiten zur Beschreibung eines Sachverhalts zu verwenden – denn dann werden die Sachverhalte bei sinnvoller Wahl der Informationseinheiten leichter verständlich und zugänglich.

[11] Wir haben selbst Erfahrungen mit der Validität dieser Annahme gesammelt: Das System *Intrexx* des Herstellers United Planet (siehe [Intr09]) ist in mancherlei Hinsicht ein Vorbild für unseren CMDSD-Prototypen. Dort werden Portale bis zu einem gewissen Grad auf Grundlage der von uns formulierten Paradigmen spezifiziert. Wir haben bei QuinScape seit 2001 mit einer Reihe produktiver Lösungen diverse technische Migrationen mitgemacht – von Frame-basierten HTML-Ansätzen bis hin zur kompletten „Ajaxifizierung“ der generierten Lösungen – und es gab nur geringe manuelle Nachbearbeitungsaufwände.

Entsprechend muss das Prinzip der Kompaktheit Hand in Hand gehen mit dem Prinzip der Abstraktion (siehe Abschnitt 5.4.2.6), da nur so die Menge an Informationseinheiten durch Chunking über abstrakte Konzepte auf eine handhabbare Zahl reduziert werden kann.

Die Einhaltung dieses Prinzips ermöglicht es den Domänenexperten, leichter die Modellierung ihrer Domäne zu verstehen, was wiederum wichtig ist für die von uns definierte Vorgehensweise in konkreten Projekten (siehe Kapitel 6, hier insbesondere die Ausführungen zu Conceptual Pair Programming in Abschnitt 6.4.3.5).

Desweiteren fordern wir:

> *Entwirf die Zielobjekte einer Modelltransformation so, dass sie die Modellelemente möglichst direkt repräsentieren.*

Damit erfüllen wir eine der zentralen Anforderungen aus dem Domain Driven Design, die gleichzeitig wichtige Optimierungspotenziale für Dienstleister bringt, die ein solches Werkzeug implementieren (siehe [Evan04a], S. 49): *„Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software, even as you seek to make it reflect deeper insight into the domain. Demand a single model that serves both purposes well, in addition to supporting a robust* UBIQUITOUS LANGUAGE.*“* Dienstleister profitieren von der Suche nach einer möglichst direkten Abbildung zwischen Modellen und Implementierungen – möglich wird dies wiederum aber erst dadurch, dass wir uns im Conceptual Programming und damit CMDSD ausschließlich mit hinreichend standardisierten Domänen beschäftigen, die bereits über ausgereifte Architekturmuster und Implementierungen verfügen[12] (siehe dazu Abschnitt 5.4.2.4).

5.4.2.6 Abstraktion

> *Abstrahiere so weit wie möglich von allen technologischen und fachlichen Details.*

[12] In einem solchen standardisierten Umfeld wird sich also eine möglichst direkte Abbildung in vielen Fällen bereits anbieten wenn nicht gar aufdrängen, da entsprechend viele praktische Erfahrungen bereits in die vorhandenen Architekturen und Entwurfsmuster geflossen sind.

Abstraktion unterstützt als Prinzip das zuvor erläuterte Prinzip der Kompaktheit
K-3.23 ▷ 73
(siehe Abschnitt 5.4.2.5 und wieder [Wein98], jetzt auf den Seiten 224–229) und erlaubt überhaupt erst die Definition aussagekräftiger und domänenorientierter Modelle. Abstraktion hat viel damit zu tun, Domänenwissen möglichst nicht-explizit bereitzustellen. Dies gilt sowohl für technische wie fachliche Domänen, beispielsweise ist es einfacher, aus Anwendersicht

- eine Gleichheitsoperation auf komplexen Objekten zu verwenden als die einzelnen Eigenschaften eines Objekts gleichsetzen zu müssen,
- eine Liste von Aktionen in einem Modell automatisch immer als Transaktion anzusehen (und gegebenenfalls über ein geeignetes Statusfeld eine Ausnahme definieren zu können) als für jede Aktionsliste eine (potenziell komplexe) Transaktionsbehandlung angeben zu müssen,
- eine abstrahierte Funktion zum Schließen von Kostenstellen zu haben als das Schließen explizit mit all seinen technischen Details definieren zu müssen[13].

Abstraktion bietet als weiteren Vorteil die Möglichkeit, technische Implementierungen unter der Haube auszutauschen oder zu variieren: So kann ein Werkzeug leichter SOA-fähig gemacht werden, wenn die fachlichen Modelle keine Aussage darüber machen, wie externe Dienste angesprochen werden sondern nur die abstrakte Schnittstelle beschreiben. Ebenso ist es leichter, mit verschiedenen Datenbanksystemen zu arbeiten, wenn z. B. die fachlichen Domänenobjekte nur über abstrakte Datentypen (wie „Währung“, „Text“ usw.) beschrieben werden und nicht als „VARCHAR(45000)“ der ähnliches.

Martin gibt in [Mart81] auf den Seiten 113–114 ein aus heutiger Sicht sehr schönes Beispiel für die Möglichkeiten zur technischen Abstraktion (und zeigt auch implizit, wie weit diese bereits heute gediehen ist): Er beschreibt dort zunächst, dass viele Anwender 1981 bereits damit überfordert sind, sich an einem System anzumelden (was zunächst verwundert) und illustriert dies dann an einem praktischen Beispiel, indem er die neun (!) Schritte zusammen mit den Rückmeldungen des Systems angibt, die zur Anmeldung an einem typischen System des Jahres 1981 erforderlich waren (wodurch die Komplexität offensichtlich wird). Heute sind Login-Vorgänge

[13] Hier zeigt sich wieder, wie wichtig das eingebrachte Domänenwissen ist: Ist das Schließen von Kostenstellen in einer gegebenen Domäne kein wiederkehrendes Konzept, macht es auch keinen Sinn, dafür abstrahierende Operationen anzugeben. Modelliert man aber z. B. ein CMDSD-Werkzeuge für Controlling-Anwendungen, kann eine solche Operation durchaus sehr viel Sinn machen, da Kostenstellen aus den unterschiedlichsten Gründen und damit aus den unterschiedlichsten Prozessen heraus geschlossen werden können, was ggf. mit viel technischer Komplexität verbunden ist (z. B. der Benachrichtung von Drittsystemen, verteilte Transaktionsbehandlung, etc.).

sehr viel einfacher, aber eine Authentifizierung gegen ein LDAP unter Berücksichtigung von Single-Sign-On-Mechanismen zu programmieren ist auch heute noch eine komplexe Aufgabe. Entsprechend sind wir für dieses Beispiel der Auffassung, dass die sinnvolle Abstraktion für Logins in einem modellbasierten System darin bestehen sollte, einmal in einem Expertenmodus die Verbindungsparameter zu konfigurieren und dann nur noch festzulegen, welche Seiten oder Prozesse mit welchen Rechten zugänglich sind. Stößt man dann auf eine Komponente, für die man nicht ausreichend Rechte besitzt, sollte das System selbständig auf einen Login-Prozess mit allen notwendigen technischen und prozessbedingten Schritten verweisen, ohne das hier eine weitere explizite Modellierung oder Programmierung erforderlich ist[14].

Zudem macht die folgende Leitlinie den informationellen Wert von Abstraktion auf der richtigen Detaillierungsebene besonders bewusst:

> *Der eigentliche Wert eines Systems besteht in dem in ihm enthaltenen Wissen um die Anwendungsdomäne.*

Der wertvollste Teil eines Softwaresystems ist das in ihm enthaltene Wissen um fachliche Abläufe, Zusammenhänge und Logik. Entsprechend soll eine klare Darstellung dieser Information bei der Modellierung eines Systems im Vordergrund stehen. Insbesondere ist die Modellierung der fachlichen Tatsachen ungleich wichtiger als die Modellierung der technischen Gegebenheiten. Verbergen sich die fachlichen Informationen aber nun in einer sehr technischen oder zu detaillierten Darstellungsform (weil die falsche Abstraktionsebene gewählt wurde), sinkt der Wert der Fachinformationen, da sie schwerer aus dem System zu extrahieren sind und insgesamt aufgrund der komplexeren Darstellung eine höhere Fehleranfälligkeit gegeben ist. Auch hier ist es aber wieder wichtig, auf die sinnhafte Modellierung der Domäne hinzuweisen: Enthält ein abstraktes Modell in einem CMDSD-Werkzeug sehr spezielle Annahmen, die einem Domänenexperten intuitiv nicht klar sein können, wird der Vorteil der Abstraktion zum Nachteil der Informationsverschleierung. Implizite Annahmen erzeugen noch mehr Fehler als unzureichend abstrakte Darstellungen, da sie eine noch höhere Gefahr von Fehlschlüssen beinhalten[15].

[14] Viele Systeme unterstützen dies mittlerweile auch, aber uns sind z. B. im Java-Portal-Umfeld viele Beispiele bekannt, in denen all dies weiterhin leidlich komplexe manuelle Programmierungen des Systems voraussetzt.

[15] Bei einer komplexen Darstellung gibt es zumindest einen sichtbaren Hinweis darauf, dass man sich ggf. detaillierter mit dem dargestellten Konzept auseinandersetzen und die Darstellung durchdenken muss, um sie zu verstehen. Verbergen falsche Abstraktionen komplexe Annahmen, ist die Chance für Fehlinterpretationen wesentlich höher, da es keinen Indikator dafür gibt, dass hier komplexe Annahmen zugrunde liegen könnten.

Victoria Livschitz beschreibt dies sehr treffend in ihrer Unterscheidung zwischen *contextual programming* und *computational programming* in [Livs05]:

> *„Let's talk about contextual and computational programming – and why it's important to create a formal framework to distinguish them.*
>
> *A functional behavior of the application cannot be expressed without a contextual model for the application domain. For example, in order to codify a simple business rule, say, "Transfer X amount of dollars from Y account to Z account,"there has to be a formal definition of the concepts of account, transfer, dollars, and amount. Once the context is set, one can write down the proper algorithm for fund-transfer function. The context, on the other hand, cannot be defined in isolation from the intended functionality of the application.*
>
> *As you can see, two sharply distinct intellectual activities are competing for the developer's attention. One aims to define a context for the application's functionality, and the other codifies the functional rules of the application's behavior within a given context. I refer to these as contextual programming and computational programming.*
>
> *Traditional computer science has always been consumed with computational programming, while contextual programming has been largely ignored until the advent of object-oriented programming (OOP). OOP established the separation of the analysis and design stages of the project (mostly concerned with contextual programming) from the implementation stage (mostly concerned with computational programming). OOP also introduced several convenient built-in context-building facilities such as classes, interfaces, and abstract data types.*
>
> *However, the explicit difference between contextual and computational programming is not supported in object-oriented languages. Instead, OOP relies on methodologies, external tools, and special-purpose languages – most notably UML – to address contextual programming outside of the programming language. Subsequently, the separation of context from computation in modern software development is based on the timing of the project's life cycle. This is a good idea wrongly framed. Software development is a continuous, iterative, and evolutionary activity. Neither context nor computation comes before the other. Both are organically related and belong to the same code base.“*

Entsprechend fordern wir eine Ausprägung von Modellen, die in der Sprache der Domänenexperten Sachverhalte beschreiben, aber hinreichend detailliert und komplex sind, um ausführbare Lösungen automatisiert zu generieren.

Aber auch während der Implementierung eines CMDSD-Werkzeugs ist Abstraktion für die Werkzeugentwickler ein wichtiges Thema, was die folgende Leitlinie und die anschließenden Ausführungen verdeutlichen, bei denen es darum geht, dass die Werkzeugentwickler sich auf die Lösung höherwertige Probleme konzentrieren sollen anstatt zu sehr in Detailfragen einzutauchen, die genauso gut durch die Verwendung existierender Komponenten (mindestens) ordentlich gelöst werden können:

> *Konzentriere dich bei der Entwicklung eines CMDSD-Systems auf höherwertige technische Fragen und nicht auf Basisdienstleistungen.*

Insbesondere durch eine sehr lebendige Open-Source-Kultur gibt es mittlerweile eine große Zahl grundlegender Technologien in verschiedensten Implementierungen, die eine Wiederverwendung selbst im Rahmen kommerzieller Systeme sinnvoll erlauben (siehe z. B. [Info09])[16]. Daher gebietet es sich als Produkthersteller geradezu, keine weiteren Energien mehr in die Implementierung von Basistechnologien zu stecken, für die es bereits gleichwertige einsetzbare Implementierungen aus dem Open-Source-Bereich gibt. Hier ist speziell der Aspekt der Gleichwertigkeit von Lösungen aus einer gewissen Distanz zu bewerten, da Entwickler dazu tendieren, eine Eigenimplementierung immer gegenüber einem Drittsystem zu präferieren – insbesondere, je mehr sie selbst an dem jeweils technischen Thema interessiert sind. Dies ist jedoch unproduktiv und daher zu vermeiden. Leider sind uns aus der Praxis viele Fälle bekannt, in denen Entwickler eines Dienstleisters sich lieber damit beschäftigen, eine technologische Frage erneut durch ein eigenes Framework (das aus Sicht der betroffenen Entwickler natürlich besser ist als alles andere am Markt) zu lösen, als einfach etwas Existierendes wiederzuverwenden und sich auf die eigentlichen fachlichen Fragestellungen zu konzentrieren. Dies deckt sich mit dem zweiten in [Lind05] formulierten Kritikpunkt an Intentional Programming (siehe dazu auch unsere Ausführungen in Abschnitt 3.1.2), denen wir nur mit dem grundlegenden Wert der Wirtschaftlichkeit begegnen können.

Informatiker im Speziellen sind allerdings von einem erstaunlichen Kreativitätsdrang motiviert, der immer wieder dazu führt, dass technische Implementierungen mit sehr

[16] Hier seien Projekte wie Subtext (siehe [Edwa09a]) aufgegriffen, die mächtige Funktionen als Open Source bereitstellen (werden). Subtext haben wir bereits in Kapitel 3 auf Seite 72 kurz vorgestellt.

hohem Engagement unter Berücksichtigung komplexer Szenarien und einer großen Hingabe zur Abbildung von flexiblen Einsatzmöglichkeiten durchgeführt werden. Zunächst mag dies positiv erscheinen und im Grunde ist es das auch – doch wenn man solche Implementierungen detailliert untersucht, fällt neben allen positiven Effekten insbesondere auf, dass projekt-, abteilungs-, teilkonzern- und konzernübergreifend immer wieder extrem viel Energie in die Realisierung von Anforderungen gesteckt wird, für die es bereits alternative Implementierungen von hoher Qualität und nachgewiesener Stabilität gibt – so zeigen empirische Ergebnisse zur Einführung von RAD-Lösungen immer wieder, dass selbst heute noch Produktivitätssteigerungen um das bis zu 15fache möglich sind (siehe Abschnitt 3.2.5 bzw. [Have05] für Details). War das diesbezügliche subjektive Gefühl schon seit langer Zeit in den Hinterköpfen von Managern und Projektleitern vorhanden, fällt doch gerade durch die zunehmende Verbreitung und Professionalisierung von Open-Source-Projekten auf, dass es sich um ein endemisches Problem handelt. Sogar (oder gerade?) in der Open-Source-Szene selbst existieren zuweilen unter einem Dach eine Vielzahl von Projekten, die ein ähnliches oder gleiches Thema fokussieren. Ein gutes Beispiel aus dem Open-Source-Bereich sind die java-zentrierten Projekte der Apache Software Foundation (ASF, siehe [ApSF06, ASFJ06]). Allein hier existieren selbst bei oberflächlicher Betrachtung mindestens sieben verschiedene Frameworks für webbasierte Anwendungen mit – teilweise subtil – unterschiedlichen Schwerpunkten (Beehive, Cocoon, Shale, Struts, Tapestry, Turbine, Wicket). Erscheint dies im ersten Moment nicht negativ – zumal durch diesen Ansatz ein kreativer Wettbewerb gefördert wird und die Apache Software Foundation im Gegensatz zu vielen Open-Source-Projekten massiv auf die Wiederverwendung von Basistechnologien setzt[17] – fällt doch auf, dass all diese Projekte trotz herausragender Ideen in bestimmten technischen Themenfeldern große Lücken in Fragestellungen von Benutzbarkeit und Dokumentation aufweisen:

- Es gibt größtenteils nur sehr unzureichende Dokumentation und erschienene Bücher spiegeln nur selten den Stand der aktuellen Entwicklung wieder. Meistens ist der aktuelle Entwicklungsstand ein Major Release oder noch weiter fortgeschritten als dies in der gedruckten Literatur nachvollziehbar wäre. Da die meisten Weiterentwicklungen im positiven Sinne häufig so gut sind, dass man sie eigentlich auch nutzen sollte, entsteht hier eine große Diskrepanz zwischen technischer Realität und effektiver Einführbarkeit dieser Technologien.
- Beispiele sind nur spärlich gesäht und schwanken sehr stark in der Qualität und Korrektheit, zumeist wiederum dadurch begründet, dass sie nicht mehr

[17] Siehe dazu zum Beispiel das Apache Commons Project in [ASFC06, OBri05].

den aktuellen Stand der Technik wiederspiegeln.

- Um fortgeschrittene Fähigkeiten von Frameworks erlernen zu können, werden Entwickler an den Quelltext der Projekte verwiesen. Dieser ist aber häufig eher unvollständig dokumentiert, so dass auch hier große Hürden aufgebaut werden.

Aber wo liegen die Ursachen für diese klaffende Schere zwischen einerseits technisch hochgradig innovativen und andererseits wenig produktreifen Frameworks und Komponenten? Nach unserer Ansicht sind die Gründe sehr naheliegend und nachvollziehbar motiviert:

- Open-Source-Projekte entstehen (ebenso übrigens wie häufig die Entwicklung von Basistechnologien im rein kommerziellen Projekt- oder Produktgeschäft) quasi nebenbei – sei es in der Freizeit oder als Untertätigkeit im Rahmen großer Entwicklungsprojekte. Dies bedeutet aber, dass die für diese Projekte zur Verfügung stehende Zeit eher knapp bemessen ist. Somit fokussieren Entwickler ganz natürlich die Themen, die für sie höchste Priorität haben und die sie auch am meisten interessieren – und Dokumentation und das Schreiben von Einführungsmaterialien gehört seit jeher eher nicht zu den Themen, die dann weit oben auf der Prioritätenliste stehen.
- Sobald ein Framework oder eine Technologie von mehr als einer Person eingesetzt wird, steigt der Test- und Fehlerbehebungsaufwand signifikant – neue oder falsche Konfigurationen verlangen dem ursprünglichen Entwickler analytisches Geschick und Talent bei der Fehlerbehebung ab, wodurch seine ohnehin beschränkten Ressourcen auf noch mehr Themen verteilt werden müssen.
- Menschen nutzen ihre Zeit bevorzugt für Tätigkeiten, die auch Spaß machen. Die Praxis zeigt, dass die Lösung komplexer technischer Probleme für Informatiker wesentlich befriedigender und interessanter ist als die Dokumentation „ja schon lange gelöster und bereits bekannter“ Ansätze. Dementsprechend werden Dokumentation und Test eher vernachlässigt während das Hinzufügen neuer technischer Eigenschaften zu einem System einen weitaus höheren Stellenwert genießt. Wir bezeichnen diese Vorgehensweise gerne als *“Happy Engineering“*, weil sich das Glücksgefühl der entwickelnden Ingenieure zwar stetig im grünen Bereich bewegt, die Anwender der Systeme (selbst wenn dies andere Entwickler sind) aber häufig etwas kurz kommen und ihre Bedürfnisse und Wünsche nur oberflächlich zur Kenntnis genommen werden[18].

[18]Vielen Dank an Matthias Graus, der diesen Begriff geprägt hat.

Diese Probleme zeigen sich in allen Bereichen der Software-Entwicklung. Selbst von der Fachwelt anerkannte Experten aus dem MSDS-Umfeld machen diesen Fehler immer wieder und nutzen so Ressourcen für die Implementierung im Prinzip bereits existierender Lösungen, anstatt sich noch stärker auf die Weiterentwicklung ihrer eigenen herausragenden Konzepte zu konzentrieren. So wird zum Beispiel bei der Vorstellung des *openArchitectureWare*-MSDS-Ansatzes die „hauseigene“ Template-Sprache *Xpand* vorgestellt (siehe [OAW, StVö05]), die – zumindest soweit aus den Erläuterungen zu entnehmen ist – keine nennenswerten Vorteile zu zum Beispiel *Velocity* oder *FreeMarker* bietet (siehe z. B. [OBri05]), dafür aber Zusatzkomplexität durch Einführung einer weiteren Templating-Syntax erzeugt, ganz zu Schweigen von dem erneuten Implementierungs-, Test- und Dokumentationsaufwand für die Entwickler des openArchitectureWare-Frameworks.

Genau hier lässt sich durch konsequente Nutzung bereits existierender Werkzeuge die Entwicklung höherwertiger Funktionen beschleunigen. Dazu ist es erforderlich, dass sich die beteiligten Entwickler von der Vorstellung lösen, dass auch die niederwertigen Dienste selbst entwickelt sein müssen, weil (und hier finden sich viele immer wiederkehrende Argumente) “man die Umgebung dann komplett unter Kontrolle hat“, “der eigene Ansatz flexibler ist“ oder “man sich nicht von anderen Technologien abhängig macht“. Diese Argumente sind genau zu prüfen und entsprechen nach unserer Erfahrung nur in den seltensten Fällen stimmen:

- Gerade bei der Neuentwicklung hat man die eigene Umgebung eben nicht zwingend im Griff, da man Ressourcen darauf verwenden muss, niederwertige Basisdienste (erneut) zu schreiben. Da die Erfahrungen der „Vorgänger“ aus den existierenden Projekten aber in den seltensten Fällen dokumentiert sind, wird man häufig die gleichen Fehler wiederholen und mit den gleichen Designfragen viel Zeit verlieren, was kaum dazu führen wird, dass das eigentliche Projekt besser kontrolliert und in den gewünschten ruhigen Bahnen abläuft. Stattdessen werden Zeit und Energien darauf ver(sch)wendet, bereits existierende Dinge mit einem vermeintlich besseren Ansatz noch mal zu bauen.
- Der eigene Ansatz wird häufig nicht flexibler sondern anders sein. Dies ist oft bedingt dadurch, dass (wie oben bereits erwähnt) existierende Frameworks nur unzureichend oder schwer verständlich dokumentiert sind und somit ein gewisser Einarbeitungsaufwand erforderlich ist, um die Fähigkeiten dieser Ansätze zu verstehen. Erstaunlicherweise neigen Informatiker häufig eher dazu, ein Thema durch eine Neuimplementierung abzudecken als sich in ein anderes existierendes Framework einzuarbeiten. Gerade in der heutigen Zeit ist dieser

Ansatz aber sträflich, da die existierenden Open-Source-Projekte von den Tausenden von anderen Entwicklern täglich genutzt und auf ihre Brauchbarkeit geprüft werden. Nur in sehr speziellen Fällen wird es daher tatsächlich so sein, dass ein weiterer neuer Ansatz benötigt wird, um einen bestimmten Aspekt besser abzudecken.

- Letztlich ist auch das Argument der Abhängigkeit von anderen Technologien eher fadenscheinig. Natürlich macht man sich bis zu einem gewissen Grad von anderen Projekten und Bibliotheken abhängig, wenn man diese benutzt und natürlich erfordert dies eine konsequente strategische Planung. Realistisch betrachtet wird die Abhängigkeit bei einer Neuimplementierung aber noch viel größer: Nach unserer Erfahrung zeigt sich in vielen Fällen, dass interne Implementierungen oft nur von kleinen (1–3 Personen starken) Gruppen getragen werden. Es stellt sich die berechtigte Frage, was mit einer internen Entwicklung passiert, wenn die internen Betreuer das Projekt oder gar den Arbeitgeber wechseln. Kommen neue Mitarbeiter in interne Projekte, müssen sie sich in jedem Fall in die Details der internen Bibliothek einarbeiten. Bei konsequenter Verwendung von entsprechenden Bibliotheken aus dem Open-Source-Umfeld oder von anderen Anbietern, besteht zumindest eine zunehmend höhere Wahrscheinlichkeit, dass die Neuzugänge bereits aus Ausbildung, Studium oder anderen Projekten mit der entsprechenden Technologie vertraut sind und sich somit sowohl Einarbeitungs- als auch Weiterentwicklungsaufwände reduzieren.

Zusammenfassend halten wir es daher für zwingend erforderlich, sich bei der Entwicklung (C)MDSD-getriebener Systeme auf das "eigene Kerngeschäft", nämlich das CMDSD-System zu konzentrieren und hier die höherwertigen Aspekte des Systems wie Modell(ab)bildung und Ergonomie weit über technische Detailimplementierungsfragen wie Templating-Sprachen, XML-Interpreter, Parser, usw. zu stellen. Auch im Rahmen der dieser Arbeit zugrunde liegenden Referenzimplementierung *PIKAS* wurde diese Vorgehensweise gewählt. Dadurch konnten die sehr begrenzten Ressourcen erfolgreich auf die Entwicklung des eigentlichen Kernsystems gelenkt werden. Nachdem anfänglich psychologische Hemmschwellen bezüglich der Flexibilität und Qualität der externen Bibliotheken überwunden waren, stellte sich sogar ein hochgradig positiver Effekt ein: Der Spaß bei der Arbeit nahm stetig zu, da sich auch sehr komplexe Module und Bausteine mit überraschend wenig Aufwand implementieren ließen und so die eigene Zufriedenheit und Erfüllung bei der Arbeit in überproportionalem Maße stieg. Durch die Konzentration auf die höherwertigen Aspekte einer Lösung und die vollständig Abstraktion von der Implementierung der niederwertigen

K-3.5 ▷ 57

K-3.24 ▷ 73

Aspekte konnte so ein System geschaffen werden, dass sprichwörtlich erheblich mehr ist als die Summe seiner Teile – siehe dazu die Evaluierung von PIKAS in Teil III.

Als praktisch erwiesen hat sich die Verwendung verschiedener Entwurfsmuster, um von der konkret verwendeten Bibliothek im benötigten Maße zu abstrahieren (siehe [GHJV96] sowie [Evan04a] zur Erläuterung der einzelnen Entwurfsmuster). Insbesondere wurden die Muster *Repository*, *Factory*, *Facade*, *Service* und *Strategy* verwendet, so dass die konkrete Implementierung mit verhältnismäßig geringem Aufwand ausgetauscht werden kann, wenn sich herausstellt, dass doch nicht alle Kernanforderungen von der Bibliothek abgedeckt werden.

Zusammenfassend adressiert dieses Prinzip also aus Sicht des Dienstleisters maßgeblich den Wert der Wirtschaftlichkeit und soll dazu motivieren, die Entwicklungsressourcen für die differenzierenden Aspekte eines Werkzeugs zu nutzen. Aus Sicht der Fachexperten wird der Wert der Zugänglichkeit adressiert, da dem Fachexperten die Handhabung das Verständnis des Werkzeugs so leicht wie möglich gemacht wird. Aus Sicht des Werkzeugentwicklers geht es vor allem darum, komplexe Sachverhalte durch eine hinreichend hohe Abstraktion möglichst einfach ausdrücken zu können, damit die Entwicklung des Werkzeugs nicht durch zu universelle Ansatzpunkte erschwert wird.

5.4.2.7 Lokalität und Linearität

> *Sammle alle für einen Sachverhalt essentiellen Informationen, so dass sie auf einen Blick verfügbar sind. Sammle dekomponierte Informationen so, dass sie gleichförmig zugreifbar sind.*

Das Prinzip der Lokalität und Linearität geht wie schon die Prinzipien der Uni-

K-3.21 ▷ 70

formität aus Abschnitt 5.4.2.3 und der Kompaktheit aus Abschnitt 5.4.2.5 auf die Arbeiten von Weinberg aus [Wein71] bzw. [Wein98], S. 229–232 zurück. Er adressiert damit die neurophysiologische Aufspaltung des Erinnerungsvermögens in ein synästhetisches und ein sequentielles Gedächtnis. Ersteres merkt sich Fakten anhand eines Gesamteindrucks, der auf einen Blick aufgenommen wird (z. B. Gesichter), letzteres speichert aufeinanderfolgende oder stark zusammenhängende Informationen (z. B. eine regelmäßige Tonfolge). Zu diesen beiden Gedächtnisformen formuliert Weinberg die Prinzipien der Lokalität und der Linearität:

- Lokalität unterstützt das synästhetische Gedächtnis, indem angestrebt wird, alle für einen Sachverhalt relevanten Informationen auf einen Blick bereitzustellen. Während es bei Weinberg noch darum geht, Programmcode möglichst kompakt auf einer Seite darzustellen, sehen wir eine zwingende Übertragbarkeit auf die Art und Weise, wie Modelle zu gestalten sind. Als Ausführung der nachfolgend vorgestellten Methoden werden wir in Abschnitt 5.4.3.8 im Rahmen unserer Systemphilosophie zur Modellbildung in CMDSD-Systemen darauf eingehen, wie durch Dekomposition themenorientierte Modelle aufgebaut werden, die einen Sachverhalt (z. B. einen Prozess oder eine Bildschirmmaske) auf einen Blick zwar vollständig darstellen, für die Erfassung der Kernaussage überflüssige Details aber in Detailmodelle verlagern (z. B. die Transitionsdetails in einem Prozess oder die Datenquellen für Maskenfelder).

  Auch hier ist wieder das fundierte Expertenwissen über die Domäne von essentieller Bedeutung, da die Entscheidung darüber, wie essentiell eine Information ist, je nach Domäne sehr unterschiedlich ausfallen kann.

- Linearität dagegen unterstützt das sequenzielle Gedächtnis und findet sich in Modellen wieder, wenn nicht-lokale Strukturen aufgrund von Dekompositionen erforderlich werden. Die Abfolge bzw. Art und Weise der Dekomposition[19] sollte möglichst linear gestaltet werden, um dem Anwender ein strukturiertes und leicht nachvollziehbares Vorgehen zu ermöglichen.

5.4.2.8 Tradition und Innovation

> *Modelliere in der natürlichsten Form.*

Das Prinzip der Tradition und Innovation geht wie schon die Prinzipien Uniformität (Abschnitt 5.4.2.3), Kompaktheit (Abschnitt 5.4.2.5) sowie Lokalität und Linearität (Abschnitt 5.4.2.7) auf die Arbeiten von Weinberg aus [Wein71] bzw. [Wein98], dieses Mal auf den Seiten 232–237 zurück. Weinberg beschäftigt sich dabei zwar vornehmlich mit der Frage, wie Programmkonstrukte (z. B. hinsichtlich ihrer Schreibweise oder hinsichtlich der Verwendung von Leerraum zwischen Anweisungen) so zu strukturieren sind, dass sie leicht verwendbar sind und keine Missverständnisse erzeugen, aber auch dies lässt sich wieder in ein Entwurfsprinzip für Modelle übertragen.

K-3.21 ▷ 70

[19] Ein Beispiel ist “immer top down“.

Bei der Modellbildung für das CMDSD-System sollten zunächst immer existierende Vorlagen aus der Domäne verwendet werden. Ein CMDSD-Werkzeug zur Generierung von Schaltungen sollte entsprechend Symboliken aus dem Schaltwerkentwurf benutzen, ein CMDSD-Werkzeug für den Entwurf virtueller Gebäude sollte sich Zeichensystematiken aus der Architektur bedienen, usw. Was die jeweils natürlichste Form für eine Domäne ist, hängt wiederum sehr stark von der spezifischen Domäne ab. Allgemein lässt sich aber experimentell nachweisen, dass „Programmiersprachen“ (bzw. Modelle) dann am besten aufgenommen und benutzt werden können, wenn sie der zugrunde liegenden natürlichsten Sprache, die als Vorbild dient, möglichst nahe kommen oder sie möglichst uniform zur Ableitung verwenden (siehe Abschnitt 5.4.2.3 sowie die Ausführungen von Weinberg in [Wein98]).

Durch dieses Prinzip wird also insbesondere der Wert der Zugänglichkeit (siehe Abschnitt 4.2.1.3) betont. Dies soll durch folgende Leitlinie noch verstärkt werden, die ergänzend der Wert der Wirtschaftlichkeit in die vorausgegangenen Betrachtungen einbezieht:

> *Begeistere und binde durch Zugänglichkeit: Es ist besser, dass Viele wenig als das Wenige viel modellieren können.*

K-3.43 ▷ 106

Ein Werkzeug setzt sich dann durch, wenn es leicht verständlich ist und Spaß bei der Arbeit bringt. Daher ist es besser, die Bedürfnisse der breiten Masse bei der Erstellung des Werkzeugs zu adressieren und es möglichst vielen Nutzern zugänglich zu machen, selbst wenn diese große Zahl von Anwendern nur einen kleinen Teil der bereitgestellten Funktionalität benutzt. So erweckt das Werkzeug ein Gefühl von Nützlichkeit und wird es erheblich leichter haben, sich zu etablieren.

Diese Maxime basiert inhärent auf den Ideen des Soziologen Mark Granovetter von der *Stärke schwacher Bindungen* (siehe dazu insbesondere [Gran95], aber auch [Gran78] und [Gran83] für die fundamentalen Grundlagen sowie [Glad01] für eine ausgezeichnete Zusammenfassung sowie eine Erweiterung auf eine Theorie epidemischer Wirkungen bei Ideen, Trends und sozialen Wirkungen). Die Grundidee ist sehr einfach zusammengefasst: Ideen oder Strömungen scheinen insbesondere dann eine nachhaltige Wirkung zu erzielen und ein breites Publikum zu erreichen, wenn Sie über sogenannte Konnektoren verbreitet werden. Diese Konnektoren zeichnen sich dadurch aus, dass sie eine Vielzahl relativ schwacher sozialer Bindungen pflegen, d. h. sie habe faktisch eine ungewöhnlich große Zahl von „entfernten Bekannten“.

Diese Konnektoren scheinen besondere Macht zu besitzen, da sie über ihr ausgeprägtes soziales Netzwerk Ideen und Trends schnell und äußerst effektiv kommunizieren können und somit eine mächtige Multiplikationsfunktion innehaben. Die bereits genannten Artikel und Studien präsentieren eine Reihe von Felduntersuchungen, die diese Theorie untermauern.

Übertragen auf die Entwicklung von Software steht die Maxime der Bindung für ein Konzept, dass eine leicht zugängliche und in ihren Grundfunktionen einfache Software dazu prädestiniert ist, eine Art "Software-Konnektor" für eine Idee zu werden. Ist eine Software leicht zugänglich und ihr Nutzen einfach zu erkennen, wird es tendenziell viele Anwender geben, die sich „mal zwischendurch" mit ihr auseinandersetzen, um sie zur Lösung ihrer lokalen Probleme zu verwenden. Sind zudem schnelle Lösungen für einfache Probleme möglich, stellt sich ein Erfolgserlebnis ein, dass auch anderen Personen kommuniziert wird, die so wiederum angeregt werden, die Software auszuprobieren. So multipliziert sich zum einen der Bekanntheitsgrad der Software und so werden zudem neue Konnektoren gewonnen, die über ihr Netzwerk die Idee und den Nutzen der Software weitertragen.

Eine ähnliche Idee liegt zum Beispiel der Programmiersprache Basic (siehe [Lien86]) zugrunde: Ursprünglich dazu konstruiert relativ leicht zugänglich zu sein, wurden Generationen von Programmierern durch Basic (in der einen oder anderen Ausprägung) an ihr späteres Berufsfeld herangeführt und noch heute sind Entwicklungswerkzeuge wie Visual Studio (siehe [Micr08]) ohne Programmiersprachen wie Visual Basic (siehe [Löff08]) kaum denkbar. Auch andere Technologien wie HTML (siehe [Wiki06n]) haben einen ähnliche Strategie verwendet, um sich nachhaltig zu verbreiten und so erscheint es insbesondere bei einem extrem kunden-, anforderungs- und nutzenorientierten Ansatz von hoher pragmatischer Ausrichtung wie dem hier vorgestellten CMDSD-Ansatz sinnvoll zu sein, Werkzeuge so zu konzipieren, dass sie als Konnektoren für die CMDSD-Idee fungieren.

Auch im Produktbereich finden sich Beispiele für solche Erfolgsgeschichten: Insbesondere Microsoft hat einige Anwendungen sehr erfolgreich am Markt positioniert, die trotz zuweilen gravierender technischer Schwächen von Anwendern begeistert aufgenommen und in vielfältigen Bereichen eingesetzt werden – insbesondere auch von Personen mit nur relativ geringer IT-Ausbildung. Beispiele sind hier Microsoft Access (siehe [Wiki09a]) für einfache Datenbankanwendungen aller Art (vom Lagermanagement bis hin zur Personal- oder Zeitverwaltung) oder Microsoft Excel (siehe [Wiki09b]) bzw. früher Lotus 1-2-3 (siehe [Wiki08f]) als Tabellenkalkulationen, die es seit ihrem Erscheinen Controlling- und Buchhaltungsunternehmen in Unter-

nehmen aller Größenordnungen ermöglichen, relativ komplexe Auswertungen ohne Hilfe von Programmierern zu erstellen. Vergleicht man dies mit dem Zustand vor 20 oder mehr Jahren, hat sich hier eine signifikante Verbesserung der Entwicklungsprozesse ergeben, da sich zu jenem Zeitpunkt umfangreiche Werke wie z. B. [Mart81] noch mit der Frage beschäftigt haben, wie Entwickler beispielsweise effizienter Auswertungen für das Management erstellen können. Heute reicht in vielen Fällen ein einfacher Datenbankexport z. B. im CSV-Format, so dass ein Controller sich die darauf basierenden Auswertungen sehr schnell selbst mit Excel erstellen kann. Uns sind zahlreiche Beispiele bekannt, in denen sogar unternehmenskritische Anwendungen auf Grundlage dieser Produkte betrieben werden (so zum Beispiel die Katalogerstellungsprozesse für einen führenden deutschenden Touristikanbieter und einen international operierenden Leuchtenhersteller[20]).

Wird die Maxime der Bindung erfolgreich für ein bestimmtes CMDSD-Werkzeug umgesetzt, so ergibt sich ein Sog, der Anwender und Entwickler weg von der klassischen techniklastigen Programmierung hin zur fachlich fokussierten konzeptionellen und anforderungsbezogenen Arbeit bewegt. Insofern ist die Maxime der Bindung als untermauerndes Argument für viele der anderen hier vorgestellten Prinzipien und als Motivation für die streng pragmatische und problemorientierte Ausrichtung von CMDSD zu sehen.

Zudem kann der Wert der Wirtschaftlichkeit (siehe Abschnitt 4.2.1.1) für Kunden wie Dienstleister adressiert werden. Bei großer Verbreitung eines Systems ergeben sich die bereits auf Seite 39 erläuterten Skaleneffekte, die Dienstleistern einen Margenvorteil und Kunden einen Qualitätsvorteil bieten.

Hier spielt auch die Maxime der Ausgewogenheit eine wichtige Rolle:

> *Achte darauf, dass alle zentralen Aspekte der betrachteten Domäne sich auch in vergleichbarer Gewichtung im CMDSD-Werkzeug wiederfinden.*

In dieser Maxime spiegelt sich eine Erfahrung aus dem Convergent Engineering wieder (siehe [Tayl95], S. 108), das genau eine solche Ausgewogenheit für die spezielle Domäne der Geschäftsprozessmodellierung fordert. Dort wird eine Ausgewogenheit zwischen Ressourcen, Organisationseinheiten und Prozessen gefordert. Eine solche Ausgewogenheit ist wichtig, um die Erwartung der Anwender zu erfüllen – eine

K-3.40 ▷ 99

[20] Um Verwechslungen zu vermeiden, sei darauf hingewiesen, dass es sich dabei *nicht* um Kunden der QuinScape GmbH handelt.

Missachtung oder Reduzierung von Domänenbausteinen, die für Domänenexperten eine zentrale und wichtige Rolle einnehmen, wird zu einem inhärenten Misstrauen und Zweifel am spezifischen CMDSD-Werkzeug führen. Das Misstrauen schwächt die Beziehung zwischen Kunden und Dienstleistern und reduziert so die Chancen für den Projekterfolg.

5.4.2.9 Inkrementelle Evolution

> *Starte klein und wachse dann („Start small then grow“).*

Dieses Prinzip fordert, dass das CMDSD-Werkzeug generell ein inkrementelles und evolutionäres Vorgehen unterstützen muss. Die initiale Version eines Systems oder Teilprozesses sollte so einfach wie möglich spezifiziert sein, d. h. dass alle unnötigen oder nicht kritischen Funktionen zunächst außen vor gelassen werden sollten. Diese zusätzlichen Anforderungen sind für eine spätere Version gründlich zu dokumentieren und zugreifbar zu halten – um den initialen Erfolg zu sichern und eine schnelle Realisierung zu ermöglichen, sollten in einer ersten Version eines Systems aber nur die absolut unabdingbaren Anforderungen umgesetzt werden, ohne die ein sinnvolles Arbeiten nicht möglich ist.

K-3.19 ▷ 66
K-3.22 ▷ 72
K-3.54 ▷ 127

Zudem sind Anforderungen analog zur Vorgehensweise beim Extreme Programming zunächst auf die einfachst mögliche Weise umzusetzen. Basierend auf der initialen Umsetzung erhält man schnell ein direktes und authentisches Feedback der Anwender und kann so sehr viel besser planen, wie Funktionen weiterentwickelt und ausgebaut werden müssen. Dies schützt insbesondere auch davor, unnötig komplexe (und damit aufwände) Umsetzungen zu realisieren, ohne dass Kunden nachhaltig das entsprechende Bedürfnis nach diesen Funktionen kommuniziert haben[21].

5.4.2.10 Rapid Prototyping

Rapid Prototyping hat geradezu spektakuläre Erfolge gerade in Projekten gefeiert, die von kleinen Teams oder Einzelpersonen abgewickelt werden (siehe [Tayl95], S. 146–147). Das ist aber genau das Umfeld, in dem sich viele mittelständische IT-Projekte positionieren, wie wir bereits in früheren Abschnitten belegt haben. Daher

[21] Besonderer Dank gebührt Manfred Stetz, den Entwicklungsleiter der Firma United Planet (siehe [Intr09]), der den einprägsamen Satz „Start small then grow“ seit mindestens 1998 wie kaum ein anderer erfolgreich lebt und ihn unseres Wissens auch geprägt hat.

ist es zwingend erforderlich, dass jedes CMDSD-System Möglichkeiten für domänenspezifisches Rapid Prototyping vorsieht. Üblicherweise wird dies durch die direkte Ausführung von Modellen oder deren Übersetzung in ausführbare Modelle erreicht. Es gilt die Regel:

> *Ein Prototyp sagt mehr als 1000 Bilder.*

Wie ein Bild mehr sagt als 1000 Worte, sagt ein Prototyp mehr als 1000 Seiten Spezifikation und 1000 Bilder. Alle Fachmodelle sind daher für den Lösungsraum so zu entwerfen, dass aus ihnen automatisiert, effizient und unmittelbar Prototypen generiert werden können. Modellierungsfehler können auf diese Weise früh erkannt und Alternativen gemeinsam mit den Architekten und technischen Entwicklern diskutiert werden. Visuell unterstütztes Denken ersetzt geschriebene und abstrakte Spezifikationen und erlaubt vor allem ein frühzeitiges Feedback, da der Spezifizierende (zumeist der Fachexperte) einen relativ direkten Eindruck von der Wirkung seiner Ideen bekommt. Erkenntnisse von James Martin (z. B. [Mart81], S. 60) unterstützen dies. Er berichtet von vielen Erfahrungen aus der Praxis, in denen eine textuelle Spezifikation – so wichtig sie auch in vielen Fällen ist – einfach nicht ausreicht, um tatsächliches Verständnis für einen komplexen Sachverhalt zu erzeugen und erst der praktische Umgang mit einem System diesen Erkenntnisprozess effizient einleitet[22]. Entsprechend fordert er als Konsequenz aus seinen Beobachtungen auch in [Mart81], S. 65, dass Prototypisierung eine der essentiellen Grundlagen für zukünftige leistungsfähigere Entwicklungsprozesse sein muss (Typ 3 der Anwendungsentwicklung in Martins Nomenklatur). Auf den Seiten 66–67 findet sich dann auch eine Zusammenfassung der zu erwartenden Leistungssteigerung durch Prototypisierung (Spalte 3), die sich mit unseren in Abschnitt 4.2.1 formulierten Erwartungen an die Wirtschaftlichkeit, Zugänglichkeit und Flexibilität eines durch Prototypisierung gestützten Ansatzes decken. Insbesondere weist Martin dort auch auf eine weitere sehr wichtige Eigenschaft von prototypisierenden Ansätzen hin: Sie stimulieren das Vorstellungsvermögen und die Kreativität der Anwender, so dass neue Anforderungen und Prozessverbesserungen leichter zu finden werden.

K-3.31 ▷ 81
K-3.44 ▷ 107
K-3.38 ▷ 91
K-3.45 ▷ 109
K-3.25 ▷ 73
K-3.53 ▷ 125
K-3.20 ▷ 67

[22] Besonders prägnant erscheint hier auch das Zitat von S. 81 von [Mart81]: *„I don't know what I want until I see it."*, das aus Anwendersicht widerspiegelt ist, wie schwierig die abstrakte Formulierung von Anforderungen an IT-Systeme tatsächlich ist. Entsprechend werden wir im Rahmen des operativen CP – also unseres CDPM-Ansatzes – auch die Methode des Conceptual Pair Programming einführen, die Anwender motiviert, durch exploratives Prototyping zusammen mit einem IT-Experten, der effizient mit den CMDSD-Werkzeugen umgehen kann, Anforderungen zu erarbeiten (siehe Abschnitt 6.4.3.5).

Eine Grundidee des CMDSD-basierten Ansatzes ist es, dass die zugrunde liegende Modellierungstechnik im Prinzip direkt eine ausführbare Spezifikation liefert. Dieser Ansatz basiert auf den Erfahrungswerten insbesondere des Intentional Programming (siehe Abschnitt 3.1.2), aber auch auf Erkenntnissen von Language Workbenches (siehe Abschnitt 3.1.6). Da der Anspruch ist, nur mit kundenverständlichen Modellen zu arbeiten, kann das geforderte System gemeinsam mit dem Kunden spezifiziert werden. Da das Modell direkt in eine ausführbare Anwendung transformierbar ist, steht somit im Grunde auch unmittelbar ein Prototyp bereit. Letztlich wird dieser Prototyp nur selten dem fertigen System entsprechen, da es immer Anforderungen geben wird, die doch nur durch Programmierung und nicht durch ein abstraktes und gleichzeitig verständliches Modell ausgedrückt werden kann. Es ergeben sich also funktionale Bausteine, die nur als Dummies existieren. Dies ist aber kein Nachteil – stattdessen sollten bei der Spezifikation entsprechender Funktionsbausteine Platzhalter verwendet werden, die zumindest textuell das erwartete Verhalten beschreiben. So kann ein ausführbares System entworfen werden, dass dem Anwender zumindest ein Grundgefühl für die fertige Lösung vermittelt. Das Rapid-Prototyping-Prinzip zielt also zudem darauf ab, die in Studien der Standish Group als signifikant erkannten Probleme der unrealistischen Erwartungen auf Kundenseite zu reduzieren (siehe Punkt 4 aus Tabelle 2.4 auf Seite 26 sowie [KiRS04, Stan03b]).

K-3.42 ▷ 105
K-3.37 ▷ 90

Gleichzeitig begegnen wir damit einem Problem des klassischen Rapid-Prototyping-Ansatzes (siehe zum Beispiel [Whit95], S. 288): Programmierte Prototypen werden technisch nur in den seltensten Fällen realen Anforderungen genügen. Da bei CMDSD aber ausführbare Modelle erzeugt werden, aus denen direkt lauffähige Anwendungen resultieren, gibt es weniger Abfallprodukte.

Zu Bedenken ist bei einem Rapid-Prototyping-basierten Vorgehen natürlich, dass ggf. wiederholte Refactoring-Phasen erforderlich sind, um die Qualität der Prototypen nicht zu sehr zum Qualitätsstandard für das eigentliche System werden zu lassen – bis hin zur vollständigen Aufgabe eines Prototypen (siehe auch [Broo95]: *„Plan to throw away one prototype. You will, anyhow.“*). Zwar gehen wir durchaus davon aus, dass die Prototypen nach Verfeinerung in vielen Fällen auch produktiv genutzt werden können, hier sind aber vielfältige Rahmenbedingungen und Risiken zu beachten, deren Umfang über diese Arbeit hinausgeht. Wie verweisen dazu auf Abschnitt 10.3.5 in unserem Ausblick. In jedem Fall ist es wichtig auf Erfahrungen und Ideen aus [Davi98] im Rahmen des *operational prototyping* zurückzugreifen, um für die jeweilige Domäne die richtige Art der Prototypisierung zu finden. Dialoggetriebene Anwendungen sind hier im Vorteil, solange nur Masken verändert werden,

da sich hier selten weitreichende Änderungen ergeben. Sobald Veränderungen aber auf Datenstrukturen durchschlagen, kann es bei einem prototyp-basierten Vorgehen recht schwierig werden, die notwendige Qualität aufrechtzuerhalten, da Änderungen ggf. weitreichende Folgen haben. Weitere Hinweise zur Komplexität von Prototyping-basierten Ansätzen und möglichen Lösungen finden sich zum Thema *Evolutionary Systems Development* in [Crinn91] und *Evolutionary Rapid Development* in [SPC97]. Eine gute Zusammenfassung als Ausgangspunkt für weitere Recherchen findet sich bei der Wikipedia unter [Wiki08b] – diese hat auch als Grundlage für unsere vorausgegangenen Hinweise gedient.

5.4.3 Methoden

Basierend auf den zuvor beschriebenen Prinzipien geben wir nun konkrete Methoden vor, die bei der Entwicklung von CMDSD-Werkzeugen unterstützen. Diese orientieren sich an dem bereits in Abschnitt 5.3 vorgestellten Vorgehensmodell.

5.4.3.1 Entwicklungsleitung durch einen Chief Programmer

> *Installiere einen Chief Programmer, der die Entwicklungsmethodik, -kultur und -strategie des CMDSD-Projekts im Rahmen des Domain Engineering verantwortet.*

Das Entwicklungsteam muss so strukturiert sein, dass es fokussiert und konzentriert an den Zielen der CMDSD-Lösungen arbeitet und dabei stets die Kundenanforderungen im Auge hat. Dazu ist ein Chief Programmer zu definieren, der die technische Projektleitung unter besonderer Berücksichtigung der strategischen fachlichen Anforderungen kontrolliert und steuert. Ihm obliegt insbesondere auch die Einführung und Weiterbildung in der von uns hier beschriebenen CMDSD-Vorgehensweise, d. h. er muss nicht nur die hier eingeführten Werte, Prinzipien und Methoden einführen, sondern auch leben und deren kontinuierliche Anwendung sicherstellen. Überwiegend wird dies der bisherige Produktmanager oder Entwicklungsleiter sein, der in den meisten Fällen auch aufgrund der überschaubaren Größe mittelständischer IT-Dienstleister umfassende technische und fachliche Fähigkeiten in einer Person vereint.

K-3.30 ▷ 80

5.4.3.2 Entwicklung eines Architekturstils

> *Lege frühzeitig einen Architekturstil für die zu instanziierenden Lösungen fest.*

Bevor die eigentliche Umsetzung eines CMDSD-Werkzeugs geplant wird, definiert das Entwicklungsteam unter Leitung des Chief Programmers einen Architekturstil für die durch das CMDSD-System generierten Lösungen.

K-3.7 ▷ 61
K-3.9 ▷ 62
K-3.10 ▷ 62

Gelingt es, mit überschaubarem Aufwand zu einem klaren Architekturstil zu definieren, steht den weiteren Schritten prinzipiell nichts entgegen. Fällt dagegen bereits hier auf, dass es große Schwierigkeiten bereitet, einen sinnvollen Architekturstil zu spezifizieren, ist zu befürchten, dass der Dienstleister womöglich nicht über ausreichend Erfahrung in der betrachteten Domäne verfügt und daher eine erneute Machbarkeitsanalyse inklusive einer intensiven Risikobewertung vorgenommen werden muss, um zu entscheiden, ob ein CP-basiertes Vorgehen wirtschaftlich und organisatorisch leistbar ist.

5.4.3.3 MDSD-basierte Software-Produktlinien

> *CMDSD-Werkzeuge sind als modellbasierte und generative Software-Produktlinie zu entwickeln.*

Wir haben Software-Produktlinien in Abschnitt 3.2.6 als ein wertvolles Vorgehensmodell identifiziert, dass bei richtiger Skalierung für den mittelständischen Kontext ein leistungfähiges Entwicklungsprinzip bereitstellt, um eine strategische Produktentwicklung aufzusetzen. Zwei Merkmale von Software-Produktlinien sind dabei besonders hervorgestochen:

K-3.34 ▷ 87

1. Zum einen liefern Software-Produktlinien den notwendigen Rahmen, um die von Hubert im Rahmen der Convergent Architecture geforderten und von uns in Abschnitt 3.1.4 beschriebenen Architekturstilsystematik einen durchgängigen Architekturstil zu definieren und auch im Rahmen einer strategischen Produktentwicklung gezielt fortzuentwickeln.

K-3.11 ▷ 63

2. Zum anderen bieten Software-Produktlinien durch die drei Regelkreise des Domain Engineerings, Application Engineerings und des Linienmanagements den organisatorischen Rahmen, um das Zielprodukt gut kontrollierbar und in enger Abstimmung mit den Kundenbedürfnissen weiterzuentwickeln.

K-3.33 ▷ 85

Wir legen daher fest:

> *Verwende die Software-Produktlinien-Methodik als Vorlage für die Entwicklung eines CP-basierten Produktansatzes. Adaptiere das Vorgehensmodell für Software-Produktlinien soweit wie möglich auf den spezifischen Domänenkontext.*

CMDSD liefert die notwendigen Optimierungshinweise für den Aufbau des Domain Engineering, CDPM (Kapitel 6) leistet Selbiges für das Application Engineering, das in Abschnitt 4.2.3.3 vorgestellte Produktgremium liefert den Rahmen für den Managementregelkreis. Wir haben auch bereits ausgeführt, dass es wichtig ist, Software-Produktlinien in ihrer methodischen Komplexität an den mittelständischen Kontext anzupassen:

- Jedes für Software-Produktlinien definierte Methodengebiet (siehe Abschnitt 3.2.6 für Details) ist auf seine Eignung für den spezifischen Kontext zu prüfen.
- Die Entscheidung sollte auf Grundlage einer Geschäftswertanalyse erfolgen, in deren Rahmen die zu erwartenden Aufwände bei der Nutzung des Methodengebiets in Verhältnis gesetzt werden zu dem Nutzen. In Kapitel 9 werden wir Formeln für Abschätzungen zur Chancen-/Risiken-Analyse eines solchen Vorhabens angeben.
- Es ist zu entscheiden und zu dokumentieren, ob das Methodengebiet unverändert, verändert oder gar nicht verwendet wird. Insbesondere sind für eine spätere Wiedervorlage die identifizierten Chancen und Risiken festzuhalten.
- In regelmäßigen Abständen (wir empfehlen basierend auf unseren Erfahrungen 6-12 Monate) sind diese Entscheidungen zu überprüfen und ggf. an den jeweiligen Erfahrungs- und Entwicklungsstand anzupassen.

Sollte sich erweisen, dass Software-Produktlinien als Methodik für den betrachteten IT-Dienstleister in weiten Teilen zu komplex sind, ist zu prüfen, ob Feature Driven Development als herunterskaliertes Entwicklungsmodell besser geeignet ist (siehe Seite 131).

K-3.46 ▷ 111

In jedem Fall gelten die folgenden Grundregeln für das weitere Vorgehen, die wir auch bereits in anderen Prinzipien und Methoden dieses Kapitels fundamentiert haben:

K-3.42 ▷ 105
K-3.47 ▷ 112
K-3.39 ▷ 93
K-3.36 ▷ 89
K-3.37 ▷ 90

- *Das gesamte System wird als modellbasiertes System konzipiert.*
- *Das gesamte System arbeitet generativ.*
- *Das gesamte System wird auf Grundlage von Domain Driven Design entwickelt.*
- *Die Domänenmodelle sind so fachlich wie möglich orientiert und vermeiden technische Aspekte.*
- *Die Domänenmodelle finden sich durchgängig wieder.*

Durch die konsequente Verwendung eines modellbasierten Vorgehens in unseren CMDSD-Werkzeugen erfüllen wir zudem die Forderung nach einer universellen Sprache aus dem Domain Driven Design (siehe Abschnitt 3.2.7 bzw. [Evan04a], S. 23ff.). Die Modelle des CMDSD-Werkzeugs sind unsere universelle Sprache, wir wir zu Realisierung domänenspezifischer Lösungen verwenden. Wir werden die Benutzung der Sprache u. a. durch die Einführung des sogenannten *Conceptual Pair Programmings* in Abschnitt 6.4.3.5 im Rahmen unserer Ausführung zur zweiten Hauptmethode des Conceptual Programmings – dem Concept Driven Project Management – in Kapitel 6 forcieren. Zudem folgen wir mit diesem Vorgehen auch Evans Hinweisen zu der Wichtigkeit eines durchgängigen Modells zur Realisierung von Lösungen[23].

Erfahrungswerte aus dem Bereich der Software-Produktlinien zeigen, dass wir durch die Fundamentierung des CMDSD-Ansatzes in Software-Produktlinien einen wertvollen Beitrag zur Steigerung der Wirtschaftlichkeit von Projekten erhalten werden. So wird z. B. in [ClNo02], S. 26–27, eine Fallstudie präsentiert, die u. a. von einer Steigerung der Wiederverwendung von 70 auf über 90%, einer Reduzierung der Auslieferungszeit von Jahren auf Monate und einer Halbierung der Systemkosten durch Softwareanteile berichtet. Zudem werden weniger als 50 Mitarbeiter in die Lage versetzt, die Arbeit von eigentlich mehr als 200 Angestellten zu erledigen und

[23] Siehe dazu [Evan04a], S. 41: *„The thrust of this book is that one model should underlie implementation, design and team communication.“*

dabei gleichzeitig noch mehr Systeminstanzen auszuliefern. Natürlich beziehen sich diese Zahlen auf sehr große und komplexe Projekte, die vermutlich vergleichsweise viel Optimierungspotenzial besaßen – dennoch ist aber anzunehmen, dass auch in kleineren mittelständischen Projekten eine nachhaltige Steigerung der Produktivität erfolgen wird, insbesondere auf Seite der Dienstleister.

Wir weisen aber auch noch mal daraufhin, dass unsere Fundamentierung von CMDSD-Werkzeugen auf Basis von Software-Produktlinien bewusst keine abschließende Aussage zur grundlegenden Implementierungsplattform macht – wir haben in Kapitel 3 eine Reihe interessanter und wertvoller Plattformen identifiziert wie z. B. Software Factories (siehe Abschnitt 3.3.3) oder Intentional-Programming-Plattformen (siehe Abschnitt 3.1.2), die maßgeblich zu einer wirtschaftlich rentablen Realisierung des CMDSD-Werkzeugs beitragen können. Die Auswahl der konkreten Plattform ist sehr kontextabhängig und wird nicht zuletzt durch die spezifische Domäne, die Vorerfahrungen der Entwickler und Architekten sowie die Stabilität der verfügbaren Plattformen determiniert. Daher werden wir diesen Aspekt hier nicht weiter betrachten, sondern beschränken uns mit dem Hinweis, dass hier eine entsprechend multikriterielle Plattformauswahl zu treffen ist. Fällt die Entscheidung für die Neuimplementierung einer Plattform, liefert generative Programmierung (siehe Abschnitt 3.3.4 bzw. [CzEi00]) wichtige methodische Beiträge zur Realisierung solcher Systeme und sollte unbedingt als Grundlage herangezogen werden.

5.4.3.4 Unternehmenskonformes Projektmanagementmodell

> *Kombiniere unternehmensspezifische bewährte Erfahrungen im Projektmanagement mit den CMDSD-Optimierungsansätzen, um so ein synergetisch wirkungsstarkes Projektmanagementmodell zu entwerfen.*

Da es sich bei CMDSD um ein domänenunabhängiges Metamodell zur Ausprägung domänenspezifischer Software-Produktlinien handelt, werden wir an dieser Stelle kein spezifisches Projektmanagementmodell für die Erstellung des CMDSD-Werkzeugs vorgeben. Das zu verwendende Projektmanagementmodell hängt von den folgenden Bedingungen ab:

K-3.13 ▷ 64
K-3.14 ▷ 64
K-3.54 ▷ 127

- Die domänenspezifischen Rahmenbedingen können einen sehr unterschiedlichen formalen Einfluss auf das zu verwendende Projektmanagementmodell haben (siehe dazu Abschnitt 5.5.4).

- Die Unternehmenskultur des IT-Dienstleisters spielt eine große Rolle. Das ideale Projektmanagementmodell kombiniert die bereits bewährten und im Unternehmen im Einsatz befindlichen Methoden mit den Werten, Prinzipien und Methoden, die wir in dieser Arbeit vorgeben, um so ein synergetisches Modell zu entwickeln, dass sowohl unsere Optimierungsansätze als auch die bereits existierenden Stärken des Dienstleisters berücksichtigt.

In jedem Fall muss die resultierende Methode folgende Anforderungen erfüllen:

- Sie muss einen klaren zur Unternehmenskultur und Domäne passenden Entwicklungsprozess vorgeben. K-3.13 ▷ 64 K-3.26 ▷ 74
- Sie muss die Rollen und Verantwortlichkeiten der Mitglieder des CMDSD-Teams klar definieren. K-3.14 ▷ 64
- Es muss ein iteratives, evolutionäres, rekursives und inkrementelles Entwicklungsmodell definiert werden. K-3.19 ▷ 66 K-3.54 ▷ 127
- Anwender sollten so früh wie möglich in die Entwicklung einbezogen werden. K-3.38 ▷ 91
- Das Vorgehensmodell sollte möglichst viele Prinzipien und Werte der agilen Software-Entwicklung berücksichtigen und leben. K-3.35 ▷ 88

5.4.3.5 Archetypisierung

Bevor wir uns mit dem Konzept der Archetypisierung beschäftigen, wollen wir zunächst den Begriff herleiten und definieren: Das Wort *Archetyp* meint laut [Wiki08e, Wikt06] *„ein Muster(exemplar), das zeitlich oder strukturell am Anfang steht, völlig neu und Vorbild für die Nachfolger ist“*. Der Duden ([DSSW97], S. 81) versteht darunter u. a. *„Original eines Kunst- oder Schriftwerkes im Gegensatz zu Nachbildungen oder Abschriften“* oder *„Urform, Musterbild“*. Wir wollen bezogen auf CMDSD-Systeme einen Archetyp wie folgt definieren:

> *Ein Archetyp ist eine Vorlage für eine domänenspezifische (Teil)Lösung. Archetypen lösen ein in sich geschlossenes Thema in zufriedenstellender Weise, wenn auch meistens nicht erschöpfend.*

Die Idee des Archetypen-Konzepts ist, dass eine Lösung über ihren kleinsten gemeinsamen Nenner (negativ formuliert) oder ihre allgemeinste und flexibelst anpassbare Form (positiv formuliert) beschrieben werden kann. Will man basierend auf einem

CMDSD-System, das Archetypen enthält, eine komplexe Lösung entwickeln, kann man sich dann entscheiden, ob man dafür ggf. den passenden Archetypen (verändert oder unverändert) als einen Lösungsbestandteil nutzen will oder über das Modellierungswerkzeug und die zu Verfügung stehenden Modelle eine neue Ausprägung für eine Lösung schafft. Wir werden im Rahmen der Verifikation unseres Ansatzes verschiedene Archetypen für Intranet- und Extranetsysteme vorstellen, darunter u. a. den *FAQ*, die *Dokumentenliste*, das *Forum* oder den *Newsletter*. Solche Archetypen geben eine (häufig relativ einfache) Lösung für ein spezifisches Problem an und bieten somit eine exzellente Grundlage für eine komplizierte Lösung (da viele Grundlagenarbeiten nicht mehr gemacht werden müssen, wenn eine neue Ausprägung erforderlich wird). Man kann diesen Ansatz z. B. mit Vorlagen in Microsoft Word (siehe [Micr09a]) oder Intrexx-Anwendungsvorlagen (siehe [Intr06]) vergleichen, wir systematisieren diesen Ansatz hier für die modellbasierte Entwicklung.

Formal gesehen ist ein Archetyp, der sich aus einer Menge von Modellen zusammensetzt, die Hülle aller zur Beschreibung des Archetypen relevanten Modelle. Daraus ergeben sich wiederum interessante Implementierungsfragestellungen, die sich am Besten an einem Beispiel illustrieren lassen: Sei ein Archetyp A_x gegeben, der u. a. ein Modell M_D zur Beschreibung einer Datenstruktur enthält. Nun wird der Archetyp A_x zweimal instanziiert. Es ergibt sich die Frage: Sollen zwei verschiedene Instanzen von M_D erzeugt werden (um die Daten disjunkt zu halten) oder beziehen sich beide Archetyp-Instanzen auf die gleiche zugrunde liegende Modellinstanz des Datenmodells. Fachlich gesehen könnten beide Antworten richtig sein. Faktisch resultieren daraus beim Entwurf eines CMDSD-Systems die folgenden Notwendigkeiten, die aber nur anhand der jeweils spezifischen Domäne abgeleitet werden können:

- Entweder beziehen sich Archetypen immer auf für sie eindeutige Instanzen von Modellen (d. h. kein Modell wird von verschiedenen Archetyp-Instanzen gemeinsam verwendet) oder
- es gibt bestimmte Modelltypen (häufig werden dies Datenstrukturen sein, es kann sich aber auch um Konnektoren, Schnittstellenbeschreibungen, etc. handeln), die in manchen Fällen von verschiedenen Archetypinstanzen gemeinsam genutzt werden. In diesem Fall sind weitere Entwurfsentscheidungen zu treffen:
    - *Wie wird die fachliche Identität eines Modells festgelegt?* Dahinter steht wieder am Beispiel illustriert die Frage „Woher weiß ich, dass ein Modell für einen Domänentyp inhaltlich einen FAQ-Eintrag beschreibt?“. Gängigste Lösung ist die Vergabe einer Identifikations-GUID[24] zusammen mit

[24] GUID steht für „global unique identifier“ und meint eine global eindeutige ID.

einer Versions-GUID, die initial der Identifikations-GUID entspricht).

- *Wie entscheidet der Anwender ergonomisch darüber, welche Modellinstanzen geteilt werden und welche neu ausgeprägt werden?* Häufig wird er sich anhand einer Auswahlliste entscheiden müssen. Schwieriger ist die Frage, wie z. B. effizient mit logischen Bezeichnern umgegangen wird, da es z. B. keinen Sinn macht, in einem Modellierungswerkzeug zweimal den gleichen logischen Namen für zwei fachlich verschiedene Dinge zu sehen, nur weil dieser Name im zugrunde liegenden Archetypen angegeben war. Je nach Domäne wird hier eine manuelle, halbautomatische oder automatische Umbenennung erforderlich sein.
- *Was passiert, wenn sich eine neue Archetypinstanz eine bereits existierende Modellinstanz mit einer bereits existierenden Archetypausprägung teilen soll, die existierende Modellinstanz aber bereits modifiziert wurde?*[25] In diesem Fall muss untersucht werden, ob es sinnvolle Verschmelzungsoperationen gibt[26] oder ob eine Wiederverwendung aufgrund bestimmter Änderungen gar nicht möglich ist und daher dem Anwender auch gar nicht erst angeboten wird.

Auch frühere Ansätze wiesen zur Archetypisierung ähnliche Ideen auf – so beschreibt z. B. Martin in [Mart81] auf den Seiten 96–99 sogenannte *packages*, die als Bausteine oder konfigurierbare Module eines Gesamtsystems zu sehen sind. Er weist aber bereits auf S. 96 unten darauf hin, dass er davon ausgeht, dass solche Packages zukünftig durch „Applikationsgeneratoren“ beschrieben sein werden, so dass man diese viel einfacher anpassen können wird. Genau diesen Gedanken haben wir mit unserer Idee modellbasierter Archetypisierung aufgegriffen. Ein weiteres Beispiel für ein System, dass mit großen Erfolg durch sogenannte Anwendungsvorlagen ein ähnliches Konzept verfolgt, ist Intrexx (siehe [Intr09]) – hier ist sogar ein Markt für Drittanbieter entstanden, die selbst entwickelte Vorlagen aufgrund ihrer Exportierbarkeit auch anderen (neuen) Kunden zur Verfügung stellen. Somit ergibt sich ein weiteres interessantes wirtschaftliches Argument für die Integration von Archetypisierung: Es ergeben sich nicht nur für Kunden Nutzeffekte durch die Möglichkeit zur Wiederver-

[25] Dies ist daran zu erkennen, dass Identifikations-GUID und Versions-GUID nicht mehr übereinstimmen; wir nehmen an, dass die Versions-GUID bei einer Veränderung durch eine neue GUID ersetzt wird.

[26] Zum Beispiel haben bei einer Datentabelle ergänzte Datenspalten keine Auswirkungen auf Archetypinstanzen, die eigentlich nur die Basisversion benutzen wollen. Veränderte Typen oder Restriktionen können bereits Auswirkungen haben und gelöschte Spalten haben diese ziemlich sicher. Daraus könnte man z. B. für Datentabellen die Anforderung an das System ableiten, das destruktive Änderungen wie Löschungen nicht zugelassen werden, was sich mit Vererbungsprinzipien in den meisten objektorientierten Sprachen decken würde.

wendung einmal konzipierter und umgesetzter Lösungen, auch Dienstleister können Skaleneffekte (siehe dazu unsere Ausführungen auf S. 39) effektiver für sich nutzen.

Der Methode der Archetypisierung kann nun für zwei verschiedene Szenarien genutzt werden, die wir nachfolgend erläutern wollen: Einerseits kann sie als Unterstützung zur Domänenbildung dienen (Abschnitt 5.4.3.5.1), andererseits ist sie auch eine Hilfe in der nachfolgend detaillierter beschriebenen Modellbildung (Abschnitt 5.4.3.5.2).

5.4.3.5.1 Domänenbildung durch Archetypisierung Die Abgrenzung einer Domäne und die Identifikation der essentiellen Charakteristika ist ein komplexes Problem – siehe dazu unsere Ausführungen z. B. in Abschnitt 3.2.7 und insbesondere bei der in Abschnitt 5.4.3.7 definierten Methode zur Domänenbildung. Insofern ist jede Unterstützung hier wertvoll, insbesondere wenn sie wie die Archetypisierung Szenarien liefert, mit den geprüft werden kann, ob bereits alle Konzepte der Domäne erfasst sind. Archetypisierung liefert also im Prinzip die Stories, die den Kern der Domäne charakterisieren.

Domänenbildung durch Archetypisierung ist eine ergänzende Methode für die nachfolgend beschriebenen Hauptmethoden, die wie folgt zur Beschreibung einer Domäne anzuwenden ist:

- Finde archetypische Anwendungsfälle aus der betrachteten Problemdomäne. Diese werden häufig die Form von Use Cases[27], Stories[28], Modulen[29] oder Programmbausteinen[30] haben. Zur Erhebung von archetypischen Anwendungsfälle bieten sich hinreichend bekannte Kreativtechniken wie Brainstorming, Brainwriting, Mindmapping, usw. an (siehe z. B. [Nöll06]).
- Beschreibe die Archetypen. Hier bieten sich beispielsweise Ablaufdiagramme o. ä. an. Auch in der Beschreibung ist es wieder wichtig, zwecks Archetypisierung das Pareto-Prinzip anzuwenden: Die Beschreibungen sollten sich auf die 80% der Standardszenarien konzentrieren. Alles, was den Charakter eines Spezial- oder Sonderfalls hat, wird mit hoher Wahrscheinlichkeit auch mehr oder minder subtil anders funktionieren müssen. Daher sollten Spezialfälle nie berücksichtigt werden sondern erst bei der Instanziierung der Archetypen als Modifikationen eingeführt werden[31].

[27] Beispiel: „Ein neuer Mitarbeiter wird in die Arbeitsrichtlinien des Unternehmens eingewiesen."
[28] Beispiel: „Ich will eine Reisekostenabrechnung machen."
[29] Beispiel: die Verwaltung der Urlaubsanträge.
[30] Beispiel: die Erstellung einer Rechnung.
[31] Ein gutes Beispiel ist der Archetyp „FAQ" (Frequently Asked Questions) für Intranets, der im Wesentlichen die Anzeige einer Liste von Fragen und Antworten beinhaltet. Zudem wird es eine

- Strukturiere die Archetypen so, dass sie sinnvolle Erweiterungspunkte enthalten. Indizien dafür finden sich anhand der Features, die aus dem Archetypen aus oben genannten Gründen herausgenommen wurden. Dies sind Stellen, die mglw. häufiger angepasst werden, um Erweiterungen zu berücksichtigen. Hier gilt generell der Grundsatz, dass erweiterungsbedürftige Schritte, die in einem Archetypen abgebildet werden, möglichst stark modularisiert werden sollten[32]. Zur Trennung von Erweiterungspunkten hat sich für uns bewährt, einen auf *Feature Driven Development* (siehe Seite 131) basierenden Ansatz zu wählen und die Features bzgl. der Frage „Werden wir dieses Feature in 80% der Fälle genau so benötigen“ nach dem Pareto-Prinzip zu beurteilen. Kann man die Frage bejahen, gehört das Feature häufig in den Archetypen, muss man sie verneinen, sollte man prüfen, ob bereits ein geeigneter Erweiterungspunkt existiert bzw. wie dieser aussehen sollte.

K-3.3 ▷ 55
K-3.18 ▷ 66
K-3.48 ▷ 114

Die Beschreibungen der Archetypen können dann in einer Art „Domain Mining“ dahingehend untersucht werden, ob es neue Konzepte, Begriffe oder Zusammenhänge gibt, die im bisherigen Domänenmodell noch nicht berücksichtigt werden. Die archetypisierende Form der Domänenbildung hat den großen Vorteil, dass Archetypen auch gemeinsam mit Kunden erarbeitet werden können und so der unmittelbare Kundenzug bereits während der Produktdefinition gegeben ist.

5.4.3.5.2 Modellbildung durch Archetypisierung Archetypen können auch zur Modellbildung eingesetzt werden – leichter wird dies fallen, wenn sie vorher auch wie in Abschnitt 5.4.3.5.1 zur Domänenbildung eingesetzt wurden, es geht aber auch ohne diesen Schritt. Voraussetzung dafür ist, dass bereits eine Modellbildung wie in

K-3.36 ▷ 89

Suchfunktionalität sowie eine Funktion zum Anlegen neuer bzw. Verändern oder Löschen bestehender FAQ-Einträge geben müssen. Nun ist aber genauso denkbar, dass in einer typischen Redaktionsumgebung ein mehrstufiges Genehmigungsverfahren gefordert sein könnte. Da Genehmigungsverfahren aber typischerweise sehr unterschiedliche Rahmenbedingungen (z. B. in der Anzahl der Genehmigungsstufen, bei der Frage, ob eine Genehmigung aus einem Kreis von Verantwortlichen ausreicht oder alle zustimmen müssen, etc.) haben, sollte dieses Konzept aus dem Archetypen herausgehalten und erst bei Instanziierungen in der speziellen gewünschten Form berücksichtigt werden. Sollten bei einem zu erstellenden Produkt dann viele FAQ-Instanzen mit der komplizierteren Semantik benötigt werden, kann dieser einmal ausgeprägt und dann als neuer Archetyp bereitgestellt werden, z. B. unter der Bezeichnung „FAQ mit Genehmigungsworkflow“. Dieses Beispiel illustriert sehr deutlich die Mächtigkeit des Archetypen-Konzepts für domänenspezifische Lösungen.

[32] Für unseren Archetypen des FAQs könnte dies z. B. bedeuten, dass die Änderung- und Anzeigen-Seiten tatsächlich unterschiedliche Seiten mit unterschiedlichen Verbindungen sind, damit es leichter wird, einen Genehmigungsworkflow einzuhängen, ohne auch die Anzeigefolge modifizieren zu müssen. Welche Form der Modularität die Richtige ist, hängt aber sehr stark von der spezifischen Domäne ab, so dass wir hier keine weiteren allgemeinen Empfehlungen machen können.

der nachfolgend beschriebenen Methode „Strukturierte Modellbildung“ (siehe Abschnitt 5.4.3.8) stattgefunden hat.

Im Einzelnen sind die folgenden Schritte notwendig, um Archetypen für die Modellbildung zu nutzen:

1. Stehen die Archetypen fest, müssen sie domänenspezifisch modelliert werden. Dazu sind die prinzipiellen Konzepte zu identifizieren, aus denen sich ein Archetyp zusammensetzt. Die Modellierung wird zunächst üblicherweise manuell und mit einem generischen grafischen Werkzeug erfolgen, um ein Gefühl für die Struktur und Komplexität der Modelle zu bekommen. Dieser Modellierungsvorgang sollte immer in einem kleinen Team aus Fach- und Technikexperten erfolgen, um das erforderliche Gesamtwissen in einen kreativen Prozess einzubringen. Hier sind insbesondere auch die Prinzipien und Methoden aus dem Agile Modeling (siehe Abschnitt 3.3.9) von Nutzen.

K-3.2 ▷ 50
K-3.38 ▷ 91
K-3.52 ▷ 120

2. Zunächst ist herauszuarbeiten, welche Modellformen benötigt werden. In interaktiven Systemen wird man häufig wiederkehrende Elemente wie Masken, Workflows u. ä. vorfinden, in anderen Domänen können dies ganz andere Elemente sein[33].

3. Die Modelle sollten in einer Top-Down-Sicht angeordnet werden, um sich vom Allgemeinen zum Besonderen vorzuarbeiten.

4. Stößt man auf Stellen, an denen sich eine explizite Modellierung als sehr komplex erweist, weil sehr viele auf komplexe Weise miteinander interagierende Modelle benötigt werden, ist zu prüfen, ob das zu modellierende Konstrukt nicht so zentral für die betrachtete Domäne ist, dass man es besser durch ein einziges wesentlich abstrakteres Modell mit einer hinreichenden Zahl von Variationsmöglichkeiten beschreibt[34].

K-3.5 ▷ 57

[33] Betrachtet man zum Beispiel DocScape, dass wir in Abschnitt 9.4.3 kurz vorstellen werden, ergeben sich zwar auch Seiten als Modelle, dann folgen aber schnell Raster, Rasterzellen, usw. um typographische und layouterische Aspekte abzubilden.

[34] In unserem Referenzbeispiel zur Lebensmittelüberwachung gibt es zu Beispiel die Anforderung, sogenannte Katalogauswahlhilfen zu unterstützen. Dabei handelt es sich im Grunde um komplexe Suchmasken auf spezialisierten Datenbeständen, die an vielen Eingabefeldern als Unterstützung für Fachkräfte zur effizienten Datenerfassung angebunden werden. Da es sich um einen immer wiederkehrenden Anwendungsfall handelt, der zudem in seiner Detailausprägung sehr spezialisiert und damit komplex ist, haben wir entschieden, ihn durch eine abstrakte Parametrisierung der Form *supports-catalog-selection-of* abzubilden, die lediglich wissen muss, welches Art von Datum erfassbar sein soll. Alles weitere leitet der Generator aus dem Kontext ab. Werden weitere Sonderfälle benötigt, können diese ebenso abstrakt dargestellt werden. Dies erspart eine unnötig komplexe, explizite Modellierung, die zudem die wiederholte Ausprägung des gleichen Merkmals erschwert und führt so zu einer deutlich effizienteren Behandlung der Anforderung.

5. Gibt es bereits fachliche Modelle, sind diese in die Modellierung einzubeziehen und insbesondere auf ihre Anwendbarkeit für eine modellbasierte Generierung von Software zu prüfen (siehe den nachfolgenden Reduktionsalgorithmus). In manchen Fällen wird man solche Modelle sogar unverändert übernehmen können. Sollten doch Erweiterungen erforderlich sein, sind insbesondere die technischen Aspekte so minimal wie möglich zu halten.

K-3.48 ▷ 114

6. Es ist darauf zu achten, dass nur die Modelle konzipiert werden, die tatsächlich zur Beschreibung der Archetypen gebraucht werden.
7. Anschließend ist zu prüfen, welche Metamodelle es geben muss, um technische Randbedingungen variabel ausprägen zu können. Häufig wird dies auf einfache Checklisten hinauslaufen.

5.4.3.6 Komplexitätsbewertung durch das Pareto-Prinzip

> *Folge dem Pareto-Prinzip, um die Sinnhaftigkeit von Funktionen und Erweiterungen eines CMDSD-Systems zu bewerten: Ziel muss es sein, 80% der normalen Aufgaben mit dem Werkzeug in 20% der normalen Zeit erledigen zu können. Zudem sollte die (Bedienung und Zielsetzung der) Erweiterung durch 80% der Anwender verstanden werden können.*

Die Bewertung von Anforderungen und Ideen zur Erweiterung eines Werkzeugs ist häufig nicht einfach, da Erweiterungswünsche in drei Kategorien fallen:

1. Es gibt klare Business-Case-getriebene Anforderungen, bei denen Einigkeit herrscht, dass sie umgesetzt werden müssen, um wirtschaftlich erfolgreich zu sein.
2. Die Anforderungen sind nicht durch einen Business Case belegbar bzw. es ist kein unmittelbarer Handlungsbedarf erkennbar[35].
3. Die Anforderungen fallen in eine Grauzone, d. h. sie haben eine gefühlte Wichtigkeit, es fehlen aber stichhaltige Belege für die Notwendigkeit.

Die letzte Art von Anforderungen ist nach unserer Erfahrung leider die häufigste Variante, was immer wieder zu komplexen Bewertungsfragen hinsichtlich des Werts

[35] Nach unseren Erfahrungen passiert dies häufig, wenn Kunden eine neue Anwendung als Ersatz für ein existierendes System spezifizieren und sich dann zu stark an – möglicherweise technisch, fachlich oder historisch überholten – Funktionsweisen (nicht Funktionen!) des Altsystems orientieren.

K-3.27 ▷ 74

einer Umsetzung führt. Eine Kosten-/Nutzen-Analyse mag zwar einen ersten Indikator für die Sinnhaftigkeit liefern, aufgrund der mit Abschätzungen verbundenen Unsicherheit gerade auch bezüglich des Nutzens erweisen sich derartige Entscheidungen aber insbesondere bei kleineren Projekten als sehr schwierig. Aufgrund der Komplexität des Themas empfehlen wir eine Erfahrungsregel, die sich in unseren Projekten bewährt hat und die auf dem Pareto-Prinzip (siehe [Wiki08c]), angewendet auf CMDSD-Werkzeugerweiterungen, beruht:

1. Ist man unsicher bzgl. der Umsetzung einer Anforderung, sollte man sie dem Pareto-Prinzip folgend dahingehend hinterfragen, wie häufig die Umsetzung der Anforderung zukünftig helfen wird, Zeit zu sparen. Reduziert ihre Umsetzung in 80% der Anwendungsfall den Aufwand auf 20%, so ist sie ein geeigneter Kandidat[36].
2. Zweites wichtiges Kriterium ist, dass die Erweiterung auch verständlich genug ist, dass sie von 80% der zukünftigen Anwendern verstanden wird.

Gelingt es, gemäß beider Kriterien für die Erweiterung zu argumentieren, ist sie ein Umsetzungskandidat und kann in die normale Projektplanung (je nach verwendetem Vorgehensmodell) aufgenommen werden[37]. Wichtig ist, dass ein Argument für die Umsetzung einer Anforderung aus Sicht eines Dienstleisters auch sein kann, dass 80% bestimmter im Marketing oder Vertrieb erschwerend wirkender Argumente durch die Umsetzung ausgeräumt werden. Aber auch nur dann, wenn Anwender ihren Sinn auch nachvollziehen können[38].

Wie motivieren sich nun die beiden Anwendungen des Pareto-Prinzips? Zum einen

[36] Man beachte, dass weiterhin das Prinzip der Domänenfokussierung gilt – wir werden keine Anforderungen umsetzen, die unsere Domäne verlassen, um das Werkzeug in seiner Klarheit nicht zu verununreinigen.

[37] Insbesondere sollte man nicht nur sich selbst sondern andere Personen von der Stichhaltigkeit der Argumentation überzeugen.

[38] In den Projekten der QuinScape GmbH ist uns folgendes Beispiel begegnet: Das Produkt *DocScape* ist ein regelbasiertes Werkzeug zur vollautomatischen datenbasierten Erzeugung hochwertiger und komplexer Dokumente, z. B. von Katalogen oder Haushaltsplänen (siehe [Docs06] bzw. Abschnitt 9.4.3). Da das System vollautomatisiert arbeitet, ist eigentlich keine Nachbearbeitung generierter Dokumente notwendig. DocScape-Kunden haben oft noch keine Erfahrung mit Vollautomatisierung und halten diese häufig für unmöglich. Daher ist eine typische Vertriebsfrage, ob man die generierten Dokumente auch manuell nachbearbeiten kann – denn so sind es die Kunden von anderen existierenden Produkten her gewohnt. DocScape verfügt im Moment nicht über eine Möglichkeit, generierte PDF-Dokumente umfangreich nachzubearbeiten – obwohl einzelne Seiten durch manuell bearbeitete Seiten ausgetauscht und Texte und Bilder in sehr begrenztem Maß verändert werden können. Für Marketing- und Vertriebsbemühungen wäre es aber sehr wertvoll, auch einen Export in DTP-Systeme wie Adobe Indesign (siehe [Adob09]) anbieten zu können, obwohl dies operativ keinen Nährwert bietet. Insofern ist diese Anforderung nach einer Kosten-/Nutzenanalyse auf die Umsetzungsliste gesetzt worden, auch wenn sie nicht sehr hoch priorisiert ist.

fordern wir eine hohe Zugänglichkeit unseres Ansatzes, so dass Funktionen, die nur von wenigen Anwendern verstanden werden, das System in seiner Bedienbarkeit unnötig belasten. Zu viele solcher Funktionen führen zum klassischen „Feature Creep“ (siehe [Wiki08d]) in der Ergonomie. Die erste oben formulierte Anwendung des Pareto-Prinzips motiviert sich aus den Werten *Wirtschaftlichkeit* und *Kundenorientierung*: Man sollte nur Anforderungen umsetzen, die auch einen Nutzen haben (also z. B. Zeit sparen, egal ob in der Entwicklung oder im Vertrieb) und zudem dem Kunden helfen (sei es, in dem sie wahrgenommene Risiken reduzieren oder tatsächlich effektivere Umsetzungen erlauben).

Somit stellt unsere Methode der Komplexitätsbewertung durch das Pareto-Prinzip ein nützliches Zusatzwerkzeug im Werkzeugkasten von Entwicklern, Architekten und Projektmanagern dar, die als Zusatzkriterium für klassische Verfahren der Anforderungsbewertung genutzt werden kann.

5.4.3.7 Domänenabgrenzung und Modellbildung durch Domain Driven Design

> *Verwende Domain Driven Design, um sowohl die Domäne in ihrem Umfang klar abzugrenzen als auch die erforderlichen Modelle, die als Grundarbeitsmittel im Rahmen des CMDSD-Werkzeugs zu realisieren sind, zu ermitteln.*

In Kapitel 3 haben wir in Abschnitt 3.2.7 den Domain-Driven-Design-Ansatz von Eric Evans als einen der wichtigsten Ansätze zur Erfassung und Formalisierung von Domänenwissen identifiziert. Zudem haben wir in Kapitel 4 in Abschnitt 4.2.2.2 das Prinzip der Domänenfokussierung für unseren Ansatz gefordert, nicht zuletzt um dem Wert der Zugänglichkeit aus Kundensicht (siehe Abschnitt 4.2.1.3) und dem Wert der Wirtschaftlichkeit aus Dienstleistersicht (siehe Abschnitt 4.2.1.1) gerecht zu werden.

K-3.39 ▷ 93

Evans beschreibt einige wichtige Regeln zur effektiven Modellbildung (siehe [Evan04a], S. 12–13 bzw. Abschnitt 3.2.7), die auch essentiell für den Entwurf von CMDSD-Werkzeugen sind.

> *Verwende die durch Domain Driven Design vorgegebenen Regeln und Empfehlungen zur Domänenabgrenzung und Modellbildung.*

Wir fassen diese Empfehlungen kurz zusammen und begründen ihre Wichtigkeit:

- *Verbinde Modell und Implementation möglichst eng.* Damit werden direkt unsere Prinzipien von Vereinfachung (Abschnitt 5.4.2.1), Problemorientierung (Abschnitt 5.4.2.2), Uniformität (Abschnitt 5.4.2.3), Kompaktheit (Abschnitt 5.4.2.5) und Abstraktion (Abschnitt 5.4.2.6) gefördert. Zudem beschreibt Evans auf den Seiten 8–12 von [Evan04a], wie genau diese enge Bindung dazu führt, dass sich Domänenexperten mit Begeisterung und einem natürlichen Verständnis von seinem Nutzen auf ein evolutionäres Vorgehen sowie Prototyping eingelassen haben – ein weiteres Prinzip, dass wir fördern wollen (Abschnitt 5.4.2.10).

K-3.37 ▷ 90

- *Kultiviere eine Sprache auf Basis des Modells.* Wir haben bereits formuliert, dass wir eine domänenfokussierte Modellbildung anstreben. Evans bestätigt diesen Ansatz, da seine Erfahrungen zeigen, dass dies zu einer deutlichen Verbesserung in der Kommunikation zwischen Entwicklern und Domänenexperten führt.

K-3.36 ▷ 89
K-3.37 ▷ 90

- *Entwickle ein wissensintensives Modell.* Evans meint damit, dass das Modell nicht ein reines Datenschema, sondern auch Restriktionen und Logik enthalten soll. Dies wird bestärkt durch unsere Forderung nach sehr ausdrucksstarken Modellelementen auf einer hohen Abstraktionsebene (siehe Abschnitt 5.4.2.6).

K-3.23 ▷ 73

- *Destilliere das Modell.* Damit meint Evans, dass die essentiellen Domänenaspekte Bestandteil des Modells sein sollen, Konzepte aber auch weggelassen werden können, wenn sie nichts zum Lösungsprozess beitragen[39].

K-3.10 ▷ 62
K-3.48 ▷ 114

- *Brainstorming und Experimentieren.* Evans meint damit eine szenarienorientierte Vorgehensweise, in der Dutzende von Alternativen durchprobiert werden. Dies spiegelt seinen Hintergrund in großen Projekten wider – er geht von sehr umfangreichen Explorationsphasen aus, die es in typischen mittelständischen Projekten allein schon aufgrund der gegebenen zeitlichen Restriktionen üblicherweise weder geben kann noch wird. Trotzdem ist auch dieser Appell wichtig, passt er doch gut zu dem von uns vorgeschlagenen Grundprinzips des *Rapid Prototypings*, das wir als exploratives Werkzeug für essentiell halten (siehe Abschnitt 5.4.2.10).

K-3.19 ▷ 66
K-3.20 ▷ 67
K-3.22 ▷ 72

Auf diese Weise können leistungsfähige Fachmodelle entwickelt werden, die als Grundlage der zu erstellenden Software-Produktlinie dienen.

[39] Er spricht sich also explizit gegen eine Vollständigkeit der Vollständigkeit halber aus – wichtiger ist es, ein handhabbares und leistungsfähiges Domänenmodell zu finden, dass eine effektive Lösung von Problemen erlaubt.

Zudem lassen sich aus dem Domain Driven Design weitere Regeln ableiten, die für die Implementierung von CMDSD-Werkzeugen erfüllt sein sollten. Ausführlichere Erläuterungen und Hinweise finden sich wiederum in unseren Ausführungen in Abschnitt 3.2.7:

K-3.39 ▷ 93

1. *Verwende die universelle Sprache deiner Domäne in deinem Werkzeug durchgängig.* Formulierungen in Meldungen und Hilfetexten sollten ebenso konsequent wie Beschriftungen von Dialogen und Modellen daraufhin geprüft werden, dass die zugrunde liegende universelle Sprache der spezifischen Domäne durchgängig und möglichst widerspruchsfrei benutzt wird.
2. *Setze in dem generierten System eine schichtenbasierte Architektur mit klarer Trennung von Verantwortlichkeiten um.* Dies hilft bei der Abstraktion von technischen Sachverhalten und unterstützt die transparente technische Evolution, indem einzelne technische Schichten ausgetauscht, erweitert oder ergänzt werden könnten, ohne dass sich zwingend Modelle ändern müssen[40].
3. *Verwende die von Evans vorgeschlagenen Standardmuster sowohl für Objekttypen als auch für den Lebenszyklus der Objekte.*

5.4.3.8 Strukturierte Modellbildung

Da die Ermittlung der richtigen Modellgranularität ein sehr komplexes Thema ist, fassen wir in dieser Methode Erfahrungswerte zusammen, die sich hierbei besonders bewährt haben. Die erste und zentralste Regel lautet:

K-3.23 ▷ 73

> *Entwerfe Modelle in der Granularität und Schachtelungstiefe, die ausreicht, um mit Kunden ein Problem aus der Domäne besprechen zu können.*

Üblicherweise wird daraus eine Top-Down-Vorgehensweise in der Modellierung resultieren, um sich mit dem Kunden im Gespräch vom allgemeinen Anliegen zum spezifischen Problemdetail vortasten zu können. In manchen Domänen kann auch eine Bottom-Up-Vorgehensweise geeigneter sein, wenn sehr spezielle Probleme zu

[40] Ein gutes Beispiel ist die Ergänzung einer neuen Präsentationsschicht – gibt es ein sauberes Schichtenmodell, in dem die Benutzerschnittstelle über eine Übersetzungsschicht mit der Geschäftslogikschicht und der Domänenschicht kommuniziert, müssen dazu maximal die Präsentations- und Übersetzungsschicht ausgetauscht werden – alle anderen Schichten sind von den erforderlichen Änderungen nicht betroffen.

lösen und dann in einen allgemeineren Kontext zu bringen sind. Üblicherweise muss man sich nur einen typischen Gesprächsverlauf eines Anforderungsgesprächs kurz anschauen, um intuitive Klarheit über ein Top-Down- oder Bottom-Up-Vorgehen zu erhalten. Zu beachten ist zudem, dass eine prinzipiell vorteilhaftere Vorgehensweise die andere Alternative nicht ausschließt[41].

K-3.40 ▷ 99
K-3.50 ▷ 115

Die zweite und nächstwichtige Regel ist:

> *Stelle nur ein Hauptkonzept in einem Modell dar.*

Hier stellt sich natürlich die Frage, was ein Hauptkonzept ist. Dies hängt wieder sehr stark von der betrachteten Domäne ab – in einem Intranetsystem kann dies ein Prozess oder View sein, in einer auf einer Tabellenkalkulation basierenden Controlling-Anwendung entweder die Liquiditäts- oder GuV-Betrachtung, usw. Wichtig ist, Konzeptebenen nicht zu vermischen, da die Modelle dann zum einen schnell unübersichtlich werden und zum anderen die Modellierung selbst komplizierter wird. In der Methode „Dekomposition der Komplexität“ in Abschnitt 5.4.3.9 werden wir weitere Mittel vorstellen, mit der die Komplexität reduziert werden kann. Es schließt sich unmittelbar die folgende Regel an:

> *Stelle nur die wirklich wichtigen Informationen dar.*

Hier ist wieder die Idee, die Informationsdichte in den Modellen so gering wie möglich zu halten, um die Diskussion mit Anwendern im Rahmen des Conceptual Pair Programming (siehe Abschnitt 6.4.3.5) so einfach wie möglich zu gestalten[42].

[41] Greifen wir wieder unserem Beispiel von Intranetsystemen in Teil III vor, so könnte man durchaus eine Prozessmodellierung damit beginnen, zunächst den zentralen View zu besprechen, diesen dann in einen Prozess und zuletzt in die Navigation einzubetten, auch wenn der umgekehrte Weg natürlicher erscheint. Dies hängt auch immer vom Einzelkunden ab und hier ist das Prinzip der Kundenorientierung anzuwenden: Fühlt sich der Kunde wohler darin, zuerst das spezielle Problem zu besprechen (häufig, um es erst mal „vom Herzen zu haben“ und Gewissheit darüber zu erlangen, ob der jeweilige Dienstleister ihm überhaupt helfen kann), sollte man dies unbedingt tun. Die Diskussion wird sich dann schnell genug durch entstehende Fragen auf den Gesamtkontext führen lassen, um so zu der Wunschvorgehensweise zurückzugelangen. Hier ist der Wert Flexibilität nicht nur in den Werkzeugen sondern auch in den Werkzeugnutzern gefragt.

K-3.38 ▷ 91

[42] Wieder bietet sich ein Vorgriff auf unser Intranetbeispiel zur Illustration an: Dort werden wir Prozessmodelle (siehe Abschnitt 8.1.8.1.2) zur Beschreibung der Interaktion zwischen einzelnen Sichten definieren. Diese enthalten zwar auch Transitionen, aber die in den Transitionen enthaltenen Aktionen werden nicht weiter dargestellt – es gibt nicht mal eine Information darüber, ob

5.4.3.8.1 Modelltypen Um die Modellbildung zu vereinfachen und insbesondere das Auswählen der richtigen Modellgranularität zu erleichtern, wollen wir nachfolgend drei Arten von Modelltypen unterscheiden. Diese betrachten unterschiedliche Detaillierungsebenen und strukturieren den Modellbildungsprozess. Zwischen den Modelltypen werden wir verschiedene Abhängigkeitstypen identifizieren (z. B. *referenziert* und *enthält*, siehe Abschnitt 5.4.3.8.4), die wiederum Grenzen definieren, um Versionierungs- und Sperrmechanismen für Modelle realisieren zu können (siehe Abschnitt 8.1.3 für ein Beispiel aus unserem Prototypen). Insbesondere Letztere werden uns dabei helfen, einfache und doch für den mittelständischen Kontext ausreichend effektive Mechanismen zur nebenläufigen Bearbeitung von Modellen auszuprägen. Wir definieren nachfolgend primäre, sekundäre und tertiäre Modelltypen.

K-3.12 ▷ 64

K-3.28 ▷ 78

Primäre Modelle Primäre Modelltypen sind häufig leicht zu identifizieren, da sie zentrale Aspekte einer Domäne abbilden. So wird sich in fast allen CMDSD-Systemen eine Modellrepräsentation von spezifischen Domänenressourcen, mit der Domäne verbundenen Organisationsstrukturen sowie domänenspezifischer Prozesse finden. Wir definieren primäre Modelltypen als die Modelle, die für die Modellierung lauffähiger Systeme von zentraler Bedeutung sind, so dass ohne sie eine Modellierung gar nicht möglich wäre (siehe dazu als Beispiel für die Geschäftsprozessmodellierung [Tayl95], S. 109).

Zudem zeichnen sich primäre Modelle dadurch aus, dass sie größere Sinneinheiten bilden, auf denen typischerweise nur ein Modellierer gleichzeitig arbeiten kann[43]. Aufgrund der geringen Teamgröße in mittelständischen IT-Projekten erlauben wir ein exklusives Sperren dieser Modelltypen zur Bearbeitung. Es besteht zwar die Gefahr, dass Leerzeiten entstehen, weil ein Entwickler auf einen anderen warten muss. Wir halten dieses Problem im mittelständischen Kontext im Gegensatz zu größeren Projekten für vernachlässigbar, da unserer Erfahrung nach Aufgaben häufig so geschnitten sind, dass ein Entwickler alleinverantwortlich für einen gesamten Teilprozess ist.

K-3.28 ▷ 78

bestimmte Aktionen enthalten sind. Die einzige dargestellte Information ist der umgangssprachliche Name der Transition in der Annahme, dass ein Kunde hier einen für ihn verständlichen und prozessrelevanten Bezeichner angeben wird, der für das intuitive Verständnis des Gesamtmodells ausreicht. Werden weitere Details benötigt, kann die Transition durch Doppelklick im Editor expandiert werden und die Modellierer können die Aktionsliste und weitere Details analysieren. Natürlich muss man hier Vorsicht walten lassen, um wichtige Informationen nicht auf zu viele Teilmodelle zu verteilen.

[43] Dies könnte z. B. ein Prozess, eine Bildschirmmaske oder ein Datentyp sein.

Sekundäre Modelle Sekundäre Modelltypen sind durch eine hohe Wiederverwendbarkeit gekennzeichnet. Sie repräsentieren wichtige Konzepte der betrachteten Domäne, die Modellierung der Domäne wäre aber – wenn auch eventuell mit starken Einschränkungen und nur unter Ergänzung einzelfallbezogener Sondermechanismen – ohne einzelne sekundäre Modelltypen möglich. Wir wollen dies an folgendem Beispiel illustrieren, das sich auf unseren Prototypen in Kapitel 8 bezieht: Wir definieren Filtermodelle für Webapplikationen als einen logischen Ausdruck über einem Datensatz, den wir vielfach wiederverwenden können, beispielsweise um

- in einem Gridcontrol (also einem Tabellenelement in einem Formularmodell) anzuzeigende Zeilen zu filtern,
- in einem Portlet eine nutzerspezifische Sicht zu definieren,
- den Auslöser für ein Agentenmodell zu definieren,
- spezielle Rechte zu modellieren.

Wir könnten für alle diese Fälle speziellere Sondermechanismen definieren. Die Verwendung eines durchgängigen Filtermodells erlaubt uns aber eine einfachere Definition des Gesamtmodells, verringert den Einarbeitungsaufwand in das Modellsystem und steigert zudem die Wiederverwendung von Code bei der Realisierung des Systems deutlich. Insofern erfüllt das Filtermodell alle Eigenschaften eines sekundären Modelltyps.

Wie dieses Beispiel zeigt, sollt es ein generelles Ziel sein, eine kleine Zahl (möglichst wiederverwendbarer) sekundärer Modelltypen zu identifieren, da sekundäre Modelltypen letztlich den Implementierungsaufwand von CMDSD-Werkzeugen aufgrund ihrer Wiederverwendbarkeit drastisch reduzieren. Zudem wirkt sich eine kleine Zahl häufig verwendeter sekundärer Modelltypen positiv auf die Ergonomie des Systems aus, da Anwender immer wiederkehrende Muster antreffen und so die Bedienung leichter einzuprägen ist.

Die im Rahmen des Convergent Engineering beschriebenen fraktalen Muster (siehe [Tayl95], S. 43) sind in Modellen häufig ein Hinweis auf sekundäre Modelle, können aber auch bei primären Modellen (zum Beispiel bei *ist-zusammengesetzt-aus*-Beziehungen) auftreten. In jedem Fall sind fraktale Muster in Modellen Punkte, an denen besonders viel Energie in das ergonomische Design der Entwurfsschnittstelle des CMDSD-Werkzeugs gelegt werden sollte. Die fraktale Natur des Musters erlaubt sehr mächtige Aussagen über die zu modellierende Entität. Anwender sollten auf fraktalen Mustern basierende Modelle daher so einfach wie möglich entwerfen

können. Hier bietet sich insbesondere die enge Zusammenarbeit mit den Fachexperten und späteren Zielnutzern des Systems an, um eine möglichst natürliche und somit intuitive Darstellung für diese sekundären Modelle zu finden.

K-3.38 ▷ 91
K-3.39 ▷ 93

Tertiäre Modelle Tertiäre Modelle sind Meta-Modelle, die technische Eigenschaften fachlich abstrahiert beschreiben (meistens Checklisten wie z. B. „Zielsystem“ mit den Wahlmöglichkeiten „low-bandwidth personal website“, „SME company website“ oder „high load public portal“, was sich dann in den Generierungsvorschriften niederschlägt, um das darunterliegende technische System auszuprägen).

Betrachtet man zum Beispiel das Spring-Framework (siehe [Spri09a, BWHS08, BSSW09]) als CMDSD-Werkzeug zur Konfiguration von JEE-Anwendungen, so sind Spring Namespaces tertiäre Modelle und gleichzeitig Erweiterungspunkte des Systems (siehe dazu auch unsere Ausführungen in Abschnitt 5.4.3.5.1 auf Seite 207 sowie die Definition auf Seite 220).

Ein weiteres gutes Beispiel für tertiäre Modelle sind Modelle, die für die Integration bestehender Altsysteme mit dem CMDSD-System erforderlich sind. Hier empfiehlt bereits das *Convergent Engineeering*, den Zugriff auf diese Systeme so weit wie möglich zu verbergen, um zu einem späteren Zeitpunkt die Altsysteme für den Anwender transparent zu ersetzen (siehe [Tayl95], S. 97–98). Für derartige Integrationsfragen sollte das CMDSD-System daher domänenspezifische Integrationsmodelle vorsehen, die eine Anbindung domänenrelevanter Altsysteme so stark wie möglich vereinfacht.

5.4.3.8.2 Modellgruppen Unter einer Modellgruppe verstehen wir eine in sich geschlossene, vollständige Menge von Modellen, die einen zusammenhängenden Aspekt der Domäne (ein Modul, einen Archetypen, einen Basisprozess u. ä.) beschreiben.

Primäre, sekundäre und tertiäre Modelle sollten gruppierbar sein, da sich durch diese Eigenschaft (die häufig relativ einfach realisierbar ist) wie nachfolgend beschrieben deutliche Einsparungseffekte hinsichtlich Entwicklungszeit und -kosten realisieren lassen. Eine Gruppe solcher Modelle sollte mit dem CMDSD-Werkzeug in einem geschlossenen Standardformat exportierbar und mit jedem anderen CMDSD-Werkzeug der gleichen Version importierbar sein. Zudem sollte das zugrunde liegende Standardformat ein offenes, nach Möglichkeit text- (und häufig XML-)basiertes Format sein, um auch anderen Anbietern die Nutzung des Formats zu ermöglichen. Auf diese Weise wird die Distribution von bewährten Modellen erleichtert: Eine sinnvolle Gruppierungsklassifikation ist dazu „die Anwendung“, also eine geschlossene Einheit

von Modellen, die eine vollständige Anwendung bzw. einen Funktionsbaustein beschreiben.

Zudem liegt der Gedanke nahe, (ggf. modifizierte) Archetypen als Modellgruppe importieren und exportieren zu können, um so das initiale Aufsetzen neuer Produktinstanzen zu beschleunigen. Wir streben also eine Wiederverwendung auf Modellebene an, indem wir eine Modellgruppe als wiederverwendbare Komponente betrachten.

K-3.11 ▷ 63

In jedem Fall müssen in einer Modellgruppe alle referenzierten und inkludierten Modelle enthalten sein. Wird eine Modellgruppe mit referenzierten Modellen importiert, ist domänenspezifisch zu entscheiden, wie innerhalb des Werkzeugs damit umzugehen ist:

- Möglicherweise wird eine vollständige Modellkopie angelegt.
- Es kann aber auch Sinn machen, nach einer Prüfung auf Modellgleichheit oder -äquivalenz ein bereits existierendes Modell zu referenzieren.
- Unter Umständen muss diese Entscheidung gar dem Anwender überlassen werden.

5.4.3.8.3 Modellfacetten Wir definieren Facetten als Modellparameter. Modelle können parametrisiert werden, so dass komplexe Subfunktionalitäten durch das einfache Setzen von Parametern gesteuert werden können.

K-3.3 ▷ 55

Ein Beispiel aus unserem in Kapitel 8 beschriebenen Prototypen ist das Transitionsmodell, das eine Aktion(smenge) beschreibt, die bei einem Zustandswechsel ausgeführt wird. Eine Facette könnte eine „Ausführungsbeschränkung“ sein. Diese wird wiederum durch das zuvor bereits kurz vorgestellte sekundäre Filtermodell realisiert mit der Semantik, dass die Aktion nur ausgeführt werden kann, wenn der Filter auf den aktuellen Daten den Wert „wahr“ zurückliefert. Analog könnte ein Eingabefeld die Facette „Sichtbare Eingabe“ mit den Werten „ja“ und „nein“ haben. Ist der Wert auf „nein“ gesetzt, wird so eine Kennworteingabe simuliert, bei der die eingebenen Zeichen nicht zu lesen sind.

Für schwerwiegendere Änderungen im Verhalten eines Modells greifen wir auf ein bewährtes Design Pattern in Form des *Strategy Patterns* (siehe [GHJV96]) zurück: Jedwedes Verhalten einer Modellimplementierung, bei dem eine problemabhängige Variantenbildung zu erwarten ist, wird durch Definition einer geeigneten Schnittstelle in der Implementierung beschrieben. Das Standardverhalten wird über mit einem Defaultwert vorbelegte Modellfacette für jede Instanz des Modellelements definiert.

Soll sich das grundlegende Verhalten des Modellelements ändern, kann der Modellierer durch Umdefinition der Modellfacette über das so umgesetzte Strategiemuster das Verhalten innerhalb der Grenzen des Modells austauschen[44]. Dies entspricht dem Prinzip der Delegation aus dem Convergent Engineering (siehe [Tayl95], S. 94), das für komplexere Verhaltensmodifikationen über eher elementare Zustandswechsel hinaus definiert wird. Um diese Form der Erweiterbarkeit möglichst optimal zu nutzen, ist das *Single Point of Responsibility*-Prinzip (siehe [Mart03]) anzuwenden. Es besagt, dass jede Klasse (in unserer Betrachtung: jeder Erweiterungspunkt) nur genau einen Grund haben darf, sich zu verändern. Es geht also darum, eine möglichst klare Verantwortlichkeit zu definieren, was die Nutzung von Erweiterungspunkten durch Dritte stark vereinfacht. Das bereits zitierte Spring-Framework macht von diesen Prinzip sehr konsequent Gebrauch und erzielt immer wieder sehr gute Bewertungen bezüglich der Klarheit seiner Architektur und seiner Verständlichkeit (siehe z. B. [Dona06]).

Der Austausch des Modellverhaltens erfordert also eine manuelle Implementierung des Spezialfalls. Dies lässt sich nach unserer Auffassung aber nicht vermeiden, da Programmiersprachen eine Möglichkeit zur Formulierung turing-vollständiger Fragestellungen sind. Um den Wert der Einfachheit zu wahren erlauben wir in CMDSD, entsprechende Erweiterungspunkte in allen Modellen vorzusehen und sie manuell in einer geeigneten Programmiersprache zu implementieren. Die Umsetzung durch Programmierung erscheint uns erheblich sinnvoller als eine weitere formale Beschreibungssprache einzuführen, die dann in eine spezifische Implementierung einer konkreten Programmiersprache übersetzt wird. Letzteres Vorgehen hat zwar Vorteile durch den universelleren und generischeren Ansatz, der auch den Wechsel der darunter liegenden Programmierplattform ermöglicht. Unserer Erfahrung nach findet ein Wechsel in einer zugrunde liegenden Programmierplattform üblicherweise aber nur dann statt, wenn es dafür eine umfassende technische Notwendigkeit gibt. Eine solche Notwendigkeit impliziert meist aber auch viele weitere Änderungen, so dass es fast immer Sinn macht, spezifische Funktionen in einer solchen Situation auch neu zu implementieren. Zudem erscheint es uns nicht schwieriger, eine Transformation von einer Programmiersprache auf eine andere Programmiersprache anzugeben als eine Transformation von einer formalen Darstellung auf eine konkrete Programmiersprache. Wir sehen sogar eine Reihe von Vorteilen in unserer Vorgehensweise:

- Die direkte Definition spezieller Funktionalitäten in einer Programmiersprache sind ein Vorteil für Programmierer, da diese ihr vertrautes Umfeld nicht

[44] Die Modellgrenzen ergeben sich durch die Definition einer festen Schnittstelle.

verlassen müssen, sondern weiter mit gewohnten Werkzeugen arbeiten können. Dies setzt natürlich voraus, dass das CMDSD-Laufzeitsystem in einer standardisierten und verbreiteten Programmiersprache implementiert wurde.

- Die Verwendung einer existierenden Programmiersprache für die Implementierung spezifischer Funktionen und Erweiterungen vereinfacht auch die Realisierung des eigentlichen CMDSD-Systems, da weder eine neue formale Darstellung für die auszudrückenden Funktionen entwickelt noch verifiziert werden muss.
- Die Formulierung von Erweiterungen in der gleichen Programmiersprache wie das CMDSD-Laufzeitsystem vereinfacht die Integration der Erweiterungskomponenten: Diese können statisch zur Erstellungszeit oder dynamisch zur Laufzeit in das System integriert und durch direkten Aufruf angesprochen werden. Es entfällt eine komplizierte Übersetzung der Aufrufe aus verschiedenen Laufzeitumgebungen.

Wir formulieren das Konzept von Erweiterungspunkten für CMDSD-Modelle unter Bezugnahme auf unsere Ausführungen zu Modellfacetten (siehe Abschnitt 5.4.3.8.3 auf Seite 218) wie folgt:

- *CMDSD-Modelle kapseln Variations- und Erweiterungspunkte der Modellimplementierung als sogenannte* Modellfacetten *mittels des Strategiemusters.*
- *Die konkrete Strategie wird in Form einer Modellfacette für das jeweilige Modellelement bzw. Modell festgelegt und kann durch den Modellierer verändert werden.*
- *Erweiterungen sind nach Möglichkeit in der gleichen oder einer eng verwandten Programmiersprache zu formulieren, wie die, in der die Implementierung der CMDSD-Laufzeitplattform vorgenommen wurde.*
- *Generell ist eine möglichst weit verbreitete und standardisierte Programmiersprache zu wählen.*

Modellfacetten sind eine modellbasierte Umsetzung des *Switching*-Konzepts des Convergent Engineering (siehe [Tayl95], S. 93). Dabei werden Variationspunkte in Klas-

sen statt durch Vererbung und Subklassen über Zustandsfelder in Klassen ausgedrückt. Dies ist immer dann sinnvoll, wenn eigentlich ein spezieller Zustand ausgedrückt werden soll, der sich aber auch ändern kann. Für komplexere Fälle wird dort das Prinzip der Delegation vorgestellt (siehe [Tayl95], S. 94).

Bei der Gestaltung des CMDSD-Modellierungswerkzeugs sind Modellfacetten mit besonderer Aufmerksamkeit zu betrachten:

- In Modellelementen werden sich häufig eine Vielzahl von Erweiterungspunkten anbieten. Hier drohen mehrere Gefahren, die aber einfach zu umgehen sind:
  1. **Problem:** Das CMDSD-Modellierungswerkzeug wird sehr aufwändig und teuer in der Entwicklung, da eine sehr große Zahl von Erweiterungspunkten zu berücksichtigen ist. Das Prinzip der Wirtschaftlichkeit würde also in der Entwicklung verletzt werden.

     Lösung: Alle denkbaren Erweiterungspunkte werden in der Analysephase auf Story Cards gesammelt. Den Story Cards wird ein Aufwand, ein Risiko und ein Geschäftswert zugewiesen. Unter Berücksichtigung des Zeitplans für die Erstellung der initialen Version des CMDSD-Werkzeugs werden aus der Menge Story Cards mit dem höchsten Geschäftswert ausgewählt und realisiert. Bei Story Cards mit gleich hohem Geschäftswert werden die Karten bevorzugt, die einen günstigeren Quotienten aus Geschäftswert und Kosten besitzen. Auf diese Weise werden alle Facetten bis zum Ablauf der zur Verfügung stehenden Zeitspanne umgesetzt. So wird unter Berücksichtigung der veranschlagten Kosten der maximal mögliche Nutzwert in Form von CMDSD-Funktionen realisiert. Verbleibende Story Cards werden durch Anregungen im Produktlebenszyklus und aus Projekten heraus um neue Story Cards ergänzt, die analog bewertet werden. Für den Releaseplan des nächsten CMDSD-Werkzeug-Meilensteins werden dann wieder Story Cards gemäß dem oben beschriebenen Vorgehen selektiert und realisiert. Dieses Vorgehen wiederholt sich während des gesamten Produktlebenszyklus.

     Alternativ können auch Ansätze zur Feature-Modellierung aus dem Bereich der Software-Produktlinien (siehe Abschnitt 3.2.6) verwendet werden.
  2. **Problem:** Die Bedienoberfläche wird sehr komplex, da eine Vielzahl von Erweiterungspunkten in der Darstellung für den Benutzer zu berücksichtigen sind. Der Wert Zugänglichkeit würde also nicht genügend beachtet.

     Lösung: Hier bietet sich eine Zweiteilung der verfügbaren Facetten an:

a) Zu jedem Modellelement wird in Form eines Wizards ein Editor bereitgestellt, der die Parametrisierung des Modellelements ermöglicht.
b) Editoren werden z. B. über Registerkarten in Themen unterschiedlicher Komplexität aufgeteilt. Die Einstellungen, die immer (bzw. am Häufigsten) vorgenommen werden, finden sich auf der vordersten Registerkarte.
c) Auf den Registerkarten werden nur die Facetten berücksichtigt, für die eine intuitive grafische Pflege oder Formulareingabe entwickelt werden kann. Alle anderen Modellfacetten werden auf einer speziellen Registerkarte für Experten untergebracht. Diese Registerkarte ist immer die letzte aller Registerkarten. Je nach Komplexität des Gesamtwerkzeugs kann es sich sogar anbieten, die Registerkarte erst durch explizite Konfiguration des Werkzeugs sichtbar zu machen.
d) Modellfacetten mit Expertencharakter dürfen durchaus in einer expertengerechten Darstellung angeboten werden. Häufig werden dazu in modernen Entwicklungsumgebungen sogenannte „Property Editoren“ verwendet, die alle Einstellmöglichkeiten in Form von (Name, Wert)-Paaren in einer geordneten Liste anbieten und die Eingabe durch intelligente Eingabefelder unterstützen (z. B. Auswahlmöglichkeiten für Farben).

Auf diese Weise erreichen wir, dass alle Modellvariationsmöglichkeiten nach Nützlichkeit und Komplexität sortiert zugreifbar sind. Unerfahrene Modellierer werden zu Beginn mit den Optionen auf der vordersten Registerkarte auskommen und sich anschließend mit zunehmender Erfahrung mit dem Werkzeug „nach hinten“ vorarbeiten können. Die Expertenoptionen, für die es keine einfache Repräsentation gibt, sind die letzte Stufe der Konfiguration, die (eventuell) sogar explizit aktiviert werden muss. Auf diese Weise wird erreicht, dass die Anwender des Systems nicht durch übermäßig viele Informationen und Zusammenhänge überfordert werden.

3. **Problem:** Die Modelldarstellung selbst wird sehr komplex, da die vielen Erweiterungspunkte grafisch ausgeprägt werden müssen. Auch hier wird der Wert Zugänglichkeit nicht ausreichend beachtet.

   Lösung:

   - Zeige nicht die vielfältigen Einstellungen eines Modells sondern nur seine kurze umgangssprachliche Beschreibung (die eine Pflichtangabe sein muss) an. Details sind dann auf verschachtelten Dialog-Ebenen zu finden und können bei Bedarf aufgerufen werden.

- Zeige nur die essentiellen Informationen (z. B. Pflichtfelder in einem Maskenmodell) an.
- Suche das richtige Modell für eine Facette[45].

5.4.3.8.4 Kompositionsprinzipien für Modelle Modelle können auf folgende Weise aus anderen Modellen komponiert werden:

- Ein Modell M_i kann andere Modelle M_j *enthalten* (Inklusion). Inkludierte Modelle sind dadurch charakterisiert, dass sie nur im Kontext des inkludierenden Modells eine Bedeutung haben.
- Ein Modell M_i kann andere Modelle M_j *referenzieren* (Referenz). Referenzierte Modelle können im Gegensatz zu inkludierten Modellen auch alleine für sich stehen.

Diese beiden Prinzipien spiegeln sich auch in den Definitionen primärer und sekundärer Modelltypen in Abschnitt 5.4.3.8.1 wider: Sekundäre Modelltypen werden per Definition immer von primären oder sekundären Modelltypen inkludiert. Primäre Modelltypen können nur andere primäre Modelltypen referenzieren.

Unabhängig von der Inklusions- und Referenzbeziehung ist aus Designsicht zu beachten, dass bevorzugt eine Topdown-Sicht auf Modellierungsschichten gewählt werden sollte. Diese Sichtweise hat sich zwecks Reduktion der Gesamtkomplexität als äußerst effektiv erwiesen (siehe z. B. [Tayl95], S. 18) und wird durch Ergebnisse der experimentellen Psychologie gestützt (siehe unsere Ausführungen auf Seite 100 sowie [Tayl95], S. 25ff.): Menschen können im Schnitt sieben bis neun unzusammenhängende Konzepte miteinander in Verbindung bringen und gleichzeitig verstehen. Diese Erkenntnis bedingt zwangsweise, dass Modelle im Gegensatz zu vielen gegenwärtigen Ansätzen (man beachte die Menge an Diagrammtypen, -elementen und Spezialisierungen z. B. in der UML, siehe auch Abschnitt 3.3.1) so einfach wie möglich gehalten werden müssen, um sie allgemein verständlich zu machen. Insbesondere erkennen wir, dass dies dann von großer Bedeutung ist, wenn eine Brücke zwischen komplexen Domänen (z. B. Fach- und IT-Domäne) geschlagen werden soll. Um trotz einer nur begrenzten Zahl sinnvoller Konzepte in einer Sicht komplexe Systeme modellieren zu können, hat sich *Chunking* (Gruppierung) als Technik bewährt ([Tayl95],

K-3.21 ▷ 70

[45] Hier bietet sich wieder das Pflichtfeld-Beispiel an: Ist die Eingabe eines bestimmten Feldes nur auf einer spezifischen Maske zwingend erforderlich, wird dies eine Facette an dem Eingabeelement sein. Muss der Wert dagegen auf allen Masken, auf denen er eingegeben werden kann, als Pflichtfeld behandelt werden, sollte diese Eigenschaft nicht auf den einzelnen Eingabeelementen definiert werden sondern direkt an der entsprechenden Eigenschaft des zugrundeliegenden Domänentyps.

S. 26–28): Wenn Menschen nur mit einer begrenzten Zahl von Konzepten zur gleichen Zeit arbeiten können, werden diese zu Bausteinen kombiniert (chunking), die dann wiederum zur Beschreibung komplexerer Systeme verwendet werden können. Damit das funktioniert, müssen zwei Grundvoraussetzungen erfüllt sein:

1. Gruppierte Elemente müssen gemeinsam ein sinnvolles Muster ergeben, das wiederum als Ganzes begriffen werden kann. Für Modelle bedeutet dies, dass sie nur eng miteinander verbundene Konzepte modellieren sollten zu denen es nach Möglichkeit direkt fassbare Entsprechungen in der Problemdomäne gibt. Modelle entsprechen also den Gruppen (chunks).
2. Elemente einer Gruppe dürfen nicht direkt mit Elementen einer anderen Gruppe interagieren. Aus Programmiersicht sind die Elemente von Gruppen gekapselt. Für Modelle bedeutet dies, dass die Abgrenzung von primären zu sekundären Modelltypen konsequent eingehalten werden muss.

Im Zusammenhang mit domänenspezifischer Modellierung erweist es sich als vorteilhaft, dass „Chunking“ ein Prozess ist, der intuitiv in der realen Welt und damit auch in den meisten Problemdomänen bereits angewandt wird: So strukturieren Software-Ingenieure ihre Programme in Modulen, Konzerne organisieren sich in Teilkonzernen, Geschäftseinheiten, Divisionen, Abteilungen, usw. Zudem gibt es Konzepte wie die „Befehlskette“ oder „Linie“, die eine natürliche Topdown-Beziehung zwischen den Elementen implizieren und so sicherstellen, das es keine ungewollte Interaktion zwischen Elementen auf verschiedenen Ebenen gibt.

Wir haben somit gezeigt, dass Inklusion und Referenz von zentraler Bedeutung für die Beziehung zwischen Modellen sind. Zudem haben wir gesehen, dass sie basierend auf diesen Prinzipien in einer Topdown-Sicht strukturiert werden müssen. Dies hält die Komplexität von Modellen beherrschbar.

5.4.3.9 Dekomposition von Komplexität

K-3.41 ▷ 101

> *Nimm nur die notwendige Menge an Details für den wesentlichen Punkt eines Modells auf.*

Wird ein Metamodell so komplex, dass es an Zugänglichkeit verliert (siehe dazu die Ausführungen in Abschnitt 4.2.1.3), so sind Dekompositionsgrenzen zu suchen, um

Details in Untermodelle zu verschieben. Dadurch kann die Anzahl an Modellelementen überschaubar gehalten und Komplexität verborgen werden. Wichtig ist, einen geeigneten Hinweis auf die verborgene Komplexität zu geben, so dass Modellierer nicht überrascht werden.

K-3.23 ▷ 73

Beispielsweise haben wir sehr gute Erfahrungen damit gemacht, kurze umgangssprachliche Hinweise in Modellen anzuzeigen, hinter denen sich dann (durchaus komplexe) Untermodelle verbergen. Abschnitt 8.1.8.2.3 enthält ein Beispiel für ein sogenanntes Filtermodell, dass in unserem Referenzbeispiel an allen Stellen wiederverwendet wird, an denen ein logischer Ausdruck (sei es zur Entscheidungsfindung oder zur tatsächlichen Filterung von Daten) benötigt wird. Ein Anwendungsfeld sind zum Beispiel Entscheidungszustände in unseren exemplarischen Prozessmodellen (siehe die ausführlichen Erläuterungen in Abschnitt 8.1.8.1.2). Mit den Bedingungen an einem Entscheidungszustand könnte prinzipiell auf zwei Weisen umgegangen werden:

K-3.4 ▷ 56

1. Das Modell zeigt den Booleschen Ausdruck, der zum Auslösen der entsprechenden Transition führt.
2. Das Modell zeigt eine abstrakte Repräsentation des Booleschen Ausdrucks.

Wir raten immer zur zweiten Option, da Boolesche Ausdrücke wiederum verschiedene Operatoren enthalten und zudem stark verschachtelt auftreten können, was zu einer deutlichen Verringerung der Zugänglichkeit eines Prozessmodells (aufgrund einer Überfrachtung des Betrachters mit Details) führen würde. Eine geeignete Abstraktion ist zum Beispiel ein vom Modellierer eingegebener umgangssprachlicher Text (beschränkter Länge – dies fördert die Formulierung prägnanter Merksätze), der ggf. mit dem Präfix „*wenn:*“ an der dem Filtermodell zugeordneten Transition ausgegeben wird. Auf diese Weise wird ein dem Modellierer geläufiges Darstellungsmerkmal (die „Beschriftung“ eines Elements) wiederverwendet, um zusätzliche Informationen auszugeben, ohne Details anzuzeigen. Operativ kann der Anwender bei Klick auf diese Beschriftung in einen Untermodelleditor geführt werden, in dem das zugeordnete Filtermodell in seiner vollen Kompelexität dargestellt wird. Es ist offensichtlich, dass beispielsweise die Beschriftung „Dokument ist abgenommen“ wesentlich prägnanter (und für Domänenexperten vermutlich sogar unmittelbar klarer) ist als ein logischer Ausdruck, der etwas formuliert wie „(alle Verantwortlichkeiten haben das Dokument abgenommen) und (es befindet sich in einem der Zustände *fehlerfrei*, *kosmetische Mängel* oder *als Case Study empfohlen*) und (es gibt eine digitale Signatur des Autors)“.

In den vorausgegangenen Ausführungen haben wir wiederholt die Bedeutung ein-

facher und intuitiver Modelle betont, die dennoch hinreichend aussagekräftig sein müssen, um sie automatisiert in ausführbare und lauffähige Systeme zu übersetzen. Nun wollen wir uns der Herausforderung stellen, derartige Modellsysteme zu entwerfen und als Grundlage für domänenspezifische CMDSD-Systeme zu beschreiben. Da CMDSD zunächst eine domänenunabhängige Modellierungs- und Entwicklungsmethode ist, geben wir einen domänenunabhängiges, allgemeines Vorgehensmodell an, das es ermöglicht, CMDSD-taugliche Fachmodelle zu entwerfen. In Kapitel 8 werden wir die Arbeitsweise des Vorgehensmodells für eine spezifische Domäne vorführen und anschließend die daraus resultierende Implementierung beschreiben, um die prinzipielle Funktionsfähigkeit unseres Ansatzes zu verifizieren.

Zu berücksichtigen ist dabei, dass das grundlegende Vorgehensmodell annimmt, dass sowohl die Anwender des Vorgehensmodells als auch die betrachtete Domäne eine Reihe von Voraussetzungen erfüllen:

1. Die Anwender verfügen über umfassende Erfahrungen mit konkreten Projekten innerhalb der spezifischen Domäne.
2. Die Anwender haben die Werte, Prinzipien und Methoden aus den Abschnitten 5.4.1, 5.4.2 und 5.4.3 verinnerlicht und sind bereit, diese konsequent anzuwenden.
3. Die Domäne ist stabil, d. h. es gibt etablierte und in praktischen Projekten bereits intensiv genutzte und bewährte Standards für Technologien, Algorithmen, Protokolle, etc., die zur Modellierung der Domäne essentiell sind.
4. Es gibt bereits etablierte Implementationen der zuvor beschriebenen stabilen Grundlagen, bevorzugt in Form lizenzrechtlich unbedenklicher Open-Source-Bibliotheken. Dieser Punkt ist aus Kostengründen wichtig – muss ein Dienstleister erst noch umfassende Basistechnologien implementieren, wird der Aufwand für die Entwicklung von CMDSD-Lösungen üblicherweise die Ressourcen mittelständischer Dienstleister weit überschreiten[46].

Die nachfolgend beschriebene Reduktionsmethode ist darauf ausgerichtet, ein hinsichtlich Zugänglichkeit und Abstraktion optimiertes Fachmodell zu erzeugen.

K-3.13 ▷ 64

[46] Interessant ist hier zum Beispiel die in Kapitel 8 betrachtete Domäne „Portallösungen“: Bis ca. Mitte 2005 waren auf der gewählten Java-Technologieplattform nur unzureichende Implementierungen wesentlicher Technologien verfügbar und erst ab Mitte 2005 haben sich mit rasanter Geschwindigkeit eine Vielzahl essentieller Open-Source-Frameworks so weit stabilisiert, dass sie zu einem leistungsfähigen Gesamtsystem kombiniert werden können. Darunter finden sich Frameworks wie Spring [Spri09a, BWHS08, BSSW09], XStream [Code09a] u. v. a. m.

1. *Prüfe, ob es für die betrachtete Domänensicht bereits eine domänenspezifische Form der Beschreibung (vorzugsweise grafisch / schematisch) gibt.*

2. *Existieren mehrere Darstellungsformen, wähle diejenige, die am Weitesten verbreitet ist. Halte die anderen Modellierungsformen aber bereit, um gegebenenfalls Modellelemente aus diesen zu adaptieren.*

3. *Untersuche die Zahl der Aspekte in dem ausgewählten Modell. Wird mehr als ein Aspekt abgebildet, sollte das Modell auf einen Aspekt vereinfacht werden.*

    a) *Prüfe, welcher Hauptaspekt in dem Modell dargestellt wird (z. B. könnten dies Zustände mit ihren Übergängen in einem Aktivitätsdiagramm sein). Dieser Hauptaspekt ist zu erhalten.*

    b) *Prüfe, ob es Teilaspekte gibt, die im Sinne einer Top-Down-Modellierung in andere Modelle ausgelagert werden können. Ist das möglich, sollten die Teilaspekte aus dem aktuellen Modell herausgenommen werden (z. B. könnte das die Beschreibung der Zustandsausprägungen in einem Aktivitätsdiagramm sein).*

    c) *Erhalte nur die dominanten Modellelemente (im Beispiel sind dies „Zustand“ und „Transition“).*

    d) *Entwerfe Untermodelle für die entfernten Teilaspekte (häufig sekundäre oder tertiäre Modelle, siehe Abschnitt 5.4.3.8.1).*

4. *Prüfe die verbleibenden Wahlmöglichkeiten im reduzierten Modell:*

    a) *Gibt es technische Alternativen, die in dem Modell dargestellt werden, so vereinfache diese durch die Einführung zusätzlicher Infrastrukturmodelle (tertiäre Modelle, siehe Abschnitt 5.4.3.8.1) oder durch die Vorgabe sinnvoller Defaultwerte, die gegebenenfalls überschrieben werden können.*

    b) *Gibt es fachliche Alternativen, die in dem Modell dargestellt werden können, vereinfache diese durch sekundäre Zusatzmodelle (siehe Abschnitt 5.4.3.8.1) oder durch die Vorgabe sinnvoller Defaultwerte, die gegebenenfalls überschrieben werden können.*

5. *Nimm alle Formen der Detailmodellierung aus der grafischen Darstellung heraus und überführe diese in Modellfacetten (siehe Abschnitt 5.4.3.8.3).*

5.4.3.10 Werkzeuginhärente Archetypen

> *Stelle eine hinreichende Menge von Archetypen bereit, um die Ausprägung von Lösungsinstanzen zu vereinfachen.*

Die im Rahmen der Archetyisierung definierten Archetypen (siehe Abschnitt 5.4.3.5) werden sowohl als „Proof of Concept“ als auch zur Ausprägung von Lösungsinstanzen mit dem CMDSD-Werkzeug umgesetzt. Sie bilden nachfolgend einen Bestandteil der Software-Produktlinie. Somit stehen von vorne herein eine Reihe werkzeuginhärenter Archetypen zur Verfügung, die für einen schnellen Projektkickoff verwendet werden können. Werkzeuginhärente Archetypen ergeben sich dabei als Modellgruppen primärer und sekundärer Modelltypen, die in Summe jeweils einen Archetypen beschreiben. Es darf optional tertiäre Modelltypen als Ergänzung zu den Archetypen geben, wenn dies die allgemeine Flexibilität des Systems erhöht.

5.4.3.11 Intention statt Imperativ

> *Modelliere das Ziel. Vermeide die Modellierung des Weges.*

Hier greifen wir auf zentrale Ideen der Intentionalen Programmierung zurück (siehe Abschnitt 3.1.2): Um eine wesentliche Leistungssteigerung in der Entwicklung zu erreichen, müssen neue Wege gegangen werden. Ein Ansatz dazu ist, Intentionen zu formulieren und diese durch auf Metaprogrammierung beruhende domänenspezifische Bibliotheken interpretieren und ausführen zu lassen. Im Gegensatz zu Intentionaler Programmierung wollen wir mit CMDSD-Werkzeugen keine vollständige IP-Plattform schaffen, sondern zielen nur darauf ab, für eine spezifische und möglichst eingeschränkte Domäne eine möglichst leistungsfähige intentionale Unterstützung zu bieten. Abschnitt 3.1.2 stellt das grundlegende Vorgehen überblicksartig vor und verweist auf weiterführende Literatur. Czarnecki und Eisenecker liefern in [CzEi00] mit ihren Ausführungen zu generativer Programmierung (und insbesondere auch in Kapitel 11 zu Intentional Programming) vielfältige Methoden und Vorgehensmodelle, um dieses Ziel zu erreichen.

Für CMDSD-Werkzeuge ergeben sich wiederum mehrere Auswirkungen: Bei allen Aktionen, die mit dem CMDSD-Werkzeug ausgeführt werden können, sollte das

Werkzeug eine maximale Menge von sinnvollen Standardeinstellungen selbständig herleiten. Zudem sollten die Modelle, die unser CMDSD-Werkzeug bereitstellt, möglichst intentional (also auf einer hohen Abstraktionsebene) gestaltet werden (siehe dazu auch unsere Ausführungen in Abschnitt 5.4.2.6). Wir verdeutlichen diese Vorgehensweise mit einem Beispiel:

K-3.23 ▷ 73

Beispiel. *Ein Anwender modelliert ein Eingabeformular zur Speicherung von Steuerdaten mit der UML.*

- *Ausgangspunkt ist ein Use Case Diagramm, das den speziellen Anwendungsfall skizziert.*
- *Der Anwender würde dann ein Klassendiagramm entwerfen, in denen er die Geschäftsobjekte aus der Maske über ihre Felder und deren Datentypen beschreibt.*
- *Zudem würde er zum Beispiel ein Aktivitätsdiagramm entwerfen, das beschreibt, welche Aktionen aus dem aktuellen Zustand heraus möglich sind.*
- *Eine Aktion könnte zum Beispiel das „Speichern“ der Daten sein. Bei einem plattformspezifischen Modell würde er zudem eventuell typisierte Parameter für die Aktionen definieren müssen.*

Im Rahmen der Codegenerierung würde aus diesem Modell eine Reihe von Klassen in einer objektorientierten Programmiersprache, eine Präsentationsschicht (zum Beispiel Webseiten) und eine nach einem Standardschema aufgebaute Bildschirmmaske erzeugt werden.

Bei Verwendung eines CMDSD-System wird eine nutzerzentrierte Sicht der Modellierung eingenommen.

K-3.26 ▷ 74

- *Der Anwender beschreibt die Existenz des Anwendungsfall (zum Beispiel durch ein Use Case Diagramm).*
- *Der Anwender modelliert die Abfolge von Seiten, die er durchlaufen muss, um den Use Case zu behandeln. Dazu zeichnet er lediglich die Seiten und die sie verbindenden Kanten mit den Aktionslisten (eine durch das Werkzeug angebotene Aktion könnte „Speichern“ sein).*
- *Der Anwender malt die Bildschirmmaske mit den Feldern, die ein Nutzer des Systems ausfüllen muss. Dabei gibt es fachliche Datentypen an („Währung“, „Datum“, etc.).*

K-3.44 ▷ 107

- *Das System generiert daraus ein vollständig lauffähiges System:*

K-3.32 ▷ 83

- *Die physikalischen Repräsentationen der fachlichen Datentypen kann das System automatisch aus der zugrunde liegenden Laufzeitplattform ermitteln.*
- *Die genaue Signatur der „Speichern"-Methode kann das System automatisch aus den Feldern der Eingabemaske ableiten. Insbesondere kann es automatisch den vollständigen Datensatz speichern.*
- *Die interne Datenstruktur zur Speicherung der in der Maske modellierten Daten kann das System ebenfalls automatisiert aus der Maske und den logischen Datentypen ableiten. Es ist keine explizite Datenmodellierung erforderlich.*

Intentionale Modellierungssystematiken sind eine wesentliche Methode, um die Zugänglichkeit von Modellen zu erhöhen – durch intentional klare Strukturen wird die Notwendigkeit verringert, viele Details in Modelle aufnehmen zu müssen, die kognitiv leichter zugänglich werden (siehe dazu die Ausführungen in Abschnitt 4.2.1.3).

Zudem bestärkt diese Methode noch einmal ein wichtiges Anliegen des Domain Driven Design (siehe [Evan04a], S. 222): *„Right up front, let's agree that we do* not *want to make procedures a prominent aspect of our model. Objects are meant to encapsulate the procedures and let us think about their goals or intentions instead."*

5.4.3.12 Kapselung von Technologien

> *Treffe im fachlichen Modell keine technischen Aussagen.*

Software-Architekturen sind für die überwiegende Mehrheit der gängigen Fachsysteme lange bekannt und erfolgreich erprobt. Daher sollten fachliche Modelle vollständig von den zugrunde liegende technischen Architekturen abstrahieren – Fachanwender wissen damit nichts anzufangen. Zudem wird der fachmodellgetriebene Generator basierend auf einem zugrunde liegenden Regelsystem die technisch sinnvolle Architektur automatisch auswählen, so dass es nicht nötig ist, sie explizit zu modellieren.

K-3.4 ▷ 56
K-3.32 ▷ 83
K-3.16 ▷ 65

Sollte es aus Anwendersicht tatsächlich sinnvolle technische Alternativen geben, sollten diese entweder in einem Meta-Modell in Form checklisten-artiger sonstiger Einstellungen modelliert werden (siehe auch Abschnitt 5.4.3.8.1 zu tertiären Modellen) oder aber als Expertenoptionen angeboten werden (siehe Abschnitt 5.4.3.15)[47].

[47] Ein typisches Beispiel für eine notwendige technische Modellierung könnte z. B. die Festlegung des zu verwendenden Datenbanksystems und Servers sein.

Dies entspricht dem Vorgehen und den Forderungen des Domain Driven Design (siehe Abschnitt 3.2.7 bzw. [Evan04a], S. 41), technologische Aspekte einer Modellierung so minimal wie möglich zu halten: *„The technical model that drives the software development process must be strictly paired down to the necessary minimum to fulfill its functions.“* Entsprechend finden sich hier auch Hinweise zur Kapselung von Techologien in Form von Mustern wie *Smart UI* und *Layered Architecture* – siehe dazu unsere Ausführungen in Abschnitt 3.2.7.

K-3.23 ▷ 73

Zudem tragen wir durch die Forderung nach der Kapselung von Technologien dem Wert der Flexibilität (siehe Abschnitt 4.2.1.2) Rechnung: Durch das Verbergen der technologischen Details wird es uns möglich, Implementierungen auszutauschen ohne dass sich Modelle ändern müssen. In Kapitel 9 geben wir in Abschnitt 9.1.2.3 Beispiele für derartige Migrationen, von denen wir während der prototypischen Realisierung unseres Ansatzes auch bereits erfolgreich Gebrauch gemacht haben.

K-3.29 ▷ 80

5.4.3.13 Erhaltung der Migrationsfähigkeit

> *Spezifikationen sind als langfristiges Gut zu behandeln.*

Fachliche Modelle sind langlebig im Vergleich zu Softwarearchitekturen. Daher sind alle Modelle in einem nicht-proprietären, offenen und dokumentierten Format zu speichern, so dass die spezifizierten Fachmodelle gegebenenfalls auch in der Zukunft durch heute noch nicht existierende fachmodellgetriebene Generatoren verwendet werden können, um Softwaresysteme auf Basis einer dann aktuellen technologischen Plattform zu generieren.

Für neu entwickelte Metamodelle bietet sich inbesondere die Speicherung der Modelle im XML-Format an (siehe [W3C06]). Die zu den Modellen gehörende Dokumentation kann dann über XML-Schema (siehe [W3C08a, W3C08b]) mit ausführlichen Kommentaren bereitgestellt werden. So wird sichergestellt, dass es eine vielfältige Werkzeugunterstützung sowohl für die manuelle als auch die automatische Bearbeitung der Modelle gibt (z. B. XML-Editoren, XSL-Transformationen (siehe [W3C07]) oder XML-Data-Binding-Bibliotheken wie [JSR222]).

5.4.3.14 Differenziere deine Lösung

> *Entwickle ausschließlich differenzierende Softwarekomponenten selbst und greife für alle nicht-differenzierenden Komponenten auf am Markt verfügbare Komponenten oder Open-Source-Projekte zurück.*

Wir definieren *differenzierende Software* als die Teile eines Systems, die das Alleinstellungsmerkmal der Lösung charakterisieren und nicht ohne weiteres durch bereits verfügbare Angebote abgedeckt werden können. *Nicht-differenzierende Software* meint die anderen Software-Komponenten und beschreibt häufig Infrastrukturtechnologien wie Webframeworks, Persistenzanbindungen, Parser, Datenkonverter, usw. Um den Wert der Wirtschaftlichkeit im Auge zu behalten, sollten Dienstleister immer anstreben, vorgefertigte Lösungen für nicht-dfferenzierende Komponenten eines Systems zu wählen und sich dabei möglichst aus dem Open-Source-Angebot bedienen. Dabei ist natürlich die Qualität der angebotenen Komponenten zu berücksichtigen. Wir empfehlen ausschließlich die Auswahl von Komponenten, die die folgenden generellen Qualitätsmerkmale erfüllen:

- Die Komponente bzw. das Framework ist bereits seit mehreren Jahren am Markt verfügbar.
- Es gibt eine aktive Community.
- Es gibt eine möglichst vollständige und aktuelle Dokumentation, sei es online oder in Buchform (letzteres ist ein zusätzliches Qualitätsmerkmal, da man daraus folgern kann, dass die Komponente erfolgreich genug ist, dass Verlage eine Möglichkeit sehen, damit wirtschaftliche Erfolge zu erzielen).
- Die Komponente ist eine der führenden Lösungen in ihrem Segment.
- Die Komponente passt harmonisch zu den anderen im Projekt eingesetzten Technologien.

5.4.3.15 Realisiere wissenskonforme Zugänglichkeit

> *Das CMDSD-Werkzeug muss unterschiedlich versierten Anwendern konform zu deren Wissensstand einen graduellen Einstieg in seine Funktionen und Möglichkeiten anbieten.*

In der Zusammenarbeit mit Fachexperten wird man immer wieder auf technisch unterschiedlich qualifizierte Partner stoßen. Für diese sollte ein möglichst einfacher Einstieg angeboten werden. Ein naheliegender Weg dafür ist, dass schwierige und selten benötigte Funktionen in Form von Expertenoptionen verborgen werden. Diese sind nur zugreifbar, wenn man sich explizit gegenüber der Oberfläche als Experte charakterisiert. Zudem empfiehlt es sich für den Einstieg, Best Practices bereitzustellen, die Anfängern einen leichten Zugang und schnelle erste Erfolge erlauben.

K-3.41 ▷ 101
K-3.50 ▷ 115

5.5 Weitere Aspekte von CMDSD

In diesem Abschnitt fassen wir unsere Überlegungen zu weiteren Aspekten zusammen, die sich insbesondere aus operativen Fragestellungen in der Umsetzungsphase von CMDSD-Werkzeugen ergeben.

5.5.1 Ausführbarkeit von Modellen

> *Alle Modelle müssen insoweit ausführbar sein, das die Modellierung ein zumindest prinzipiell nachprüfbares Systemverhalten nach sich zieht.*

Diese Anforderung ist besonders wichtig, um gemeinsam mit dem Kunden besprechen zu können, inwieweit die aktuelle Modellierung seinen Erwartungen und Anforderungen entspricht.

K-3.45 ▷ 109

Die Forderung der Ausführbarkeit von Modellen basiert auf Erkenntnissen, die bei der Nutzung prototypischer Vorgehensmodelle zur Softwareentwicklung gewonnen wurden (siehe Abschnitt 3.4.1.2). Zudem ist die direkte Ausführbarkeit ein wichtiger Schritt zur Verkürzung der Kommunikationskette zwischen Domänenexperten und Implementierern (siehe auch unsere Ausführungen in Abschnitt 2.2.1 ab Seite 28). So erfordert selbst das bewährte Verfahren des *Convergent Engineering* (siehe Abschnitt 3.1.1), dass die definierten Modelle vor der Übergabe an die Implementierer verfeinert werden, in dem Verantwortlichkeiten in Methoden übersetzt werden. Diese Übersetzung trägt die bereits von uns formulierten Risiken bzgl. der Verfälschung von Intentionen in sich. Direkt ausführbare Modelle dagegen machen diesen Schritt weitgehend überflüssig. Sind sehr spezielle Funktionalitäten gefordert, erleichtert das bereits beschriebene Prinzip von Erweiterungspunkten (siehe Abschnitt 5.4.3.8.3 auf

K-3.31 ▷ 81
K-3.53 ▷ 125
K-3.54 ▷ 127

K-3.3 ▷ 55

Seite 220) die Spezifikation der Schnittstellen. Die dabei verwendeten Schnittstellen können möglicherweise durch automatisierte Tests abgesichert werden (siehe dazu unseren Ausblick in Abschnitt 10.3.4).

5.5.2 Behandlung der Modelltransformationen

Um Modelle in ausführbare Systeme zu überführen, gibt es mehrer prinzipielle Ansätze, die wir hier überblicksartig vorstellen wollen:

1. Modelle können direkt in eine bereits existierende Technologie übersetzt werden, ggf. mit weiterführenden Anpassungen und Ergänzungen[48]. Es gibt eine Reihe sehr guter Gründe, die zugrunde liegenden Technologien nicht direkt zu verwenden:

   a) Technologie-Migrationen führen nicht zu Änderungen an den Modellen[49].

   b) Modelle werden einfacher[50].

2. Die Modelle werden interpretiert. Modellinstanzen sind in dieser Variante Eingabeparameter in codierte Komponenten, die das Modell analysieren und dann die eigentlichen Aktionen anstoßen[51].

3. Die Modelle werden in mehreren Phasen in eine Zielrepräsentation überführt[52].

[48] So werden z. B. die von uns in Abschnitt 8.1.8.1.2 vorgestellten Prozessmodelle in Spring-Webflow-Konfigurationen (siehe [Spri09b]) übersetzt.

[49] Bei einer Migration von Spring Webflow 1.0 auf Spring Webflow 2.0 wäre das z. B. der Fall gewesen, da sich hier das Konfigurationsformat deutlich geändert hat. Aufgrund der Abstraktion konnten wir die Migration ohne Veränderung der Modelle durch reine Anpassung der Transformationen vornehmen.

[50] Z. B. müssten in Spring Webflow explizit Transitionen für jede Tabelle auf einer Maske angegeben werden, um Paging zu ermöglichen. In unseren abstrakteren View-Modellen reicht es, wenn Widgets als *Pageable* markiert sind. Der Generator erzeugt dann automatisch die notwendigen Transitionen für Spring Webflow und das abstrakte Prozessmodell bleibt einfacher und überschaubarer.

[51] Unser Prototyp PIKAS aus Kapitel 8 behandelt so z. B. Filtermodelle – diese können je nach Kontext zur Laufzeit interpretiert oder in einen Teil einer WHERE-Klausel für die darunterliegende Datenbank transformiert werden. Da in beiden Fällen Laufzeitkontextinformationen berücksichtigt werden müssen, dienen die Filtermodelle als Eingabeparameter in eine Transformator-Komponente, die die Umwandlung erst zur Laufzeit ausführt.

[52] PIKAS realisiert dies z. B. bei der Übersetzung von View-Modellen. Diese werden im PIKAS-Transformationsschritt zunächst in JSF-Seiten (siehe [JSFC09]) übersetzt. Die JSF-Seiten werden dann vom Laufzeitcontainer auf Servlets (siehe [JSR315]) abgebildet. So wird eine mehrstufige Transformation realisiert.

5.5.3 Behandlung von Variabilität

Durch die von uns definierte Form der Modellbildung sind alle aus dem Bereich der Software-Produktlinien bekannten Variabilitätstypen abbildbar (siehe [BKPS04], S. 14–16). Variationspunkte ([BKPS04], S. 17–19) werden je nach Anforderung über primäre, sekundäre oder tertiäre Modelltypen, Modellfacetten oder adaptierte werkzeuginhärente Archetypen wie folgt abgebildet:

- *Variabilität in Features* wird auf der Makroebene durch die werkzeuginhärenten Archetypen abgebildet. Auf der Mikroebene definieren die verfügbaren Modellelemente der einzelnen Modelle die Feature-Variabilität.
- *Variabilität in Abläufen* wird durch domänenspezifische Ablaufmodelle beschrieben. In den meisten Fällen (insbesondere bei interaktiven Systemen) werden von Prozess- und Workflowmodellen inspirierte CMDSD-Modelle eine entsprechende Modellierung von Abläufen ermöglichen. Wiederum werden durch werkzeuginhärente Archetypen standardisierte Workflows vorgegeben, die kundenspezifisch anpassbar sind.
- *Variabilität im Datenumfang* ergibt sich durch die domänenspezifischen Datenmodelle sowie ggf. Modelle für die Visualisierung von Daten. Hier ist zu unterscheiden zwischen Hinzufügungen, Löschungen und Änderungen:
  - Hinzufügungen können jederzeit in den Datenmodellen erfolgen. Allerdings sind dann die entsprechenden Visualisierungsmodelle zusätzlich anzupassen. Zudem kann es erforderlich sein, existierende tertiäre Modelle zu überarbeiten, wenn diese Schnittstellen zu anderen Systemen beschreiben.
  - Löschungen können auf Ebene der Datenmodellierung oder der Visualisierungsmodellierung stattfinden. In vielen Fällen wird es einfacher sein, gewisse Felder in den werkzeuginhärenten Archetypen nur aus den Visualisierungsmodellen statt auch aus den Datenmodellen zu entfernen: Die Datenmodelle werden potenziell an einer Vielzahl von Stellen referenziert, so dass derartige Änderungen aufwändiger sind. Möglich sind aber beide Vorgehensweisen.
  - Änderungen können eine sehr unterschiedliche Komplexität haben:
    - Bei einfachen Datentypänderungen ohne Präzisionsverlust (von Ganzzahl auf Fließkommazahl) können die Änderungen trivial sein.
    - Bei Änderungen mittlerer Komplexität (z. B. von „Datum“ auf „Datum und Uhrzeit“ kann eine solche Änderung in den Datenmodel-

len einfach sein, aber eine manuelle Nachbearbeitung von Visualisierungsmodellen erfordern, um zusätzliche Eingabemöglichkeiten zu schaffen.

  - Bei komplexen Änderungen („Text“ nach „Status“) kann die Änderung aus einer Sequenz von „Löschung“ und „Hinzufügung“ bestehen.

  In jedem Fall ist zu beachten, dass Änderungen bei bereits vorinitialisierten Datenbeständen Datenverluste oder Konsistenzprobleme nach sich ziehen können. Inwieweit solche operationalen Probleme durch das Werkzeug behandelt werden können und sollten, ist von der Domäne und den wirtschaftlichen Fragestellungen bezüglich des notwendigen Implementierungsaufwandes und des Kundennutzens abhängig.

- *Variabilität im Datenformat* wird häufig bei CMDSD-Werkzeugen kein Thema sein, da das Werkzeug prinzipiell von derartigen technischen Details abstrahiert. Handelt es sich um eine fachlich motivierte Frage (wie z. B. „Sollen die Katalogdaten in unserem CMDSD-basierten Produktdatenmanagementsystem im BMECat- oder TecDoc-Format exportiert werden?“, siehe dazu [ScLK05, Tecd09]), so können entsprechende Fragestellungen durch tertiäre Hilfsmodelle oder Modellfacetten realisiert werden. Sind gänzlich neue Formate ein häufiger denkbares Szenario, kann es sogar Sinn machen, entsprechende Erweiterungspunkte in den Modellen vorzusehen.

- *Variabilität im Systemzugang* ist eher eine organisatorische als eine technische Frage und somit problemlos durch CMDSD-Ansätze abbildbar. Technisch damit verbundene Fragestellungen werden häufig die Benutzerschnittstellen unterschiedlicher Zugangswege betreffen, die nachfolgend diskutiert werden.

- Die *Variabilität in Benutzerschnittstellen* ist eine sehr komplexe Fragestellung, da sich Generierung zwar einerseits dafür anbietet, unterschiedliche Zielplattformen zu bedienen (z. B. Web, mobiles Endgerät und Terminal), es andererseits aufgrund der drastischen Detailunterschiede in der Plattformtechnologie bisher aber keine weitverbreiteteten und bewährten Standards für eine entsprechende Vorgehensweise gibt. In regelmäßigen Abständen werden neue Standards definiert, die dieses Ziel verfolgen (in jüngerer Vergangenheit z. B. Java Server Faces (JSF) und ASP.NET, siehe [JSFC09, Wiki09g]), aber es fehlen noch die vorzeigbaren Erfolge, die die Nachhaltigkeit dieser Ansätze illustrieren. Insofern werden CMDSD-Systeme häufig auf eine bestimmte Zielplattform festgeschrieben sein, schon um die Entwicklungskosten der CMDSD-Systeme überschaubar zu halten. Denkbar ist aber in jedem Fall eine Unterstützung

verschiedener Benutzerschnittstellen über eine Ausprägung entsprechender Modellfacetten in primären oder tertiären Modellen.

- *Variabilität über Systemschnittstellen* kann trivialerweise domänenspezifisch durch primäre, sekundäre oder tertiäre Modelltypen abgebildet werden.
- *Variabilität in der Qualität* wird üblicherweise durch Modellfacetten primärer, sekundärer oder tertiärer Modelltypen abgebildet. Häufig werden CMDSD-Werkzeuge aus Kostengründen zumindest in den initialen Versionen nur einen bestimmten Qualitätsgrad oder bestimmte Qualitätsmerkmale unterstützen.

Somit können durch CMDSD-Modelle auch alle Domänen modelliert werden, für die sich Software-Produktlinien eignen. Da sich der grundlegende Prozess der Entwicklung eines CMDSD-Systems sehr stark bei den Vorgehensweisen der Entwicklung einer Software-Produktlinie bedient und diese Vorgehensweisen nur im Rahmen der Fachmodellbildung erweitert, gelten alle Vorteile von Software-Produktlinien auch für CMDSD-Ansätze. Im nächsten Kapitel werden wir demonstrieren, wie sich die Stärken von CMDSD-Modellen nutzen lassen, um weitere Optimierungspotenziale während der eigentlichen Projekte (also der Produktausprägung) zu realisieren.

Interessanterweise finden sich die erfolgreichsten Prinzipien objektorientierter Modellierung (siehe [Tayl95], S. 34) auch in den von uns definierten Fachmodellen wieder:

- Die Spezialisierung von Klassen wird durch eine Kombination von Modellfacetten und Erweiterungspunkten realisiert.
- Die Kollaboration von Klassen wird durch die Verbindung von Modellfacetten (über Referenzen) und dem Top-Down-Entwurf in der Modellkomposition ermöglicht (durch Inklusion und Referenz).
- Die Komposition von Klassen spiegelt sich in der Kombination von Modellfacetten und Topdown-Entwurf in der Modellkomposition wider.

5.5.4 Adaption von CMDSD

Bisher haben wir CMDSD als domänenunabhängiges Metamodell zur Ausprägung domänenspezifischer Modellierungswerkzeuge beschrieben. Wie bereits in Abschnitt 4.3.3 ausgeführt, werden wir die Frage der Adaption auf spezifische Domänen nur kurz für die wichtigsten Aspekte anreißen. Die wichtigste Frage für CMDSD sind dabei die werkzeugbezogenen Anforderungen, die sich unmittelbar aus der betrachteten Domäne ergeben:

- Sind besondere rechtliche oder sicherheitstechnische Rahmenbedingungen zu berücksichtigen, die eventuell eine Prüfung oder Zertifizierung der Software erfordern? Falls dies der Fall ist, sollten das im Rahmen von CDPM definierte Projektmanagementmodell die entsprechenden Rahmenbedingungen insbesondere hinsichtlich Dokumentation und Nachvollziehbarkeit schaffen. Die Software selbst kann in diesem Fall die Zertifizierung durch die konsequente Verwendung automatisierter Tests unterstützen.
- Sind häufige gesetzliche Anpassungen zu erwarten? Ist dies der Fall, sollten von vorne herein entweder Fachmodelle oder Modellfacetten entworfen werden, die eine regelmäßige Anpassung erlauben.
- Sind besondere Dokumentationen zu erstellen? In diesem Fall ist zu prüfen, ob diese Dokumentation ganz oder teilweise aus den Modellen generiert werden kann, um den Dokumentationsprozess so zu vereinfachen und zu beschleunigen.
- Wird auf weit verbreitete Standards zurückgegriffen? Dann ist zu prüfen, welche Standardmodelle bereits verfügbar sind und wie diese in den eigenen Ansatz integrierbar sind. Häufig kann es vorteilhaft sein, eine Übersetzungsschicht zwischen den eigenen Modellen und den Standardmodellen vorzusehen – insbesondere falls auch diese sich dynamisch weiterentwickeln, weil dann kontinuierliche Aufwände durch Anpassungen an den eigenen Modellen vermieden werden können[53].

Wie diese beispielhaften Punkte zeigen, sind letztlich Checklisten der besonderen Rahmenbedingungen für die betrachtete Domäne zu erstellen und für jede Rahmenbedingung sind die Konsequenzen für das Werkzeug und den Entwicklungsprozess zu definieren.

In Abschnitt 9.4 zeigen wir beispielhaft die Nutzung von CMDSD für weitere Einsatzfelder. In Abschnitt 10.3.7 werden wir auf die Adaption von CMDSD auf andere Kontexte als den Mittelstand eingehen.

5.6 Zusammenfassung

Wir haben in diesem Kapitel mit dem Ansatz des *Conceptual Model Driven Software Development (CMDSD)* einen Ansatz zur Entwicklung von Werkzeugen auf

[53] Hier kann wieder das Beispiel „Spring Webflow“ aus unserem Prototyp PIKAS herangezogen werden: Obwohl unsere Prozessmodelle eine hohe Ähnlichkeit zu den Webflow-Modellen besitzen, haben wir entschieden, dafür eine abstraktere Modellierungsform anzubieten – siehe dazu auch Fußnote 49.

Grundlage modellbasierter und generativer Software-Produktlinien vorgestellt. Dazu haben wir in Abschnitt 5.1 zunächst einen kurzen Überblick über die spezifischen Ziele des synergetischen Ansatzes gegeben:

- Es soll eine Infrastruktur geschaffen werden, die auch die Bewältigung komplexer Projekte in einer spezifischen Domäne unter den gegebenen mittelständischen Restriktionen ermöglicht.
- Die zugrundeliegenden Werkzeuge sollen auf für Domänenexperten verständlichen Fachmodellen beruhen.
- Sie bewegen sich auf einem Abstraktionsniveau, das dennoch die automatisierte Generierung von Lösungen ermöglicht.
- Die explizite Modellierung von Domänenwissen erhält dieses Spezialwissen in den Unternehmen statt es in technischen Modellen zu verbergen.

Anschließend haben wir das konkrete Vorgehens in Abschnitt 5.2 motiviert, indem wir den Nutzen einer besonders engen Kommunikation zwischen Domänen- und IT-Experten dargestellt und den Nutzen der Verwendung von Standardarchitekturen beschrieben haben. Anschließend haben wir das prinzipielle Vorgehensmodell in Abschnitt 5.3 vorgestellt. Nachfolgend haben wir die Werte und Prinzipien des *Conceptual Programming* verfeinert und diese zu spezifischen Methoden konkretisiert, die die Umsetzung von CMDSD-Werkzeugen unterstützen (Abschnitt 5.4). Besonders hervorzuheben sind dabei:

- unsere Methode der Archetypisierung, die sowohl zur Abgrenzung der Domäne und Beschreibung der grundlegenden Modelle dient als auch als Mittel, um einen initialen Werkzeugkasten sinnvoll zu befüllen, der dann die Ausprägung von Lösungsinstanzen nachhaltig beschleunigt (siehe insbesondere Abschnitt 5.4.3.5,
- unsere Klassifikation von Modelltypen und ihren Eigenschaften für einen pareto-basierten und durch seine Pragmatik äußerst mittelstandsorientierten Ansatz zur präziseren Einordnung und Komplexitätsbewertung von Modellen (siehe Abschnitt 5.4.3.8) und
- unser Ansatz zur Reduktion der Komplexität in Einzelmodellen, der üblicherweise zu einer Top-Down-Dekomposition von Modellsichten führt (siehe Abschnitt 5.4.3.9).

Auf diese Weise streben wir zu einem Entwicklungsmodell, das auf Basis abstrahierender und auch vereinfachender (bzw. standardisierender) Modelle die Ausprägung

spezifischer Problemlösungen erlaubt. Wir messen dieser Idee großes Potenzial zu, da unserer Erfahrung nach viele Kundenprobleme zu einem großen Teil durch weitgehend standardisierte Vorgehensweisen gelöst werden können, die sich heute in den verfügbaren Frameworks und Bibliotheken der meisten Programmiersprachen aber nur sehr unzureichend widerspiegeln.

So schätzt bereits Martin in [Mart81] auf S. 83, dass in vielen Projekten weniger als 10% der Anforderungen tatsächlich so beschaffen sind, dass sie formale Programmiersprachen zu ihrer Lösung benötigen. Dies deckt sich mit unseren Erfahrungen aus dem mittelständischen Umfeld. Wir möchten aber noch mal betonen, dass aus unserer Sicht der (neue) kritische Erfolgsfaktor in der konsequenten Beschränkung der betrachteten Domäne liegt und dies auch den essentiellen Unterschied z. B. zu den existierenden MDA-Ansätzen ausmacht: Nur durch Beschränkung auf einen viel überschaubareren Problembereich kann die Modellierungssprache in endlicher Zeit so ausdrucksfähig gestaltet werden, dass sie tatsächlich 90% oder mehr der typischen Domänenanforderungen durch Modellierung auf hoher Abstraktionsebene abdecken kann.

Abschließend sei noch mal darauf hingewiesen, dass insbesondere Archetypen eine sehr leistungsfähige Möglichkeit sind, um die Erstellung von Prototypen bei der Softwareentwicklung maßgeblich zu beschleunigen (siehe dazu auch die Diskussion in Abschnitt 3.4.1.3). Hier zeigen sich die immanenten Stärken modellbasierter Vorgehensweisen: Allgemeine Vorlagen für domänenspezifische Lösungen können bereits bei der Erstellung des CMDSD-Systems erzeugt werden und bewirken in jedem nachfolgenden CMDSD-Projekt Einsparungen sowohl im Prototyping als auch bei der Fertigstellung eines Systems.

Zudem bieten sich weitere Optimierungspotenziale, da Projekterfahrungen wiederum in die Archetypen zurückfließen können. Wichtig ist dabei aber, die Archetypen nicht mit Funktionalität zu überfrachten, da sie sonst ihre Flexibilität und Anpassbarkeit verlieren. Als Grundprinzip gilt hier, dass Funktionalitäten nur dann direkt in einen Archetypen einfließen sollten, wenn mindestens 80% aller Lösungen, die auf einem Archetypen beruhen, die entsprechende Funktionalität benötigen. Andernfalls ist es besser, die entsprechenden Erweiterungen in Form von Best Practices in der Dokumentation oder Hilfe zu dem CMDSD-System zu erfassen.

Kapitel 6

Concept Driven Project Management (CDPM)

„Reisender, es gibt keine Straßen, Straßen entstehen im Gehen.“
— Weisheit aus Spanien

Das *Concept Driven Project Management (CDPM)* ist eine Projektmanagement-Meta-Methode, die sich darauf fokussiert, CMDSD-basierte Projekte in engster Abstimmung mit dem Kunden bzw. den Fachexperten auf Kundenseite durchzuführen und dabei den Schwerpunkt auf die fachlichen Probleme zu legen. Derartige Projekte erfordern ein konsequentes Umdenken aller Projektteilnehmer, da sie sich in zentralen Teilen von der Vorgehensweise in klassischen IT-Projekten unterscheiden. Wesentliche Unterschiede sind zum einen die viel stärkere Einbeziehung des Kunden (die in dieser Form sonst im Wesentlichen nur aus dem Extreme Programming bekannt ist, siehe [BeAn05] zum Thema „On Site Customer“) sowie die Konzentration auf rein fachliche Aspekte einer Aufgabenstellung mit dem Ziel, möglichst früh und schnell ein ausführbares System zu erzeugen, anhand dessen die Spezifikation verifiziert und qualitativ getestet werden kann.

Praktische Erfahrungen zeigen, dass Agilität in Projekten, die modellgetriebene Ansätze verfolgen, nicht alleine durch den Einsatz entsprechender MDA- bzw. MDSD-Werkzeugen erreicht werden kann. Ebenso erforderlich ist es, die richtige Form des Softwarentwicklungsprozesses auszuwählen und allen Projektmitarbeitern zu kommunizieren (siehe z. B. [Ther05]). Das hier vorgestellte CDPM-Modell kombiniert dazu unsere Erfahrungen aus einer Vielzahl praktischer Projekte basierend auf den CP-Ideen mit den grundlegenden konzeptionellen Erkenntnissen aus Kapitel 3.

CDPM stellt weder eine neue noch eine in sich geschlossene Projektmanagementme-

thode vor, da auch hier wieder in Analogie zu unseren Ausführungen zum CMDSD-Ansatz in Kapitel 5 je nach Domäne unterschiedliche Anforderungen zu berücksichtigen sind. Stattdessen geben wir eine Reihe von Prinzipien und Methoden an, die mit den für einen Dienstleister bzw. eine Domäne geeigneten und bereits existierenden Vorgehensmodellen kombiniert werden. So wird das jeweilige Vorgehensmodell optimal für den Einsatz der in Kapitel 5 konzipierten und entwickelten CMDSD-Werkzeuge vorbereitet. CDPM geht also davon aus, dass ein CMDSD-Werkzeug bereitsteht, mit dem die domänenspezifischen Projekte abgewickelt werden.

Abschnitt 6.1 stellt die grundlegenden Ziele von CDPM vor, die wir auf Basis der zuvor beschriebenen CMDSD-Werkzeuge erreichen wollen. Abschnitt 6.2 motiviert den Ansatz noch einmal kurz durch Betrachtung der operativen Herausforderungen in mittelständischen IT-Projekten. Abschnitt 6.3 beschreibt CDPM überblicksartig und verweist dazu auf die Methoden, die für das Meta-Modell von zentraler Bedeutung sind. Diese werden zusammen mit den grundlegenden Werten und Prinzipien im Rahmen der detaillierten Beschreibung des CDPM-Ansatzes in Abschnitt 6.4 ausführlich dargestellt. Es folgt Abschnitt 6.5, der weitere bei der praktischen Umsetzung relevante Gesichtspunkte vorstellt. In Abschnitt 6.6 fassen wir die besonderen Leistungen dieses Kapitels noch einmal kurz zusammen, bevor wir unseren CP-Ansatz in seiner Gesamtheit noch einmal kritisch im anschließenden Kapitel 7 von anderen existierenden Ansätzen abgrenzen.

Wir wollen hier betonen, dass der Hauptteil unserer Arbeit sich bereits im CMDSD-Ansatz manifestiert. CDPM stellt „nur“ noch das Finetuning für das operative Projektgeschäft bereit, um den Nutzen der CMDSD-Werkzeuge zu maximieren. Dennoch sind die hier vorgestellten Bausteine sehr wichtig, da wir in unseren Projekten beobachten konnten, dass eine falsche oder unzureichende Nutzung der Vorteile von CMDSD-Werkzeugen Projektchancen auch in das Gegenteil umkehren können. Insbesondere tendieren Entwickler dazu, die Möglichkeiten des CMDSD-Werkzeugs wieder zu sehr ausdehnen zu wollen, um eine spezifische technische Lösungsimplementierung zu realisieren und übersehen dabei häufig die bereits existierenden Mittel, die das Werkzeug bietet – siehe dazu unsere Ausführungen in Abschnitt 5.4.2.2 ab Seite 175.

Wie bereits in Abschnitt 3.6 erklärt, werden wir die von uns hergeleiteten Konzepte als Randmarkierung referenzieren, wenn sie für unsere Ausführungen bedeutsam sind.

6.1 Ziele

„Selbst eine schwere Tür hat nur einen kleinen Schlüssel nötig.“
— Charles Dickens (1812–1870), Englischer Schriftsteller

> *Concept-Driven Project Management soll als synergetische Verschmelzung aus bewährten und für eine Domäne geeigneten Projektmanagementverfahren und ergänzend definierten Prinzipien und Methoden die Abwicklung von individuellen Projekten dahingehend optimieren, dass Kunden im direkten Kontakt mit CMDSD-Beratern Probleme beschreiben, Lösungen umsetzen, prototypisch erproben und so in möglichst kurzen iterativen und evolutionären Zyklen eine hinreichend gute Lösung im zur Verfügung stehenden Budgetrahmen realisieren.*

Um dies zu erreichen, verweisen wir auf die Ziele unserer CMDSD-Methode aus Abschnitt 5.1 und wiederholen die wichtigsten Hinweise zum mittelständischen Kontext, die für die hier erfolgende Definition von CDPM von besonderer Bedeutung sind:

- Häufig wird eine Vorgehensweise gewünscht, die Charakteristika agiler Prozesse aufweist (siehe auch [FrWe02]): K-3.35 ▷ 88
  - Projektergebnisse sollen so früh wie möglich verfügbar sein.
  - Eine enge Abstimmung mit Fachabteilungen ist erforderlich und erwünscht.
  - Anforderungen ändern sich stetig und häufig auch kurzfristig.
- Es stehen nur begrenzte Budgets für die Umsetzung von IT-Projekten zur Verfügung. So entsteht ein Bedarf nach kurzen Projektlaufzeiten und entsprechend kurzen Kommunikationswegen für die Abstimmung von Anforderungen.

Alle nachfolgend vorgestellten Prinzipien und Methoden zielen daher darauf ab,

1. die Kommunikationsdistanz zwischen IT- und Fachexperten zu minimieren, um ein möglichst intensives gegenseitiges Verständnis für die Herausforderungen und tatsächlichen alltäglichen Probleme des Anderen zu schaffen. Dies K-3.38 ▷ 91
K-3.35 ▷ 88

erzeugt zusätzliches Vertrauen in die Einschätzungen des jeweiligen Partners, was in kritischen Projektmomenten hilft, um schnell eine Lösungsalternative abzustimmen.

2. die Abstraktionsmöglichkeiten von CMDSD möglichst oft zum Tragen zu bringen, damit technischer Diskussionsbedarf minimiert und fachlicher Diskurs maximiert wird.

K-3.37 ▷ 90

3. jederzeit ein frühes und auf praktischen Beobachtungen basierendes Feedback zu ermöglichen.

K-3.35 ▷ 88

4. durch den Einsatz von Rapid Prototyping Fachanwender dabei zu unterstützen, kreative Lösungen für ihre Probleme zu finden und gleichzeitig wichtige von unwichtigen Anforderungen zu trennen, um die zur Verfügung stehenden Budgets für die Kernanforderungen zu nutzen.

K-3.20 ▷ 67
K-3.31 ▷ 81

6.2 Das fachmodellgetriebene Manifest (Teil II)

Analog dazu, wie wir in Abschnitt 5.2 versucht haben, die Schwerpunkte von IT-Experten weg von technologischen Details und hin zur Domänenexpertise zu verschieben, wollen wir im Rahmen von CDPM die Gewichtung von organisatorischer und formaler Behandlung von Projekten hin zu offener Kommunikation und ehrlichem Diskurs bewegen.

Projekte sind nach unserer Erfahrung heute häufig von formalen Prozessen bestimmt[1]:

- Es gibt einen formalen Angebotsprozess, der darauf abzielt,
  - aus Sicht des Dienstleisters den Profit im Rahmen des Projekts zu maximieren und insbesondere keine unnötigen Zugeständnisse zu machen,
  - aus Sicht des Kunden möglichst viel Leistung für möglichst wenig Geld zu erhalten und zunächst einmal möglichst viele Funktionen zu fordern, häufig in der Hoffnung, dabei nichts Wichtiges zu vergessen.
- Es gibt einen formalisierten Anforderungsprozess (manchmal auch erst nach der Angebotserstellung), in dem beide Seiten versuchen, für sich einen maximalen Nutzen zu definieren.

[1] Wir merken aber sofort an, dass die nachfolgenden Punkte vor der eigentlichen Entwicklungsphase mit zunehmender Projektgröße stärker ausgeprägt sind und in kleineren Projekten nicht immer so deutlich zum Tragen kommen. Alle geschilderten Probleme ab dem Entwicklungsprozess beobachten wir dagegen in Projekten jeder Größenordnung.

- Es gibt einen Entwicklungsprozess, der für Kunden häufig sehr undurchsichtig und schwer nachvollziehbar ist und in dessen Rahmen ebenso häufig aufgrund (realer oder fälschlich eingebildeter) technischer Notwendigkeiten Veränderungen in der Domänenmodellierung vorgenommen werden. Zudem werden Probleme in der Realisierung selten offen mit dem Kunden diskutiert, sondern es werden „Workarounds“ entwickelt, die zwar meistens pragmatisch aber auch eher kurzsichtig ein erfolgreiches Projekt ermöglichen sollen. Dadurch verpasst man einerseits die Chance, gemeinsam tragfähigere Lösungen zu entwickeln und läuft andererseits Gefahr, die Evolutionsmöglichkeiten eines Systems nachhaltig zu beschränken, weil die Summe der „Workarounds“ zu schwerwiegenden architektonischen Problemen führen.
- Tests werden nur teilweise automatisiert durchgeführt, sondern überwiegend (wenn überhaupt) in textuellen Testspezifikationen beschrieben. Häufig werden selbst auf diese Weise manuell durchzuführende Tests bei wiederholten Abnahmen in unzureichender Präzision ausgeführt. Insbesondere erfolgen selten Nachbesserungen an einmal definierten Tests.

Wir wollen durch die nachfolgend vorgestellten Prinzipien und Methoden eine nachhaltige Vertrauensgrundlage zwischen Kunden und Dienstleister schaffen, die es erlaubt, viele der nur schwer vorhersehbaren Probleme, auf die man üblicherweise in Projekten stößt, in schneller und pragmatischer Abstimmung aus dem Weg zu räumen. Wir haben mit diesem Ansatz sehr gute Erfahrungen gemacht und glauben, dass einige wenige Bausteine klassische Projektmanagementansätze dahingehend ergänzen können, dass sie auf eine gesündere Basis gestellt werden[2].

6.3 CDPM im Überblick

Um die zuvor beschriebene und von Vertrauen und Offenheit geprägte Projektkultur zu schaffen, gehen wir wie folgt vor:

- *Wähle als Dienstleister eine Projektmanagementmethode, die für die betrachtete Domäne angemessen ist und zu der bereits möglichst viele Erfahrungen existieren.* Dies gewährleistet, dass sich die Mitarbeiter in einem vertrauten Rahmen bewegen und bereits hinreichende Erfahrungen in der Anwendung der Methode existieren. Agile Projektmanagementmethoden sind hier klar zu

K-3.26 ▷ 74

[2] Mit „gesünder“ meinen wir dabei, dass weniger auf formalen Aspekten der Projektabwicklung beharrt und mehr über die tatsächlichen Probleme und die dafür sinnvollen Lösungen gesprochen wird, um beiderseitig ein gutes Projektergebnis zu erzielen.

bevorzugen (siehe Abschnitt 3.4.3), Feature Driven Development ist häufig ein gut zugänglicher Kompromiss für Unternehmen, die klassischere präskriptive Ansätze gewohnt sind (siehe Seite 131).

K-3.35 ▷ 88
K-3.46 ▷ 111

- *Gestalte die Methode (falls erforderlich) so um, dass die Entwicklung iterativ, evolutionär, rekursiv und inkrementell erfolgen kann.* Dies kann insbesondere dadurch erreicht werden, dass anstelle komplexer Spezifikationen im Vorfeld der Projektumsetzung nur die Projektziele möglichst präzise abgestimmt und die einzelnen dazu erforderlichen Schritte erst im Rahmen des Projektes gemeinsam geplant werden. Die prinzipielle Machbarkeit muss natürlich im Vorfeld geklärt werden.

K-3.54 ▷ 127

- Erweitere das bestehende Projektmanagementmodell um die folgenden Methoden:

  - *Beginne die eigentliche Projektarbeit mit einem philosophischen Projektkickoff* (Abschnitt 6.4.3.1). Dadurch werden alle Beteiligten mit dem Vorgehensmodell und seinen Besonderheiten vertraut gemacht und auf die gemeinsamen Projektziele eingeschworen. Zudem erhalten Sie einen vertrauensbildenden Überblick über die einzusetzenden Werkzeuge und Methoden.

K-3.12 ▷ 64

  - *Führe die zentralen Akteure im Rahmen eines oder mehrerer CP-Coachings in die spezifischen Modelle und Methoden des Vorgehensmodells ein* (Abschnitt 6.4.3.2). So erhalten alle Beteiligten ein Mindestmaß an Übung und Sicherheit, das ihnen für die tatsächlichen Aufgaben im Projekt hinreichend viel Selbstvertrauen und Zugang zu der prinzipiellen Vorgehensweise geben soll. Zudem führt dies dazu, dass alle Beteiligten als vollwertige Partner einbezogen werden und ihnen bewusst die Wichtigkeit ihrer Rolle verdeutlicht wird.

K-3.13 ▷ 64
K-3.14 ▷ 64

  - *Führe die Problemanalyse durch prototypische Entwürfe durch und versuche anhand beispielhafter Szenarien die Einzelaspekte des Problems zu erarbeiten* (Abschnitt 6.4.3.3). So werden die Projektbeteiligten anhand möglichst konkreter Fragestellungen und greifbarer Prototypen an das oder die Detailprobleme herangeführt und es beginnt ein kreativer Lösungsprozess, der im schnellen Wechsel zwischen Entwurf und Umsetzung schrittweise auf eine hinreichend gute Lösung hinführt.

K-3.38 ▷ 91

  - *Realisiere die Entwürfe durch Rapid Development* (Abschnitt 6.4.3.4). Dabei handelt es sich um eine von uns entworfene zweiteilige Methode, die

K-3.31 ▷ 81
K-3.52 ▷ 120

zwischen kurzen Modellierungs- und Implementierungsphasen[3] wechselt, um so möglichst schnell greifbare und erfassbare exemplarische Lösungen in Form von Prototypen zu erarbeiten. Die Beteiligten entscheiden dann im weiteren Verlauf, ob sie Prototypen verwerfen, ausbauen oder sogar als endgültige Lösungen übernehmen.

- *Führe die eigentliche Implementierung einer Lösung durch Conceptual Pair Programming durch* (Abschnitt 6.4.3.5). Idee von Conceptual Pair Programming ist es, das aus dem Extreme Programming bekannte Pair Programming in erweiterter Form umzusetzen, indem ein Fach- und ein CMDSD-Experte gemeinsam mit dem CMDSD-Werkzeug einen Lösungsentwurf umsetzen und evaluieren. Die daraus resultierende Diskussion und der Austausch zwischen Kunden- und Dienstleisterrepräsentanten führt zu einer schnellen Intensivierung des Problemverständnisses und erzeugt zudem gegenseitiges Vertrauen.

K-3.2 ▷ 50
K-3.22 ▷ 72

- *Änderungen werden in Refactoring-Phasen in die Prototypen oder endgültigen Modelle übernommen* (Abschnitt 6.4.3.6). So werden neue Ansätze und Ideen möglichst schnell ausprobiert und die initialen Prototypen nähern sich Schritt für Schritt dem endgültigen System an.

K-3.54 ▷ 127

Ergänzend sind die folgenden begleitenden Maßnahmen kontinuierlich anzuwenden:

- *Das erworbene Fachwissen über die kundenspezifische Ausprägung der betrachteten Domäne wird in Form eines agilen Glossars verwaltet* (Abschnitt 6.4.3.8). Auf diese Weise wird sichergestellt, dass das erarbeitete Spezialwissen erhalten und für alle Beteiligten zugreifbar bleibt. Werden neue Konzepte identifiziert, finden diese einen sofortigen Eingang in das Projekt und stehen allen Stakeholdern wieder unmittelbar zur Verfügung.

K-3.6 ▷ 60

- *Stelle sicher, dass es eine Dokumentation zur Evolution des Systems z. B. in Form eines lebenden Angebots gibt* (Abschnitt 6.4.3.7). Wichtig ist uns hier nicht, jede Detailentscheidung zu dokumentieren – davon gibt es in den meisten Projekten so viele, dass der Versuch, diese nachzuhalten bereits im Ansatz zum Scheitern verurteilt zu sein scheint. Wichtiger ist es, die zentralen und wichtigen Ideen eines Entwurfs in ihrer Weiterentwicklung für alle greifbar festzuhalten, so dass der essentielle Stand des Projekts für alle Beteiligten sichtbar ist und Feinjustierungen von außen

[3] Wir bezeichnen diese Unterphasen als *Rapid Modeling* und *Conceptual Pair Programming*.

schnell eingebracht werden können.

6.4 CDPM als Paradigma

CDPM lässt sich am Besten auf Grundlage agiler Vorgehensmodelle umsetzen, die dann um die nachfolgend beschriebenen Werte, Prinzipien und Methoden erweitert werden. Abschnitt 6.4.1 geht dazu kurz auf die zugrunde liegenden Werte ein, Abschnitt 6.4.2 beschreibt die wesentlichen und durchgehend zu berücksichtigenden Prinzipien und Abschnitt 6.4.3 schließlich gibt die zentralen Methoden des CDPM an, die insbesondere in das zugrunde liegende Vorgehensmodell einzupflegen sind.

6.4.1 Werte

Wie bereits in Abschnitt 4.3.1 ausgeführt fußt CDPM analog zu CMDSD auf den gleichen Werten wie CP, also *Wirtschaftlichkeit* (Abschnitt 4.2.1.1), *Flexibilität* (Abschnitt 4.2.1.2) und *Zugänglichkeit* (Abschnitt 4.2.1.3). Auch hier sind weder Änderungen noch Ergänzungen erforderlich. Allerdings werden wir wie im Rahmen unserer Ausführungen zu CMDSD die aus den Werten resultierenden Prinzipien nachfolgend verfeinern. Wirtschaftlichkeit wird weiterhin das oberste Ziel bleiben, im Rahmen des operativen Projektgeschäfts werden wir aber insbesondere die Werte der Flexibilität und Zugänglichkeit betonen, die durch die besonders zugängliche Ausgestaltung der CMDSD-Werkzeuge gegeben sind.

6.4.2 Prinzipien

Mit den nachfolgend erläuterten Prinzipien zielen wir darauf ab, dass allgemeine Projektklima möglichst positiv, angenehm und unkompliziert zu gestalten. IT-Projekte weisen häufig für alle Beteiligten einen relativ hohen Stressfaktor auf, da sich zum einen Fachexperten in für sie wenig interessante technische Herangehensweisen einfinden und zum anderen IT-Experten innerhalb möglichst kurzer Zeit ein Verständnis für eine ungewohnte Domäne oder sehr spezielle Domänenprobleme erlangen müssen. Beide Probleme relativieren sich bereits bis zu einem gewissen Maße dadurch, dass die in Kapitel 5 vorgestellten CMDSD-Werkzeuge auf einem sehr hohen und untechnischen Abstraktionsniveau versuchen, eine Domäne zu modellieren und so sowohl die Fachexperten von unnötigen technischen Details abschirmen als auch

K-3.50 ▷ 115

den IT-Experten mehr Zeit geben, sich statt mit Technik mit der Problemdomäne zu befassen. Dennoch wird es aufgrund des in Abschnitt 5.4.3.6 angewendeten Pareto-Prinzips immer wieder die Notwendigkeit zur Beschäftigung mit der jeweils anderen Domäne geben. Die daraus resultierenden Potenziale für Missverständnisse und schwierige Kommunikationssituationen wollen wir durch geeignete Prinzipien abfedern.

K-3.37 ▷ 90

Wir werden die folgenden Verfeinerungen der zentralen Prinzipien aus Abschnitt 4.1.2 vornehmen:

- als Verfeinerungen des Prinzips der Kundenorientierung aus Abschnitt 4.2.2.1:
  - *Respekt* (Abschnitt 6.4.2.1) als grundsätzliches Prinzip für einen möglichst angenehmen Umgang zwischen allen Projektbeteiligten,
  - *Fairness* (Abschnitt 6.4.2.2) als Grundlage für den gegenseitigen vertrauenswürdigen Umgang mit Änderungen, die sich erst im Projektverlauf ergeben,
  - *Offenheit und Ehrlichkeit* (Abschnitt 6.4.2.3), um kooperative Prozesse zu vereinfachen und nachhaltiges Vertrauen zwischen den Beteiligten zu schaffen.
- als Verfeinerungen des Prinzips der Domänenfokussierung aus Abschnitt 4.2.2.2:
  - *Uniformität* (Abschnitt 6.4.2.4) als Mittel, um der spezifischen Kundenkultur Rechnung zu tragen,
  - *Kompetenz statt Komplexität* (Abschnitt 6.4.2.5, um Projektpartner als gleichwertig und freundlich gesinnte Partner anzuerkennen.
- als Verfeinerung des Prinzips der Automatisierung aus Abschnitt 4.2.2.3:
  - *CMDSD-basiertes Vorgehen* (Abschnitt 6.4.2.6), um den konsequenten Einsatz eines CMDSD-Werkzeugs sicherzustellen.
- als Verfeinerungen des Prinzips der Agilität aus Abschnitt 4.2.2.4:
  - *Konsequenz* (Abschnitt 6.4.2.7), um die Potenziale von CMDSD operativ zu realisieren,
  - *Direktheit* (Abschnitt 6.4.2.8), um Kommunikations- und Abstimmungsprozesse möglichst effizient zu gestalten,
  - *Explizivität und Transparenz* (Abschnitt 6.4.2.9), um für einen allgemein anerkannten Wissensstand zu sorgen,

- *Bedarfsorientierung* (Abschnitt 6.4.2.10), um den Gesamtaufwand zu minimieren und die verfügbaren Ressourcen auf die tatsächlichen Probleme zu lenken,
- *Min-Max* (Abschnitt 6.4.2.11) als Bekräftigung des agilen Prinzips (insbesondere aus dem Extreme Programming und dem Test Driven Development), wirklich nur das aktuell relevante Problem zu bearbeiten und nicht für eine schwer vorhersehbare Zukunft vorauszuplanen,
- *Start small, then grow* (Abschnitt 6.4.2.12) als Mittel, um durch möglichst früh realisierten Geschäftswert eine Aufbruchsstimmung im Projekt zur Förderung der Akzeptanz der neuen Lösung zu erzeugen.

Abbildung 6.1 fasst die detailliertere Untergliederung der generelleren CP-Prinzipien analog zu Abbildung 5.1 auf Seite 174 noch einmal bildlich zusammen.

6.4.2.1 Respekt

> *Behandle dein Gegenüber so wie Du behandelt werden willst.*

Respekt meint als Prinzip insbesondere

- seinen Projektpartner ernst zu nehmen.
- Anliegen, die dem Projektpartner wichtig sind, als Äußerung eines echten Problems zu begreifen und respektvoll zu hinterfragen, auch wenn Sie zunächst seltsam erscheinen mögen. Häufig verstehen Fachexperten nicht wirklich, was ein IT-Experte ihnen mitteilen will, da sie weder das hinter einem Wunsch stehende technische Problem erkennen (können) noch die zur Erläuterung verwendeten Begrifflichkeiten und Zusammenhänge in ihrer Gänze erfassen (können). Umgekehrt erscheint IT-Experten manche fachliche Notwendigkeit in ihrer Bedeutung unverständlich, überraschend oder wenig ernst[4]. Beide Seiten müssen

[4]Wir haben selbst mehrfach die Erfahrung gesammelt, dass sich eine Runde von Fachexperten in Zusammenarbeit mit höheren und höchsten Managementebenen stundenlang z. B. über die richtige Farbe eines Schalters auf einer einzelnen Maske ausgetauscht hat und in ihrem intensiven Diskurs gar kein Interesse an den tagelang vorbereiteten technischen Präsentationen zu Architektur und Struktur des eigentlichen Systems hatte. Dahinter steckte weder eine Missachtung für die Leistung der IT-Experten noch handelte es sich um eine „komisch ausgeartete Diskussion“. Nach mehrstündiger Diskussion war endlich eine Farbe festgelegt und das Meeting wurde – zur Überraschung der anwesenden IT-Berater – als großer Erfolg und wichtiger Durchbruch in dem Projekt gewertet. Woran liegt dies? Desinteresse an Technologie? Ja und nein. Besagter Schalter war Auslöser eines wichtigen und in der Vergangenheit zu Problemen

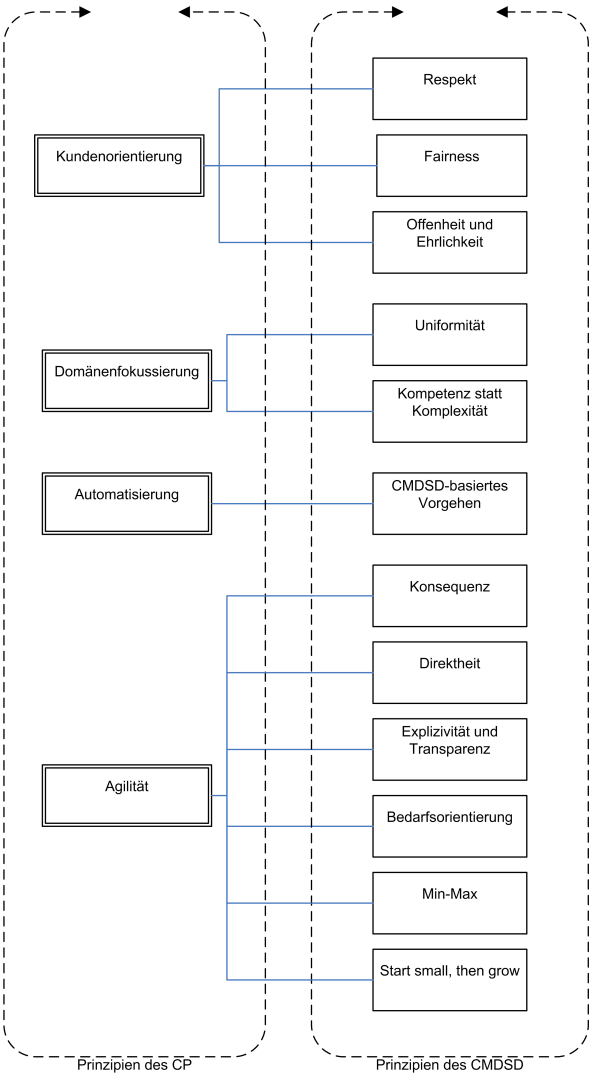


Abbildung 6.1: Prinzipien des CDPM

lernen, das hinter nicht leicht verständlichen Aussagen wichtige Anforderungen und Probleme stehen können und daher insbesondere unverständliche Aussagen zu hinterfragen und zu klären sind. Andernfalls droht eine Haltung der Form „die interessieren sich ja gar nicht für mein Problem“, was dann zu „Workarounds“ und taktisch wie strategisch unglücklichen Entscheidungen führen kann.

- Anfragen mit der gebührenden Aufmerksamkeit und Ernsthaftigkeit zu beant-

führenden Prozesses und hatte daher eine große fachliche und historisch motivierte Bedeutung für das Zielsystem. Entsprechend intensiv und aus Sicht der Fachseite vollkommen angemessen wurde er diskutiert. Dieses Beispiel zeigt unserer Ansicht nach sehr anschaulich, das scheinbar triviale Diskussionen einen durchaus wichtigen Hintergrund haben können und Fachexperten daher in ihrer Motivation im Zweifelsfall entsprechend ernst zu nehmen sind.

worten. Insbesondere gilt dies auch, wenn Anfragen zum wiederholten Male gestellt werden. Der Fragende weist damit (mglw. unbeabsichtigt) darauf hin, dass er einen Gesamtsachverhalt immer noch nicht vollständig erfasst hat. Entsprechend sind gerade wiederholte Anfragen nicht als Belästigung anzusehen, sondern man sollte versuchen, diese durch eine neue Form der Erklärung noch einmal zu beantworten, um so Schritt für Schritt Verständnis für das enhaltene Problem zu erzeugen. Unsere Erfahrung zeigt, dass dieser Prozess – so mühsam er zuweilen sein kann – im weiteren Projektverlauf häufig sehr positive Effekte haben kann, da das so erzielte tiefere Verständnis die Projektbeteiligten zu echten Partnern macht, die sich zumindest gleichwertig fühlen, was wiederum positive Auswirkungen auf das Projektklima hat.

6.4.2.2 Fairness

Belohne Gleiches mit Gleichem und maximiere den langfristigen Profit.

Die nachfolgenden Prinzipien beschreiben verschiedene Herangehensweisen, um offen und ehrlich mit Kunden umzugehen und dies auch nachhaltig zu forcieren. Bei einem offenen Umgang miteinander wird man immer wieder Probleme thematisieren und erst im Verlauf des Projekts den bestmöglichen Weg zu einer sinnvollen Lösung identifizieren. Damit Offenheit und Ehrlichkeit tatsächlich Vertrauen schaffen können, muss aber auch die Fairness zwischen den Partnern gewahrt bleiben. Insbesondere sollten Gefallen auf Grundlage von Gegenseitigkeit erwidert werden. Das Gleichgewicht zwischen Geben und Nehmen soll also gewahrt bleiben.

Wir haben festgestellt, dass ein offenes Thematisieren des fairen Umgangs (bei Bedarf) meist zielführend ist, insbesondere wenn es gelingt, mit sachlichen und objektiven Argumenten die gegeneinander aufzuwiegenden Zugeständnisse darzustellen. Dazu empfiehlt es sich, eine pragmatische und doch hinreichend präzise Nachverfolgung des Projektverlaufs vorzunehmen (siehe dazu die Methode „Lebendes Angebot“ in Abschnitt 6.4.3.7).

6.4.2.3 Ehrlichkeit und Offenheit

> *Sorge für unkomplizierte Verhältnisse im Projekt, indem alle Abläufe zwischen Kunden und Dienstleistern ehrlich und offen gehandhabt werden.*

Ein effizientes Vorgehen zwischen ungleichen Partnern[5] erfordert eine möglichst vertrauensvolle Grundlage. Am leichtesten lässt sich Vertrauen durch Offenheit und Ehrlichkeit herstellen.

Unter Offenheit und Ehrlichkeit verstehen wir in Bezug auf unser Anliegen insbesondere:

- Auf Wunsch sollte der Kunde jederzeit Einblick in die Prozesse und Arbeitsweisen des Dienstleisters erhalten.
- Probleme sollten unbeschönigt angesprochen werden. Kompetenz beweist man nicht durch das Verschweigen von Problemen, sondern durch den sicheren Umgang mit ihnen sowie das Anbieten von Lösungen. Wir empfehlen daher ehrliche Gespräche mit dem kompetenten Aufzeigen von Alternativen und Auswegen zu verbinden.
- Günstigere und ungünstigere Alternativen bei der Realisierung (insbesondere auch aus Sicht des Dienstleisters) sollten offen angesprochen werden. Häufig zeigt sich, dass auch die Ansprechpartner auf Kundenseite an einem ruhigen und entspannten Projektablauf interessiert sind und daher entsprechend oft kooperativ mit dem Dienstleister zusammenarbeiten werden. Viele Kunden erhoffen sich nach unserer Erfahrung dadurch auch kleinere Begünstigungen, wenn sich zeigt, dass Details in der Anforderungsspezifikation vergessen wurden (siehe auch die Hinweise zum fairen Umgang in Abschnitt 6.4.2.2).
- Sei ehrlich in Bezug auf Stärken und Schwächen. Es bringt häufig wenig, Schwächen zu verheimlichen (da sich diese im Projektverlauf und insbesondere bei

[5] Wir sprechen hier von ungleichen Partnern, da wie mehrfach geschildert einerseits ausgewiesene Domänenexperten mit häufig eher geringen IT-Kenntnissen auf IT-Experten mit vergleichsweise geringen Domänenkenntnissen treffen. Im Gegensatz zu Großunternehmen sind die Unterschiede häufig größer, da mittelständische Unternehmen meist Nischenanbieter mit einer eher begrenzten Personaldecke sind, so dass sich einzelne Mitarbeiter sehr stark auf ihr jeweiliges Fachgebiet konzentrieren und daher eher rudimentäre ergänzende Kenntnisse aufweisen. In Großunternehmen dagegen werden häufig bewusst Schnittstellen zwischen Fach- und IT-Abteilungen geschaffen, die sich leidlich gut in beiden Bereichen auskennen und daher als Ansprechpartner für IT-Experten dienen.

enger Zusammenarbeit sowieso früher oder später offenbaren). Auch das Vorgaukeln von Stärken ist eine nicht unübliche Praxis (und zuweilen durch harte Konkurrenz in den Marketing- und Vertriebsprozessen motiviert), ist nach unserer Erfahrung aber auch nicht zielführend, da sich das nachträgliche Aufdecken fehlender Stärken als sehr nachteilig für die gemeinsame Vertrauensbasis erweisen kann. Wir haben die Erfahrung gemacht, dass der offene Umgang mit Stärken und Schwächen auch nicht häufiger zu einem Verlust von Projekten führt als der beschönigende Umgang.

Desweiteren fordern wir:

> *Sei so früh wie möglich offen und ehrlich.*

Fachanwender sind häufig viel Leid aus gescheiterten Projekten aus der Vergangenheit gewohnt. Gar zu oft wurden Ihnen neue Technologien als Heilsbringer und Allheilmittel verkauft, nur um nach Monaten intensiver und anstrengender Bemühungen erkennen zu lassen, dass auch das neue Wundermittel nur ein altes Werkzeug in neuer Verpackung war. Entsprechend misstrauisch reagieren Fachanwender auf allzu euphorische Versprechungen. Unsere Erfahrung zeigt, dass es daher unbedingt Sinn macht, nicht immer nur Stärken und Vorzüge zu preisen sondern auch von Beginn an offen mit Schwächen und Problemen von Ansätzen umzugehen. Ein solch offener Umgang erzeugt Vertrauen und wirkt glaubwürdig (und ist es letztlich auch). Anwender reagieren darauf zwar vielleicht kurzfristig mit Schmunzeln und vielleicht Spott – mittel- und langfristig wissen sie diese Ehrlichkeit zu schätzen und verzeihen daher auch eher Probleme und Fehler. Wird ein solcher Umgang von vorne herein konsequent gepflegt, zeigt sich zumeist auch, dass auch terminliche Verzögerungen zwar als Ungemach aber nicht als Katastrophe gewertet werden. Insbesondere eskalieren Probleme nach unserer Erfahrung dadurch viel seltener und selbst in Eskalationsfällen kann eine einfachere Aufarbeitung stattfinden, da aufgrund einer offenen Projektkommunikation nach Auswegen gesucht werden kann.

Das Organisationsmuster *Take No Small Slips* aus [CoHa05] (S. 54–55) ist hier ebenfalls sehr hilfreich: Sollte es trotz aller Vorkehrungen zu Verzögerungen kommen, ist es besser, einmalig eine größere Terminverschiebung in Kauf zu nehmen als wiederholt zu knapp gesetzte Ausweichtermine nicht halten zu können. Letzteres untergräbt die eigene Glaubwürdigkeit ebenso wie die des Projektverantwortlichen auf Kundenseite, da auch er die Verzögerung gegenüber seinen Vorgesetzten kommunizieren muss.

6.4.2.4 Uniformität

K-3.41 ▷ 101

> *Behandle Gleiches gleich.*

Mit diesem Prinzip sprechen wir die Gleichbehandlung von Konzepten bei der Erstellung von Lösungsinstanzen an. So wie wir in Abschnitt 5.4.2.3 gefordert haben, dass Modell- und Werkzeugelemente, die Gleiches meinen, auch gleich benannt und behandelt werden sollten, gilt auch für konkrete Projekte, dass kundenspezifische Konzepte immer gleich benannt und behandelt werden sollten. Insbesondere bedeutet dies, sich auf die Sprache des Kunden einzulassen und mit ihm gemeinsam ein Vokabular abzustimmen, dass für das Projekt von Bedeutung ist (siehe auch Abschnitt 6.4.3.8).

K-3.21 ▷ 70

Zudem kann dies auch bedeuten, kundenspezifische Vokabeln zu verwenden, die von der Standard-Nomenklatur des CMDSD-Systems abweichen. Dies ist in jedem Fall besser als dem Kunden neue Begriffe beibringen zu wollen, da der erforderliche Umlernprozess normalerweise in der zur Verfügung stehenden Zeit nicht schnell vollzogen werden kann.

6.4.2.5 Kompetenz statt Komplexität

> *Sichere deine Existenz durch Kompetenz anstelle von Komplexität.*

Eine häufige Neigung von Experten sowohl auf IT- wie auch auf fachlicher Seite ist es, ihre Daseinsberechtigung durch eine Verklausulierung ihrer erbrachten Leistung abzusichern (siehe dazu unsere Ausführungen auf Seite 176). Dies kann aber nur eine kurzfristige Lösung sein – wird die eigene Kernkompetenz durch neue Entwicklungen überholt, hilft die mühsam aufgebaute Positionierung nicht weiter. Ganz im Gegenteil – die Befürworter neuer Ansätze werden tendenziell sogar eine schlechtere Meinung vom bisherigen Meinungsführer haben, da ihnen seine eher komplexen Darstellungen den Umstieg erschweren. Dadurch wird die Existenzberechtigung des bisherigen Leistungsinhabers zusätzlich in Zweifel gezogen und er hat sich so mglw. unbewusst in eine schwierige Position manövriert.

K-3.26 ▷ 74

Daher ist es unserer Erfahrung nach für beide Seiten besser, offen und kommunikativ auf möglichst einfache und unkomplizierte Weise Wissen bereit zu stellen und sich

K-3.2 ▷ 50

so zu einem geschätzten und wertvollen Ansprechpartner zu machen. Insbesondere hilft dies dem Betroffenen auch selbst, dabei mehr Problemkompetenz aufzubauen, da er in den Genuss eines intensiveren Austauschs mit Problemsuchenden kommt und so implizit den Wert seiner eigenen Kenntnisse stetig steigert.

Für Fachexperten bedeutet dieses Prinzip, dass sie sich darin üben müssen, ihre Kenntnisse anschaulich zu vermitteln. Für IT-Experten meint es häufig, dass diese sich intensiver mit den realen statt den vermeintlichen Anforderungen der betrachteten Domäne befassen müssen. Die so resultierende persönliche Weiterentwicklung sichert beiden Seiten, dass sie nicht durch neue Technologien überflüssig werden sondern ihre nachhaltigen Erfahrungen auch in einen kontinuierlichen Verbesserungsprozess einbringen können.

6.4.2.6 CMDSD-basiertes Vorgehen

> *Verwende ein geeignetes CMDSD-Werkzeug als Modellierungsgrundlage.*

Wir haben bereits in Kapitel 5 ausführlich dargestellt, welche Optimierungspotenziale die von uns spezifizierten CMDSD-Werkzeuge besitzen – dieses Prinzip hält noch einmal fest, dass die Möglichkeiten eines CMDSD-Werkzeugs im Rahmen des CDPM auch voll ausgeschöpft werden sollen. Insbesondere ist sicherzustellen, dass auch der Kunde die Möglichkeiten des Werkzeugs zumindest überblicksartig erfasst und die Grenzen versteht, da er seine Anforderungen dann besser auf die zur Verfügung stehende Plattform abstimmen kann. Dabei ist er natürlich von entsprechend geschulten Experten des Dienstleisters zu unterstützen.

6.4.2.7 Konsequenz

> *Wende konsequent alle Methoden des CDPM an. Beachte im Rahmen des Projektmanagements in allen Fällen die zugrunde liegenden Prinzipien.*

Der Wert der Wirtschaftlichkeit wird bereits durch das Prinzip der Agilität maximiert – Agilität hilft dabei, unvorhersehbare Projektrisiken in kleinen Portionen zu begegnen (siehe die Abschnitte 2.2.4 und 3.4.3). Wie dort erläutert erfordert Agilität

aber auch sehr viel Disziplin, da agile Vorgehensmodelle zwar sehr pragmatische und kommunikationsorientierte Prinzipien und Methoden forcieren, aber ebenso leicht an Effektivität verlieren, wenn diese nicht kontinuierlich und nachhaltig benutzt werden. Damit die von uns geforderten agilen Grundsätze also ihre Wirkung entfalten können, ist ein besonders konsequentes Vorgehen erforderlich.

Dieses Prinzip bezieht sich eigentlich auf alle Aspekte von Conceptual Programming. Insbesondere aber im operativen Projektgeschäft müssen die Potenziale von CP im Allgemeinen und CMDSD im Speziellen realisiert werden. Werden daher die nachfolgend beschriebenen Methoden nicht konsequent genutzt, wird sich auch nur eine begrenzte Effektivitätssteigerung erzielen lassen, die die Aufwände der Vorarbeiten in vielen Fällen nicht rechtfertigen werden.

6.4.2.8 Direktheit

Kommuniziere ohne Umwege.

Mit diesem Prinzip zielen wir darauf, eine möglichst direkte und unmittelbare Form der Kommunikation zu suchen. Persönliche Gespräche sind Telefonaten vorzuziehen, Telefonate sind besser als E-Mails, etc. Lebt man dieses Prinzip, zeigt sich schnell, dass so Missverständnisse leichter aus der Welt geräumt werden und explorative Prozesse deutlich an Geschwindigkeit gewinnen können.

K-3.22 ▷ 72
K-3.45 ▷ 109

So einfach dieses Prinzip zu sein scheint, so schwierig erweist es sich doch in der praktischen Umsetzung, da alle Beteiligten häufig gute Gründe finden, eher indirekt zu kommunizieren. Oft angeführte Gründe sind Nicht-Erreichbarkeit des Gegenübers, andere Ablenkungen aus dem Tagesgeschäft, etc. Wir empfehlen daher, einen entsprechenden Coach (z. B. den Projektleiter) initial über derartige Vorgänge wachen zu lassen und ihm Zeit zu geben, die anderen Projektmitarbeiter darin zu unterstützen, sich an eine möglichst direkte Kommunikation zu gewöhnen.

Wir weisen darauf hin, dass die direkte Form der Kommunikation die Beteiligten nicht davon entbindet, Ergebnisse geeignet zu dokumentieren (siehe dazu die in Abschnitt 6.4.3.7 beschriebene Methode ab Seite 276).

6.4.2.9 Explizivität und Transparenz

> *Spezifikationen sind nur dann gültig, wenn sie von Fachexperten verstanden und mit diesen abgestimmt sind.*

Da Fachanwender naturgemäß am Besten wissen, welche *fachlichen* Anforderungen umzusetzen sind, ist die Fachspezifikation ausschließlich gemeinsam zwischen Fachanwendern und Architekten bzw. Entwicklern zu erstellen. Als Grundlage können natürlich Vorarbeiten von beiden Seiten dienen – diese sind aber immer in gemeinsamen Sitzungen anhand des vorliegenden Fachmodells zu diskutieren, validieren und abzuzeichnen. Optimalerweise ist das Ergebnis solcher Abstimmungen eine mit dem CMDSD-Werkzeug erstellte Menge von Fachmodellen.

Diese Maxime zielt unmittelbar darauf ab, die in Studien der Standish Group als signifikant erkannten Probleme der unvollständigen und wechselnden Anforderungen sowie der unrealistischen Erwartungen zu beseitigen (siehe die Punkte 1, 4 und 6 aus Tabelle 2.4 auf Seite 26 sowie [KiRS04, Stan03b]).

Um dieses Prinzip erfolgreich umsetzen zu können, gilt zudem:

> *Entscheidungen müssen für alle Beteiligten transparent sein.*

Damit wollen wir dem häufig zu beobachtenden Problem der „heimlichen Dokumentation“ entgegenwirken, bei dem eine Partei in dem Projekt mutmaßliche Entscheidungen als gegeben annimmt und diese gar nicht oder für andere intransparent dokumentiert. Wir fordern daher, dass alle Entscheidungen allen Beteiligten kommuniziert werden müssen. Insbesondere müssen entsprechende schriftliche Dokumentationen für alle Betroffenen (bevorzugt in Form von Modellen oder Prototypen) leicht zugänglich sein.

6.4.2.10 Bedarfsorientierung

> *Die konsequente Erfüllung der Kundenanforderungen stellen das oberste Maß für die Qualität eines Systems dar.*

Letztlich geht es bei jedem Softwaresystem darum, bestimmte Kundenanforderungen umzusetzen. Häufig sind diese im Vorfeld eines Projekts nicht ausreichend detailliert worden und in vielen Projekten stellt sich spät heraus, dass auch auf Kundenseite noch eine erschreckende Unklarheit über das eigentliche Projektziel herrscht. CDPM geht an Projekte heran mit dem Anspruch, die Anforderungen des Kunden als zentrales Thema zu sehen. Anforderungen müssen explizit und der Status ihrer Bearbeitung muss in jeder Projektphase klar sein. Auf diese Weise kann eine wesentlich höhere Projektqualität erreicht und letztlich die Kundenzufriedenheit maximiert werden. Um diesem Anspruch gerecht zu werden, ist alles zu tun, um das Anforderungsmanagement zu einem integralen Bestandteil von CMDSD-Systemen zu machen. Häufig wird sich dabei aber zeigen, dass eine entsprechende technische Implementierung sehr komplex ist, wogegen organisatorische Vereinbarungen bzw. diszipliniertes Vorgehen (siehe auch Abschnitt 6.4.2.7) sehr gut funktionieren. Hier zahlt sich wiederum der Einsatz geeigneter agiler Methoden (siehe Abschnitt 3.4.3) aus. Wichtig ist dabei die Verwendung pragmatischer Ansätze: Insbesondere im Rahmen der Prototypisierung in den CPP-Phasen (siehe Abschnitt 6.4.3.5) fallen bislang nicht berücksichtigte Anforderungen schnell auf.

K-3.5 ▷ 57

Aktuelle Studien (so zum Beispiel [Bitp02, KiRS04, Stan03b]; siehe dazu auch die Ausführungen auf S. 25) belegen, dass viele Projekte scheitern oder einen nicht zufriedenstellenden Abschluss finden, weil die Anforderungsanalyse unterschätzt bzw. nicht konsequent genug durchgehalten wird. Als zentrale Faktoren sind zu nennen, dass

- Anforderungen unvollständig spezifiziert werden,
- die Kunden nicht ausreichend an der Erfassung der Anforderungen beteiligt waren,
- Kundenerwartungen aufgrund technischer Unkenntnis, unkonsequenter Kommunikation seitens des Anbieters oder Missverständnissen unrealistisch sind und
- Anforderungen sich während eines Projektes ändern.

Diese Probleme sind altbekannt und es gibt eine Vielzahl von Ansätzen und Empfehlungen, wie diesen Problemen zu begegnen ist (siehe z. B. [RoRo99, Schi02]). CDPM sieht in einem konsequenten Anforderungsmanagement kein Komplexitätsproblem, sondern eine Möglichkeit, die Kundenorientierung des Gesamtprozesses zu betonen, indem Kunden durch geeignete und nachfolgend beschriebene Methoden kontinuierlich in die Entwicklung einbezogen werden und so stetig Feedback geben können und

müssen.

Auf diese Weise wird die strategische Projektplanung unterstützt. Projektmanager wie -entwickler erhalten ein besseres Gefühl für das Gesamtziel, dass der Kunde verfolgt. Zudem können sie ihre Aktivitäten in den aktuellen Projektphasen auf strategische Ziele abstimmen und spätere Umsetzungsphasen somit frühzeitig vorbereiten. Die Kenntnis über zukünftig geplante Entwicklungen erlaubt es zum Beispiel, Datenstrukturen oder Bildschirmmasken bereits so zu planen, dass zukünftige Entwicklungen leichter möglich oder zumindest nicht behindert werden. Wichtig ist dabei, nicht jede mögliche Erweiterung berücksichtigen zu wollen. Dies ist im Normalfall ohnehin nicht möglich, da das sich stetig wandelnde Geschäftsumfeld langfristige Planungen nur schwer möglich macht.

Um strukturiert Anforderungen zu erheben, empfiehlt sich ein *Top-Down-Ansatz*: Zunächst sollte die strategische Zielrichtung und das allgemeine zu lösende Fachproblem ermittelt und dann erst die generellen Anforderungen stufenweise verfeinert werden. Anforderungen können – entweder direkt und gemeinsam mit dem Anwender – mit den Mitteln des zugrunde liegenden CMDSD-Systems im Rahmen eines Prototyps umgesetzt werden oder aber – sollten sich die Anforderungen als zu kompliziert erweisen – zunächst in textueller Form oder als Modellskizzen (siehe Abschnitt 3.3.9 zu agiler Modellierung) spezifiziert werden. Um Anforderungen möglichst hoher Qualität und Schärfe zu erhalten, empfiehlt sich die Verwendung offener Fragen gegenüber dem Kunden (siehe auch S. 111 in [KiRS04]):

- Was ist das Problem, welches gelöst werden soll?
- Warum ist das Problem zu lösen?
- Welche Organisationseinheiten sind von dem Problem betroffen?
- Welche Organisationseinheiten müssen in den Problemlösungsprozess einbezogen werden?
- Wo genau ist das Problem aufgetreten?
- Wie könnte das Problem gelöst werden?
- Womit wird das Problem zur Zeit – möglicherweise unzureichend oder unbefriedigend – umgangen?
- Welche quantitativen Aussagen können mit dem Problem verbunden werden? (Anzahl von Kunden, Größe von Dokumenten, Schnelligkeit der Bearbeitung, etc.)
- Wann muss das Problem unbedingt gelöst sein?

Die Standardliteratur zu Requirements Engineering liefert hier vielfältige weiterführende Hinweise (siehe z. B. [RoRo99, Rupp07]).

6.4.2.11 Min-Max

> *Realisiere immer nur das tatsächlich Benötigte.*

Hinter diesem Prinzip steht das (zwar offensichtliche aber erstaunlich oft nicht nachhaltig genug verfolgte) Prinzip, mit minimalem Aufwand maximalen Geschäftswert zu erzielen. In der Praxis neigen Fachanwender dazu, neue IT-Systeme nach dem Vorbild bereits existierender Lösungen zu entwerfen, wodurch sie sich neuen und besseren Ansätzen (zumeist unbeabsichtigt, manchmal aber, weil sie nichts Neues lernen wollen) verschließen. Analog neigen IT-Experten dazu, eine möglichst flexible und erweiterungsfähige Architektur zur sinnvollen Behandlung zukünftiger Anforderungen entwerfen zu wollen – wodurch sie all zu häufig für eine nicht eintretende Zukunft frühzeitig Zusatzaufwände erzeugen.

Fachmodellgetriebene Software-Entwicklung soll sich immer auf ein klar umrissendes Problemfeld beschränken, um die Beherrschbarkeit der erforderlichen Modelle sicherzustellen. Insbesondere sind Eigenschaften und Möglichkeiten, die aus technischer Sicht reizvoll sind oder zusätzliche Flexibilität für zukünftige Erweiterungen erlauben, aus den Fachmodellen konsequent herauszulassen, wenn es von Fachanwenderseite keinen unmittelbaren Anwendungsfall für die Nutzung dieser Konzepte gibt. So wird gewährleistet, dass die Projektziele überschaubar bleiben und von allen Beteiligten leicht verstanden werden können. Es reicht vollkommen aus, derartige Erweiterungsmöglichkeiten im Rahmen der Modellierung auf ihre prinzipielle Machbarkeit zu prüfen – sie müssen aber nicht sofort vorgenommen werden.

K-3.48 ▷ 114

6.4.2.12 Start small, then grow

> *Realisiere einen Geschäftswert so früh und schnell wie möglich.*

In Analogie zu unseren Ausführungen in Abschnitt 5.4.2.9 ab Seite 195 sollten auch Lösungsinstanzen so klein und unkompliziert wie möglich begonnen werden, damit sich möglichst schnell ein initialer Erfolg einstellt, indem bereits frühe Prototypen

einen realen Geschäftswert aufweisen. Dies führt zu einer grundlegend positiven Stimmung im Projekt, da sich bereits früh Erfolgserlebnisse einstellen, die dann auch kommuniziert werden können und bei nur indirekt Beteiligten (z. B. dem Management oder späteren Anwendern) zu einer positiven Grundhaltung führen. Zudem nehmen frühe Erfolge die Angst auf Seiten der Fachanwender vor neuen Technologien und Anwendungen, was dann bei der tatsächlichen Einführung der Lösung zu einer konstruktiveren und zielführenderen Atmosphäre führt.

K-3.25 ▷ 73
K-3.31 ▷ 81
K-3.53 ▷ 125

Außerdem lassen sich weiterführende Anforderungen besser anhand eines frühzeitig funktionierenden Prototyps beurteilen und Fachanwender werden sich leichter damit tun, komplexere Anforderungen anhand eines Beispielsystems zu verfeinern als dies bei einer reinen dokumentationsorientierten Anforderungsanalyse.

6.4.3 Methoden

Die nachfolgend beschriebenen Methoden für das CDPM fußen grundlegend auf den Erkenntnissen aus Abschnitt 3.4. Dabei wird insbesondere auf den Eigenschaften in-

K-3.54 ▷ 127

krementeller, iterativer, rekursiver und evolutionärer Vorgehensmodelle aufgebaut, was sich inhärent in der Struktur der CDPM-Modellierungsphasen widerspiegelt (siehe dazu die Ausführungen in Abschnitt 3.4.1.3). Die Grundlagen für unser methodisches Vorgehensmodell basieren auf den Erkenntnissen aus [BoTu03b], in denen es um ein effektives und effizientes Gleichgewicht aus Agilität und Disziplin geht. Der inhärente Wert von Disziplin ist in einem planvollen und strukturierten Vorgehen zu sehen[6]. Ebenso unstrittig ist die Notwendigkeit, den Grad der Planung und Vorstrukturierung nicht zu übertreiben[7]. Hier bietet ein inkrementeller, iterativer,

K-3.54 ▷ 127

rekursiver und evolutionärer Grundprozess einen exzellenten Kompromiss zwischen Agilität und Disziplin, da er einerseits stark strukturiert ist, andererseits die Annäherung an das Ziel in kleinen Schritten ermöglicht.

K-3.49 ▷ 115

CDPM verfügt sowohl über eine Mikroebene als auch über eine Makroebene. Die Mikroebene des Vorgehensmodells ist wie in Abbildung 6.2 dargestellt strukturiert

[6] Ein solch konsequentes Vorgehen schafft z. B. die Möglichkeit, Prozesse und Organisationen stetig in ihrer Leistungsfähigkeit zu verbessern sowie die korrekte Arbeitsweise zu verifizieren, siehe [BoTu03b], S. 12–13.

[7] Siehe dazu wiederum [BoTu03b], S. 13: *„Of course, planning can also cause problems. If too strictly applied, plans and process can impede innovation or lead to mechanical, checklist mentality, where the object of the endeavor becomes so focused on the process that the product **(often along with the customer)** is afforded secondary status.“* Fett hervorgehoben haben wir die Stelle, die wir für besonders beachtenswert halten, da sie das grundlegende CP-Prinzip der Kundenorientierung (siehe Abschnitt 4.2.2.1 ab S. 152 adressiert.

und beschreibt, wie einzelne Modelle oder zusammenhängende Modellgruppen erstellt werden. Die einzelnen Phasen des in Abbildung 6.2 dargestellten Mikromodells haben die folgende Bedeutung:

- In einem kurzen *Kickoff* werden das Thema des aktuellen Rapid-Development-Schritts (siehe dazu Abschnitt 6.4.3.4 ab Seite 271) sowie Ideen zu seiner Struktur und Bearbeitung besprochen.
- In einer *Analyse* werden sowohl existierende Altsysteme, das mglw. bereits in Teilen oder als Prototyp existierende neue System sowie Vorarbeiten und -erfahrungen auf Kunden- und Dienstleisterseite zum Thema des aktuellen Schritts gesammelt und strukturiert.
- Falls bereits ein Prototyp oder ein neues Teilsystem existiert, werden ggf. *Refactoring*-Operationen geplant, die sich aus den vorzunehmenden Erweiterungen ergeben.
- In einer anschließenden *Rapid Development*-Phase werden die geplanten Erweiterungen detaillierter modelliert und dann durch *Conceptual Pair Programming* (siehe Abschnitt 6.4.3.5 ab Seite 273) realisiert.
- Die Implementierung wird anschließend *validiert*. In Abhängigkeit von der spezifischen Domäne und der betrachteten Teildomäne sind hier sehr unterschiedliche Herangehensweisen und Aufwände denkbar. Beispielhaft seien die folgenden Herangehensweisen genannt:
  - Die Validierung erfolgt unmittelbar im Anschluss an das Conceptual Pair Programming, indem die anwesenden Domänenexperten den Prototypen beispielhaft anzuwenden. Dies bietet sich insbesondere für kleine Teilprobleme und die Lösung nur lokal relevanter Fragestellungen an (wenn also davon ausgegangen werden kann, dass die anwesenden Domänenexperten bereits einen Großteil der späteren Nutzerbasis ausmachen).
  - Die Validierung wird durch eine ausgewählte Fachabteilung (oder Teile davon) vorgenommen. Man spricht hier von einem *Test- oder Pilotbetrieb*. Dieses Vorgehen bietet sich bei komplexeren Lösungen oder Implementierungen an, die eine Reihe von Fachabteilungen berühren. Pilotanwender können dann zunächst mit geringem Aufwand geschult werden und Erfahrungen in der Nutzung der neuen Lösung sammeln. Feedback wird zeitnah gesammelt und in weiteren Rapid-Development-Phasen in das System eingearbeitet.
  - Die Validierung folgt einem formalen und dokumentationsorientierten

Prozess. Dieses Vorgehen wird insbesondere dann erforderlich sein, wenn die betrachtete Domäne speziellen gesetzlichen Auflagen unterliegt.

Selbstverständlich sind auch Kombinationen dieser Herangehensweisen denkbar.

Für kleine Projekte ist es absolut ausreichend, nur mit der Mikroebene zu arbeiten. Unsere Erfahrung zeigt, das Projekte mit nur einem zentralen System oder mehreren Teilsystemen aber einer geschätzten Gesamtgröße von ca. 30 Personentagen allein mit der Mikroebene von CDPM arbeiten sollten, um den Verwaltungs- und Strukturierungsoverhead der Makroebene zu vermeiden.

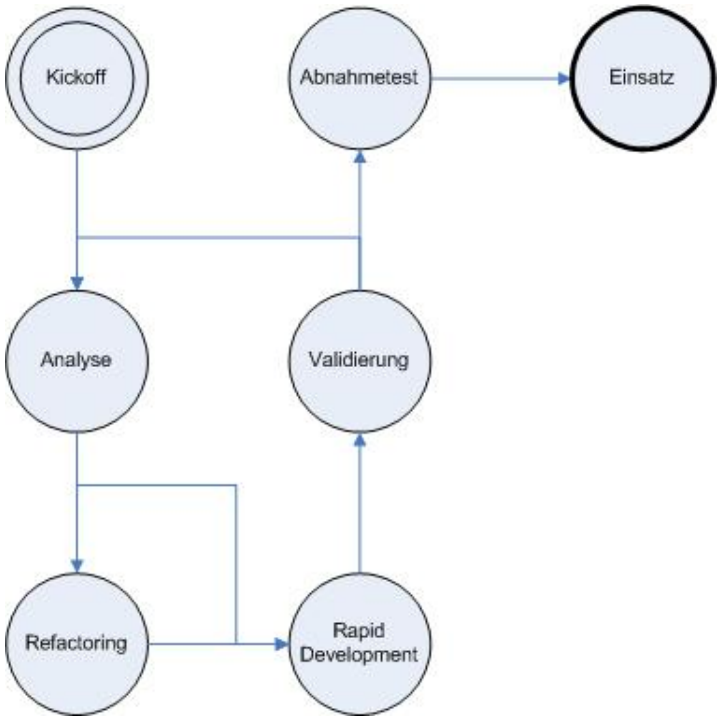


Abbildung 6.2: Mikroebene des CDPM

Die Makroebene von CDPM hat zwei essentielle Komponenten:

1. Zum einen legt sie die spezifische Projektmanagementmethode fest, für die wir die bereits in diesem Kapitel beschriebenen Prinzipien sowie die nachfolgenden Methoden verlangen.
2. Zum anderen beschreibt sie die prinzipielle Vorgehensweise (also insbesondere die Reihenfolge), in der Modelle erstellt werden. Diese Reihenfolge ergibt sich jeweils aus der betrachteten Domäne. Wir empfehlen eine Top-Down-Vorgehensweise (siehe dazu unserer Ausführungen auf Seite 260).

Darüber hinaus erfordert die Makroebene von CDPM keine erweiterten Definitionen. Hier hat sich die Verwendung eines klassischen inkrementellen, iterativen, rekursiven und evolutionären Vorgehensmodells bewährt, wobei jedes Inkrement gemäß des zuvor beschriebenen Mikromodells entwickelt wird. Diese Kombination betont die agilen Vorteile der Mikroebene und kombiniert sie mit den Stärken der Familie der inkrementellen, iterativen, rekurisven und evolutionären Vorgehensmodelle (siehe Abschnitt 3.4.1.3), um so auch für größere Projekte zu skalieren.

Abbildung 6.3 zeigt das Zusammenspiel von Mikro- und Makroebene im Concept Driven Project Management.

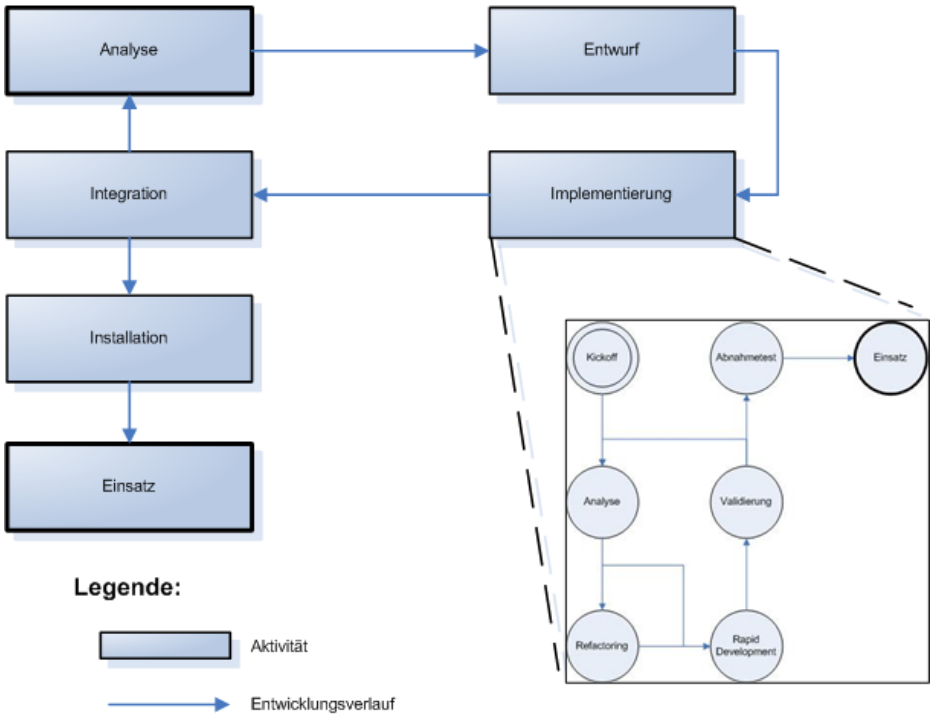


Abbildung 6.3: Makroebene des Concept Driven Project Management

Nachfolgend stellen wir die einzelnen CDPM-Methoden detaillierter vor und verweisen noch einmal auf Abschnitt 6.3 auf Seite 245 für einen Gesamtüberblick über die Anwendung der Methoden.

6.4.3.1 Philosophischer Projekt-Kickoff

> *Führe zu Beginn des Projekts die CP-Systematik, die Werkzeuge und die verwendeten Modelle ein, so dass eine gemeinsame Basis für das weitere Vorgehen existiert.*

Ziel des philosophischen Projekt-Kickoffs ist es, allen Beteiligten die besonderen Notwendigkeiten (insbesondere eine aktive Beteiligung und Mitverantwortung am Erfolg) klar zu machen und diese Erkenntnisse in die Gesamtmethodik einzubetten. Insofern werden bei einem philosophischen Projektkickoff üblicherweise die folgenden Informationen vermittelt:

- Überblick über Conceptual Programming
- Kurzvorstellung der spezifischen CMDSD-Ausprägung (insbesondere der zu verwendenden Werkzeuge)
- Überblick über die spezifische methodische Projektvorgehensweise (also CDPM) als Hauptteil des Kickoffs mit besonderer Betonung der Vorteile und Stärken der spezifischen Vorgehensweise, aber auch Hervorhebung der besonderen Anforderungen an aktive Beteiligung und Disziplin der Projektbeteiligten (also Einschwörung auf das Verantwortungsbewusstein und die persönliche Mitverantwortung am Projekterfolg)
- Gemeinsamer Beschluss zur Einhaltung des vorgestellten Vorgehens, um alle Beteiligten zu einem gemeinsam verantwortlichen Team zu formen

Je nach Branche, Unternehmen und Teilnehmerkreis ist die sinnvollste Vermittlungsform zu finden – je praxisnäher die Vermittlung gelingt, desto wahrscheinlicher ist ein Erkenntnisgewinn. Insofern bieten sich insbesondere Beispiele, Rollenspiele und Fallstudien als Ergänzung zum reinen Vortrag an, wobei Meinungsträger und Entscheidungsfinder unbedingt einbezogen werden sollten.

Der Kickoff findet üblicherweise in Form eines Meetings mit allen Beteiligten statt, bei dem die gemeinsamen Ziele noch mal explizit ausgesprochen und das weitere Vorgehen vorgestellt wird. Der Kickoff-Termin sollte möglichst kurz gehalten werden. Sobald wie möglich im Anschluss sollten die konkreten Projektarbeiten beginnen. Ein typischer Kickoff-Termin sollte nicht länger als einen halben Tag dauern, häufig reichen auch ein bis zwei Stunden. Bei kurzen Kickoffs kann der Kickoff auch mit dem CP-Coaching kombiniert werden (siehe dazu Abschnitt 6.4.3.2). Dies hat

häufig den Effekt, dass die neuen Ansätze und Vorgehensweisen für eine positive Aufbruchstimmung bei den Kunden sorgen. Diese positive Energie wirkt sich nach unserer Erfahrung nachhaltig auf den Erfolgswillen in den ersten Arbeitssitzungen aus. Ziel muss es im weiteren Verlauf sein, diese positive Grundhaltung durch schnell vorzeigbare Prototypen zu erhalten, da so die Experten auf Kundenseite zur aktiven und maximal produktiven Mitarbeit motiviert werden – denn sie erleben direkt, wie ihre Ideen und Anregungen in Software umgesetzt werden.

Auch hier greifen wir wieder Erkenntnisse zur sinnvollen Kombination von Agilität und Disziplin auf. Zum einen erzwingen wir die bewusste aktive Integration aller Projektbeteiligten, so dass es in problematischen Phasen mental schwerer wird, sich aus der Verantwortung zu entziehen und fördern so die in [BoTu03b], S. 16 formulierte Erkenntnis zur Notwendigkeit aktiver Beteiligung *„Processes cannot work without the active support of the participants“* als auch die Notwendigkeit, Verantwortungsbewusstein einzufordern (wiederum aus [BoTu03b], S. 16): *„. . . Just like management, they must realize that the process should be given a certain amount of allegiance or else the entire strategy may fail.“*

K-3.2 ▷ 50
K-3.38 ▷ 91

6.4.3.2 CP-Coaching

> *Fördere realistische Erwartungen an das konkrete Projekt durch einen gleichmäßigen Informationsstand über die Grenzen, Möglichkeiten und besonderen Anforderungen der eingesetzten Methoden.*

CP-basierte Projekte unterscheiden sich aufgrund ihrer hohen fachlichen Fokussierung und des Bestrebens nach konsequentem Rapid Prototyping maßgeblich von klassischen Software-Projekten, in denen die Fachexperten nach Erstellung einer (zumeist rein textuellen) Anforderungsdefinition nur in unregelmäßigen Abständen Kontakt zum umsetzenden Projektteam haben. Da der Nutzen von CP aber auf einer extrem engen und offenen Zusammenarbeit zwischen technischen und fachlichen Experten beruht und die fachliche Sicht der Domäne zentral für das gesamte Projekt ist, ist es nach unserer Erfahrung besonders wichtig, die daraus resultierende veränderte Form des Umgangs miteinder im Projekt sowie die daraus resultierenden andersartigen Anforderungen an die fachlichen wie technischen Projektmitarbeiter konsequent zu kommunizieren.

Abhängig von der Größe des Projektteams bietet sich hier ein ein- oder zweitägiger gemeinsamer Workshop der zentralen Personen des Projektteams (oder bei kleineren

Projektteams sogar aller Projektmitglieder an), in denen die Rahmenbedingungen des Projektes kommuniziert werden. Insbesondere sind hier die folgenden Themen hervorzuheben:

- *Grundidee des CP-Ansatzes:* Fokussierung auf fachliche Themen und somit eine zentrale Rolle der Fachexperten. Es ist zu betonen, dass der Projekterfolg unmittelbar von der Qualtität der Zusammenarbeit zwischen den Fachexperten und den technischen Experten abhängt. Psychologisch gesehen macht es hier häufig Sinn, gegenüber den Fachexperten auch die Abhängigkeit der Techniker von dieser Zusammenarbeit zu betonen: Die technischen Experten können nur dann erfolgreich sein, wenn sie eng mit den Fachexpxerten zusammenarbeiten und von diesen in alle fachlichen Zusammenhänge eingewiesen werden. Dabei gilt es insbesondere Hemmschwellen abzubauen, um frühzeitig Kommunikationsproblemen vorzubeugen, wobei die folgenden nicht selbstverständlichen Punkte allen Projektteilnehmern gegenüber zu betonen sind:
  - *Es gibt keine dummen Fragen.* Der Erfolg der technischen Umsetzung hängt am korrekten Verständnis der komplexen fachlichen Zusammenhänge. Insofern muss es beiden Seiten ausdrücklich erlaubt sein, Fragen wiederholt zu stellen, bis die Zusammenhänge tatsächlich klar sind. Durch die Verwendung geeigneter Werkzeuge (z. B. eines agilen Glossars, siehe Abschnitt 6.4.3.8) werden dabei beidseitige Reibungverluste verringert.
  - *Es gibt keine "heiligen Kühe".* Fachliche Anwender mögen es gewohnt sein, dass „gewisse Dinge schon immer so gehandhabt wurden". Technische Anwender mögen es gewohnt sein, dass „gewisse Dinge immer so implementiert werden". Beide Seiten sind ausdrücklich dazu angehalten, die Sinnhaftigkeit solcher Traditionen zu hinterfragen, um möglichst umfassende Verbesserungen aller Aspekte eines Systems zu erreichen. Dies darf nicht in endlose Diskussionen ausarten und Festlegungen müssen letztlich zeitnah durch einen entsprechenden Architekturverantwortlichen geregelt werden. Neue Erkenntnisse und Überlegungen zu weitreicherenden Verbesserungen können entsprechende Festlegungen aber wieder in Frage stellen. Zu beachten ist, dass entsprechende Umwälzungen eher in frühen Projektphasen stattfinden können und sollten – späte Änderungen der Spezifikation oder Gesamtarchitektur werden für gewöhnlich zu erheblichen Zusatzkosten und Projektverzögerungen führen.
  - *Vorstellung des CMDSD-Manifests (siehe Abschnitt 5.2).* Ein Einblick in die Grundidee von CMDSD hilft erfahrungsgemäß beiden Seiten, die

Sinnhaftigkeit und Motivation bestimmter Vorgehensweisen besser zu verstehen und zu verinnerlichen.

- *Vorstellung des verwendeten Prozessmodells:* Hier wird in einem kurzen Überblick die konkrete Projektmanagementmethode mit den CDPM-spezifischen Ausprägungen und Anpassungen vorgestellt.
- *Vorstellung der verwendeten Werkzeuge:* Es ist insbesondere für Fachexperten wichtig, ein Gefühl und einen Eindruck von den Grenzen und Möglichkeiten der im Projekt zur Anwendung kommenden Werkzeuge zu erhalten. Hier geht es nicht um eine marketing-orientierte (und somit zuweilen zu optimistische) Darstellung der Möglichkeiten eines Werkzeugs, sondern um eine realistische, mit konkreten Erfahrungen und Empfehlungen angereicherte Sicht auf den vorhandenen Werkzeugkasten. Dabei sind insbesondere auch die Grenzen und Einschränkungen zu betonen, um keine falschen oder unrealistischen Erwartungen an das Projekt zu wecken. Letztere führen erfahrungsmäß in späteren Projektphasen zu einer erheblichen Ernüchterung, die die Projektkommunikation massiv belasten kann, da sich die Fachexperten von den Versprechungen der Technik womöglich hinter das Licht geführt fühlen und bei Nichterfüllung dieser Erwartungen dann kontraproduktiv und sogar aktiv behindernd auf das Projekt einwirken. Letztlich steigt auch die Glaubwürdigkeit von Beratern, wenn diese die Welt nicht immer nur in rosigen Farben darstellen – auch Kunden habe schon lange erkannt, dass eine große Lücke zwischen Marketing und Realität klafft.

Diese Vorgehensweise dient außerdem dazu, Erkenntnisse aus dem Bestreben nach einer effektiven Balance zwischen Agilität und Disziplin Rechnung zu tragen. So formulieren Boehm und Turner in [BoTu03b], S. 16 *„Practitioners must be well trained in both the practice and philosophy of the plan-driven approach“*. Aufgrund der ungewohnt aktiven Rolle der Fachexperten in CP im Allgemeinem und im CDPM im Besonderen machen wir dieses Training also zu einem formalen Bestandteil des CDPM-Prozesses.

6.4.3.3 Analyse durch Design

Analysiere die Domäne durch den Entwurf konkreter Systeme und die Modellierung ihres Zusammenspiels.

Dieses Vorgehen basiert auf dem von Tayler vorgeschlagenen „Analysis by Design"-Ansatz (siehe dazu [Tayl95], S. 148–152). Die Grundidee ist dabei – mit großer Ähnlichkeit zu den postulierten Zielen von RAD-Anhängern[8] – dass es Fachanwendern leichter fällt, Anforderungen zu beschreiben und fachliche Zusammenhänge darzustellen, wenn Sie in die Lage versetzt werden, dies anhand möglichst greifbarer Objekte und Systeme zu tun.

K-3.25 ▷ 73
K-3.31 ▷ 81

Wie bereits in Abschnitt 5.4.2.10 ausgeführt, hat Rapid Prototyping geradezu spektakuläre Erfolge insbesondere in Projekten gefeiert, die von kleinen Teams oder Einzelpersonen abgewickelt werden (siehe [Tayl95], S. 146–147). Insofern ist ein essentielles Grundprinzip für alle CP-basierten Projekte, so frühzeitig und so oft wie möglich zu Rapid Prototyping zu greifen, um Domänenexperten die Auswirkungen der definierten Modelle zu illustrieren und so die Verifikation der Modelle nachhaltig zu unterstützen.

Zu Bedenken ist, dass es bei der Entwicklung eines komplexen Prototyps mit CMDSD-Werkzeugen immer einen Punkt geben wird, an dem der erste Entwurf den realen Anforderungen nicht mehr genügt: Sei es, weil man einen eleganteren Weg der Modellierung findet oder neue Anforderungen aufkommen, die initialen Anforderungen widersprechen. Dann gilt wie für das klassische Rapid Prototyping: Man verwirft den Prototypen und beginnt auf Grundlage der bislang gesammelten Erfahrungen mit einem neuen Entwurf. Da sich CMDSD von vorne herein auf die fachliche Ebene konzentriert, wird auffallen, dass spätere Prototypen leichter und effektiver zu entwerfen sind als bei klassischen Rapid Prototyping.

Man sollte Rapid Prototyping also immer mit der Erwartung betreiben, den Prototypen wegzuwerfen. Insbesondere ist es wichtig, dass Manager und Budgetverantwortliche ebenso wie alle Kunden diesen Punkt in seiner vollen Bedeutung verstehen, da sonst unrealistische Erwartungen bezüglich des Fertigstellungszeitpunktes geweckt werden (siehe dazu auch [CoHa05], S. 51).

Positiv an CDPM-basiertem Rapid Prototyping ist, dass eine erfolgreiche und vollständige Abbildung der fachlichen Anforderungen im Prototyp üblicherweise auch zu einer fast fertigen Anwendungskomponente führt. Dies ist dadurch bedingt, dass klassische Rapid-Prototyping-Probleme wie Stabilität, Laufzeitverhalten oder Skalierbarkeit durch die zugrunde liegenden CMDSD-Komponenten weitgehend berücksichtigt werden. Das CMDSD-Werkzeug generiert ein System, das seinem Modell genügt, so dass es kaum technische Unterschiede zwischen einem prototypischen

[8] Siehe dazu Abschnitt 3.2.4.

und einem endgültigen Modell gibt[9].

Praktisch wird dieses Vorgehen insbesondere auch bei der Methode des Conceptual Pair Programmings (siehe Abschnitt 6.4.3.5) angewendet.

6.4.3.4 Rapid Development

> *Entwickle Prototypen durch eine Kombination aus* Rapid Modeling *und* Conceptual Pair Programming *weiter.*

Auf Grundlage eines der in Kapitel 5 beschriebenen CMDSD-Werkzeuge werden die eigentlichen Entwicklungsarbeiten im Rahmen des CDPM in zwei Phasen realisiert. Diese sind:

1. Rapid Modeling
2. Conceptual Pair Programming

Hauptziel ist dabei, möglichst schnell zu einem Erkenntnisstand zu gelangen, der es erlaubt, die beschriebenen Anforderungen prototypisch umzusetzen, um sie dann anhand von Prototypen solange zu verfeinern, bis eine akzeptable Lösung entsteht. Dabei kann die gesamte Entwicklung anhand eines Prototypen erfolgen oder es können eine Reihe von Prototypen aufeinander aufbauen.

In einer freien Modellierungsphase – dem sogenannten *Rapid Modeling* – müssen zunächst die domänenspezifischen Konzepte, die im Rahmen einer Problembeschreibung von Bedeutung sind, unter Verwendung von Agile-Modeling-Methoden abgegrenzt und miteinander verbunden werden (siehe auch Abschnitt 3.3.9). Hier arbeiten Domänen- und CMDSD-Experten eng zusammen, um die Anforderungen zu umreißen. Ziel ist dabei nicht, eine vollständige Lösung zu erarbeiten, sondern genug Informationen zu haben, um zeitnah in die zweite Entwicklungsphase, das nachfolgend beschriebene Rapid Development, einzusteigen. Es ist zu beachten, dass bei gutem Domänenverständnis auch häufig direkt mit dem Rapid Development begonnen werden kann. Lediglich bei noch stark unstrukturierten Anforderungen macht

K-3.52 ▷ 120

K-3.2 ▷ 50
K-3.38 ▷ 91

[9] Wesentliche Unterschiede ergeben sich insbesondere durch die Verwendung produktiver Datenmengen, die deutlich umfangreicher sein können als in einem Prototyp. Auch hier empfiehlt es sich, den Prototypen frühzeitig zu Testzwecken mit realen Datenmengen zu befüllen und das daraus resultierende Laufzeitverhalten zu analysieren. So können auch Fachanwender vergleichsweise schnell eine Aussage dazu treffen, ob das Antwortverhalten des Systems ihren Anforderungen genügt oder spezifische Szenarien gezielt zu optimieren sind.

es Sinn, zunächst die Anforderungen über reines Agile Modeling einzugrenzen, da die dort beschriebenen Methoden häufig effektiver sind als die Arbeit mit einem Computerprogramm, wenn es darum geht, Ideen schnell aufzunehmen und weiterzuentwickeln.

Der zweite Schritt beinhaltet die eigentliche Entwicklung der Lösung. Dazu werden normalerweise zunächst unter Verwendung des CMDSD-Werkzeugs Prototypen entwickelt. Zentrale Methode ist hier das sogenannte *Conceptual Pair Programming*, dem wir einen eigenen Abschnitt widmen (siehe Abschnitt 6.4.3.5 ab Seite 273).

Sind Integrationsszenarien abzubilden, bietet es sich häufig an, zunächst ein idealisiertes System mit simulierten Schnittstellen aufzubauen und dann erst die Legacy-Systeme zu integrieren. Dabei ist Wert auf eine saubere Domänenmodellierung zu legen, ohne sich mit Altlasten aus den existierenden Datenstrukturen zu befassen. Diese verhindern Optimierungsversuche und belasten neue Systeme durch Annahmen und Architekturen, die womöglich gar keine Bedeutung mehr haben. Entwirft man zunächst ein neues Domänenmodell, können dann die Integrationsszenarien und -schnittstellen in weiteren Entwicklungsschritten ausgearbeitet werden (siehe auch [Tayl95], S. 66).

K-3.19 ▷ 66
K-3.22 ▷ 72
K-3.54 ▷ 127

Die von uns vorgestellte Form des Rapid Development greift also auch das bewährte „Analyse durch Design“ Prinzip des Convergent Engineering auf (siehe Abschnitt 6.4.3.3 bzw. [Tayl95], S. 151ff.).

Die Beachtung der folgenden Hinweise fördert die erfolgreiche Durchführung von Rapid-Development-Phasen (siehe auch [Tayl95], S. 68ff.):

- Ziele für eine Sitzung sollten ambitioniert, aber realistisch sein. Um die Akzeptanz der agilen Vorgehensweise zu erhöhen, sollten in den initialen Entwicklungsphasen die besonderen Stärken des jeweiligen CMDSD-Werkzeugs ausgespielt werden, um die Begeisterung und Kooperationsbereitschaft auf Kundenseite zu maximieren.
- Es ist wichtig, sich nicht in Details zu verstricken. Zunächst ist das Grobmodell zu definieren, dann kann das Gesamtmodell durch schrittweise Verfeinerung erstellt werden. Wie beim *Convergent Engineering* ist es von Vorteil, technische Details und insbesondere programmiersprachenspezifische Fragestellungen auszublenden.
- Jedes Mitglied im Team ist gleichranging zu behandeln. Das gilt insbesondere für Workshops. Auf diese Weise wird die Produktivität maximiert. Rangunterschiede aufzubrechen ist nicht immer einfach, so dass auch hier wieder

die Unterstützung von Managementseite des Kunden großer Bedeutung zukommt. Insbesondere ist vorgelebte Gleichberechtigung durch das Management in Workshops von großem Wert.

6.4.3.5 Conceptual Pair Programming

K-3.2 ▷ 50
K-3.31 ▷ 81
K-3.38 ▷ 91

> *Entwickle Anforderungen und Lösungen durch explorative Prototypisierung in Paaren aus Domänen- und CMDSD-Experten. Entwickle so gewonnene Prototypen zu produktiven Systemen.*

Ziel des Conceptual Pair Programming ist es, dass Fachexperten zusammen mit IT-Experten (die exzellente Kenntnisse über das zur Modellierung verwendete CMDSD-Werkzeug besitzen müssen) gemeinsam Anforderungen und die dazugehörigen Lösungen durch iterative Prototypisierung auf Modellbasis erarbeiten. Eric Evans sagt dazu in [Evan04a], S. 62–63: *„Any technical person contributing to the model must spend some time touching the code, whatever primary role he or she plays on the project. Anyone responsible for changing code must learn to express a model through the code“* und verstärkt dies noch, indem er in [Evan04a], S. 62 fordert: *„Every developer must be involved in some level of discussion about the model and have contact with domain experts.“* Wir nehmen dies zum Anlass, Conceptual Pair Programming als verbindliche Methode des CDPM zu definieren: IT-Experten werden kontinuierlich mit Domänenexperten an der Entwicklung der Lösungen arbeiten, die Erhebung und prototypische Modellierung von Anforderungen wird ein essentieller Bestandteil des operativen Projektablaufs.

Durch diese Vorgehensweise maximieren wir zudem die auch in allgemeineren agilen Vorgehensmodellen als essentielle Notwendigkeit erkannte enge Zusammenarbeit zwischen Entwicklern und Kunden (siehe [BoTu03b], S. 18–19): *„Agile methods universally need close relationships with the customer and users of the systems under development. Requirements and validation are dependent on the customers' describing, prioritizing, and refining their needs throughout the development. The cycle of building functionality, obtaining feedback, and evolving the system based on that feedback is highly dependent on an informed and involved customer. The customer also establishes acceptance criteria and tests.“* Conceptual Pair Programming erzeugt eine direkte Beteiligung an weiten Teilen der Entwicklung (durch unmittelbare Modellierungssitzungen zusammen mit einem CMDSD-Experten und direkte Begutachtung

so entwickelte Prototypen), so dass der Kommunikationsweg minimiert wird. Zudem tragen wir durch diese Vorgehensweise dem Kommunikationsverlust pro Kommunikationsschritt Rechnung, indem wir eine unmittelbare Kommunikationslinie schaffen. Das zugrunde liegende Problem haben wir in Abschnitt 2.2.1 ab S. 28 sowie in Abbildung 2.6 dargestellt.

Zudem systematisieren wir damit einen Effekt, der bei der Verwendung von Software-Produktlinien als sehr nützlich erkannt wurde (siehe [ClNo02], S. 22, unten): *„We have seen situations in which customers of products in the product line sit on the configuration control board for the core assets“*. Der Kunde wird in die Lage versetzt, direkten und ganz unmittelbaren Einfluss auf die von ihm gewünschten Lösungen zu nehmen. Gleichzeitig wird er dadurch auch stärker in die Verantwortung genommen. Erfahrungswerte aus unseren Projekten weisen diese Vorgehensweise nahezu durchgängig als nutzbringend aus: Je stärker sich ein Kunde für eine Lösung verantwortlich fühlt, desto mehr wird er sich engagieren und auch alle verfügbare Zeit und Energie einbringen, um das Projekt zu einem maximalen Erfolg zu führen. Dies wirkt sich auch sehr positiv auf das Arbeitsklima aus, da jegliches inhärente Technik- oder Methodenmisstrauen durch die unmittelbare Möglichkeit zum Sammeln von Erfahrungen deutlich reduziert wird. Zudem setzt dies Konzept 3.20 ▷ 67 um, da der Kunde durch den engen Bezug zum Entwicklungsprozess verstärkt dazu motiviert wird, eigene Ideen und Ansätze aktiv einzubringen – insbesondere ist diese Form der Motivation wesentlich stärker als wenn Kunden nur in dokumentationsorientierte (und damit eher theoretische und entsprechend trockene) Tätigkeiten einbezogen werden.

K-3.20 ▷ 67

Auch tragen wir Erfahrungen aus dem Domain Driven Design (siehe [Evan04a], S. 60–62) Rechnung – dort gesammelte Erfahrungen zeigen, wie wichtig es ist, sogenannte *Hands-On Modelers* in Projekten einzusetzen. Eric Evans meint damit insbesondere (siehe [Evan04a], S. 61): *„If the people who write the code do not feel responsible for the model, or don't understand how to make the model work for an application, then the model has nothing to do with the software. If developers don't realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it.“* Durch den Einsatz von CMDSD-Werkzeugen (siehe dazu unsere Ausführungen in Kapitel 5) haben wir bereits erreicht, dass in unseren zugrunde liegenden Werkzeugen Modelle die nahezu ausschließliche und universelle Sprache für die Entwicklung domänenspezifischer Lösungen sind. Durch das hier beschriebene *Conceptual Pair Programming* verstärken wir diesen von Evans geforderten Effekt nicht nur, indem wir quasi zwangsweise ausschließlich *Hands-On Modeler* in Projek-

ten einsetzen sondern zudem noch einen maximal aktiven Austausch zwischen den Domänen- und IT-Experten forcieren: Denn diese kommen aufgrund des Conceptual Pair Programmings regelmäßig im Projektverlauf zusammen, um gemeinsam zu modellieren und damit direkt auch zu programmieren. Zudem teilen sie aufgrund der Rapid-Prototyping-Funktionalitäten der Werkzeuge alle Erfolgs- und Miserfolgserlebnisse, was zu einer Stärkung der Teambindung führt. Gleichzeitig wirken wir den von Evans beschriebenen klassischen Problemen modellgetriebener Ansätze entgegen (siehe wiederum [Evan04a], S. 61): *„Meanwhile, when a modeler is separated from the implementation process, he or she never acquires, or quickly loses, a feel for the constraints of implementation. The basic constraint of* MODEL-DRIVEN DESIGN *– that the model supports an effective implementation and abstracts key domain knowledge – is half-gone, and the resulting models will be impractical. Finally, the knowledge and skills of experienced designers won't be transferred to other developers if the division of labor prevents the kind of collaboration that conveys the subleties of coding a* MODEL-DRIVEN DESIGN.*“* Entwickler, Domänenexperten und Modellierer rücken zusammen, da sie an allen diesen Aufgaben beteiligt sind.

Insbesondere tragen wir hier im Operativen den Erkenntnissen Rechnung, die wir gewonnen haben, als wir Rapid Prototyping als extrem effizientes Mittel zur explorativen Spezifikation von Anforderungen identifiziert haben (siehe Abschnitt 5.4.2.10 ab S. 195). Das maßgeblich anschauliche Zitat stammte von S. 81 aus [Mart81]: *„I don't know what I want until I see it.“*, das aus Anwendersicht widerspiegelt ist, wie schwierig die abstrakte Formulierung von Anforderungen an IT-Systeme tatsächlich ist. Analog zu Abschnitt 5.4.2.10 wird Conceptual Pair Programming als explorative Methode definiert.

Wir haben bereits in Abschnitt 5.4.2.10 auf Risiken und Grenzen der Prototypisierung und insbesondere der entstehenden Prototypen hingewiesen. Entsprechend ist der Teil *„Entwickle so gewonnene Prototypen zu produktiven Systemen.“* der zu Beginn dieses Abschnitts formulierten Maxime so zu verstehen, dass ein möglichst sinnvoller Weg genutzt werden muss, um aus einem Prototyp ein produktives System zu machen. Hier gibt es eine Reihe von Möglichkeiten:

- Der Prototyp wird als so gut empfunden, dass er direkt oder mit minimalen Ergänzungen und Änderungen in die Produktion überführt werden kann.
- Der Prototyp dient als konzeptionelle Vorlage (quasi als Ersatz oder als Ergänzung einer textuellen Spezifikation), um durch CMSD-Experten die produktive Version entwickeln und diese dann durch Fachexperten validieren zu lassen.
- Der Prototyp dient als Grundlage für eine produktionsgeeignetere Version des

Zielsystems, die – wie im vorhergehenden Punkt erläutert – entwickelt und als Grundlage für weitere iterative Prototypisierungsschritte verwendet wird, um sich so durch gezielte Zwischenrefactorings einer sinnvollen Produktionslösung anzunähern.

Es ist im Einzelfall zu entscheiden, welche dieser Vorgehensweisen die geeignete ist. Aufgrund der Leistungsfähigkeit eines CMDSD-Systems ist es nach unserer Erfahrung häufig aber sehr effizient möglich, anhand eines Prototypen eine komplett neue Version zu entwickeln – in jedem Fall geht dies drastisch effizienter als bei klassischer manueller Programmierung.

6.4.3.6 Refactoring

K-3.39 ▷ 93

Plane ein kontinuierliches Refactoring der Lösung.

Diese Methode setzt die Forderungen von Eric Evans nach einem kontinuierlichen Refactoring einer gerade entstehenden Lösung um (siehe dazu unsere Erläuterungen in Abschnitt 3.2.7 auf Seite 92). Wir wollen mit dieser Methode noch einmal explizit darauf hinweisen, wie wichtig es ist, entsprechend dem Vorgehen der T-Technik[10] Erkenntnisse aus den vertikal orientierten Modellierungsphasen in die horizontale Domäne zurückzuspeisen. Dies kann Refactoring-Operationen nach sich ziehen, bei denen das Gesamtsystem hinsichtlich seiner Details optimiert wird und den neuen Ergebnissen gemäß hin angepasst wird. Entsprechend sehen wir eine explizite Berücksichtigung von Refactoring vor (siehe dazu auch die einzelnen in Abschnitt 6.3 beschriebenen Schritte).

6.4.3.7 Lebendes Angebot

Es muss mindestens ein Dokument geben, das den fortlaufenden Stand der Diskussionen widerspiegelt.

Um im Rahmen eines offenen und fairen Umgangs miteinander zu arbeiten, bietet es sich bei einer agilen Vorgehensweise an, eine pragmatische Dokumentation des

[10] Hinweise zur T-Technik finden sich in Abschnitt 6.5.5.

Projektstandes zu finden. Am Sinnvollsten hat sich für uns in vielen Projekten ein Fortschreiben des ursprünglichen Angebots bzw. der ergänzenden Angebotsdokumentation erwiesen[11]. Dahinter steckt folgende Idee:

- Das Angebot beschreibt mindestens das Projektziel und den groben Rahmen (und in manchen Fällen auch wesentlich mehr Details).
- Das Angebot wurde initial von beiden Seiten akzeptiert.
- Wenn sich Änderungen im Projektverlauf ergeben, sind diese gemeinsam abzustimmen.
- Die abgestimmten Änderungen sind schriftlich im Angebot zu fixieren und als Änderungen zu markieren. Die so fixierten Änderungen sind an alle Beteiligten zu kommunizieren und durch diese zu bestätigen.
- Sobald die Änderungen abgestimmt sind, können sie umgesetzt werden.
- Ergibt sich die Notwendigkeit von Kompromissen, kann die fortgeschriebene Angebotsdokumentation als Grundlage für ein Aufwiegen der bisherigen gegenseitigen Zugeständnisse verwendet werden. Wir empfehlen diesen formalen Vergleichsprozess aber erst dann, wenn sich auf der rein kommunikativen und informellen Ebene keine Einigung erzielen lässt. Ein zu frühes Zurückgreifen auf diesen eher formalen Prozess kann zu Irritationen führen, da eine vertrauensvolle Basis durch unnötige Formalismen belastet wird.

Alternativ ist ein ebenfalls sehr gut denkbarer Ansatz für Projekte, dass der aktuelle modellbasierte Prototyp des Systems als lebendige Dokumentation dient. Dazu muss der Kunde einen Ansprechpartner für den gesamten Projektverlauf abstellen und diesem auch genug Zeit gewähren, sich kontinuierlich mit dem Projekt zu beschäftigen. Über ein bereitzustellendes (möglichst einfaches) Ticketsystem kann der Kunde alle ihn störenden Punkte an dem Prototypen erfassen und deren Diskussions- und Umsetzungsstatus nachverfolgen.

Es sei darauf hingewiesen, dass manche Domänen besondere Dokumentationspflichten erfordern (z. B. im Umgang mit Menschenleben, Gefahrstoffen oder sicherheitskritischen Anlagen) – in einem solchen Fall kann diese Methode natürlich nicht verwendet werden, sondern es ist auf das durch die Domäne vorgeschriebene Dokumentationsverfahren zurückzugreifen.

[11] Unserer Erfahrung nach sind Angebote bei kleinen und mittelständischen Unternehmen in der Regel sehr viel informeller und pragmatischer aufgebaut als bei größeren IT-Projekten. Nur selten gibt es ein formales Lasten- und Pflichtenheft – viel häufiger trifft man auf eine begrenzte Menge story-artiger Formulierungen, die die Zielsetzung des Systems umreißen. Dies liegt u. a. darin begründet, dass in vielen Fällen ausführliche Analysephasen gar nicht budgetierbar sind (siehe dazu wiederum unsere ausführlichen Erläuterungen in Kapitel 2).

6.4.3.8 Agiles Glossar

> *Ermögliche allen Projektbeteiligten, dass projektspezifische Vokabular einzusehen und weiterzuentwickeln.*

Da in CMDSD-getriebenen Projekten die fachliche Modellierung von Systemen fokussiert wird, ist auch die Verwendung der korrekten Fachbegriffe von zentraler Bedeutung. Da sich sowohl in der Spezifikations- als auch in der Implementierungsphase immer wieder neue Erkenntnisse über fachliche Zusammenhänge ergeben (insbesondere, wenn ein fachliches Thema zum ersten Mal bearbeitet wird), ist es wichtig, eine entsprechende Dokumentation der Fachbegriffe verfügbar zu machen.

Um allen Projektbeteiligten die gemeinsame Wartung und Pflege einer solchen Dokumentation zu ermöglichen, sollte ein sogenanntes *agiles Glossar* bereitgestellt werden. Als agiles Glossar verstehen wir eine Dokumentationsplattform, die die folgenden Eigenschaften erfüllt:

- Alle Projektbeteiligten haben Zugriff auf die Dokumentationsplattform.
- Alle Projektbeteiligten können Inhalte ergänzen, verändern und löschen.
- Alle Projektbeteiligten können Verknüpfungen zwischen den Inhalten verändern.
- Die Dokumentationsplattform wird als einzige Plattform für die Glossarpflege verwendet. Sind aus formalen Gründen Auszüge in anderen Dokumenten (Angeboten, etc.) erforderlich, sind diese durch entsprechende manuelle oder automatisierte Exporte bereitzustellen.
- Die Dokumentationsplattform sollte so einfach wie möglich zu bedienen sein.

Grundidee des agilen Glossars ist also, eine lebende und einfach zu bedienende sowie für alle verfügbare Plattform zu haben, in der die Projektbeteiligten ihr Wissen leicht einstellen und wieder abrufen können. Aktuell bietet sich insbesondere die Verwendung von Wikis als Grundlage für eine solche Dokumentationsplattform an (siehe zum Beispiel [Wiki09h]).

Ein agiles Glossar erlaubt bei konsequenter Verwendung zudem die Generierung von Mehrwert – mit den richtigen Werkzeugen kann aus einem Wiki zum Beispiel automatisiert ein Nachschlagewerk oder ein Referenzhandbuch extrahiert werden (siehe z. B. [CoHa05], S. xii). Somit wird auch die Vorbereitung von Schulungen und Weiterbildungsmaßnahmen direkt aus in Projekten erzielten Ergebnissen vereinfacht.

6.5 Weitere Aspekte von CDPM

In diesem Abschnitt fassen wir weitere Aspekte des CDPM zusammen, die nicht so bedeutsam sind, dass sie aus unserer Sicht in das Haupt-Meta-Modell integriert werden müssen, zumal sie in der nachfolgenden eher formalisierten und institutionalisierten Form nur zeitweilig Anwendung finden. Zudem geben wir weitere Erfahrungen wieder, die wir im Umgang mit CDPM gesammelt haben.

6.5.1 Teamorganisation

Ziel der nachfolgend beschriebenen Teamorganisation ist es, Reibungsverluste durch komplexe Organisationsstrukturen, politische Abhängigkeiten, eine unübersichtliche Zahl von Meetings und weitläufige Verantwortlichkeiten zu verringern, um so einen harmonischen und effizienten Projektablauf sowie eine enge Zusammenarbeit mit der Fachseite zu ermöglichen. Dies wird in KMU-Projekten trotz der geringen Projektgröße wieder wichtig, da wie bereits in Kapitel 2 ausgeführt häufig eine sehr unmittelbare Zusammenarbeit zwischen allen Stakeholdern erforderlich ist und dabei aufgrund der kürzeren Projektlaufzeiten ähnliche Probleme wie bei Großprojekten auftreten.

Die Teamgröße sollte in allen Sitzungen so überschaubar wie möglich gehalten werden. Bewährt haben sich nach unserer Erfahrung insbesondere Workshops mit insgesamt zwei bis sechs Teilnehmern. Darüber hinaus besteht die Gefahr, sich in unergiebigen Diskussionen zu verlieren. Sitzungen mit mehr als 10 Teilnehmern sollten unbedingt vermieden werden, da sie nur selten effektiv sind. Sie sind lediglich für Kickoff-Veranstaltungen und Projektabschlussmeetings relevant, um das gemeinschaftliche Erlebnis zu betonen (siehe dazu Abschnitt 6.4.3.1 auf Seite 266).

Der nachfolgende Abschnitt 6.5.2 geht näher auf die in einem CDPM-basierten Team zu besetzenden Rollen ein.

6.5.2 Rollen

CDPM greift den rollenbasierten Ansatz aus der *Contextual Design*-Methode heraus. Hier wurde erkannt (siehe z. B. [BeHo98]), dass Prozessveränderungen einfacher werden, wenn sich die Prozessteilnehmer in ihrem Handeln an Rollen und nicht an langen Listen von Regeln orientieren können. Untersuchungen in diesem Umfeld

haben ergeben, dass sich die Teilnehmer durch lange Regellisten sehr stark abgelenkt fühlen und sich letztlich mehr mit dem Erinnern an und Verarbeiten von Regeln beschäftigen als mit ihren eigentlichen Aufgaben. Sich dagegen mit einer Rolle zu identifizieren führt (bei klarer Definition) der Rollen auf natürliche Weise dazu, dass die notwendigen Regeln mehr oder minder automatisch eingehalten und intuitiv berücksichtigt werden. Dies wird noch verstärkt durch den folgenden Erfahrungswert aus [CoHa05], S. 27:

> *[...] Finally, recognize that no matter who you are,* you *can't change the organization. People must change themselves. So enlist allies. [...]*

Ziel muss es also sein, neue Methodiken so zugänglich, überzeugend und intuitiv wie möglich zu machen, um auf diese Weise möglichst einfach Verbündete zu gewinnen und so die Organisation von innen heraus zu verändern durch eine Veränderung in den Köpfen der handelnden Personen. Zudem werden die Erfahrungen aus dem *Convergent Engineering* über optimale Teambildung aufgegriffen (siehe [Tayl95], S. 53ff.). Wir schlagen daher im Rahmen des CDPM die nachfolgend beschriebenen speziellen Rollen vor:

- Domänenexperten (Abschnitt 6.5.2.1)
- CMDSD-Experten (Abschnitt 6.5.2.2)
- Mediatoren (Abschnitt 6.5.2.3)
- Protokollführer (Abschnitt 6.5.2.4)

6.5.2.1 Domänenexperten

Domänenexperten setzen sich nach Möglichkeit aus den versiertesten und erfahrensten Anwendern auf Kundenseite aus der jeweiligen Problemdomäne zusammen. Wir weisen explizit darauf hin, dass es sehr sinnvoll sein kann, auch Experten auszuwählen, die erst seit kurzer Zeit bei dem jeweiligen Kunden tätig sind, aber trotzdem über intensive Erfahrungen mit der jeweiligen Domäne verfügen. Letzteres fördert den kreativen Gedankenaustausch während der Modellierung und hilft zudem, implizite Annahmen aufzudecken und zu hinterfragen. Ist die Optimierung der existierenden Prozesse ein Projektziel, so kommt eine solchen Durchmischung von Domänenexperten eine besondere Wichtigkeit zu, um die maximal mögliche Kreativität für den Modellierungsprozess zu aktivieren. Zudem ist es hilfreich, wenn das Management die Domänenexperten unterstützt. Nur wenige Dinge sind so motivierend wie ein expliziter Hinweis des Managements, dass

K-3.20 ▷ 67

1. alle Punkte offen diskutiert werden können und Kritik offen ausgesprochen werden soll ohne negative Auswirkungen befürchten zu müssen,
2. argumentierbare Veränderungen des Ist-Zustandes auch tatsächlich umgesetzt und erprobt werden,
3. Optimierungen tatsächlichauch gegen Widerstände durchgesetzt werden, solange der Nutzen nachhaltig darstellbar ist (zum Beispiel durch „Return On Investment“ Berechnungen).

Typische Aufgaben von Domänenexperten sind:

- Fachanalysen
- Wissensaufbau
- Fachliche Entscheidungsfindung
- Anforderungsdefinition
- Validierung und Abnahme

Bei länger laufenden Projekten können auch Domänenexperten selbst im Rapid Development aktiv werden, was hilft, die Akzeptanz des CP-Ansatzes noch weiter zu erhöhen, da Mitarbeiter auf Kundenseite plötzlich eine sehr aktive und bedeutsame Rolle einnehmen können.

6.5.2.2 CMDSD-Experten

Der CMDSD-Experte analysiert und modelliert die Anforderungen der Domänenexperten während der Arbeitssitzungen. Er muss dazu über exzellente Kenntnisse des CMDSD-Werkzeugs inklusive seiner Möglichkeiten und Grenzen verfügen. Gute Domänenkenntnisse sind ebenso wichtig, um die Domänenexperten hinsichtlich der effektivsten Umsetzung ihrer Anforderungen mit dem vorliegenden CMDSD-Werkzeug sinnvoll beraten zu können.

In der freien Modellierungsphase (Rapid Modeling, siehe Abschnitt 6.4.3.4) übernimmt der CMDSD-Experte die Rolle des agilen Modellierers (siehe [Ambl02a]), in der Conceptual Pair Programming Phase führt er die Domänenexperten durch das CMDSD-System und modelliert mit ihnen die konkrete Lösung bzw. die dazu erforderlichen Prototypen. Haben die Domänenexperten umfassende Erfahrungen mit dem spezifischen CMDSD-Werkzeug gesammelt, können sie diesen zweiten Part unter Beratung des CMDS-Experten auch selbst übernehmen (siebe oben). Ein solches Vorgehen verstärkt Respekt und Vertrauen zwischen den Projektbeteiligten.

Typische Aufgaben des CMDSD-Experten sind:

- Machbarkeitsanalysen
- Rapid Modeling
- Rapid Development
- Erweiterte Anforderungsanalyse[12]

6.5.2.3 Mediatoren

Bei einer größeren Arbeitsgruppe sollte ein Mediator eingebunden werden. Je nach Qualifikation des CMDSD-Experten kann diese Rolle bei Gruppen von bis zu vier oder fünf Personen auch durch den CMDSD-Experten selbst übernommen werden. Ab drei Personen ist dann aber fast immer ein zusätzlicher Protokollführer erforderlich.

Der Mediator sollte umfangreiche Moderationskenntnisse besitzen. Zudem sind ausgewiesene Fachkenntnisse in der Geschäftsprozessanalyse sowie der Systemanalyse von Vorteil.

Typische Aufgaben des Mediators sind:

- Diskussionssteuerung
- Einsatz und Steuerung von Kreativitätstechniken
- Vermittlung und Schlichtung in Diskussionen
- Kompromisssuche

6.5.2.4 Protokollführer

Protokollführer sollten in jeder Conceptual Pair Programming Sitzung zugegen sein. In kleinerem Kreis (bis drei Personen), bei Personalengpässen oder bei äußerst begrenzten Budgets auf Kundenseite kann es sich dabei um den CMDSD-Experten handeln. Bei einer größeren Zahl von Teilnehmern (insbesondere Domänenexperten) empfiehlt es sich, diese Rolle unbedingt mit einer separaten Person zu besetzen.

K-3.20 ▷ 67
K-3.38 ▷ 91

[12] Damit meinen wir insbesondere, dass der CMDSD-Experte als Sparringspartner für den Domänenexerten fungieren und dessen Anforderungen kritisch hinterfragen soll. Häufig entwickeln sich dabei Ideen für einfachere oder alternative Lösungsansätze.

Hauptaufgabe des Protokollführers ist es, während des Rapid Modeling (siehe Abschnitt 6.4.3.4) und in einem geringeren Maße während des anschließenden Conceptual Pair Programming alle Entscheidungen, Ideen und Anforderungen der anderen Teilnehmer zu dokumentieren und bei Bedarf wieder anzusprechen. Insbesondere obliegt ihm die Überwachung des agilen Glossars (siehe Abschnitt 6.4.3.8). Dies entbindet Domänen- wie CMDSD-Experten davon, sich während des kreativen Modellierungsvorgangs durch die notwendige Buchhaltung einzuschränken und zu behindern.

Optimalerweise hat der Protokollführer gute Kenntnisse sowohl im vorliegenden CMDSD-Werkzeug als auch in der Problemdomäne, im Zweifelsfall reichen aber auch solide Kenntnisse im CMDSD-Werkzeug. Dies ist besser als der umgekehrte Fall (also gute Kenntnisse über die Problemdomäne, aber keine Kenntnisse im CMDSD-Werkzeug), da diese Konstellation häufig dazu führt, dass implizite Annahmen über spezielle Domänenausprägungen nicht angesprochen und schon gar nicht dokumentiert werden. Das kann zu großen Problemen bei der anschließenden Umsetzung führen und ist daher zu vermeiden.

Protokolle sollten allen Projektbeteiligten zeitnah zur Verfügung gestellt werden, um so die Möglichkeit zu Korrekturen zu geben. Dies fördert das Prinzip der Explizivität.

Typische Aufgaben des Protokollführers sind:

- Protokollierung
- Überwachung und Nachverfolgung von Protokollinhalten
- Verwaltung von Protokollen (Verschicken, Archivieren, usw.)
- Pflege der lebendigen Spezifikation (siehe Abschnitt 6.4.3.7)

6.5.3 Organisatorische Einführung von CDPM

Neue Vorgehensmodelle in bestehende Organisationen einzuführen ist immer eine Herausforderung, da sich Methoden, Prozesse und Menschen anpassen müssen, um effektiv mit einem neuen Modell arbeiten zu können. Mit CDPM ist dies nicht anders, zumal CDPM einen sehr starken Paradigmenwechsel weg von der weithin betriebenen code-zentrierten und damit sehr vertrauten Form der Software-Entwicklung hin zur einer modellorientierten und vollständig auf Kundenbedürfnisse, -begrifflichkeiten und -sichtweisen bezogenen Arbeitsweise bedeutet.

Um den zu erwartenden organisatorischen, politischen und inhaltlichen Widerständen im Rahmen der Einführung zu begegnen, empfiehlt sich ein ebenso einfaches

wie schrittweises Vorgehen, das sich in den folgenden Phasen vollzieht (in Anlehnung an [BuKn02], S. 17):

1. Einführung der Schlüsseltechnologien (CMDSD-Werkzeug) anhand eines konkreten kleinen Projektes,
2. Schulung der fundamentalen Prinzipien und Werte durch einen erfahrenen Coach in der Kickoff-Phase des Projekts,
3. Schrittweise Einführung des gewählten Vorgehensmodells

Besondere Sorgfalt ist auf die Auswahl des richtigen initialen Projektes zu legen. Hier sind zwei Fälle zu unterscheiden:

1. Soll CDPM in Kombination mit CMDSD für interne Projekte in einem Unternehmen eingeführt werden,
   a) so sollte zunächst ein Projekt ausgewählt werden, das nicht unter besonderem Zeitdruck steht. Andernfalls wird das Erlernen der neuen Methoden und Werkzeuge aller Wahrscheinlichkeit nach aus Zeitgründen scheitern,
   b) empfiehlt es sich zumindest für das initiale Projekt mit einem externen Berater, Coach oder Dienstleister zusammenzuarbeiten, um unter Anleitung an die neuen Methoden und Werkzeuge herangeführt zu werden. Dies verhindert auch auf effektive Weise Projektverzögerungen durch interne politische oder organisatorische Grabenkämpfe von Parteien, die mglw. ein Interesse daran haben, dass CDPM nicht eingeführt wird (um beispielsweise ihre eigenen vermuteten oder tatsächlichen Kompetenzbereiche zu wahren),
   c) ist es sehr hilfreich, von Beginn alle potenziellen Stakeholder (auch solche, die erst in zukünftige Projekte einbezogen sein werden) über das Projekt zu informieren und sie über erreichte Erfolge, Einsparungsmöglichkeiten, etc. auf dem Laufenden zu halten. Hier sind auch „Meta-Stakeholder“ wie z. B. ein Betriebsrat, etc. eng einzubeziehen, da die erfolgreiche Einführung von CDPM/CMDSD deutliche Auswirkungen auf die Arbeitsumwelt sowohl von IT- als auch fachlichen Abteilungen hat.
   d) muss die Managementlinie über alle Aktivitäten auf dem Laufenden gehalten werden, um einen zunehmenden Rückhalt für dein Einführungsprozess zu erlangen und sicherzustellen, dass Dritte kein verfälschendes Bild der Projektaktivitäten hinter dem Rücken der Verantwortlichen vermitteln, um durch solche Maßnahmen ihre eigenen Ziele durchzusetzen.

2. Wird CDPM als neues Verfahrensmodell für die Bearbeitung externer Projekte (also in der Rolle als Dienstleister) eingeführt, ist eine nachhaltigere und breiter angelegte Strategie von Nöten. Hier spielt die Größe des Unternehmens eine wichtige Rolle. Soll eine Veränderung im Großen herbeigeführt werden, sind erheblich aufwändigere Anstrengungen erforderlich, als wenn nur ein Projektteam oder eine Abteilung ihre Arbeitsweise verändern will. Im Einzelnen bedeutet dies:

   a) Zunächst müssen interne Meinungsträger identifiziert und für das neue Thema gewonnen werden.

   b) Die betroffenen Fachabteilungen sollten durch Demonstrationen und Workshops (die praktische Beispiele beinhalten müssen) mit der neuen Thematik vertraut gemacht werden.

   c) Internes Marketing mit konsequenter Berichtserstattung über Best Practices, Success Stories, etc. ist erforderlich.

   d) Die internen Meinungsträger müssen in die Lage versetzt werden, mit CMDSD und CDPM umgehen zu können.

   e) Eine Pilotabteilung muss gewonnen werden, die mit der Einführung beginnt und hier analog zu der Beschreibung oben für die interne Einführung von CDPM und CMDSD vorgeht.

   f) Die Pilotabteilung muss als Meinungsträger aufgefordert und in die Lage versetzt werden, andere Abteilungen über die Erfolge und Schwierigkeiten der Einführung auf dem Laufenden zu halten.

   g) Im nächsten Schritt müssen großflächige Schulungen und Workshops abgehalten werden, um mit Unterstützung der Meinungsträger auch die Organisation als Ganzes an das neue Thema heranzuführen.

Insgesamt wird aus den obigen Ausführungen schnell klar, dass die Einführung von CDPM nicht wirklich eine technische Herausforderung sondern eher von psychologischen, sozialen und organisatorischen Aspekten dominiert wird. Dies ist bedingt durch die nachhaltige und konsequente Veränderung der IT-Prozesse, die eine entsprechende Einführung nach sich zieht und kann direkt verglichen werden mit den Herausforderungen, die die Einführung konsequenten Kundenziehungsmanagements auf betriebswirtschaftliche Prozesse und Organisationseinheiten hat (siehe z. B. [Crmn06]). Hier sind vergleichbare Widerstände zu erwarten und ähnliche Anstrengungen zu leisten. Im übertragenen Sinne handelt es sich bei CP sogar um „totales Kundenbeziehungsmanagement in allen IT-Projekten“.

6.5.4 Strategisches Anforderungsmanagement

Eine wichtige Phase eines jeden CMDSD-basierten Projektes und somit auch die bedeutendste Phase nach dem eigentlichen Projektabschluss im CDPM-Prozess ist das Management der neuen Anforderungen. Dabei sind die folgenden Einzelaufgaben zu unterscheiden:

- Es sind die Anforderungen zu ermitteln, die in zukünftigen Projektphasen umgesetzt werden sollen. Diese sind zu sammeln, aus Anbietersicht zu priorisieren und dem Kunden vorzulegen, um nachfolgende Projektphasen zu planen.

  Bei diesem Schritt handelt es sich interessanterweise zu einem nicht unwesentlichendem Teil um eine vertriebliche Aufgabe, da es letztlich darum geht, weitere Budgets für kommende Projektschritte zu erhalten. Durch die vertrauenswürdige Form der Zusammenarbeit, die durch den Einsatz von CMDSD und CDPM zwischen Kunden und Anbietern aufgebaut wird, gestalten sich derartige Planungsaktivitäten häufig einfacher als bei klassischen Projekten, in denen die Budgetdiskussionen viel häufiger durch Unstimmigkeiten im bisherigen Projektverlauf belastet werden.

  Umgekehrt führen vertriebliche Notwendigkeiten häufig zu Problemen im Projektverlauf: Es wird zu viel versprochen, Lösungen werden in der Hoffnung auf einen schnellen Abschluss als trivial dargestellt, etc. Hier empfiehlt sich eine konzentrierte Presales-Arbeit: Da CMDSD-Experten (siehe Abschnitt 6.5.2.2) sowohl über gute Fachkenntnisse als auch über exzellente Werkzeugkenntnisse in dem zugrunde liegenden CMDSD-Werkzeug verfügen, können sie sich sowohl im Interesse des Dienstleisters als auch des Kunden im vorprojektlichen Umfeld dafür einsetzen, ein solides Fundament für Projekte zu schaffen. Da der CMDSD-Experte mit hoher Wahrscheinlichkeit selbst an so akquirierten Projekten beteiligt sein wird (und bewusst beteiligt werden sollte), hat er somit ein ureigenes Interesse Projekte so zu akquirieren, dass sie auch auf realistischen Randbedingungen basieren.

  Vorteil ist, dass CP-basierte Ansätze dazu führen, dass Kunden einen tieferen Einblick in die Komplexität verschiedener Anforderungen und die Auswirkungen auf den Umsetzungsprozess erhalten, so dass zukünftige Planungen mit realistischeren Zeitabschätzungen und Meilensteinen versehen werden. Nach unserer Erfahrung neigen Kunden, die bereits Erfahrungen mit CMDSD gesammelt haben, viel eher dazu, Empfehlungen von Anbietern zu folgen, da das offene Miteinander in den Projekten für eine solidere Vertrauensbasis sorgt.

Zudem hat der Kunde üblicherweise einen viel tieferen Einblick in die Details des Entwicklungsprozesses, so dass er selbst auch fundierter abschätzen kann, warum ein bestimmter Lösungsweg möglicherweise sehr aufwändig oder sehr teuer ist.

Dies resultiert letztlich in einer angenehmeren Planung zukünftiger Projektschritte für beide Seiten: Der Anbieter kann offener mit Aufwänden umgehen und der Kunde kann mit höherer Sicherheit auf die Abschätzungen des Anbieters vertrauen, da er in der späteren Umsetzungsphase einen hinreichend konkreten Einblick in die Realisierung bekommt, um dann bemessen zu können, ob die Abschätzungen realistisch waren.

- Es sind bereits umgesetzte Anforderungen ebenso wie die zur Umsetzung geplanten Anforderungen daraufhin zu prüfen, ob sie effektiv mit den vorhandenen Mitteln des CMDSD-Werkzeugs umgesetzt werden können. Ist dies nicht der Fall muss untersucht werden, ob es eine schematische und hinreichend verallgemeinerbare Form der Umsetzung gibt. Ist das der Fall muss auch in die Produktplanung für das CMDSD-Werkzeug eine Anforderung zur Umsetzung einer entsprechenden Funktionalität aufgenommen werden.

  Wie immer gilt für die Umsetzung von Anforderungen die allgemeine CMDSD-Philosophie: Eine modulare Erweiterung sollte nicht sofort versuchen, jeden denkbaren Grenzfall abzudecken sondern lieber einen leicht verständlichen und hinreichend generischen Ansatz verfolgen. Ist eine Erweiterung leicht zu erklären und dennoch für eine Vielzahl von Situationen einsetzbar, so ist das besser als ein Baustein, der alle möglichen Situationen abdeckt, aber schwer zu konfigurieren oder gar nur in einer langwierigen Schulung oder Anleitung zu erklären ist. Letztere Vorgehensweise ist unbedingt zu vermeiden, da sie den effektiven Einsatzwert des Werkzeugs mindert.

K-3.50 ▷ 115

6.5.5 Besonderheiten der Analysephasen

Das Vorgehen in den Analysephasen geht von der sogenannten *T-Technik* aus ([Tayl95], S. 61). Dabei wird in der initialen Analysephase die Domäne in ihrer Gesamtheit betrachtet. Die zentralen Geschäftsprozesse und -objekte werden auf grobgranularer Ebene identifiziert und für die nachfolgenden Schritte vorstrukturiert[13]. In allen nachfolgenden Sitzungen werden Einzelaspekte detaillierter ausgearbeitet.

[13] In unserem Fall bedeutet dies eine Vorstrukturierung der nachfolgenden Rapid-Development-Schritte.

Die dabei neugewonnenen Erkenntnisse fließen dann wieder in die Gesamtmodellierung zurück. Abbildung 6.4 stellt diesen Zusammenhang dar.

Bei hochgradig fokussierten Projektthemen, die sich auf einen zentralen Prozess beschränken, kann die initiale Analysephase häufig direkt in eine Rapid Development Phase übergehen bei der ein Prototyp entsteht, der zunächst die zentralen Bestandteile des Prozesses beschreibt. In diesem Fall wird Analysephase durch die Rapid Modeling Unterphase des Rapid Development Schritts ersetzt (siehe Abschnitt 6.4.3.4).

Handelt es sich dagegen um ein Projekt, in dem eine Reihe zentraler Prozesse umgesetzt und optimiert werden sollen, sind diese in der initialen Analysephase zu identifizieren, bezüglich ihres Umfangs zu charakterisieren und in ihren Zusammenhängen darzustellen. Dabei geht es noch nicht darum, präzise Schnittstellen abzustimmen. Stattdessen werden die essentiellen Kommunikationspfade festgelegt. Zudem sind für alle Teilprozesse die zentralen Stakeholder zu identifieren. Letztlich werden die identifizierten Prozesse in eine sinnvolle Reihenfolge gebracht. Diese Vorbereitungen dienen als Grundlage für die nachfolgenden Rapid Development Phasen. Jeder Teilprozess wird in mindestens einer Rapid Development Phase behandelt. Bei sehr kleinen Aufgaben kann natürlich mehr als eine Rapid-Development-Phase pro Tag durchgeführt werden – wir empfehlen dazwischen aber ausreichende Pausen, um sich mental hinreichend auf das nächste Thema einstellen können.

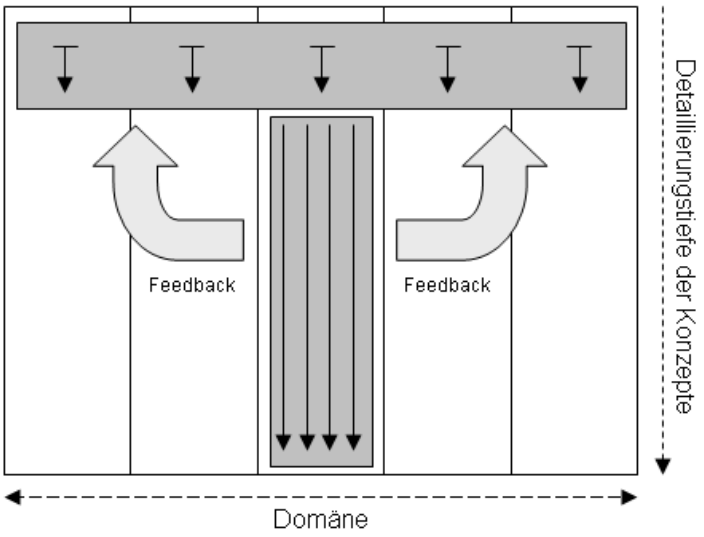


Abbildung 6.4: Feedback bei Anwendung der T-Technik

Die folgenden Ergebnisse sollten aus der initialen Analysephase resultieren:

- eine Liste der zu erreichenden Ziele

- eine Liste der Aufgaben, die durch das zu realisierende System abgewickelt werden
- eine Liste der Prozesse, die in dem System abgebildet werden sollen
- eine Liste der Verantwortlichen, die an den Prozessen teilnehmen und von der Abbildung der Aufgaben profitieren
- eine Liste der Domänenobjekte, die modelliert werden müssen. Zu diesem Zeitpunkt geht es nicht darum, die Eigenschaften dieser Objekte zu beschreiben. Das ist Thema der nachfolgenden Rapid Development Phasen. Wichtig ist eine Klärung der relevanten Domänenobjekte und ihrer Abhängigkeiten. Zudem wird zu diesem Zeitpunkt das agile Glossar (siehe Abschnitt 6.4.3.8) initialisiert und das Sammeln von Abweichungen zu den Standardbegriffen der Domäne beginnt.
- eine Liste der Systeme, die mit der zu erstellenden Lösung integriert werden sollen. Wichtig ist, an dieser Stelle ein grundlegendes Verständnis für die Komplexität der Integration zu bekommen, damit weitere Rapid Development Phasen entsprechend vorbereitet werden können. Hierzu ist es hilfreich, die Kerneigenschaften der jeweiligen Systeme abzufragen (z. B. Plattform, Anzahl Nutzer, Datenvolumen, prinzipielle Formate und Schnittstellen, siehe auch [Tayl95] S. 64).
- eine grobe Priorisierung der Umsetzungsreihenfolge und der wesentlichen Meilensteine für die weiteren Projektschritte
- eine Liste von Aufgaben für Dienstleister wie Kunden. Vornehmlich wird es um die Informationsbeschaffung zu Fragen gehen, die im Rahmen der Erstanalyse aufgworfen werden. Diese Fragen können den Kunden (bzgl. seiner Fachdomäne) oder den CMDSD-Dienstleister (bzgl. der Leistungsfähigkeit des Werkzeugs in bestimmten Aspekten, bzgl. allgemeiner technischer, fachlicher oder organisatorischer Fragestellungen, usw.) betreffen.

6.5.6 Adaption von CDPM

Zur Adaption von CDPM gelten ähnliche Aussagen, wie wir sie bereits zur Adaption von CMDSD in Abschnitt 5.5.4 gemacht haben: Es gibt eine Reihe essentieller Fragen, die zu klären sind, um das hier vorgestellte CDPM-Metamodell auf eine spezifische Domäne anzupassen. Dazu gehören die folgenden Fragen:

- *Sind durch die spezifische Domäne besondere Anforderungen an das Projektmanagementmodell festgelegt (z. B. die Forderung nach dem V-Modell im öffentlichen Umfeld)?* Falls dies so ist, müssen die Anforderungen natürlich explizit berücksichtigt werden. Im Zweifelsfall überlagern sie auch die hier formulierten Methoden und Prinzipien. Es ist aber auch durchaus möglich, zu dem Schluss zu kommen, dass die betrachtete Domäne nicht für CDPM und damit auch nicht für CP geeignet ist. Dies ist meist daran zu erkennen, dass zentrale Prinzipien oder eine Mehrheit der Methoden (und insbesondere das Rapid Development) in der betrachteten Domäne nicht anwendbar sind.
- *Welche Erfahrungswerte besitzt der Dienstleister mit Projektmanagementmethoden?* Normalerweise macht es Sinn, bewährte Methoden weiter zu verwenden, da die Einführung sowohl neuer Werkzeuge als auch neue organisatorischer Methoden zur gleichen Zeit das Risiko birgt, sich selbst zu übernehmen.
- *Hat der Kunde bestimmte Erwartungen an Projekte in der betrachteten Domäne?* Für jede Erwartung ist zu prüfen, welchen Wert diese für eine effiziente Abwicklung von Projekten hat. Werden Erwartungen durch das neue Vorgehensmodell nicht mehr erfüllt, muss der Kunde im Rahmen des philosophischen Projektkickoffs (siehe Abschnitt 6.4.3.1) über die Gründe und Motivation für die vorgenommenen Änderungen informiert werden.

Analog zu unseren Ausführungen zur CMDSD-Adaption in Abschnitt 5.5.4 zeigt sich erneut, dass auch für CDPM Checklisten aufzustellen sind, die eine genaue Betrachtung der zugrunde liegenden Domäne ermöglichen. Da dies auch hier den Rahmen der Arbeit überschreitet, schließen wir unsere Betrachtungen mit der Bemerkung, dass die konkrete Ausgestaltung von CDPM für eine Domäne häufig einfacher ist als die Entwicklung der CMDSD-Werkzeuge, da hier unmittelbar die vorhandenen Domänenerfahrungen des Dienstleisters aufgegriffen werden können.

6.6 Zusammenfassung

Analog zu unseren Ausführungen zu CMDSD in Kapitel 5 haben wir in diesem Kapitel mit dem Ansatz des *Concept Driven Project Management* ein Metamodell zur operativen Projektabwicklung für CMDSD-basierte Projekte vorgestellt. Dazu haben wir zuerst die Ziele des Modells in Abschnitt 6.1 definiert, insbesondere:

- die Minimierung der Kommunikationsdistanz zwischen Domänen- und IT-Experten,

- die Abstraktion technischer Details zwecks Maximierung des fachlichen Diskurses,
- die Maximierung frühzeitigen Feedbacks zwecks Fehlervermeidung und
- eine Prototypen-basierte Vorgehensweise, um die Feedbackqualität zu maximieren.

Anschließend haben wir unseren Lösungsansatz in Abschnitt 6.2 motiviert, indem wir dargestellt haben, das eine offene und ehrliche Diskussionskultur zwischen Kunden und IT-Dienstleistern eine Vertrauensbasis schafft, die eine wertvolle Hilfe bei der Vermeidung der in Abschnitt 2.2.1 auf Seite 26 identifizierten und heute immer noch in IT-Projekten vorherrschenden Probleme ist. Schließlich haben wir das prinzipielle Vorgehensmodell in Abschnitt 6.3 vorgestellt. Nachfolgend haben wir die allgemeinen Werte und die darauf aufbauenden Prinzipien des *Conceptual Programming* verfeinert und diese zu spezifischen Methoden konkretisiert, die den Aufbau eines CDPM-Projektmanagentmodells unterstützen (Abschnitt 6.4). Besonders wichtig sind dabei:

- die starke Betonung einer offenen, ehrlichen, direkten und konsequenten Kommunikation zwischen Dienstleister und Kunden, um eine solide Grundlage und nachhaltige Basis für das eigentliche iterative, inkrementelle, rekursive und evolutionäre Vorgehen im Projekt zu schaffen sowie
- die Methode des *Conceptual Pair Programming*, die durch eine intensive und direkte Zusammenarbeit zwischen Domänen- und IT-Experten einen besonders intensiven und nachhaltigen Wissensaufbau ermöglicht.

Über die Wirkung dieser Maßnahmen werden wir im Rahmen unserer Verifikation in Teil III der Arbeit berichten.

Zudem stellen wir fest, dass CDPM all die Möglichkeiten zur Optimierung von Geschäftsprozessen bietet, die die Stärke des *Convergent Engineering* ausmachen (siehe [Tayl95], S. 54), dass aber andererseits die Optimierung von Prozessen von uns nicht als integraler Bestandteil eines jeden Projektschritts angesehen wird. Dies sehen wir im Bezug auf mittelständische Unternehmen als zusätzliche Stärke, da Zeit- und Budgetbeschränkungen Optimierungen möglicherweise nicht in jeder Projektphase ermöglichen. Umgekehrt erlaubt das von uns definierte iterative und evolutionäre Vorgehen im Projekt, Optimierungen als neue Aufgabe in einem beliebigen späteren Schritt zu definieren und auch aufgrund des modellbasierten Charakters des CMDSD-Systems umzusetzen, so dass die Leistungsfähigkeit des Gesamtansatzes hierdurch auf eindrucksvolle Weise bestätigt wird.

Kapitel 7

Abgrenzung zu existierenden Ansätzen

„Ich weiß nicht, ob es besser wird, wenn es anders wird.
Aber es muss anders werden, wenn es besser werden soll.“
— Georg Christoph Lichtenberg (1742–1799), Deutscher Schriftsteller und Wissenschaftler

In diesem Kapitel wollen wir kurz die wesentlichen Unterschiede zwischen den für uns bedeutsamen Grundlagen aus Kapitel 3 und unserem Ansatz des *Conceptual Programming* zusammenfassen. Dabei geben wir noch einmal einen kurzen Überblick über die von uns entwickelten Neuerungen. Außerdem verweisen wir in den Abschnitten 7.1.10 und 7.1.11 auf zwei namensverwandte Ansätze, von denen aber nur der erste prinzipielle Berührungspunkte zu unserem Lösungsansatz beinhaltet.

7.1 Einzelbetrachtung

In diesem Abschnitt grenzen wir die verschiedenen zugrunde liegenden Lösungsansätze differenziert von *Conceptual Programming* ab.

7.1.1 *RAD

Auf den ersten Blick gibt es tatsächlich große Ähnlichkeiten zum Rapid Application Development (RAD, siehe die Abschnitte 3.2.4 und 3.2.5), auf den zweiten Blick zeigen sich aber wichtige Unterschiede:

- Alle bekannten RAD-Ansätze konzentrieren sich darauf, die nicht-domänenspezifischen Aufgaben eines Programmierprojektes zu vereinfachen

und nicht-invasive Möglichkeiten anzubieten, um diese möglichst einfach zu erledigen. Domänenspezifische Fragestellungen werden nicht[1]) oder nur am Rande adressiert.

- RAD-Werkzeuge bewegen sich auf einer niedrigeren Abstraktionsebene. So stellen sie z. B. durchaus Mechanismen zur Verwaltung von Datenbankverbindungen bereit. SQL-Zugriffe müssen aber überwiegend manuell kodiert werden und verteilte Datenquellen werden normalerweise nicht direkt unterstützt. Wir werden in der Vorstellung von PIKAS illustrieren, wie wir durch die Formulierung intentionaler Programmmechanismen eine wesentlich höhere Abstraktionsebene erreichen (siehe z. B. Abschnitt 8.1.8.1.4 für Datenthematiken).
- RAD-Werkzeuge sind letztlich in der Praxis nur eine andere Form von Entwicklungsumgebungen, die sich aber nicht mit einem ganzheitlichen Konzept zur Abgrenzung von Domänen, der Steuerung von Projekten und der Adressierung horizontaler Domänen (wie z. B. unserem Beispiel vom Mittelstand) beschäftigen.

Insofern sind die Erfahrungswerte zu RAD-Ansätzen eine wertvolle Grundlage bei der Konzeption unserer CMDSD-Werkzeuge (insbesondere was z. B. Rapid Prototyping angeht, siehe Abschnitt 5.4.2.10), unser Konzept ist aber weiterführender, da wir Domänenabstraktionen zur Optimierung der Entwicklungsprozesse fordern und umsetzen.

7.1.2 Convergent Engineering

Convergent Engineering nach Taylor (siehe Abschnitt 3.1.1) weist philosophisch ähnliche Ideen wie Conceptual Programming auf, unterscheidet sich aber in einer Reihe essentieller Punkte, die wir nachfolgend zur Verdeutlichung der wesentlichen Neuerungen und Weiterentwicklungen unseres CP-Ansatzes aufzeigen wollen.

Eine erste wesentliche Unterscheidung zwischen Convergent Engineering und Conceptual Programming sind die Arbeitsgrundlagen der jeweiligen Technologie. Wie auf Seite 52 ausgeführt, nutzt Convergent Engineering objektorientierte Technologien und Modellierungsverfahren als Grundlage, insbesondere spezifische Klassen und Objekte. Conceptual Programming hebt die Abstraktion auf eine höhere Stufe, indem abstrakte (also technologie-unabhängige) und domänenspezifische Modelle als

[1] Man könnte maximal eine technische Problemdomäne annehmen, die sich mit klassischen Ein- und Ausgabeprozessen, usw. beschäftigt. Hier fehlt aber die klare Abgrenzung zu klassischen Programmiersprachen.

Grundlage verwendet werden, um die Verständniskluft zwischen Domänenexperten und IT-Experten zu reduzieren (wie ausführlich in den Abschnitten 2.2 und 4.2.2 beschrieben).
Des weiteren werden – wie auf S. 53 für Convergent Engineering ausgeführt – die Modelle nach hinreichender Verfeinerung an Implementierer übergeben. Es gibt also im Gegensatz zum Conceptual Programming eine viel deutlichere Trennung zwischen Modellierern und Programmierern mit allen in Kapitel 4 erläuterten Nachteilen. Insbesondere erfolgt eine (üblicherweise händische) Übersetzung der Anforderungen in Methoden zur Umsetzung dieser Anforderungen.
Schließlich geht Taylor weiter von der Verwendung des Wasserfallmodells aus (siehe unsere Ausführungen auf S. 53), was ein gänzlich anderes Vorgehen als das von uns vorgeschlagene iterative, inkrementelle und evolutionäre Vorgehensmodell mit all den Vorteilen dieser Familie von Vorgehensmodellen ist (siehe dazu Abschnitt 3.4.3).
Neben dieser eher spezifischen Unterschieden haben wir bereits in Abschnitt 3.1.1 auf S. 53 den Versuch kritisiert, ein Gesamtmodell für Geschäftsanwendungen zu entwickeln (was im Prinzip im direkten Widerspruch zu unserer Forderung nach stärkerer Domänenfokussierung steht, die entsprechenden Argumente finden sich in den Abschnitten 2.2.2 und 4.2.2.2). Wir verweisen noch einmal auf die Argumentation von Seite 53: Taylors These von der Notwendigkeit eines integrierten Gesamtmodells für Fach- und IT-Systeme erscheint für den Mittelstand nicht relevant, da vielfältige Projekterfahrungen z. B. aus dem SAP-Umfeld zeigen, dass die Komplexität eines solchen Gesamtmodells so groß ist, dass entsprechende Projekte nur mit gewaltigen finanziellen und organisatorischen Aufwand erfolgreich bewältigt werden können. Dies passt aber genau nicht zu den von uns ermittelten üblicherweise geltenden Rahmenbedingungen.

Zudem ist Convergent Engineering aufgrund seiner Fundamentierung in objektorientierten Prinzipien relativ technisch aufgestellt und leidet somit auch an den technischen Schwächen dieser Ansätze. Wie bereits am Ende von Abschnitt 3.1.1 aufgeführt hat sich mittlerweile gezeigt, dass objektorientierte Ansätze auch über eine Reihe von Modellierungsschwächen verfügen und häufig entweder zu grob- oder feingranular sind, so dass weitere Ansätze wie AOP oder COP zunehmend bedeutsamer werden (siehe [FCEl04, Qi4j08]).

Letztlich mangelt es Convergent Engineering an Erweiterungspunkten, um branchenspezifischer Optimierungsansätze in das Grundverfahren einbringen zu können – eine weitere Domänenspezialisierung ist also nicht vorgesehen und somit entsprechend aufwändiger durchzuführen, sollte sie wünschenswert sein.

7.1.3 Convergent Architecture

Hubert hat in seinen Ausführungen zur Convergent Architecture (siehe Abschnitt 3.1.4) durch die Definition von Architekturstilen einen leistungsfähigen Rahmen für unseren Lösungsansatz geliefert. Sein konkreter Entwurf der Convergent Architecture war dagegen nicht weiterführend für den von uns betrachteten Kontext der mittelständischen IT-Projekte, da er sich auf Technologien (wie UML) und Methoden (wie den Rational Unified Process) stützt, die wir als nicht zweckmäßig oder weiterführend für typische mittelständische IT-Projekte identifiziert haben (siehe unsere Ausführungen in Kapitel 3). Im Gegensatz zu Architekturstilen sind wir aber auch in unserem Vorgehensmodell durch Konkretisierung weiter gegangen als Hubert:

- Wir haben konkrete Prinzipien und Methoden identifiziert, die zur Ausprägung von mittelstandsorientierten Lösungsansätzen relevant sind.
- Wir haben mit unserem CP-Ansatz ein vollständiges und konkretes Metamodell definiert, dass zur Ausprägung von domänenspezifischen Lösungen verwendet werden kann. Insbesondere sind wir dabei kompatibel zu Architekturstilen nach Hubert geblieben, was im Rahmen zukünftiger Forschungsarbeiten ermöglichen müsste, Conceptual Programming auch auf Großprojekte zu adaptieren.
- Wir haben mit CMDSD einen konkreten und doch domänenunabhängigen Architekturstil zur Ausprägung domänenspezifischer Lösungsansätze entworfen.
- Wir haben mit CDPM ein konkretes organisatorisches und doch domänenunabhängiges Vorgehensmodell entworfen, dass die von Hubert geforderten Eigenschaften bzgl. organischer Ordnung und der Evolution von Organisationen unterstützt.
- Wir haben gezeigt, wie CMDSD und CDPM eng integriert besondere Optimierungspotenziale besitzen.

Damit ist es uns gelungen, die allgemeinen und doch relativ handlungsunspezifischen Ausführungen von Hubert deutlich zu konkretisieren, um den von uns betrachteten mittelständischen Kontext leistungsfähiger adressieren zu können.

7.1.4 Software-Produktlinien

Mag dieser Abschnitt zunächst verwunderlich erscheinen (denn CMDSD basiert ja inhärent auf Software-Produktlinien, siehe Abschnitt 5.4.3.3), geht es uns darum, klar

zu machen, dass CP mehr als „nur eine Software-Produktlinie“ ist. Insbesondere weist die gesamte CP-Systematik und damit auch die CMDSD-Methodik spezielle Merkmale auf, die den Ansatz zwar immer noch zu einer Form von Software-Produktlinie machen[2], darüber hinaus aber neue Ansätze für die Entwicklung mittelständischer Software-Produktlinien definieren, die für das Ziel unserer Arbeit – die effektive Umsetzung von komplexeren IT-Individualprojekten im Mittelstand – von hoher Bedeutung sind.

Im Einzelnen nehmen wir folgende Modifikationen an der allgemeinen Form von Software-Produktlinien vor, um unsere Zielsetzung zu erreichen[3]:

- Wir definieren jede „CMDSD-Softwareproduktlinie“ zweigeteilt:
  - Es gibt ein domänenspezifisches Modellierungswerkzeug zur Generierung von Lösungen. Das Werkzeug muss nicht, kann aber selbst als Software-Produktlinie angesetzt sein[4].
  - Es gibt einen oder mehrere archetypisch modellierte Lösungsansätze für die betrachtete Domäne (die mit diesem Werkzeug erzeugt werden). Diese sind immer als Software-Produktlinie ausgelegt und stellen alle typischen Eigenschaften (insbesondere auch hinsichtlich der Variabilität) zur Verfügung.

  Wir gehen also immer von einem zweigeteilten Werkzeugansatz sowie einem modellbasierten und generativen Vorgehen aus.
- Der CP-Ansatz geht immer von einer geschlossenen Modellierung der eigentlichen Domänenlösung aus, d. h. es wird immer die gesamte Lösung vollständig modellbasiert dargestellt und es gibt keine Ausnahmen von dieser Regel, damit eine maximale Variabilität bei der Ausprägung von einzelnen Lösungen existiert. Software-Produktlinien erlauben hier prinzipiell ein beliebiges Herangehen mit allen Vor- und Nachteilen (höhere Flexibilität versus uneinheitlicheren Strukturen).
- Conceptual Programming als Gesamtsystematik verlangt Conceptual Pair Programming als eine der zentralen Methoden bei der Ausprägung von Einzellösungen (siehe Abschnitt 6.4.3.5), Software-Produktlinien im Allgemeinen erlauben ein beliebigeres Vorgehensmodell bei der Umsetzung von Lösungen. Auch

[2] Das ist so gewollt, um von den Erfahrungswerten zu und Vorteilen von Software-Produktlinien profitieren zu können.

[3] Wir werden die Begründungen der Einzelentscheidungen nicht mehr wiederholen – diese wurden ausführlich in den vorausgegangenen Kapiteln, insbesondere in Teil I und II, beschrieben.

[4] Dies wird im Wesentlichen durch die betrachtete Domäne und das Vermarktungsmodell bestimmt.

hier besteht wieder ein Gegensatz zwischen einer flexibleren Vorgehensweise im Allgemeinen und einem kostenoptimierteren Ansatz, der Spätfolgekosten durch stärkere Früherkennung vermeidet, dafür aber höhere initiale Investitionen in die zugrunde liegenden Werkzeuge voraussetzt.

- Software-Produktlinien sind eigentlich ein relativ dokumentationsintensives Vorgehensmodell, Conceptual Programming (und insbesondere CMDSD) weichen diesen dokumentationsintensiven Prozess durch ein grundlegend agileres Vorgehensmodell auf (siehe dazu unsere Ausführungen in Abschnitt 5.4.3.3).

So wird offensichtlich, dass der von uns vorgestellte Ansatz eine konsequente Weiterentwicklung von Software-Produktlinien im Hinblick auf das Ziel der Arbeit ist.

7.1.5 Domain Driven Design

Eric Evans (siehe Abschnitt 3.2.7 bzw. [Evan04a]) hat in seiner Arbeit zu Domain Driven Design maßgebliche Grundlagen für das von uns formulierte Vorgehensmodell des Conceptual Programmings geschaffen. Dennoch weist Conceptual Programming eine Reihe neuer Eigenschaften auf:

- Während Domain Driven Design aufgrund eines ausführlichen explorativen Vorgehensmodells sowohl das Domänenmodell als auch seine Umsetzung iterativ und evolutionär erarbeitet, macht sich Conceptual Programming die *economies of scope* (siehe Seite 39) zunutze, um auf Grundlage von Software-Produktlinien und einem sogenannten CMDSD-Werkzeug die effiziente Umsetzung von komplexen Projekten im Mittelstand zu ermöglichen. Wir gehen also grundsätzlich davon aus, dass wir ein Modellierungswerkzeug erstellen (bzw. von einem anderen Dienstleister erhalten haben), mit dem wir domänenspezifische Projekte umsetzen. Das Werkzeug muss die in Kapitel 5 geforderten Anforderungen erfüllen, um unseren Erwartungen an Wirtschaftlichkeit, Zugänglichkeit und Flexibilität gerecht zu werden.
- Conceptual Programming spezialisiert sich auf die Realisierung kleiner und mittlerer IT-Projekte. Dazu haben wir die grundlegenden Ideen von Evans aufgegriffen, um mit Ihnen die Vorarbeiten und die eigentliche Realisierung eines CMDSD-Werkzeugs zu ermöglichen. Die Komplexität der Domänenabgrenzung verlagern wir aber in den Werkzeugentwurf, so dass Kunden und Domänenexperten davon profitieren können, dass die komplexen technischen Entscheidungen bereits getroffen wurden und nur selten erneut diskutiert werden müssen. Stattdessen können Sie sich auf die Modellierung des konkreten

Problems mit einer bereits gefestigten domänenspezifischen Sprache konzentrieren.

Die Ideen von Evans liefern also einen essentiellen Beitrag zu für die erste Hauptmethode unseres Lösungsmodells, indem sie uns eine bewährte Sammlung von Entwurfsmustern und Methoden zur Realisierung von CMDSD-Werkzeugen bereitstellen. Conceptual Programming geht aber einen Schritt weiter und liefert dann auch ein konkretes Vorgehensmodell, um mit Endkunden gemeinsam basierend auf dieser Implementierung einer domänenspezifischen Sprache konkrete Projekte schnell und strukturiert umzusetzen. Dazu haben wir insbesondere konkretere Prinzipien und Methoden angegeben, als Evans dies zu leisten vermag, da er sich insbesondere darauf fokussiert, Domänen in technisch sauberen Architekturen abzubilden.

7.1.6 Intentional Programming

Wie in Abschnitt 3.1.2 ausgeführt, befasst sich Intentional Programming (IP) insbesondere mit der Bereitstellung einer dynamisch erweiterbaren auf Metaprogramming basierenden Plattform zur Ausprägung, Visualisierung und Eingabe domänenspezifischer Sprachen. Die Abgrenzung zu unserem Ansatz ist somit klar:

- IP verfolgt einen generischen Plattformansatz, dessen lange Entwicklungszeit (mehr als 13 Jahre, erste uns bekannte Veröffentlichungen von Simonyi zu diesem Thema stammen aus dem Jahr 1995, siehe [Simo95]) ihn für den von uns betrachteten mittelständischen Kontext nicht brauchbar macht[5]. Entsprechend haben wir unseren Ansatz durch Fokussierung auf eine Domäne und eine Einschränkung der Generizität bei gezielter Optimierung der besonderen Anforderungen des Mittelstandes (siehe dazu unsere Ausführungen in den Kapiteln 2 und 5) unter Berücksichtigung einiger Grundideen des IP (intentionale Abstraktion, grafische Modellierung) entwickelt.
- IP liefert keine Ansätze oder Ideen für die konkrete Nutzung des Ansatzes im Zusammenspiel mit Kunden. Zwar wird darauf eingegangen, dass grafisch ausdrucksstarke und domänenspezifische Spracherweiterungen anzustreben sind, das Problem der Vermischung (und damit potenziellen Komplexität) der resultierenden Sprachfamilien wird aber bestenfalls in der Sekundärliteratur (siehe z. B. [CzEi00], S. 559–562) ausführlicher betrachtet. Wir haben durch die Zweiteilung unseres CP-Ansatzes in CMDSD (den IP näher stehenden werkzeug-

[5] Wir haben bereits darauf hingewiesen, dass IP-Plattformen – sobald sie verfügbar werden – eine gute Grundlage für die Entwicklung von CMDSD-Werkzeugen sein könnten.

orientierten Aspekt) und CDPM (den in IP nicht betrachtet organisatorischen und operativen Aspekt) einen konsistenteren und umfassenderen sowie insbesondere in sich geschlosseneren Beitrag zur Ausprägung domänenspezifischer Vorgehensmodelle geliefert als dies bei IP der Fall ist.

- IP beschäftigt sich primär mit der internen Repräsentation, Interpretation und Transformation der durch IP benötigten Strukturen – zur eigentlichen DSL-Bildung werden keine Aussagen gemacht. Insbesondere gibt es keine uns bekannten Ausführungen und Best Practices zur Entwicklung von DSLs mit dem IP-Ansatz. Wir haben dagegen in Kapitel 5 eine Vielzahl von Hinweisen zur geeigneten Fachmodellbildung gegeben.

Zusammenfassend lässt sich also sagen, dass unser CP-Ansatz zwar Ideen des IP aufgreift, insgesamt aber ein ganzheitlicheres und für den Mittelstand optimiertes Vorgehensmodell bietet, dass insbesondere in sich geschlossen alle Aspekte moderner IT-Projekte betrachtet.

7.1.7 Concept Programming

Für das in Abschnitt 3.1.3 vorgestellte *Concept Programming* gelten aufgrund der dort bereits identifizierten Nähe zum *Intentional Programming* weitgehend unsere Ausführungen aus Abschnitt 7.1.6: Es wird eine spezifische generalisierte Plattform zur Definition von konzeptbasierten Sprachen vorgestellt. Der Urheber Dinechin liefert aus unserer Sicht allerdings wertvolle Anregungen für die Frage, wie kundenorientiertere Modellierungs- und Codierungssprachen hinsichtlich ihrer Zielerreichung durch die Verwendung der Pseudo-Metriken *syntactic noise*, *semantic noise*, *bandwidth* und *signal/noise ratio* bewertet und optimiert werden können. Diese Anregungen haben wir bei der Definition von CMDSD aufgegriffen und zur Definition unseres Vorgehensmodells genutzt (siehe Abschnitt 5.3). Unser Ansatz des Conceptual Programming ist insofern analog zu unserer Argumentation gegenüber dem Intentional Programming eine deutliche Erweiterung und Fortschreibung der Ideen des Concept Programming, ohne dass wir uns auf eine bestimmte Repräsentationsform festlegen.

7.1.8 Language Workbenches

Language Workbenches (siehe Abschnitt 3.1.6) sind per Definition von Martin Fowler Werkzeuge, die Language Oriented Programming (siehe Abschnitt 3.2.3) bzw. die

Verwendung domänenspezifischer Sprachen (siehe Abschnitt 3.2.8) unterstützen. Insofern sind die von uns in Kapitel 5 beschriebenen CMDSD-Werkzeuge auch Language Workbenches. Während Fowler aber eine reine (und wichtige!) Begriffsbildung betreibt, haben wir das Konzept unter besonderer Betrachtung mittelständischer Rahmenbedingungen dahingehend erweitert, dass wir notwendige und zu vermeidende Eigenschaften mittelstandsgeeigneter Language Workbenches (unter dem Begriff CMDSD-Werkzeug) charakterisiert haben. Wir haben die Begriffsbildung von Fowler also um neue Aspekte aus dem von uns betrachteten mittelständischen Kontext erweitert und das allgemeine Konzept konkretisiert und verfeinert.

Wir wollen noch kurz auf die wesentlichen Kritikpunkte Fowlers an der Verwendung von DSLs zur Lösung von Problemen eingehen und zeigen, wie wir diese Probleme gemindert oder vermieden haben (wir weisen darauf hin, dass CMDSD-Modelle in Fowlers Terminologie sowohl interne als auch externe DSLs sein können):

- Das Problem der *symbolic integration* (also der mangelnden Integration unterschiedlicher Sprachen in einer Entwicklungsumgebung) mindern wir durch die Forderung eines integrierten CMDSD-Entwicklungswerkzeugs, dass darauf optimiert ist, eine spezifische Domäne optimal und möglichst abstrakt zu modellieren (siehe Kapitel 5).
- Das Problem der *language cacophony* („eine Sprache zu lernen, ist bereits schwer – mehrere Sprachen zu lernen, ist noch schwerer“) reduzieren wir dadurch, dass wir zum einen nur eine fachliche Domäne betrachten, die wir auch aus fachlicher Sicht modellieren (so dass sich dem Anwender bzw. Kunden ein geschlosseneres Bild ergibt, siehe Abschnitt 4.2.2.2), zum anderen dadurch, dass wir fordern, dass die natürliche Beschreibung der Domäne (aus Fachexpertensicht) zur Modellbildung und Modellierung verwendet wird (siehe Abschnitt 5.4.2.8).
- Das Problem der *symbolic barrier* (also der mangelnden Propagation von Änderungen über Sprachgrenzen) reduzieren wir dadurch, dass signifikante Codeteile des ausführbaren Systems direkt aus den Modellen generiert und somit gar nicht für die Entwickler und Modellierer sichtbar werden.
- Das Problem beschränkter Syntax bei internen DSLs vermeiden wir dadurch, dass wir uns in unserem Ansatz nicht auf interne DSLs festlegen mussten – erweist sich die Syntax der zugrunde liegenden Programmiersprache also als zu begrenzt, können wir auf andere Sprachen ausweichen. Exemplarisch haben wir dies auch in Abschnitt 8.1.8.2.2 anhand unseres Prototypen beschrieben. Dies wirkt natürlich bis zu einem gewissen Grad kontraproduktiv zu unseren

Bemühungen, eine *language cacophony* zu vermeiden. Unseres Erachtens ist das aber ein untergeordnetes Problem, da wir nur in Ausnahmefällen auf weitere Sprachen zurückgreifen.

- Die essentiellen Probleme der *lay programmer* sowie der *COBOL inference* (also der Frage, inwieweit sich Programme ausschließlich durch Domänenexperten schreiben lassen – was Fowler anzweifelt) sind wir konstruktiv begegnet, indem wir mit CDPM ein Verfahren definiert haben, dass durch seine Conceptual-Pair-Programming-Methodik (siehe Abschnitt 6.4.3.5) natürliche Synergien von Domänen- und IT-Experten durch ein kooperatives Entwicklungsmodell verbindet und die so resultierenden Stärken durch das zugrunde liegende CMDSD-Werkzeug betont. Wir glauben also auch nicht daran, dass Programmierer vollständig zu ersetzen sind – wir denken aber schon, dass sie zunehmend höherwertige Aufgaben übernehmen werden (müssen) und insbesondere die enge Zusammenarbeit und Kommunikation mit den Domänenexperten einen zentralen Teil zukünftiger Stellenbeschreibungen ausmachen wird.
- Als verbleibenden Hauptkritikpunkt sehen wir wie Fowler das Problem der Entwicklungskosten für CMDSD-Werkzeuge. Diese haben wir zu mindern versucht, indem wir zum einen auf bewährte Methodiken wie Software-Produktlinien aufsetzen und zum anderen eine Reihe von grundlegenden Plattformen für die CMDSD-Werkzeugentwicklung identifiziert haben, die große Kostenvorteile bringen können (z. B. Software Factories und Plattformen für Intentional Programming). Des weiteren haben wir die Erwartungen an das Werkzeug durch Fokussierung auf bestimmte Prinzipien verdichten können, so dass viele potenziell teure Flexibilitäten nicht berücksichtigt werden müssen (siehe wiederum unsere ausführlichen Beschreibungen in Kapitel 5). Zudem vermeiden wir insbesondere die potenziell auf der Implementierungsseite sehr aufwändige Forderung nach einer Möglichkeit für Anwender, neue DSLs zu definieren – wir beschränken uns bewusst auf eine Domäne mit fest definierten Erweiterungspunkten zur flexiblen Modellierung innerhalb der Domäne. Dadurch wird die Implementierung der zugrunde liegenden Werkzeuge konzeptionell wie operativ drastisch vereinfacht. Als Beispiel sei dazu unser Prototyp herangezogen: Während die IP-Plattform von [InSo09] nach mehr als 13 Zeitjahren Entwicklung (und vermutlich wesentlich mehr Mannjahren) immer noch nicht allgemein verfügbar ist, kann unser Prototyp nach ca. 1,5 Zeitjahren (und knapp 3 Mannjahren Entwicklungsaufwand) bereits in Projekten für die Entwicklung innerhalb einer komplexen Domäne verwendet werden (siehe dazu unsere Ausführungen in Abschnitt 9.1.1).

- Das offene (und auch für unseren Ansatz geltende) Problem des *vendor lock-ins* (also der Abhängigkeit von einem Anbieter) können wir nur bedingt relativieren: Natürlich ist diese Abhängigkeit mindestens durch das CMDSD-Werkzeug gegeben. Theoretisch haben Dienstleister aufgrund der Standardisierung der Domäne die Möglichkeit, zumindest in einem gewissen Maße standardisierte Formate bereitzustellen[6]. Insofern sehen wir dieses Problem analog zur Alternative zu CP: dem Kauf von Standardsoftware. Auch hier existiert eine Abhängigkeit zum Anbieter. Insofern gelingt es uns zwar nicht, dieses Problem zu lösen, unser Ansatz verstärkt das Problem aber auch nicht.

7.1.9 Model-Driven Engineering

MDE greift eine Reihe der Ideen auf, die sich auch in unseren Optimierungspotenzialen für mittelständische IT-Projekte aus Kapitel 2 widerspiegeln (siehe Abschnitt 3.3.6): Insbesondere sind die Ausführungen zu den vielschichtigen Dimensionen, die bei der Konzeptionierung eines modellbasierten Ansatzes zu berücksichtigen sind sowie die Unterteilung der modellbasierten Entwicklung in Makro- und Mikroprozesse zu nennen. Wir haben letztere Idee auch aus anderen in Kapitel 3 betrachteten Verfahren abgeleitet und in unserer Methode des CDPM in Kapitel 6 aufgegriffen. Wir sind allerdings deutlich weiter gegangen als der MDE-Ansatz, in dem wir für die horizontale Domäne „Mittelstand“ nicht nur die zu betrachtenden Dimensionen (durch Formulierung der unserem Vorgehensmodell zugrunde liegenden Werte) detailliert sondern auch konkretere Angaben zur Modellbildung sowie eine Erweiterung für die zugrunde liegende Projektmanagementmethode definiert haben, die konkrete Handlungsanweisungen zum Umgang mit den verschiedenen Dimensionen gibt und so ein Metamodell für die Ausprägung vertikaler mittelständischer Domänen bereitstellt.

7.1.10 Conceptual Programming nach Reiss

Im Rahmen von Recherchen zur Namensgebung für unser Verfahren sind wir u. a. auf die Arbeiten von Steven P. Reiss gestoßen, der in seiner Einleitung zu [Reis89] formuliert: *„We introduce conceptual programming as a process model based on our programming experiences. Conceptual programming means having programmers work*

[6] Obwohl wir bezweifeln, dass sie dies aufgrund strategischer Überlegungen und des Schutzes von Märkten tun werden.

directly with their conceptual models, using their own terms or languages rather than those dictated by the computer. To be useful and effective, conceptual programming must be supported by a powerful programming environment. We have developed such an environment, GARDEN, and are beginning to get feedback on its use.“ Zudem postuliert er die folgenden Punkte:

- Entwickler sollen direkt mit multi-dimensionalen und visuellen Konzepten arbeiten, um Programme zu entwickeln.
- Der gesamte Entwicklungsprozess von Analyse über Entwurf und Implementierung bis hin zur Wartung soll mit einer einheitlichen Metapher behandelt werden.
- Spezifikationen sollten direkt ausführbar oder aber mindestens für die Prototypisierung verwendbar sein.
- Abstrakte Repräsentationen können hierarchisch oder überlappend sein.
- Fachliche Repräsentationen werden als Entwicklungssprachen angesehen.
- Die Modellierung dieser grafischen Repräsentationen soll durch ein entsprechendes Werkzeug unterstützt werden. Reiss schlägt hierfür das von ihm entwickelte System GARDEN (siehe [Reis89], S. 2 unten bzw. [Reis87]) vor.

Hier zeigen sich starke Ähnlichkeiten zu dem von uns formulierten Lösungsansatz zur Effizienzsteigerung in mittelständischen IT-Projekten. Insbesondere [Reis87] zeigt dann aber, dass Reiss einen anderen Lösungsansatz verfolgt, der sich auch in seiner weiteren Forschungsarbeit widerspiegelt, die sich zunehmend von seiner initialen Definition von Conceptual Programming entfernt und immer stärker die Visualisierung von Systemen zur Laufzeit oder die Validierung von Spezifikationen fokussiert (siehe [Reis09]):

- GARDEN erlaubt die Definition objektorientierter Sprachen, die spezifische Domänen beschreiben und neben einer textuellen Darstellung auch eine grafische Repräsentation besitzen können (siehe [Reis87], S. 225). Der Fokus liegt allerdings auf der Repräsentation spezialisierter objektorientierter Sprachen – allgemeinere Ansätze werden nicht berücksichtigt.
- So wie Reiss die Unterstützung verschiedener Sprachen in Analogie zu den verschiedenen Modelltypen in einem CMDSD-Werkzeug in seinem System fordert (siehe [Reis87], S. 227 bzw. Abschnitt 5.4.3.7), geht aber immer davon aus, dass eine einheitliche Plattform für die Definition aller Domänenausprägungen verwendet wird. Wir haben aus Vereinfachungsgründen dagegen gefordert,

ein möglichst domänennahes, auf Standards basierendes spezifisches Entwicklungswerkzeug zur Lösung der Domänenprobleme zu verwenden. Hier sehen wir wieder das Problem, dass universellere Ansätze sehr schnell eine extreme Komplexität erzeugen, was die Ausprägung neuer Lösungen erschwert (siehe dazu die Ausführungen auf Seite 34).

- GARDEN erlaubt nicht die Verwendung existierender Programmiersprachen oder Bibliotheken, sondern setzt ausschließlich auf die eigene Implementierung eines Interpreters für objektorientierte Sprachen (siehe [Reis87], S. 230). Dies stellt aus heutiger Sicht eine massive Einschränkung für die praktische Relevanz des Systems dar, da umfassende Arbeiten zur Entwicklung einer vollständigen Programmiersprache erforderlich wären, um praktische relevante Lösungen umzusetzen.
- Das Layout der grafischen Modelle scheint (soweit in [Reis87], S. 230 erschließbar) automatisch zu erfolgen. Dies ist gerade für ein kundenorientiertes und leicht zugängliches System hochgradig problematisch: Auch Reiss weist auf die Probleme mit automatischen Layoutheuristiken hin, die nicht zwingend zu ansprechenden (sprich: verständlichen) Ergebnissen führen.
- Die grafischen Beispiele aus [Reis87], S. 235 zeigen im wesentlichen (nicht besonders ansprechend) formatierte grafische Äquivalente zu Datenflussdiagrammen – der Wert eines solchen Ansatzes ist insbesondere für komplexe Probleme im Vergleich zu den kleinen beschriebenen Beispielprogrammen recht fragwürdig. Insbesondere geben die Beschreibungen von Reiss keinen Hinweis darauf, wie ein komplexerer Abstraktionsmechanismus arbeiten würde, der komplexe (und potenziell andere Systeme integrierende) Operationen abstrahiert beschreibt.

Zusammenfassend scheint das „Conceptual Programming“ von Reiss eine der vielen Vorläuferideen zu sein, die letztlich zur MDA und UML geführt haben. Insofern bietet der Ansatz die gleiche Grundidee, die wir verfolgen, setzt aber auf andere Mittel und scheint letztlich auch gescheitert zu sein, da das System keine praktische Relevanz erlangt hat.

Zusammenfassend sehen wir die Ähnlichkeit der grundlegenden Ideen als Bestätigung für unseren Ansatz – auch Reiss betont immer wieder, dass Feldstudien und Erfahrungen zeigen, dass die Kommunikation zwischen Entwicklern und Experten gefördert und die Zahl der Übersetzungsschritte zwischen Anforderungsdefinition und Implementation verkürzt werden muss. Reiss wählt dafür einen weniger integrierten und stärker auf objektorientierte Sprachen ausgerichteten Ansatz, der ins-

besondere keine Elemente enthält, die speziell die Beziehung zwischen Kunden und Entwicklern fördern. Zudem basiert sein Ansatz auf einem spezifischen implementierten Werkzeug – dem GARDEN-System. Er liefert keine Ansatzpunkte oder Hinweise auf die Entwicklung weiterer Werkzeuge für andere Problemkreise, die nicht rein objektorientiert zu behandeln sind – beispielsweise mit regelbasierten Systemen oder mit deklarativer Programmierung.

7.1.11 Abgrenzung zu Conceptual Programming nach Computing Research Laboratory (CRL)

Auch [PfHa91] verwenden den Begriff *Conceptual Programming* und die Abkürzung *CP* – wir wollen der Vollständigkeit halber nur darauf hinweisen, dass sie sich inhaltlich aber mit der graphentheoretischen Repräsentation von Wissen beschäftigen und hierfür eine Lösung entwickelt haben. Außer der zufälligen Namensgleichheit gibt es also keine uns bekannten Gemeinsamkeiten zu unserem Ansatz.

7.2 Zusammenfassung

Wir haben in diesem Kapitel unseren CP-Ansatz deutlich zu elf anderen Vorgehensmodellen abgegrenzt. Die elf Verfahren weisen auf den ersten Blick eine besonders starke Ähnlichkeit zu CP auf. Wir konnten aber zeigen, dass unser Ansatz eine neuartige Kombination etablierter und bewährter Verfahren darstellt, da es keinen vergleichbaren Ansatz gibt, der gleichzeitig hochwertige technische Abstraktionen mit einer automatisierten und modellbasierten Generierung von Lösungen verbindet und zudem noch eine agile Umsetzung durch die enge Zusammenarbeit von IT- und Domänenexperten in den Mittelpunkt des Vorgehensmodells stellt. Zwar verwenden die betrachteten Ansätze Teilaspekte unseres Vorgehensmodells, versäumen es aber, diese synergetisch zu kombinieren, um daraus einen höheren Nutzen insbesondere auch für den von uns beschriebenen mittelständischen Kontext zu ziehen.

In Teil III dieser Arbeit werden wir nun eine prototypische Umsetzung unseres Lösungsansatzes für eine bestimmte Lösungsdomäne beschreiben und die dabei gewonnenen Erkenntnisse und Erfahrungswerte zusammenfassen.

Teil III

Verifikation des CP-Ansatzes

Im dritten Teil wird das zuvor formulierte CP-Vorgehensmodell mit seinen Hauptmethoden CMDSD und CDPM anhand eines Praxisbeispiels verifiziert. Als Grundlage dient dazu ein Projekt der QuinScape GmbH (siehe [Quin09]), das in Kooperation mit der BALVI GmbH (siehe [Balv09]) entwickelt wird, um für die Domäne „Webbasierte Datenpflegeanwendungen und Portale“ ein CP-basiertes Vorgehensmodell zu entwickeln.

QuinScape agiert seit 2001 am Markt als Spezialist für webbasierte Intranet- und Extranetsysteme sowie Portale. Ein Geschäftsfeld setzt solche Lösungen auf Basis von Individualprojekten mit JavaEE-Technologien um. Ein zweites Geschäftsfeld realisiert derartige Lösungen mit dem Standardprodukt Intrexx (siehe [Intr09]).

BALVI ist deutscher Marktführer für Software zur Überwachung der Tiergesundheit, der Lebensmittel- und der Futtermittelsicherheit für öffentliche Auftraggeber und verfügt hier über einen Marktanteil von 100% – alle deutschen Bundesländer setzen mittlerweile die Software BALVI iP ein. Konsequenterweisewill das Unternehmen seine Tätigkeit auf internationale Märkte ausweiten und dort seine auf Delphi basierende Plattform durch neue (Java-)Technologien ersetzen. Letzterer Schritt ist ein langfristiges Ziel: In den nächsten Jahren werden zunächst webbasierte Zusatzprodukte um das Standardprodukt herum aufgebaut und zu einem noch zu planenden Zeitpunkt wird das alte Produkt durch das dann existierende neue Produkt abgelöst. BALVI und QuinScape kooperieren in einer engen Partnerschaft, um dieses Ziel zu erreichen.

Der Autor dieser Arbeit ist einer von drei geschäftsführenden Gesellschaftern der QuinScape GmbH und als Chefarchitekt und technischer Projektleiter zur Realisierung einer CP-basierten Lösung für die nächste BALVI-Produktgeneration tätig. Das Projekt hat Ende 2007 basierend auf dem damaligen Stand dieser Arbeit begonnen und läuft seitdem kontinuierlich.

Unterstützt wird das Projekt durch ein Forschungsprojekt im Rahmen des IKT.NRW-Förderprogramms (siehe Abschnitt 9.3.3 ab Seite 402 für weitere Details). Da das Projekt noch läuft und nach aktueller Planung aufgrund der Fokussierung auf die Domäne „Lebensmittelüberwachung mit webbasierten Datenpflegeanwendungen bzw. Portalen“ umfangreiche konzeptionelle lebensmittelüberwachungsbezogene Vorarbeiten durch die BALVI GmbH zu leisten sind, gehen wir in der aktuellen Planung von einer Fortführung des Projekts bis mindestens Ende 2010 aus, wobei seit 2009 erste Projekte auf Basis des sogenannten PIKAS-Frameworks[7] laufen. Da

[7]Dies ist der interne Name für das CMDSD-Werkzeug. Kapitel 8 beschreibt das Werkzeug genauer.

das System basierend auf agilen Ansätzen kontinuierlich weiterentwickelt und auch umfangreichen Refactorings unterzogen wird, stellt die Beschreibung in diesem Teil der Arbeit naturgemäß nur eine Momentaufnahme dar.

QuinScape und BALVI sind beides mittelständische Unternehmen mit jeweils 40–50 Mitarbeitern. Für die Umsetzung des CP-Ansatzes haben sich die Partner aus Ressourcengründen zu einem gestaffelten Vorgehen entschlossen, bei dem das Gesamtsystem in mehreren Phasen realisiert wird:

1. Initial wird die Modellbildung für eine ausreichende Menge von grundlegenden primären, sekundären und tertiären Modellen vollzogen, mit denen sich erste Anwendungen realisieren lassen. In den ersten Projekten findet eine Domänenmodellbildung für die noch unspezifischere und nachfolgend zu definierende Domäne „Portale“ statt. In späteren Phasen werden spezifischere Modelle für die Lebensmittelüberwachung ergänzt[8]. Die Modelle werden in Phase 1 auf Basis von XML-Dateien modelliert – genauere Details werden in Kapitel 8 erläutert.
2. In einer zweiten Phase wird eine Eclipse-Erweiterung realisiert, die das Editieren der XML-basierten Modelle stark vereinfacht und Entwickler nachhaltig in ihren Tätigkeiten unterstützt. Diese Entwicklungsarbeiten laufen parallel zur prinzipiellen Systementwicklung und haben im März 2009 mit einer Diplomarbeit (siehe [Kand09]) begonnen.
3. In einer dritten Phase wird das CMDSD-Werkzeug zur grafischen Modellierung umgesetzt. Bereits jetzt existiert für alle heute XML-basierten Modelle ein Konzept für die grafische Realisierung der Modelle. Details hierzu stellen wir in Kapitel 8 vor.

Vorteil dieses gestaffelten Vorgehens ist, dass bereits mit den Ergebnissen von Phase 1 erste Projekte umgesetzt werden können. Dabei werden die Modelle ausschließlich von Inhouse-Entwicklern verwendet. Das frühe Feedback erlaubt eine Anpassung und Optimierung der Modelle. Zudem können die Erfahrungswerte in der Nutzung der Modelle in die (einfachere) Entwicklung des Eclipse-basierten Werkzeugs in Phase 2 einfließen, während parallel dazu die Modelle erweitert und optimiert werden. Erst wenn die Modellbildung auf diese Weise hinreichend stabilisiert wurde, beginnt die (relativ aufwändige und komplexe) Implementierung der grafischen Editoren, um das

[8] Die interne Bezeichnung für das in dieser ersten Phase entstehende Grundgerüst ist PIKAS (als Akronym für Portal Implementation Kit: Application (Server ∨ Studio ∨ System), je nach dem, ob man über die Server-Laufzeitumgebung, das grafische Entwicklungswerkzeug oder das Gesamtsystem spricht.

Gesamtsystem damit abzurunden. Die Arbeiten der Phase 3 werden voraussichtlich Ende 2009 beginnen.

In Kapitel 8 werden wir die grundlegende Modellbildung beschreiben. Abschnitt 9.4 in Kapitel 9 beschreibt weitere Erfahrungswerte aus anderen parallel laufenden Projekten sowie bei der Herleitung des Ansatzes gewonnene Erkenntnisse. Kapitel 10 schließt die Arbeit mir einer Zusammenfassung der erzielten Ergebnisse sowie einem Ausblick auf zukünftige Aktivitäten ab.

Kapitel 8

CP am Beispiel „Portalsysteme“

„Ein gutes Beispiel ist der beste Lehrmeister.“
— Deutsches Sprichwort

Wie in der Einleitung zu Teil III dieser Arbeit beschrieben werden wir zunächst nur eine eingeschränkte und noch vergleichsweise allgemeine Domäne – Portale im mittelständischen Kontext – betrachten. Vorteil dieses Ansatzes ist es, dass nur wenig Vorkenntnisse für das Verständnis erforderlich sind und wir zudem anhand eines weithin bekannten Themas zeigen können, wie vergleichsweise einfach Modelle sein können, um komplexe Lösungen der Domäne zu modellieren. Wir werden uns im weiteren Verlauf auf die Domäne „*mittelständische Portalsysteme*“ fokussieren. Die horizontale Eingrenzung auf mittelständische Anwendungen resultiert aus der Fokussierung dieser Arbeit auf die Optimierung mittelständischer IT-Projekte. Die Auswahl des Themas *Portalsysteme* ergibt sich daraus, dass der Autor dieser Arbeit sich seit 1998 beruflich mit der Realisierung von Portallösungen beschäftigt und ein entsprechend tiefgreifendes Domänenwissen aufgebaut hat.

Wie in der Einleitung zu diesem Teil der Arbeit (ab Seite 307) ausgeführt, werden wir uns in unserer prototypischen Realisierung an einem konkreten Projekt aus dem Bereich der Lebensmittelsicherheit orientieren. Wir werden in den weiteren Ausführungen aus Platzgründen nicht den gesamten Herleitungsprozess darstellen, sondern nur die Ergebnisse vorstellen und begründen. Ergebnis des Entwicklungsprozesses ist das CMDSD-Werkzeug PIKAS, das bei der QuinScape GmbH auch in produktiven Projekten eingesetzt wird.

In Abschnitt 8.1 werden wir aufbauend auf der von uns in Abschnitt 5.3 (ab Seite 168) beschriebenen Vorgehensweise den spezifischen CMDSD-Ansatz für mittelständische Portalsysteme herleiten. Dabei werden wir besonderes Augenmerk auf

die Modellbildung legen und in Abschnitt 8.1.8 auf die besonders relevanten primären, sekundären und tertiären Modelle eingehen, indem wir zu jedem Modell die zugrunde liegende Idee, eine ausführlichere Beschreibung des Modelltyps, Hinweise zu einer besonders geeigneten grafischen Modellierung sowie Ideen für eine Vielzahl erweiterter Modellfacetten angeben. Zudem werden wir die technische Umsetzung der jeweiligen Modelle durch einen generativen Ansatz skizzieren. In Abschnitt 8.2 werden wir auf den Einsatz von CDPM im Rahmen dieser spezifischen Domäne eingehen. Abschnitt 8.3 fasst die Ergebnisse dieses Kapitels zusammen.

8.1 CMDSD am Beispiel „Portalsysteme“

Wir folgen in diesem Abschnitt dem Strukturüberblick zu CMDSD, den wir in Abschnitt 5.3 ab Seite 168 detailliert vorgestellt haben. Die Schritte 9 bis 10 werden im Rahmen der Modellerläuterungen zu Schritt 8 in Abschnitt 8.1.8 kurz beschrieben. Schritt 11 der CMDSD-Vorgehensweise wird hier nicht weiter vertieft, da sich das entsprechende Gesamtsystem noch in Entwicklung befindet und durch laufende Projekte Stück für Stück aufgebaut wird.

8.1.1 Schritt 1: Auswahl der Domäne

> *Wir werden ein System zur CP-basierten Entwicklung mittelständisch orientierter Portallösungen entwickeln. Unsere Domäne bezeichnen wir also als „mittelstandsorientierte Portale“.*

Portale wurden in der Vergangenheit auf vielfältige Weise definiert, insbesondere auch aufgrund ambitionierter Marketingaussagen vieler Portalhersteller – wir wollen daher präzise beschreiben, wie wir nachfolgend den Portalbegriff verwenden. Wir greifen für unsere Definition zunächst auf die folgende Basisdefinition aus [Wiki08h] bzw. originär aus [KGHV06] zurück: *„Ein Portal ist [...] eine Applikation, die [...] einen zentralen Zugriff auf personalisierte Inhalte sowie bedarfsgerecht auf Prozesse bereitstellt. Charakterisierend für Portale ist die Verknüpfung und der Datenaustausch zwischen heterogenen Anwendungen über eine Portalplattform. Eine manuelle Anmeldung an den in das Portal integrierten Anwendungen ist durch Single-Sign-On nicht mehr notwendig, es gibt einen zentralen Zugriff über eine homogene*

Benutzungsoberfläche. Portale bieten die Möglichkeit, Prozesse und Zusammenarbeit innerhalb heterogener Gruppen zu unterstützen.“

Wir wollen aber über den reinen Aspekt als Informationsaggregator hinausgehen und die prozessunterstützende und -optimierende Bedeutung von Portalen betonen, da hier wieder das bereits in den vorausgegangenen Kapiteln immer wieder zitierte Dogma gilt, dass mittelständische Unternehmen Lösungen überwiegend mit dem Ziel einsetzen, Kosteneinsparungen oder Marktvorteile zu realisieren, die unmittelbar greifbar sind. Ein mittelbarer Vorteil durch die Beschleunigung der Informationsflüsse ist nach unserer Erfahrung im Mittelstand nur ein sekundäres Argument. Dazu zitieren wir wieder [Wiki08h] bzw. originär [Pusc04]: *„Sie können 'als web-basierte, personalisierbare und integrierte Zugangssysteme zu internen und externen Applikationen definiert [werden], die der Unterstützung von Kunden-, Lieferanten und Mitarbeiterprozessen dienen und welche die grafische bzw. audiovisuelle Frontend-Integration (auch über verschiedene Portale hinweg) umsetzen. Dadurch verschaffen sie internen und externen Benutzern einen rollen-basierten, prozessorientierten Zugang zu einem umfassenden Set an aufeinander abgestimmten Mehrwertdiensten. Sie ermöglichen dies durch die Bereitstellung übergreifender Dienste wie Sicherheit, Personalisierung etc. Der Nutzen für den Portalbenutzer ist die Backend-Integration dieser Services.'“*. Zuguterletzt wollen wir noch die Definition zu *Prozessportalen* aus [Wiki07] aufgreifen: *„Ein Prozessportal wird häufig mit einem Unternehmensportal gleichgesetzt oder verwechselt. In einem Unternehmensportal werden diverse Services und Anwendungen zur Verfügung gestellt. Das Prozessportal hingegen ist für die Ausführung von Unternehmensprozessen zuständig und ist damit ein Teil eines Unternehmensportals. Es nutzt die zur Verfügung stehen Anwendungen und Services im Kontext eines definierten und workflowgestützten Prozesses. Der Begriff Prozessportal wird üblicherweise in den Bereichen von Service- und Verwaltungsprozessen benutzt. In einem Prozessportal werden Prozesse nicht nur dokumentiert, sondern - wie der vorherige Absatz vermuten lässt – automatisiert. Im Gegensatz den häufig anzutreffenden digitalen Organisationshandbüchern werden hier die Prozesse tatsächlich ausgeführt – immer mit der richtigen Information zur richtigen Zeit am richtigen Platz.“*

In Kombination führen uns die Ausführungen des vorangegangenen Abschnitts zu der folgenden Definition, die wir als Grundlage für unsere weiteren Entwicklungsarbeiten bei einem CP-basierten Ansatz zur Realisierung mittelständischer Portallösungen verwenden werden:

Definition ((Unternehmens)Portal)**.** *Ein (Unternehmens)Portal ist ein webbasier-*

tes System mit den folgenden Eigenschaften:

1. *Es bietet einen zentralisierten Zugang zu heterogenen (d. h. lokalen wie entfernten) Datenquellen und Prozessen.*
2. *Die Inhalte adressieren verschiedene Stakeholder, die einen Bezug zu dem Unternehmen besitzen, z. B. Geschäftskunden, Mitarbeiter, Lieferanten und Endkunden*[1]*.*
3. *Es erlaubt die Kollaboration von Anwendern, d. h. die gemeinsame Be- und Verarbeitung von Dokumenten, Daten und Prozessen.*
4. *Informationen werden personalisiert aufbereitet, Prozesse als Workflows rollen- und bedarfsgerecht angeboten.*
5. *Es stellt übergreifende Infrastrukturdienste wie Single-Sign-On, Sicherheit, Benutzerverwaltung, Personalisierung, Datenzugriff, etc. bereit.*

Im Folgenden werden wir die Begriffe *Unternehmensportal* und *Portal* synonym verwenden.

Wir weisen noch auf Abbildung 8.1 hin, da hier in einem guten Überblick viele der essentiellen Bestandteile bzw. beteiligten Stakeholder an einem Unternehmensportal dargestellt werden, die uns auch in den weiteren Ausführungen immer wieder begegnen werden. [Pres09] gibt einen ausgezeichneten Überblick über die Bestandteile eines solchen Unternehmensportals.

Der Autor dieser Arbeit hat die Rolle des Chief Programmers für das genannte Kooperationsprojekt übernommen.

Motivation der QuinScape GmbH

QuinScape ist seit 2001 als Spezialist für die Entwicklung von webbasierten Intranet-, Extranet- und Portalsystemen am Markt aktiv. QuinScape erstellt dazu einerseits Lösungen auf Basis des Standardprodukts Intrexx Xtreme (siehe [Intr09]) und realisiert andererseits Individuallösungen auf Basis verschiedener Java-Frameworks (insbesondere [Spri09a], aber u. a. auch [JSFC09, ASFS09, SunM09a, JSR315]). Zwischen diesen beiden Ansätzen existiert ein Bereich von Kundenanforderungen, der

[1] Wir fordern hier keine Exklusivität, da in der Praxis alle Formen von Portalen zu finden sind. Häufig wird aus schon aus Gründen der Datensicherheit Sinn machen, Portale nach Zielgruppen voneinander zu trennen. Dem entgegen stehen aber genauso häufig Aufwands- und Geschwindigkeitsvorteile, wenn alle Stakeholder in einem eng integrierten System arbeiten.

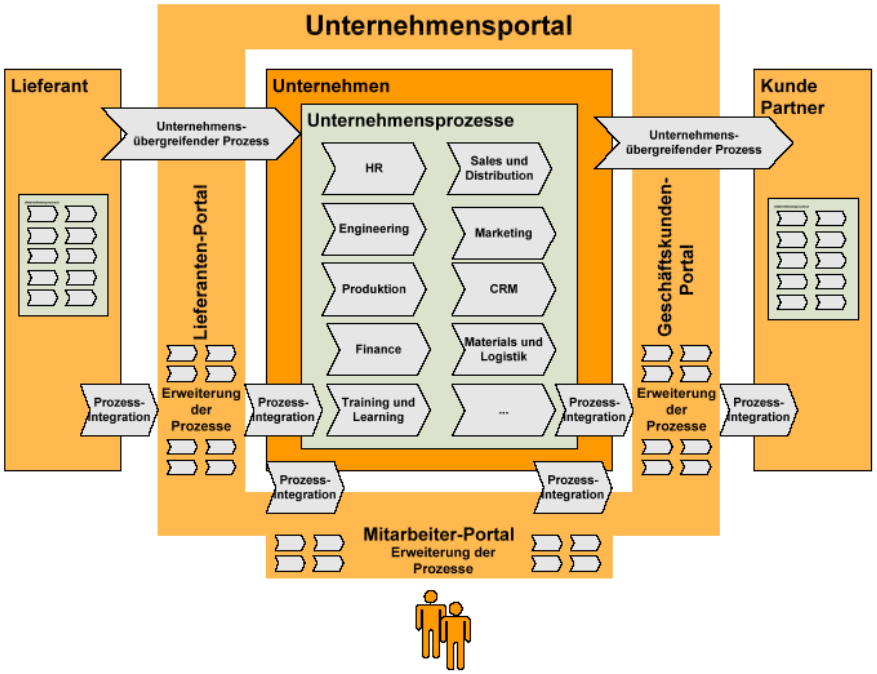


Abbildung 8.1: Unternehmensportale im Überblick (aus [Gurz08], S. 10)

durch beide Ansätze nur unzureichend abgedeckt wird: Eine Reihe von Kunden verfügen über Anforderungen, die ein über ein Standardprodukt hinausgehendes Maß an Flexibilität erfordern, das zu deutlich erhöhten Kosten in der Umsetzung führt. Andererseits sind die Projektbudgets nicht umfangreich genug, um eine Individualentwicklung von Grund auf zu ermöglichen, da viele Standardkomponenten (von der Navigation bis zur Rechteverwaltung) immer wieder benötigt werden und deren Individualentwicklung insbesondere für das mittelständische Umfeld zu teuer ist. QuinScape bewegt sich also genau in dem in Kapitel 2 beschriebenen Spannungsumfeld.

Motivation von QuinScape bei der Entwicklung von PIKAS ist also die Schaffung eines Standards für die domänenorientierte und generative Entwicklung von Portalanwendungen, die bei klassischen Ansätzen einen vergleichsweise hohen Programmier- und Anpassungsaufwand nach sich ziehen, so dass die Verwendung fertiger Standardprodukte eher ungeeignet ist. Andererseits sollen nur hinreichend standardisierte Prozesse besonders effizient abgebildet werden können, um die typischen repetitiven Tätigkeiten bei der Entwicklung klassischer formularbasierter Anwendungen bei manueller Programmierung abzuschaffen[2].

[2]Dies gilt insbesondere in den Bereichen Internationalisierung, Datenbindung, Validierung, Datenquellenanbindung, Navigation, Rechteverwaltung, etc.

Motivation der BALVI GmbH

BALVI hat wie bereits in der Einführung zu Teil III ab Seite 307 mit dem System BALVI iP ein sehr leistungsfähiges System zur Überwachung der Tiergesundheit, der Lebensmittel- und der Futtermittelsicherheit im Segment öffentlicher Auftraggeber etabliert. Ziele von BALVI sind daher die Ausweitung auf das internationale Geschäft sowie die Migration auf eine zukünftig nachhaltig tragfähige Technologieplattform. Das aktuelle BALVI-iP-System ist als Windows-Fat-Client auf Basis der Programmiersprache Delphi (siehe [Wiki09i]) realisiert, Kundenanforderungen bewegen sich aber zunehmend in Richtung webbasierter Lösungen aufgrund der einfacheren Installations- und Updatefähigkeit der Systeme sowie der erheblich einfacheren Verteilung der Lösungen.

Zudem sollen neue Anwendungskomponenten nach einem prozessbezogeneren Paradigma entwickelt werden, bei dem die jeweils aktuell durchzuführende Tätigkeit im Vordergrund steht. Dadurch werden Einarbeitungs- und Schulungsaufwände reduziert, da der Ablauf von Prozessen klarer strukturiert werden kann. Das gegenwärtige System verfolgt einen sehr konfigurierbaren und universellen Ansatz, der insbesondere Experten und sogenannte Power-User in ihrer Arbeit unterstützt, da es auf vielen Masken und Dialogen möglich ist, sehr komplexe und umfangreiche Anpassungen am System vorzunehmen. Nachteilig wirkt sich hier aus, dass Gelegenheitsanwender von der Vielzahl der Möglichkeiten schnell überfordert werden und sich schwer damit tun, Standardanforderungen umzusetzen.

8.1.2 Schritt 2: Identifikation des Architekturstils

Für die Definition unseres Architekturstils greifen wir auf das Entwurfsmuster *Layered Architecture* aus dem Domain Driven Design zurück (siehe Abschnitt 3.2.7). Webanwendungen sind mehrschichtige Anwendungen, die über eine schlanke Protokollschicht (normalerweise HTTP) die Kommunikation zwischen Client und Server abwickeln.

Wir verwenden die in Abbildung 8.2 dargestellte Schichtenarchitektur mit den folgenden Festlegungen:

- Die *Client-Schicht* realisiert die grafische Benutzeroberfläche. Da wir Portale implementieren wollen, wird es sich dabei um eine Mischung aus HTML und JavaScript handeln. Wir legen zudem fest, dass wir auf aktive Inhalte (wie

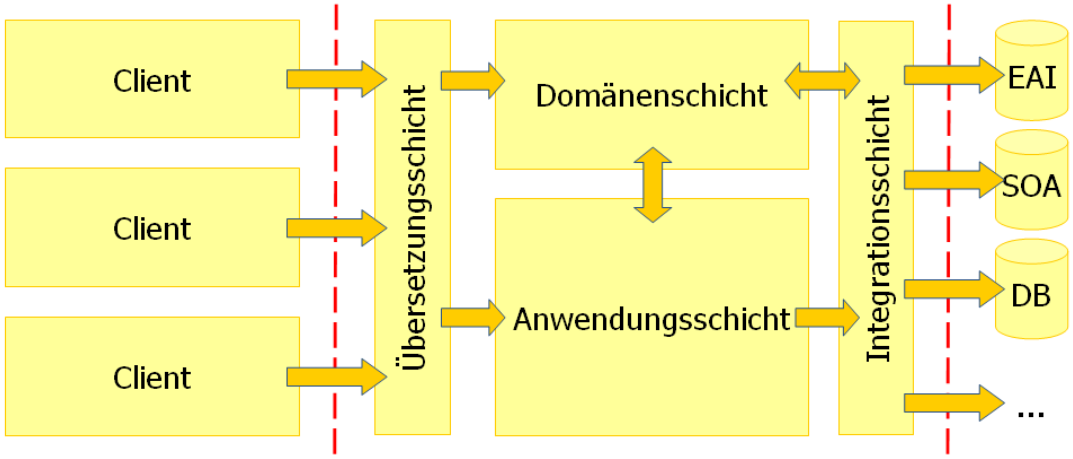


Abbildung 8.2: Schichtenmodell für PIKAS

Java Applets, Flash, usw.) verzichten wollen, wenn es keine zwingenden technischen Gründe dafür gibt[3].
Desweiteren legen wir für diese Schicht fest, dass sie nach Möglichkeit vollautomatisch aus Modellen, die eine logische Strukturierung von Benutzersichten vornehmen, generiert werden soll. Anreicherungen um moderne Funktionen wie AJAX-Support, Funktionsfähigkeit ohne JavaScript, Barrierefreiheit nach SAGA 4 (siehe [Bund08], S. 108 zur „Barrierefreien Informationstechnik Verordnung (BITV)“), usw. sollen automatisch durch den Generator eingewoben werden, um so eine konsistent hochwertige Benutzeroberfläche bereitzustellen.

- Die *Übersetzungsschicht* transformiert Werte aus fachlichen Objekten für die Darstellung auf dem Client und nimmt in umgekehrter Richtung Eingaben vom Client entgegen und transformiert diese wieder in fachliche Objekte. Zudem steuert sie den allgemeinen Kontrollfluss.
  Die Übersetzungsschicht wird vollständig durch den Generator bereitgestellt. Dieser beinhaltet das notwendige Wissen über die verwendeten Client- und Servertechnologien und kann daher ein automatisches Mapping zwischen Client- und Serverrepräsentation bereitstellen. Für den Anwender des Frameworks ist die Übersetzungsschicht also transparent, sie dient zur Steigerung der Evolutionsfähigkeit des Gesamtsystems.

- Die *Domänenschicht* kapselt die domänenspezifische Objektmodellierung der fachlichen Domäne. Wir legen fest, dass für jedes Domänenobjekt eine Factory

[3] Dadurch erhöhen wir die Benutzbarkeit der Portale für Anwender mit körperlichen Einschränkungen, reduzieren die Ladezeiten der Seiten und sichern eine breitere Zugänglichkeit, da keine Browser-Erweiterungen installiert werden müssen.

und ein Repository (siehe Abschnitt 3.2.7) bereitgestellt werden müssen – diese werden durch den Generator in spezifischen Instanziierungen konfiguriert werden und beruhen auf verschiedenen Implementierungen.
Die Domänenschicht soll aus Domänenmodellen generiert werden. Sie enthält alle Fachobjekte, Sichten, usw. In dieser Schicht werden zudem die Verbindungen zur Integrationsschicht definiert. Wir wollen dabei sowohl lokale Datenbanksysteme wie auch nahezu beliebige andere externe Datenquellen unterstützen.

- Die *Anwendungsschicht* stellt alle Operationen bereit, die sich nicht aus dem Kontext eines einzelnen Aggregats (siehe wiederum Abschnitt 3.2.7) berechnen lassen. Insbesondere werden hier alle Funktionen gesammelt, die über einer Menge von (potenziell unterschiedlichen) Domänenobjekten definiert sind.
  Die Anwendungsschicht kapselt die komplexe Geschäftslogik sowie die Workflowbeschreibungen des Systems, die Benutzerinteraktionen und Hintergrundprozesse steuern. Sie setzt sich zusammen aus generierten (z. B. der konkreten Implementierung der Workflowsteuerung) und manuell programmierten Anteilen (z. B. komplexer Geschäftslogik). Wir legen fest, dass aus unserer Sicht Programmcode in einer geeigneten Programmiersprache die beste Repräsentationsform für komplexere Logik ist und wir daher eine gute Unterstützung für entsprechende Programmierschnittstellen bereitstellen werden.
- Die *Integrationsschicht* wickelt über das DAO-Entwurfsmuster[4] alle Zugriffe auf Datenquellen und -senken ab. Sie instanziiert somit Domänenobjekte und wird ihrerseits sowohl von der Service- als auch Domänenschicht zur Speicherung von Objekten benutzt. Sie ist verbunden mit verschiedenen Backendsystemen, die die eigentliche Speicherung übernehmen (z. B. Datenbanksystemen, Messaging-Systemen oder Web Services).

Wir weisen darauf hin, dass die klar getrennten Client- und Übersetzungsschichten es uns ermöglichen werden, UI-Implementierungen so transparent auszutauschen, dass die tieferliegenden Schichten dadurch nicht beeinflusst werden[5]. Gleichermaßen können die Details der Objektpersistierung durch die explizite Definition der

[4] DAO steht für Data Access Object. Ein DAO stellt elementare Operationen zum Anlegen, Ändern, Löschen und Finden von Objekten (üblicherweise eines Typs) bereit. Weitere Details finden sich unter [Wiki09j].

[5] Faktisch haben wir dies im Rahmen der prototypischen Entwicklung bereits erprobt: Unsere initiale UI-Implementierung beruhte auf einem eigenen Web Framework, dass wir im Frühjahr 2008 innerhalb von sechs Wochen durch eine Implementierung auf Basis von JSF und Spring Faces ausgetauscht haben. Tatsächlich waren dazu keine Änderungen an den tieferliegenden Schichten oder den Modellen erforderlich, sondern es mussten lediglich die Client- und die Übersetzungsschicht neu implementiert werden.

Integrationsschicht ausgetauscht werden ohne andere Komponenten strukturell zu beeinflussen.

Zudem legen wir für die Implementierung fest, dass jedes Implementierungsartefakt entweder eindeutig einer Schicht oder aber der Kommunikationsschnittstelle zwischen zwei Schichten zuzuordnen sein muss. Desweiteren legen wir fest, dass wir für die Entwurfsmuster aus [GHJV96] zur Kommunikation zwischen Entwicklern verwenden werden. Zudem definieren wir Namenskonventionen für die Implementierung spezifischer Konzepte auf Java-Code-Ebene[6].

Auf der technischen Ebene definieren wir die folgenden Rahmenbedingungen:

- Wir werden Java als gegenwärtig verbreitetste und mit sehr umfassenden Zusatzbibliotheken ausgestattete Programmiersprache als Hauptimplementierungssprache verwenden (siehe [SunM09b]).
- Wir werden das Spring-Framework als Infrastrukturframework verwenden (siehe [Spri09a]). Spring ist das gegenwärtig modernste Open-Source-Java-Framework zur Abstraktion von Komplexitäten der Java-Enterprise-Spezifikation. Zudem glänzt es durch eine exzellente und sehr kompakte Dokumentation. Wir erwarten uns dadurch signifikante Implementierungsvorteile.
- Wir werden JavaServer Faces (JSF, siehe [JSFC09]) als Standardwebtechnologie verwenden, da JSF ein ausgereiftes und flexibles Komponentenmodell besitzt.
- Wir werden Spring Webflow (siehe [Spri09b]) zur Implementierung der UI-Flusssteuerung und für den Kontrollfluss der Backendprozesse verwenden. Insbesondere nutzen wir dazu auch die Spring-Faces-Integrationsschicht für die Anbindung an JSF.
- Wir werden jQuery (siehe [Resi09]) als Grundlage unserer JavaScript-Funktionen verwenden.
- Wir werden FreeMarker (siehe [Free09a]) als Grundlage für die Templating-Engine unseres Generators verwenden. Wir verzichten bewusst auf komplexere Frameworks wie openArchitectureWare (siehe [OAW]), da die notwendigen Einarbeitungsaufwände uns in keinem sinnvollen Verhältnis zu den relativ simplen zu implementierenden Generatoren zu stehen scheinen.

[6] Diese wollen wir hier nicht weiter ausführlich vorstellen. Beispiele sind: Alle Modellimplementierungen enden auf `Model`, ein Service endet immer auf `Service`, eine Factory immer auf `Factory`. Gibt es für eine Schnittstelle nur eine sinnvolle Implementierung, so erhält diese den gleichen Namen wie die Schnittstelle selbst, ergänzt um den Suffix `Impl`. Zudem geben wir eine Paketstruktur für die Java-Packages vor und definieren mit AspectJ Architektur-Aspekte, die die Einhaltung von Konventionen überwachen.

- Wir werden Hibernate (siehe [RedH09]) als Grundlage für die lokale Persistenzimplementierung verwenden, wobei wir auf Spring-Abstraktionen zurückgreifen, um die tatsächlichen Zugriffe zu implementieren.

8.1.3 Schritt 3: Planung der CMDSD-Werkzeugentwicklung

Für die CMDSD-Entwicklung entscheiden wir uns aus Ressourcengründen für ein initial FDD-basiertes Vorgehensmodell (siehe unsere Ausführungen zu Feature Driven Development ab Seite 131). Da in der initialen Implementierungsphase keine vollständige Anforderungsdefinition vorliegt und sich viele Detailanforderungen erst mit den ersten Projekten ergeben werden, wollen wir uns darauf konzentrieren, die als besonders wichtig identifizierten Features möglichst schnell umzusetzen. Geplant ist ein gradueller Umstieg auf einen formalen Software-Produktlinien-Ansatz gemäß unserer Ausführungen von Schritt 3 auf Seite 169 in Abschnitt 5.3.

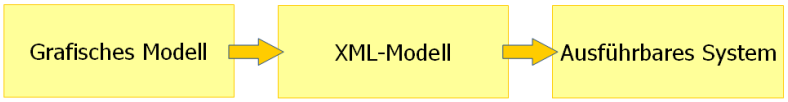


Abbildung 8.3: Modellrepräsentationen in PIKAS

Für das CMDSD-Werkzeug sehen wir die in Abbildung 8.3 prinzipielle Architektur vor. Modelle werden grafisch entworfen und in einem XML-Format serialisiert. Der Generator greift auf die serialisierten XML-Modelle zu und transformiert diese in (ggf. kompilierte) Implementierungen der Zielarchitektur. Hier legen wir fest, dass wir für die initiale Implementierungsphase noch keine grafischen Modelle implementieren, sondern uns auf die XML-Ebene begeben und auf dieser Basis die ersten Systeme spezifizieren werden. Damit wollen wir erreichen, dass

- keine Aufwände in die aufwändige Entwicklung grafischer Editoren investiert werden, da diese immer wieder geändert werden müssen, während sich die grundlegenden Modelle noch stabilisieren,
- die sehr begrenzten zur Verfügung stehenden Ressourcen[7] daran arbeiten, möglichst schnell ein für reale Projekte geeignetes Rahmen-Framework bereitzustellen,
- die Qualität der Modelle so frühzeitig wie möglich praktisch verifiziert werden kann,

[7] Im Schnitt werden uns gerade mal 2–3 Entwickler zur Verfügung stehen.

- möglichst früh durch Kundenprojekte zusätzliche Mittel für die Weiterentwicklung akquiriert werden können.

Daher werden die XML-Modelle zusätzlich darauf ausgerichtet sein, eine möglichst gut menschenlesbare domänenspezifische Sprache zur Beschreibung von Portalszenarien bereitzustellen. Der *Chief Programmer* muss daher zudem darauf achten, dass die XML-basierte DSL insbesondere auch dafür geeignet ist, später in grafischen Editoren gut integriert werden zu können. Dies erreichen wir durch die folgenden Maßnahmen:

- In den Abschnitten 3.2.2 und 3.3.6 haben wir auf ein klassisches Problem von CASE-Werkzeugen hingewiesen: Das nebenläufige Arbeiten an CASE-basierten Lösungen war häufig kaum möglich, da aufgrund der sehr generischen Natur von CASE-Werkzeugen natürliche Domänengrenzen fehlten, um Gruppen von Artefakten als zusammengehörig zu erkennen und diese entsprechend nebenläufig bearbeiten zu können. In Abschnitt 5.4.3.8 haben wir für unseren Ansatz beschrieben, wie Modelle in primäre, sekundäre und tertiäre Modelle aufgeteilt werden können. Wir wollen diese Segmentierung nun nutzen, um kurz anzudeuten, wie nebenläufiges Arbeiten in PIKAS zu einem hohen Grad ermöglicht werden kann.

  Die in Abschnitt 8.1.8 zu beschreibenden Modelle werden wir so auf XML-Artefakte (also Dateien) verteilen, dass die Modellierungseinheiten, die sinnvollerweise durch einen Entwickler exklusiv angepasst werden (also z. B. ein View, ein Prozess, eine Teilnavigation) können. Dies erlaubt eine einfache Parallelisierung der Arbeiten. Wir weisen darauf hin, dass für größere Projektteams sicherlich eine feingranularere Form der Parallelisierung erforderlich sein würde. Da im mittelständischen Kontext Projektteams aber selten mehr als fünf Personen enthalten werden, erscheint diese grobgranulare Form der Aufteilung mehr als ausreichend. Zudem werden wir Sperrmechanismen vorsehen, die wie folgt arbeiten:

  - Es werden verändernde und lesende Rechte unterschieden.
  - Primäre Modelle stellen die zentrale Grenze für nebenläufige Arbeiten dar, d. h. wir erlauben ändernde Operationen immer nur durch eine Person zur gleichen Zeit für ein primäres Modell. Wird ein primäres Modell in einem grafischen Editor geöffnet, fordert der Editor schreibende Rechte am aktuellen betrachteten Modell und lesende Rechte an allen referenzierten Modellen an. Diese Anforderungen werden wie folgt behandelt:
    * Schreibrechte werden gewährt, wenn keine andere Instanz Lese- oder

Schreibrechte angefordert hat.

* Leserechte werden gewährt, wenn keine andere Instanz Schreibrechte angefordert hat.

Werden alle Rechte gewährt, darf der Anwender mit der Bearbeitung des aktuellen Modells beginnen, andernfalls werden alle Anforderungen zurückgenommen.

Beispiel. *Entwickler Meier will einen View verändern. Die direkten Abhängigkeiten des Views sind alle enthaltenen Modelle (Widgets, Parts, Subviews). Die indirekten Abhängigkeiten sind der übergeordnete Prozess, die refenzierten Domänentypen, etc. Das Entwicklungswerkzeug fordert nun schreibenden Zugriff auf die direkten Abhängigkeiten und lesenden Zugriff auf die indirekten Abhängigkeiten. Besitzt kein anderer Entwickler schreibenden Zugriff auf indirekte Abhängigkeiten und besitzt auch niemand lesenden Zugriff auf die direkten Abhängigkeiten (insbesondere den View selbst), so wird Meier der Zugriff gewährt und er kann die zusammenhängende Modellgruppe verändern. Will er nun in einem referenzierten Domänentypen eine Eigenschaft ergänzen, muss er auch für den Domänentyp schreibenden Zugriff anfordern. Wird der Domänentyp wiederum von niemand anderem gelesen oder beschrieben, so kann er hier fortfahren.*

Theoretisch sind natürlich auch feingranularere Mechanismen möglich. Zum Beispiel könnte das System für die direkten Abhängigkeiten prüfen, welche indirekten Abhängigkeiten sie wiederum erzeugen und dann nur Veränderungen an direkt abhängigen Modellen verhindern, deren indirekte Abhängigkeiten bereits zum Schreiben gesperrt sind. Alternativ könnte der schreibende Zugriff differenzierter betrachtet werden: So könnte nach Einfügen, Ändern und Löschen differenziert werden und ergänzende Änderungen werden auch dann zugelassen, wenn andere bereits einen lesenden Zugriff auf das Modell besitzen (das Modell wird durch Erweiterungen ja nicht verfälscht und die anderen Instanzen bekannte Version behält daher grundsätzlich ihre Gültigkeit). Letztlich glauben wir aber, dass dies eine einzelfallbasierte Frage ist: In den meisten Fällen sind mittelstandsorientierte Entwicklungsteams zu klein als dass ein so detailliertes (und entsprechend komplex zu realisierendes) Sperrkonzept benötigt würde. Entsprechend wollen wir unsere Betrachtungen zur nebenläufigen Bearbeitung von Modellen damit schließen, das wir feststellen, dass unsere Konzepte zur Modellbildung aus Abschnitt 5.4.3.8 sowie zur Komposi-

tion von Modellen (siehe Abschnitt 5.4.3.8.4) uns hinreichende Mittel an die Hand geben, um für typische mittelständische Projektgrößen eine nebenläufige Bearbeitung zu ermöglichen und so einen der zentralen Kritikpunkte an CASE-Werkzeugen zu entschärfen.

- Es wird darauf geachtet, dass alle Informationen über Abhängigkeiten und Beziehungen zwischen den Modellen explizit modelliert werden. Es werden ausschließlich Konventionen und Vorgehensweisen verwendet, die ein Softwaresystem automatisiert interpretieren kann, um so eine vollständig Verwertung der Modelle zu garantieren.
- Bevor ein Modell umgesetzt wird, wird ein Grobkonzept für eine grafische Modellierung entwickelt, um sicherzustellen, dass die XML-Beschreibung alle für den Editor relevanten Informationen enthält.

Außerdem ist ein Konzept zu entwickeln, dass es erlaubt, auf XML-Ebene sowohl eine grundlegende Lösungsimplementierung bereitzustellen, diese aber auch mit möglichst geringer Code-Redundanz erweiterbar und anpassbar zu gestalten. Dazu greifen wir auf das bereits in Abschnitt 5.4.3.8.3 auf Seite 218 beschriebene *Strategy Pattern* zurück: Technische Detailaspekte der Modellumsetzung werden in lokalen Strategieschnittstellen gekapselt. Entsteht Bedarf für eine Flexibilisierung dieser Schnittstellen werden sie ggf. durch XML-Attribute exponiert, um so zum Beispiel die Einbindung projektspezifischer Schnittstellen zu ermöglichen. Als Zwischenlösung wird bei verallgemeinerbaren Implementierungsvarianten die XML-Signatur des jeweiligen Modells dahingehend erweitert, dass die verschiedenen sinnvollen Implementierungsalternativen durch ein passendes deklaratives XML-Attribut angesteuert werden können. Ein Beispiel könnte die Unterstützung verschiedener relationaler Datenbanksysteme sein, die durch ein XML-Attribut wie z. B. `portal-db-system="oracle"` oder `portal-db-system="postgres"` in einem geeigneten tertiären Modell deklariert werden sollte.

Alternative Speicherformen zum XML-Format wie z. B. die Speicherung in einer Datenbank sind trivial denkbar. Der Vorteil des dateibasierten Ansatzes ist, dass wir bereits sehr früh in der Entwicklung des Systems mit gängigen Entwicklungswerkzeugen Modelle manipulieren können. Trotz der einfachen Repräsentation als XML-Dateien bietet diese Vorgehensweise eine Reihe von Vorteilen (neben der frühzeitigen Pflegemöglichkeit): Insbesondere können wir theoretisch von der konkreten Syntax unsere Modelle abstrahieren, wenn wir dies wollen. Es wäre leicht denkbar, den Parser für die Modelle so zu erweitern, dass er auch andere Formate als XML (z. B. basierend auf der Dateiendung) erkennt und diese vor der eigentlichen Com-

pilierung zunächst in das XML-Format übersetzt. Damit wird es möglich, selbst auf der Dateiebene (also bevor grafische Modelle implementiert sind) eine möglichst einfach zugängliche Repräsentationsform für die jeweiligen Modelle zu wählen, die potenziell lesbarer als XML ist. Denkbar wäre z. B. für Navigationsmodelle eine zeilenbasierte Darstellung, bei der jede Zeile einen Navigationseintrag repräsentiert und die Einrückungstiefe die Gliederungsebene der Navigation beschreibt. So bräuchte man gar kein Markup und die Zeileneinträge würden sich auf die ID des Prozessstartzustandes sowie den anzuzeigenden Navigationstext (repräsentiert durch ein Übersetzungstag) beschränken. Der Vorteil liegt klar auf der Hand: Unser Ansatz bietet einen guten Erweiterungspunkt, um auch textuelle Modelle so pflegbar wie möglich zu machen. Damit tragen wir den in [Fowl06b] von Fowler angesprochenen Diskussionspunkten zur Frage der sinnvollsten konkreten Syntax für eine DSL Rechnung, indem wir transparent und sehr einfach Alternativformate erlauben können, wenn uns dies sinnvoll erscheint (siehe auch unsere Ausführungen zu Language Workbenches in Abschnitt 3.1.6).

8.1.4 Schritt 4: Identifikation der Archetypen

Bevor wir in die detaillierte Modellbildung einsteigen, wollen wir zunächst eine Auswahl archetypischer Anwendungen für die Portaldomäne betrachten. In über 200 Projekten der QuinScape GmbH zur Realisierung von Intranet-, Extranet- und Portallösungen sind wir immer wieder auf die folgenden Archetypen gestoßen, die sich in verschiedensten Ausprägungen in den meisten Systemen wiederfinden[8]. Gewichtungs- und Anpassungsgrad sind dabei zwar unterschiedlich, die Grundmuster aber dennoch repräsentativ:

- Formularbasierte Anwendung (Abschnitt 8.1.4.1)
- FAQ (Abschnitt 8.1.4.2)
- Dokumentenliste (Abschnitt 8.1.4.3)
- Forum (Abschnitt 8.1.4.4)
- Blog (Abschnitt 8.1.4.5)
- Aufgabenverwaltung (Abschnitt 8.1.4.6)

[8] Typische Ausprägungen können sein: Urlaubsanträge, Reisekostenabrechnungen, Vorlagenverwaltungen, Mitarbeiterlisten, Kantinenpläne, Aufgabenverwaltungen, Projektsteuerungen, CRM-Systeme, Ressourcenverwaltungen, Dokumentenverwaltungen, Media-Asset-Management-Systeme, Redaktionssysteme, Dashboard-Anwendungen, Angebotsverwaltungen, Personaleinsatzplanungen, Qualitätsmanagementlösungen, Arbeitsschutzsicherheitslösungen, Innovationsmanagementlösungen, usw.

- Kalender (Abschnitt 8.1.4.7)
- Ressourcenverwaltung (Abschnitt 8.1.4.8)
- Wiki (Abschnitt 8.1.4.9)
- Kombinationsanwendungen (Abschnitt 8.1.4.10)

8.1.4.1 Formularbasierte Anwendung

Wir beginnen unsere Beschreibungen mit dem generischsten Archetypen für Module in Portalsystemen bzw. Intranets und Extranets: Dies ist die formularbasierte Anwendung. Diese dient zur Pflege von Daten und Steuerung von Workflows. Auch wenn diese Art von Anwendung nach unserer Erfahrung der häufigste Anwendungsfall ist, lässt sich keine spezifische Funktionliste zu ihrer Beschreibung angeben. Stattdessen identifizieren wir eine Reihe prinzipieller Anwendungsmuster dieses Archetypen, aus denen wir generische Anforderungen an unser System ableiten:

- Daten werden häufig in Master-Detail-Beziehungen verwaltet, um zunehmend spezifischere Drilldowns zu erlauben (siehe [Wiki08g]). Allgemeiner gesprochen wird ein relationales oder objektorientiertes Datenmodell zur benutzergesteuerten Navigation entlang der definierten Relationen bereitgestellt.
- Aus Gründen der Klarheit und Übersichtlichkeit wird meistens sinnvollerweise entweder eine Liste von Objekten oder ein einzelnes Hauptobjekt (ggf. mit verknüpften Unterobjekten) bearbeitet werden (siehe Master-Detail). Insofern muss es einfach sein, ein dediziertes Hauptobjekt je Detailseite zu identifizieren, dass z. B. auch als Aufhänger für einen Navigationspfad dienen kann (siehe Breadcrumb, [Wiki09c]).
- In einzelnen Sichten sollen Informationen möglichst aufgabengerecht angezeigt werden (d. h. dass eine rollenbasierte Datenfilterung stattfindet, was wir in Abschnitt 8.1.5.5 weiter ausführen).
- Mit der Verwaltung von Daten gehen durchaus komplexe Prozesse mit Verzweigungen, Genehmigungsstufen, alternativen Bearbeitungsmöglichkeiten, etc. einher, so dass wir derartige Prozesse abbilden können müssen (siehe Abschnitt 8.1.5.5).
- Daten zu komplexen Fachobjekten müssen über mehrere Seiten verteilt werden können.
- Reine Ansichtsseiten müssen ebenso wie Pflegeseiten realisierbar sein.

Viele Beispiele lassen sich für formularbasierte Anwendungen finden: Mitarbeiterverwaltungen, Listen von Marketingdokumenten, Reisekostenabrechnungen, Urlaubsanträge, Investitionsanträge, Ressourcenverwaltungen, Buchungsanwendungen, etc. Wie wir in den nachfolgenden Archetypen sehen werden, lassen sich viele von ihnen bei freierer Auslegung auch als formularbasierte Anwendungen auslegen. Wir werden aber auch sehen, dass die nachfolgenden Archetypen jeweils spezielle Zusatzfunktionen erfordern, die wiederum für eine Vielzahl von über den Archetypen hinausgehenden Szenarien wichtig sind, so dass wir sie explizit beschreiben.

8.1.4.2 FAQ

FAQ steht für „Frequently Asked Questions“ und ist einer im Usenet geborener Begriff für Listen mit häufig gestellten Fragen und Antworten. In typischen Portallösungen gibt es viele Arten von FAQs: Checklisten bei der Neueinstellung von Mitarbeitern, Einführungen für neue Mitarbeiter, aufgabenspezifische Wissenssammlungen, Online-Hilfen, technische Hilfsanleitungen, usw.

FAQs sind wie folgt charakterisiert:

- Sie enthalten eine Liste von Fragen und Antworten.
- Fragen werden textuell erfasst, Antworten können auch Bilder, Links und andere komplexe Objekte enthalten.
- Nutzer differenzieren sich (mindestens) in Leser und Redakteure.
- Leser können die Liste der Fragen einsehen und durchsuchen (üblicherweise stichwortbasiert) und die dazugehörenden Antworten lesen.
- Redakteure können Frage-Antwort-Paare anlegen, ändern und löschen.

Die folgenden Funktionen nehmen wir explizit nicht in den Archetypen auf, da sie unserer Erfahrung je nach Kunde sehr unterschiedlich gehandhabt werden und so besser kundenspezifisch ergänzt als auf Basis einer häufig unpassenden Default-Implementierung verändert werden:

- Wir beachten keine Genehmigungsworkflows, da diese manchmal einstufig, manchmal mehrstufig, mit einer unterschiedlichen Zahl von Verantwortlichen und erforderlichen Zusatzleistungen, etc. ausgeprägt sein können.
- Wir sehen keine Funktion zur Kommentierung von Einträgen vor, da dies nicht immer gewünscht ist und auch hier wieder unterschiedlichste Genehmigungsprozesse eine Rolle spielen können.

- Wir sehen kein Vorschlagswesen für Verbesserungen vor, da auch hier viele Varianten denkbar sind (z. B. ein zentrales Vorschlagswesen, eine E-Mail-Adresse, ein formularbasiertes Vorgehen mit oder ohne Rückmeldung, öffentliche oder private Vorschläge).

8.1.4.3 Dokumentenliste

Dokumentenlisten existieren in unterschiedlichster Ausprägung: Es kann sich dabei um Wissensdatenbanken, Vorlagen- oder Antragsverwaltungen, persönliche oder öffentliche Archive, usw. handeln. Häufig werden diese themenspezifisch unterschiedlich ausgeprägt sein.

Dokumentenlisten sind archetypisch durch die folgenden Anforderungen charakterisiert:

- In einer (filterbaren) Übersichtsliste werden alle verfügbaren Dokumente angezeigt. Auch das Durchsuchen der Anhänge muss möglich sein.
- Dokumente werden über Metadaten charakterisiert. Es sind mindestens die Bezeichnung, eine optionale Kurzbeschreibung, eine Stichwortliste, Einstellungsdatum und Einsteller sowie ein Anhang (in den meisten Fallen das eigentliche Dokument) vorzusehen.
- Anhänge können heruntergeladen werden.
- Dokumenteinträge können angelegt, verändert oder gelöscht werden.
- Dokumenteinträge können für die Dauer der Bearbeitung vor verändernden Zugriffen anderer geschützt (gesperrt) werden.

Die folgenden Funktionen haben wir bewusst außen vorgelassen, da sie wiederum sehr spezifisch unterschiedlich ausgeprägt werdne können und wir sie daher flexibel über Modellfacetten konfigurierbar machen wollen:

- Wir unterscheiden nicht zwischen verschiedenen Dokumentenzuständen (privat, öffentlich, Draft, usw.).
- Wir definieren wieder keine Freigabe- und Genehmigungsworkflows.
- Wir sehen zunächst keine Benachrichtigungsfunktionen zu Änderungen vor.

8.1.4.4 Forum

Foren gibt es in verschiedensten Ausprägungen: Beispiele sind Schwarze Bretter, öffentliche oder private Informationsbörsen, Diskussionsrunden zur Förderung des

Informationsaustauschs zwischen Kunden, Mitarbeitern, Partnern, usw., Supportforen.

Charakteristische Funktionen sind:

- Es können themenspezifische Gruppen angelegt werden.
- Nutzer werden (mindestens) nach Lesern, Redakteuren und Administratoren unterschieden.
- Je Gruppe können Lese-, Schreib- und Kommentierungsrechte festgelegt werden.
- Leser können alle Einträge einer Gruppe und die dazugehörigen Kommentare (bzw. “Followups“ als Usenet-Begrifflichkeit) einsehen. Zudem können Sie einzelne oder alle Gruppen nach Stichworten durchsuchen.
- Berechtigte Leser können Kommentare anfügen. Dabei können sie den Bezugstext ganz oder teilweise zitieren.
- Redakteure können neue Einträge schreiben.
- Administratoren können Gruppen anlegen, ändern oder löschen. Ebenso können Sie Einträge löschen. Zudem können Sie die Rechte von Rollen und Anwendern konfigurieren. Zuguterletzt können Sie Themen „schließen“, so dass keine weiteren Kommentare bzw. Followups möglich sind.

Auch hier nehmen wir weder Freigabe- noch Genehmigungsworkflows auf und berücksichtigen weder Anhänge noch komplexere Verschachtelungen von Threads (wir bieten lediglich eine listenartige Verschachtelung an, da sich diese mittlerweile in der Mehrzahl der Foren durchzusetzen scheint).

8.1.4.5 Blog

Blogs sind gegenwärtig ein sehr weit verbreitetes Kommunikationsmittel, um einer interessierten Öffentlichkeit mehr oder minder themenspezifische Informationen zukommen zu lassen. In Unternehmen werden Blogs zunehmend auch gerne sowohl als interne wie auch externe Marketing-Instrumente sowie als persönlicherer Ersatz für den klassischen Newsletter („Geschäftsführungsblog´“) eingesetzt. Es gibt aber auch eher technische und organisatorische Einsatzfelder wie z. B. Serverblogs („welche Änderungen wurden wann und von wem an einem Server vorgenommen?“).

Ein Blog ist charakterisiert durch die folgenden wesentlichen Funktionen (siehe [Wiki09k]):

- Einträge werden in umgekehrter chronologischer Reihenfolge ausgegeben.
- Einträge können kommentiert werden.
- Über einen sogenannten Feed können Einträge abonniert und mit speziellen Readern gelesen werden[9].
- Artikel können mit *Tags* (textuellen Markierungen) versehen werden, die den Artikel stichwortartig indizieren.
- Über sogenannte *Tag Clouds* (Tagwolken) erhält man einen Überblick über die am häufigsten thematisierten Begriffe.

Wieder lassen wir aus den bereits genannten Gründen Anmelde- und Genehmigungsworkflows außen vor und verzichten auch auf Features, die insbesondere für öffentliche Blogs wichtig sind (wie z. B. Permalinks, Trackbacks, usw.), da diese je nach Einsatzszenario technische Schwierigkeiten erzeugen könnten und je nach Anwendungsfeld sehr unterschiedlich gehandhabt werden.

8.1.4.6 Aufgabenverwaltung

Aufgaben werden überall verwaltet. Dabei wird differenziert nach Themen-, Kunden- oder Event-spezifischen Aufgaben, öffentlichen oder privaten Aufgaben, Personen oder Gruppen zugeordneten Aufgaben, usw.

Eine grundlegende Aufgabenverwaltung muss die folgenden Funktionen anbieten:

- Aufgaben müssen in gefilterten Listen einsehbar sein.
- Aufgaben haben Stammdaten wie Bezeichnung, Fälligkeitsdatum, Priorität, Verantwortlicher, Beschreibung, Status.
- Aufgaben unterliegen einem statusbezogenen Workflow.

Weiterführende Funktionen wie Benachrichtigungsworkflows, etc. sehen wir im grundlegenden Archetypen nicht vor, da auch hier wieder unterschiedlichste Ausprägungen und Konfigurationen denkbar sind.

8.1.4.7 Kalender

Kalender gibt es wiederum in vielen Varianten: Öffentliche oder private Kalender, Veranstaltungs-, Meeting-, Feiertags-, Urlaubs- oder Geburtstagskalender, usw. Allen gemeinsam sind:

[9] Als Format wird üblicherweise RSS (siehe [Wiki09q]) angeboten.

- Kalendarische Ansichten von Tagen, Wochen, Monaten und Jahren,
- die Möglichkeit, Termine anzulegen, zu ändern oder zu löschen,
- Suchmöglichkeiten.

Wir vermeiden auch hier Festlegungen zu der Frage, wer Termine eintragen oder ändern darf und vermeiden auch komplexe Funktionen wie Terminabstimmungen, etc., da diese wiederum sehr stark durch den jeweiligen Kontext beeinflusst werden.

8.1.4.8 Ressourcenverwaltung

Ressourcenverwaltungen sind ein Sonderfall von Kalendern, da für eine Kombination aus Ressource, Termin und Dauer die Ressource als belegt zu kennzeichnen ist. Auch hier gibt es wieder vielfältig denkbare Ausprägungen: Fuhrparkverwaltungen, Raumverwaltungen, Geräteverwaltungen (Beamer, Notebooks, usw.), Bibliotheken u. v. a. m. Allen gemein sind:

- Sie besitzen Übersichtsmatrizen, die für eine gefilterte Menge von Ressourcen und einen Zeitraum den Belegungsstatus anzeigen.
- Ressourcen können angefordert werden und das System muss dann entscheiden, ob die Ressource noch verfügbar ist oder nicht.
- Resourcen können angelegt, geändert und gelöscht werden.

Bei der Ressourcenverwaltung lassen wir insbesondere den Aspekt der Genehmigung außen vor, da hier vielfältigste Möglichkeiten denkbar sind. Wir beschränken uns stattdessen darauf, von einer sofortigen Genehmigung auszugehen und würden in der Implementierung des Archetypen entsprechende Erweiterungspunkte vorsehen, die es erlaube, zusätzliche Aktionen und Schritte einzuplanen.

8.1.4.9 Wiki

Wikis (siehe [Wiki09h]) sind mittlerweile ein beliebter Weg, um das kollaborative Sammeln und Verteilen von Wissen zu strukturieren. Wir verzichten auf eine ausführliche Beschreibung und beschränken uns auf die Definition des minimalen Funktionsumfangs, dass wir für einen Archetypen planen würden:

- Benutzer können neue Einträge anlegen oder bestehende Einträge verändern.
- Einträge werden mit einer einfachen Auszeichnungssprache formatiert, die es auch Gelegenheitsanwendern erlaubt, typische Strukturen wie Überschriften, Listen und Tabellen in einem simplen Format einzugeben.

- Einträge können über Namensreferenzen miteinander verknüpft werden.
- Einträge werden mit einer Änderungshistorie versioniert, um ältere Versionen wiederherstellen zu können.

8.1.4.10 Kombinationsanwendungen

Es muss möglich sein, Anwendungen zu entwerfen, die sich aus den zuvor angedeuteten und weiteren Funktionsbausteinen zusammensetzen. So enthält zum Beispiel ein CRM typischerweise eine Reihe formularbasierter Teilanwendungen (zur Erfassung von Kundenstammdaten, Ansprechpartnern, Aufgaben, Besuchsberichten, usw.), Kalender (für Kundentermine, Fälligkeiten, usw.), Aufgabenverwaltungen (für vertriebliche oder organisatorische Aufgaben), Dokumentenverwaltungen (für Angebotsvorlagen, Vertriebsunterlagen, Präsentationen, usw.) und vieles mehr.

8.1.5 Schritt 5: Identifikation der Anforderungen

Wir beschränken uns in diesem Abschnitt auf die Angabe der funktionalen Anforderungen. Die nicht-funktionalen Anforderungen haben keine Auswirkungen auf die prinzipiell verfügbaren Modelle, sondern schlagen sich in den Templates für den Generator wieder (um z. B. Sicherheitsfunktionen an zentraler Stelle zu integrieren, Entscheidungen bzgl. der Zugriffsmechanismen auf Datenstrukturen zentral auswählbar zu machen, usw.).

8.1.5.1 CI-Konformität

Generell müssen wir sicherstellen, dass kontextspezifische Anwendungen ausgeprägt werden können. Insbesondere ist dazu auch wichtig, dass Aussehen der Anwendung(en) an eine vorgegebene Corporate Identity anpassen zu können. Zudem muss es möglich sein, Navigationsstrukturen lösungsspezifisch auszuprägen.

8.1.5.2 Flexible Navigation

Navigationselemente sind üblicherweise hierarchisch angeordnet. Die Verknüpfung mit den durch Navigationseinträge startbaren Anwendungen oder anzeigbaren Inhalten ergibt einen Graphen, da Anwendungen und Inhalte auch über mehrere Wege erreichbar sein können. Zudem muss es möglich sein, dynamisch erweiterbare

Navigationsstrukturen definieren zu können (z. B. ergeben sich bei einem Content Management System oder einem Wiki Teile der Navigation erst zur Laufzeit, wenn Anwender beginnen, Inhalte einzustellen oder zu verändern). Zudem muss es flexible Erweiterungspunkte für die Verwaltung von Navigationsrechten geben, da insbesondere in dynamisch aufgebauten Navigationsstrukturen mit benutzergepflegten Inhalten vielfältige und komplexe Rechtekonzepte denkbar sind.

8.1.5.3 Mehrstufige Workflows

Einfache wie komplexe Workflows mit Entscheidungen, Workflowübergängen, usw. müssen in den Modellen abbildbar sein. Workflows können dabei sowohl anwendergetrieben als auch automatisiert ablaufen. Zudem müssen sie parametrisierbar sein.

8.1.5.4 Hintergrundprozesse

Workflows müssen auch ohne direkten Anstoß durch einen Anwender (z. B. zeitgesteuert) initiierbar sein.

8.1.5.5 Personalisierung

Es muss verschiedene Möglichkeiten zur Personalisierung geben, mindesten sind dies:

- Navigationsstukturen passen sich aufgrund der aktuellen Rechte an.
- Daten müssen rechtebasiert filterbar sein.
- Aktionen müssen rechtebasiert aktivierbar sein.

8.1.5.6 Datenintegration

Das System muss mindestens die Modellierung relationaler Datenmodelle unterstützen. Zudem muss es möglich sein, sowohl lokal vorhandene Daten als auch externe Datenquellen zwecks Zusammenführung, Weiterverarbeitung und Anzeige zugreifen zu können.

8.1.5.7 Geschäftslogik

Das System muss die Realisierung von Geschäftslogik in verschiedenen Detaillierungsgraden unterstützen:

- Im wesentlichen gleichförmige portalspezifische Logiken sollten durch das Modell auf hohem Abstraktionsniveau abgebildet werden. Zum Beispiele sollten Rechte zwar konfiguriert werden, die Einhaltung der Rechte sollte aber durch das Laufzeitsystem automatisiert berücksichtigt werden, ohne das weitere explizite Prüfungen modelliert oder programmiert werden.
- Einfache Logikbausteine sollten durch Scripting integrierbar sein.
- Komplexe Bausteine sollten als standardisierte Komponenten integrierbar sein.
- Entscheidungen über Workflowalternativen und prinzipielle Aktionstrigger sollten direkt in den Workflowmodellen repräsentiert werden.

8.1.5.8 Zustände und Übergänge

Workflows müssen sich aus klar definierten Zuständen mit ebenso klar definierten Zustandsübergängen zusammensetzen. Zudem muss es möglich sein, Zustandswechsel mit Aktionen zu verbinden, die im Rahmen des Übergangs ausgelöst werden.

8.1.5.9 Tagging

Verschiedene der vorgestellten Archetypen verwenden sogenannte Tagging-Mechanismen, um Entitäten der Domäne mit zusätzlichen semantischen Markierungen zu verbinden. Da dies eine typische Web-2.0-Herangehensweise ist, sollte das System auch abstrakte Mechanismen zur Verwaltung von Tags bereitstellen.

8.1.5.10 Suche

Das System muss Suchfunktionen über die vorhandenen Datenbestände bereitstellen. Die Suchfunktionen müssen konfigurierbar sein und zudem die auf den zu durchsuchenden Datenbeständen konfigurierten Rechte berücksichtigen.

8.1.5.11 Portlets

Es müssen Portlets bereitgestellt werden, um individuelle Benutzersichten auf das System konfigurieren zu können. Wir sehen aber vor, dass Portlets ausschließlich auf explizit dafür vorgesehenen Informationsseiten konfiguriert werden können, da wir auch hier sehr gute Erfahrungen damit gemacht haben, Portlets als Einstiegspunkte in bildschirmfüllende Workflows und Module zu betrachten[10].

8.1.5.12 Rollenbasierte Rechte

Rechte sollten über Rollen konfigurierbar sein. Rollen entsprechen dem aufgabenbezogenen Verständnis der meisten Organisationen und sind daher eine sehr verständliche und naheliegende Form der Rechteorganisation. Nutzerbezogene Einzelrechte werden wir zwar erlauben, raten aber aus Pflegegründen davon ab, da sich bei Änderung der fachlichen Nutzerrolle nur schwer wieder eine korrekte technische Rechtekonfiguration herstellen lässt.

8.1.6 Schritt 6: Domänenabgrenzung

Um den initialen Implementierungsaufwand zu reduzieren, werden wir aus den obigen Anforderungen Portlets ausschließen, da es mit den anderen Konzepten bereits möglich ist, prozessbasierte Benutzungssemantiken abzubilden. Portlets sind ein sicherlich wichtiger aber technisch auch komplexer Zusatzaspekt. Wir halten ihn aber für nachträglich gut ergänzbar, da abstrakt gesehen ein Workflowmodell lediglich als „portlet-fähig“ charakterisiert werden muss, damit ein Generator die Verfügbarkeit des entsprechenden Workflows auf Basis der Portlet-Spezifikation sicherstellen kann. Technisch verbergen sich dahinter eine Reihe komplexer Fragestellungen (AJAX-Fähigkeit, Konflikte zwischen JavaScript-Libraries, usw.), die den initialen Aufwand unserer Planung übersteigen und daher auf einen späteren Zeitpunkt verschoben werden.

[10] Die heute auch immer wieder anzutreffende Alternative, Portlets als Repräsentationsmedium für alle Seiteninhalte eines Portals (und damit auch alle Seiten) zu verwenden, scheint uns eher eine Konsequenz der bisherigen mäßigen Akzeptanz rein portlet-basierter Produkte zu sein. Portlets sind aufgrund der vergleichsweise komplexen internen Kommunikationsstrukturen und der immer noch relativ geringen Standardisierung aus unserer Sicht ein denkbar ungeeignetes Mittel, um mit ihnen vollständige Systeme zu modellieren. Die uns vorliegenden Projekterfahrungen bestätigen dies, da die Aufwände für die Umsetzung einer Lösung mit einem reinen Portletansatz im Schnitt mindestens 30% über den Kosten für eine klassische webbasierte Umsetzung zu liegen scheinen. Wir präferieren daher einen Mischansatz, bei dem jede Technologie für die Aspekte verwendet wird, für die sie am Besten geeignet ist.

8.1.7 Schritt 7: Identifikation der Modelle

Wir sehen unser Portal als große Menge miteinander verknüpfter Konfigurationen für einzelne Verhaltensaspekte des Systems. Eine Modellinstanz ist für uns somit eine konkrete Konfiguration eines abstrakten Konzepts.

In diesem Abschnitt beschreiben wir die prinzipielle Struktur der Modelle, mit denen wir Portale beschreiben werden. Wir legen als allgemeine Entwurfsprinzipien für Modelle fest:

- *Alles ist ein Modell.* Damit legen wir fest, dass jeder Aspekt eines Portals als Modell dargestellt wird. Wir wollen damit erreichen, dass im Gegensatz zu uns bekannten Systemen wie Intrexx ([Intr09]) oder Adenin ([Aden09]) keine Spezialfälle im System existieren, die durch manuelle Programmierung in einer Art realisiert wurden, die keinen sinnvollen Eingriff von außen erlauben.
- *Wir definieren Erweiterungskonzepte für Modelle.* Auf diese Weise wollen wir erlauben, Einzelaspekte von bereits definierten Modellen zu variieren, um so einfache Erweiterungen für Spezialausprägungen zu erlauben[11].
- *Wir modellieren in der jeweils angemessensten Form.* Damit wollen wir erreichen, dass der Vorgang der Modellierung so einfach wie möglich gehalten wird. Zum Beispiel werden wir bei der Modellierung komplexer Datenbankzugriffe direkt SQL verwenden, da dies die angemessendste domänenspezifische Sprache zu sein scheint und die Verwendung zusätzlicher Abstraktionen keine Vorteile bietet[12].
- *Wir modellieren in einer Top-Down-Vorgehensweise.* Damit ermöglichen wir es, mit Kunden zuerst auf einer abstrakten und allgemeinen Ebene in eine Anforderungsdiskussion einzusteigen und die identifizierten Anforderungen schrittweise zu verfeinern. So wird ein natürlicher Wissensaufbau unterstützt.
- *Alle Modelle sind global eindeutig identifizierbar.* Wir werden dies erreichen, indem wir eine global eindeutige ID je Modell verwenden, um es überall ein-

[11] Beispiele sind hier:

- In einem existierenden Domänenobjektmodell sollte es leicht möglich sein, neue Eigenschaften einzuführen – insbesondere, ohne Code duplizieren zu müssen.
- In einem Workflowmodell sollte es möglich sein, einzelne View-States zu variieren, ohne den Rest des Modells kopieren zu müssen.

[12] Häufig führen zusätzliche Abstraktionsschichten sogar dazu, dass neue Probleme zum Beispiel in der Beschreibung effizienter Abfragen auftauchen, da die Transformation von abstrakteren Beschreibungen in konkrete Abfragen zu Effizienzverlusten in der Abfrageformulierung führen kann.

deutig referenzieren zu können. Dies erlaubt es uns, im Rahmen der Evolution unseres Systems zu jedem Zeitpunkt neue Funktionen einzuführen, die sich eindeutig auf andere Systembestandteile beziehen können.

8.1.7.1 Top-Down-Systematik

Wie zuvor beschrieben, wollen wir Systeme in einer Top-Down-Vorgehensweise entwerfen, die es erlaubt, Anforderungen mit zunehmendem Detaillierungsgrad zu beschreiben. Wir bewegen uns also vom Allgemeinen zum Besonderen. Dies wird sich in der Modellstruktur widerspiegeln:

1. Die XML-Modellrepräsentationen werden (wie es für XML typisch ist) in einem hierarchischen Baum organisiert sein.
2. Die global eindeutigen IDs ermöglichen beliebige Referenzen zwischen Modellen, so dass ein gerichteter Graph entsteht.
3. Die Baumstruktur gewinnt in Richtung der Blätter an Detaillierung.

Eine interessante Frage ist bei dieser Form der Modellierung, welche XML-Modelle in anderen Modellen enthalten sind und welche in eigenen Dateien abgelegt werden. Wir legen fest, dass primäre Modelle immer eigenständige XML-Dateien sein werden. Manchmal wird dies auch für tertiäre Modelle gelten. Das entscheidende Kriterium dafür ist: „Würde es bei einem grafischen Editor Sinn machen, das entsprechende Modell losgelöst von anderen Modellen in einem entsprechenden Modelleditor bearbeiten zu können?“. Wird diese Frage mit „Ja“ beantwortet, kategorisieren wir das Modell als einzeldateibasiertes XML-Modell, so dass wir später bei der Implementierung von Nebenläufigkeit intuitiv sinnvolle Grenzen für das Setzen von Sperren vorfinden (siehe Abschnitt 8.1.3 auf Seite 319).

8.1.7.2 Modellhierarchie

Wir werden folgende konkrete Hierarchie für unsere Modelle verwenden[13] – Abbildung 8.4 stellt diese Hierarchie im Überblick dar:

[13] Die konkrete Ausprägung der Hierarchie basiert auf langjährigen Projekterfahrungen im Portal-, Intra- und Extranetumfeld der QuinScape GmbH. Wir können sie nur dadurch begründen, dass wir beobachtet haben, dass die vorgegebene Aufteilung insbesondere bei technisch weniger versierten fachlichen Ansprechpartnern einen graduellen und gut zugänglichen Einstieg in die Thematik erlaubt. Bei technisch versierteren Anwendern hat sie geringere Bedeutung, führt aber umgekehrt auch nicht zu einer Verkomplizierung der Darstellung.

- Das Portal-Modell beschreibt die prinzipiellen Eigenschaften eines Portals (z. B. den Namen, die verfügbaren Workflows und Hintergrundprozesse, Domänentypen, Assoziationen zwischen Typen, diverse technische Festlegungen wie die verfügbaren Stages, Integrationsschnittstellen, Systemdatentypen u. v. a. m.). Wichtig daran ist, dass die meisten Festlegungen sich auf dieser implizit durch Definition der nachfolgenden Modellschicht ergeben und Anwender wenige explizite Festlegungen treffen müssen (im wesentlichen den Namen des Portals sowie die verfügbaren Integrationsschnittstellen). Unterhalb des Portalmodells sind die folgenden Mengen von Untermodellen definiert, die wesentliche Aspekte des Systems beschreiben:
  - Navigationsmodelle,
  - Prozessmodelle[14],
  - Domänentypen,
  - Relationen zwischen Domänentypen,
  - Agentenmodelle[15],
  - Übersetzungsmodelle,
  - Stages.

In diesen Modellen werden weitere wichtige Untermodelle enthalten sein (z. B. kann ein Prozessmodell View-Modelle[16], Transitionsmodelle[17] und Aktionsmodelle[18] enthalten).

Auf Implementierungsebene definieren wir zudem, dass das Laufzeitsystem jederzeit über eine technische Repräsentation des Gesamtmodells verfügen können muss, um daraus Schlussfolgerungen für implizite Funktionsimplementierungen abzuleiten. Daraus folgt, dass es neben den hier definierten Hauptmodelltypen eine Vielzahl zusätzlicher Hilfsmodelltypen geben wird, die als Container für die Hauptmodelle fungieren und somit insbesondere spezifische Such- und Zugriffsfunktionen bereistellen können[19].

[14] Fortan werden wir den Begriff „Workflow“ durch den Begriff „Prozess“ austauschen, da wir mit einem Modell sowohl benutzergetriebene als auch automatisierte Abläufe beschreiben wollen und den Begriff „Workflow“ als zu interaktionsgetrieben betrachten. Hierbei handelt es sich um eine Festlegung, die wir treffen, da Kunden unserer Erfahrung nach häufiger die fachliche Begrifflichkeit „Prozess“ als die technische Nomenklatur „Workflow“ verwenden. Die Begriffe sind fortan also als Synonyme anzusehen.

[15] Diese beschreiben die Ereignisse und Aktionen, die Hintergrundprozesse auslösen können.

[16] View-Modelle repräsentieren einzelne Masken oder Sichten des Portals.

[17] Transitionsmodelle repräsentieren Zustandsübergänge

[18] Aktionsmodelle abstrahieren Geschäftslogik.

[19] Ein einfaches Beispiel sind Modelle für Domänentypen: Diese werden Modelle für Eigenschaften enthalten. Wir werden dabei zwischen verschiedenen Eigenschaftstypen (z. B. atomare Typen

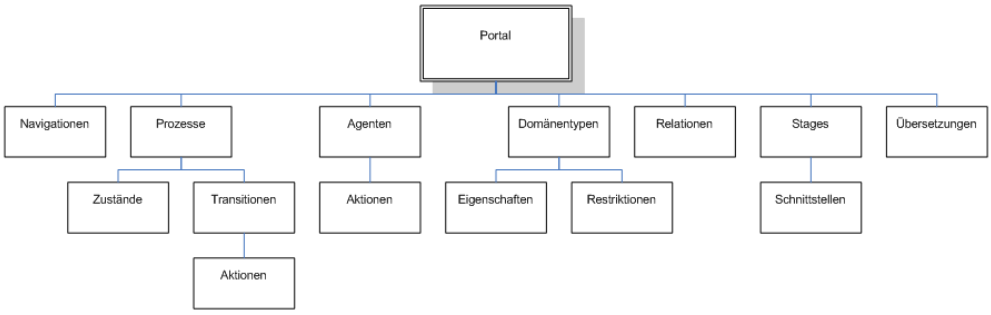


Abbildung 8.4: Logische Modellhierarchie in PIKAS (Auszug)

8.1.7.3 Scopes

Um die verschiedenen Datenstrukturen und -inhalte, die bei der Abarbeitung eines Prozesses oder auch prozessübergreifend für die Arbeit mit einem System relevant sind, zu berücksichtigen, definieren wir sogenannte Datenkontexte. Ein Datenkontext hat eine klare Lebensdauer, die beschreibt, wann der jeweilige Kontext reinitialisiert wird und wo er sichtbar ist. Wir unterscheiden:

- den *SessionContext*. Dieser existiert für die Dauer einer Sitzung, also solange ein Anwender an einem System angemeldet ist. Er ist global sichtbar.
- den *ConversationContext*. Wir definieren eine Konversation als den Teil einer Session, der an ein spezifisches Browserfenster (bzw. in Portlets an ein spezifisches Portlet) gebunden ist. Der *ConversationContext* ist für alle Prozesse, die in diesem Bereich ablaufen, sichtbar.
- den *ProcessContext*. Dieser Kontext existiert, solange sich ein Anwender in einem Prozess bewegt. Er ist für alle Akteure in einem Prozess sichtbar.
- den *ViewContext*. Dieser Kontext enthält alle Daten eines Views und existiert, bis der Nutzer den View verlassen hat.
- den *DomainContext*. Dieser Kontext ist von einer Konversation abhängig und beschreibt den Datenpfad, über den ein Anwender navigiert ist (vergleiche Breadcrumbs in Abschnitt 8.1.4.1 auf Seite 324). Er wird insbesondere dazu benutzt, um Eigenschaftsreferenzen in Views und Aktionen gezielt auflösen

versus berechnete Typen) unterscheiden. Um Eigenschaftsreferenzen leicht einer konkreten Eigenschaft zuordnen zu können, gruppieren wir alle Eigenschaften eines Domänentyps in Untergruppen von typspezifischen Eigenschaften. Konkret bedeutet dies, dass auf XML-Ebene beispielsweise ein `domain-type` eine `property-collection` enthält, die wiederum typspezifische Eigenschaftsmengen (wie ein `domain-type-property-set` oder ein `formula-property-set`) enthält.

zu können. Das Laufzeitsystem befüllt diesen Pfad automatisch anhand der Navigationspfade, die der Anwender durch das Portal nimmt.

Theoretisch wäre auch noch ein Session-übergreifender *UserContext* denkbar, der anwenderspezifische Informationen (z. B. Konfigurationen für Masken) sitzungsübergreifend speichert. Für die initiale Version des Systems werden wir diesen Kontext durch Erweiterungen des Systemobjekts *User* repräsentieren.

Jeder Datenkontext kann in der initialen Version von PIKAS die folgenden Datentypen aufnehmen:

- Eigenschaften (also atomare Werte wie Zahlen, Texte, usw.),
- Domänenobjekte (Instanziierungen von Domänentypen),
- Listen von Domänenobjekten.

Zudem stellen Datenkontexte Zugriffs- und Veränderungsoperatoren für die enthaltenen Elemente sowie Listeniteratoren bereit.

Abbildung 8.5 stellt die verschiedenen Kontexte mit ihren hierarchischen Abhängigkeiten dar.

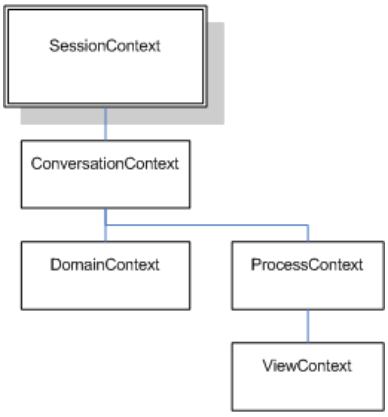


Abbildung 8.5: Datenkontexte in PIKAS

8.1.8 Schritt 8: Modelldefinition und -reduktion

Im nächsten Schritt beschreiben wir die Modelle detaillierter. Aus Platzgründen werden wir hier nur die wesentlichen Eigenschaften vorstellen – je nach Modell kann es eine Reihe weiterer Untermodelle oder Modellfacetten geben, um das Detailverhalten des Laufzeitsystems so über die Modelle zu steuern. Wir gliedern die Modelle

dabei gemäß unserer Ausführungen in Abschnitt 5.4.3.8 in primäre Modelle (Abschnitt 8.1.8.1), sekundäre Modelle (Abschnitt 8.1.8.2) und tertiäre Modelle (Abschnitt 8.1.8.3).

Immer wenn wir nachfolgend einen sogenannten Bezeichner als eine Eigenschaft eines Modellelements angeben, gehen wir davon aus, dass es sich bei dem Bezeichner nur um einen Repräsentanten handelt, der in Abhängigkeit von Sprach- und Landeseinstellungen des aktuell angemeldeten Anwenders vom Laufzeitsystem in den korrekt internationalisierten Text übersetzt werden. Für Bezeichner erlauben wir als gültige Zeichen Buchstaben, Whitespaces und Ziffern, so dass es möglich ist, sprechende Bezeichner festzulegen, die die spätere Übersetzung vereinfachen.

Zudem trägt jedes Modell eine global eindeutige ID, die wir nachfolgend voraussetzen. Alle nachfolgend beschriebenen Verweise werden technisch durch Referenzieren der ID des Ziel-Modells umgesetzt.

8.1.8.1 Primäre Modelle

Wir fassen alle Modelle als primäre Modelle auf, für die wir einen eigenen komplexen Haupteditor in unserer grafischen Lösung vorsehen wollen. Diese Modelle sind somit Aggregate, die immer nur von einem Anwender bearbeitet werden können[20].

8.1.8.1.1 NavigationModel

Idee Ein Navigationsmodell beschreibt eine Navigationsstruktur in einem Portal. Navigationsstrukturen sind üblicherweise als Bäume aufgebaut, wobei die Knoten eines solchen Baums Einsprungspunkte auf Inhalte oder Prozesse repräsentieren. In Kombination mit diesen Zielpunkten ergibt sich also ein Graph. Es muss mindestens ein Navigationsmodell je Portal geben, es kann theoretisch beliebig viele geben. Navigationsmodelle beschreiben in unserer Definition also den äußeren Rahmen der Navigation, der sich in der Navigation des Laufzeitsystems widerspiegelt. Die innere Struktur von Anwendungen wird durch die nachfolgend definierten Prozessmodelle charakterisiert (siehe Abschnitt 8.1.8.1.2 auf Seite 342).

[20] Bei einigen Modellen wie der Navigation oder Prozessen könnte man für größere Projekte diskutieren, ob es Sinn macht, diese exklusiv für andere Entwickler zu sperren. Da wir im mittelständischen Umfeld mit sehr kleinen Teams arbeiten, erscheint uns dies nicht relevant zu sein.

Beschreibung

- Ein Navigationsmodell *muss* einen Wurzelknoten enthalten.
- Ein Navigationsmodell *kann* unter jedem Knoten wieder eine Liste von Knoten enthalten.
- Ein Knoten im Navigationsmodell *muss* eine Bezeichnung haben. Diese wird für die Beschriftung des zu generierenden Menüeintrags benutzt.
- Ein Knoten im Navigationsmodell *kann* auf einen Prozessstartzustand verweisen.

Eine Menüstruktur wird in den folgenden Schritten aus einem Navigationsmodell erzeugt:

1. Die Generierung beginnt mit dem Wurzelknoten.
2. Für den aktuellen Knoten wird geprüft, ob dieser auf einen Prozessstartzustand verweist.
   a) Ist dies der Fall, wird geprüft, ob der Anwender auf den entsprechenden Prozess zugreifen kann. Falls ja, wird der Knoten in einen Menüpunkt übersetzt. Anschließend werden alle Untermenüpunkte rekursiv der gleichen Prüfung unterzogen.
   b) Falls nein, wird diese Prüfung für alle direkten Kinder wiederholt:
      i. Enthält mindestens ein Kind einen erreichbaren Prozesszustand, so werden der aktuelle Knoten zusammen mit den Kindern mit erreichbaren Prozessen in die Menüstruktur aufgenommen. Der erste über ein Kind erreichbare Prozess wird zudem zum Zielzustand für den aktuellen Elternknoten.
      ii. Enthält kein Kind einen erreichbaren Prozesszustand, werden weder der Knoten noch direkte oder indirekte Kinder in die Navigation aufgenommen.
3. Für alle aufgenommen Kinder werden wiederum deren Kinder (also die Enkel des aktuellen Knotens) geprüft.

Grafische Modellierung Navigationsmodelle betrachten gemäß unserer Definition den äußeren Navigationsrahmen von Portalen. Sie sind in ihrer Definition aus einer Vorgehensweise hervorgegangen, die wir häufig verwenden, um mit (Erst)Kunden den Umfang eines geplanten Projektes abzustecken: Dabei nähert man sich von einer allgemeinen Idee ausgehend zunehmend spezielleren Aspekten. Dazu hat sich für

uns die Verwendung von Mindmaps (siehe [Wiki09m]) bewährt, deren (reduzierte) grafische Repräsentation wir auch für die Visualisierung von Navigationsmodellen verwenden wollen.

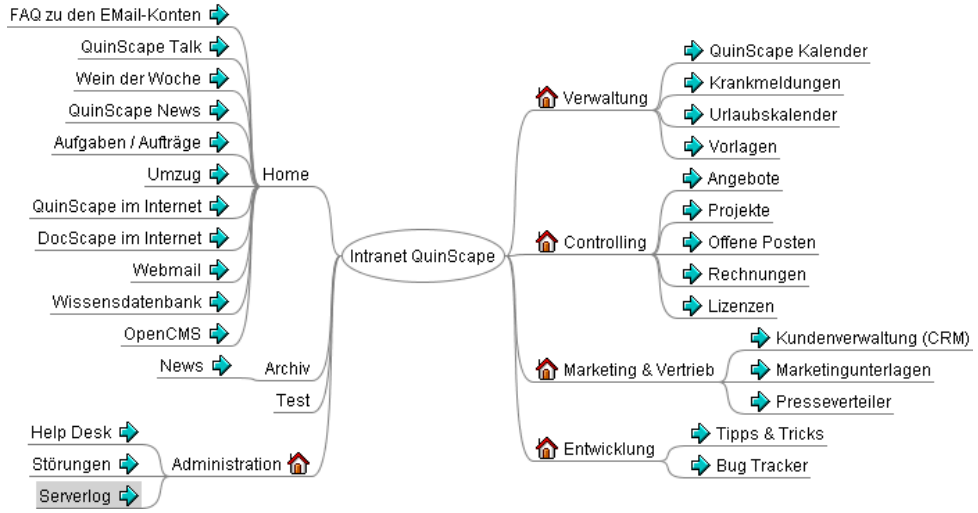


Abbildung 8.6: Grafische Repräsentation von Navigationsmodellen

Abbildung 8.6 zeigt eine exemplarische Umsetzung eines Navigationsmodells mit allen notwendigen Basiselementen: Jeder Knoten trägt eine Bezeichnung. Mit einem Haus markierte Knoten sind Übersichtsseiten, die z. B. alle enthaltenen Untermenüpunkte visualisieren oder aber eine Portletseite anbieten[21]. Mit einem Pfeil markierte Knoten verweisen auf einen Prozessstartzustand. Dieser wird bewusst nicht weiter charakterisiert, da wir aus Abstraktionsgründen in dieser Modellsicht davon ausgehen, dass die Bezeichnung des Knotens hinreichend sprechend ist. Der generierbare Prototyp erlaubt zudem eine schnelle Verifikation des korrekten Verhaltens. Nicht weiter markierte Knoten dienen der reinen Strukturierung und verändern den aktuellen Prozess nicht – sie werden im Menü auch nicht selbst anklickbar sein (es sei denn, um die enthaltenen Unterknoten einzublenden)[22].

[21] Dies ist eine portalspezifische Konfiguration. Denkbar ist, dass jede dieser Seiten auf einen global definierten Übersichtsprozess verzweigt, der dann je nach Portalausprägung eine spezielle Form von Übersichtsseite realisiert. Gegegebenenfalls bringt die Basisausprägung eines Portals bereits eine Reihe entsprechender Prozessvorlagen mit.

[22] Im Rahmen der weiteren Reduktion könnte man noch überlegen, die Pfeile zu entfernen und dafür ein Symbol wie z. B. ein leeres Blatt für in der bisherigen Darstellung nicht weiter markierte Knoten einzuführen – uns erschien es aber merkwürdig, die Anwesenheit (einer Prozessverzweigung) durch die Abwesenheit einer Grafik und die Abwesenheit weiterer Funktionalität durch die Anwesenheit einer Grafik zu kennzeichnen, so dass wir die gegenwärtige Darstellung gewählt haben.

Erweiterte Modellfacetten Für Navigationsmodelle sind vielfältige Erweiterungen denkbar, von denen wir einige beispielhaft nennen wollen:

- Ein Attribut könnte kennzeichnen, dass sich sich ein aufgerufener Prozess in einem neuen Fenster öffnet.
- Ein Attribut könnte auf einen anderen Navigationsbaum verweisen, der an seiner statt importiert wird und dessen Wurzelknoten den aktuellen Knoten ersetzt. Dies erlaubt die Modularisierung von Navigationsstrukturen, indem in einer Basisimplementierung z. B. für den Navigationsbereich eine Standardnavigation vorgegeben wird, diese aber über den beschriebenen Importmechanismus das Einhängen weiterer Navigationselemente erlaubt.
- Ein Attribut könnte festlegen, ob die zuvor beschriebene zu importierende Navigationsstruktur den aktuellen Knoten ersetzt oder lediglich neue Kinder einfügt. Dies erlaubt eine genauere Parametrisierung der Knoten.

Technische Umsetzung Das Navigationsmodell dient als direkte Datenstruktur zur Visualisierung der Navigation. Pro Request wird für den aktuellen User und den aktuellen Kontext die aktuelle Sicht auf das Navigationsmodell berechnet (z. B. in Abhängigkeit von Rechten). Die eigentliche Navigation wird über ein Faceletsbasiertes JSF-Template (das sich wiederum projektspezifisch anpassen lässt) gezeichnet – es bleibt bei der Definition der Navigation also z. B. offen, ob diese in Baumform oder als Pulldown-Menüs visualisiert werden.

8.1.8.1.2 ProcessModel

Idee Prozessmodelle repräsentieren Workflows. Durch ein Prozessmodell können dabei sowohl interaktive Workflows wie auch automatisierte (also im Hintergrund ablaufende automatisierte) Workflows charakterisiert werden. Wir legen fest, dass zur Übersicht ein Prozess immer entweder als interaktiv oder im Hintergrund ablaufend zu definieren ist[23].

[23] Soll aus einem interaktiven Prozess ein entsprechender Hintergrundprozess angestoßen werden, erlauben wir dies durch Auslösen eines Agenten (siehe Abschnitt 8.1.8.1.6). Hintergrundprozesse stoßen interaktive Prozesse dadurch an, dass sie Zustände von Datenobjekten verändern. Da Webanwendungen üblicherweise keine proaktive Push-Benachrichtigung von Anwendern unterstützen (und dies auch nur Sinn machen würde, wenn der Anwender immer einen Browser mit dem Portal geöffnet hätte), sehen wir diesen Mechanismus als ausreichend an.

Beschreibung Prozessmodelle setzen sich wie folgt zusammen:

- Ein Prozessmodell enthält eine Menge von Zuständen und eine Menge von Transitionen.
- Ein Prozessmodell *kann* einen prozessspezifischen Datenkontext definieren (siehe Abschnitt 8.1.7.3 auf Seite 337).
- Eine Transition *muss* einen Start- und einen Endzustand miteinander verbinden.
- Eine Transition *kann* eine Liste von Aktionen enthalten, die ausgelöst und sequentiell ausgeführt werden, wenn die Transition feuert (siehe Abschnitt 8.1.8.2.1 auf Seite 361 bzw. 8.1.8.2.2 auf Seite 362).
- Die folgenden Zustände sind vorgesehen[24]:
  - Startzustände (*StartStates*): Diese Zustände dienen als Einsprungpunkt aus der Navigation heraus. Wir modellieren sie explizit (statt direkt aus der Navigation auf einen anderen geeigneten Zustand zu verweisen), um zum einen direkt bei Aktivierung eines Prozesses über die erste Transition auch Aktionen ausführen zu können und um zum anderen einen sauber modellierten Zustand zu haben, an dem Eingabeparameter einer Schnittstelle beschrieben werden können.
    Startzustände dürfen nur mit Ansichts-, Entscheidungs-, Aktions- und Auswertungszuständen verbunden werden.
  - Endzustände (*EndStates*): Endzustände stoppen die Ausführung eines Prozesses. Bei einem interaktiven Prozess springt dieser auf die Startseite des Portals, bei einem Hintergrundprozess wird die Ausführung des Prozesses beendet.
    Endzustände dürfen mit keinem weiteren Zustand verbunden werden.
  - Entscheidungszustände (*DecisionStates*): Entscheidungszustände modellieren logische Verzweigungen. Jede Transition, die aus einem Entscheidungszustand führt, sollte eine mit einem Filtermodell (siehe Abschnitt 8.1.8.2.3 auf Seite 365) markierte Transition enthalten. Wird ein Entscheidungszustand erreicht, werden nacheinander die Filterbedingungen der ausgehenden Transitionen geprüft und es wird die erste Transition ausgeführt, deren Filter ein wahres Ergebnis liefert[25].

[24] Wir werden im folgenden nur noch die englischen Bezeichnungen verwenden

[25] Wir definieren die Abwesenheit eines erforderlichen Filters ebenfalls als wahren Wert. Somit kann die letzte Transition dieser Liste auch ohne Filtermodell angegeben werden, um sicherzustellen, das eine Transition in jedem Fall feuert.

Entscheidungszustände dürfen nur mit anderen Entscheidungszuständen, Startzuständen, Endzuständen, Ansichtszuständen, Aktionszuständen, Rücksprungzuständen und Auswertungszuständen verbunden werden.

- Ansichtsszustände (*ViewStates*): Ansichten definieren Interaktionen mit der Benutzeroberfläche, es handelt sich also um abstrakte Repräsentationen von Bildschirmmasken. Ereignisse werden durch Schalter auf der Maske ausgelöst und sind mit den jeweils gewünschten Transitionen verbunden. Ansichtsszustände sind so zentral für ein Portal, dass wir sie in einem separaten primären Modell beschreiben (siehe Abschnitt 8.1.8.1.3 auf Seite 347).
  Ansichtszustände dürfen
- Aktionszustände (*ActionStates*): Aktionszustände gruppieren Aktionslisten, die von verschiedenen anderen Zuständen aus aktivierbar sein sollen. Sie sind eine Spezialisierung von Entscheidungszuständen, indem der Zustandsknoten selbst eine Aktionsliste erhält (siehe wiederum Abschnitt 8.1.8.2.1 und 8.1.8.2.2).
- Rücksprungzustände (*BackStates*): Diese Zustände modellieren ein sehr spezielles Problem von grafischen Navigationsflüssen: Häufig gibt es Detailmasken, die von verschiedensten Hauptmasken aufgerufen werden können. Wird eine solche Detailmaske verlassen, erwartet der Anwender, zur aufrufenden Hauptmaske zurückzukehren. Da wir aber definiert haben, dass Transitionen immer genau einen Start- und einen Endzustand haben, ist dies nicht direkt abbildbar, sondern wir könnten ohne Rücksprungzustände immer nur zu einer definierten Maske zurückspringen (oder müssten jedes Mal komplexe Entscheidungszustände modellieren). Da dieser Anwendungsfall aber extrem häufig auftritt, führen wir Rücksprungzustände ein, die dynamisch eine Zielmaske charakterisieren. Rücksprungpunkte sind immer Ansichtszustände, die eine der folgenden Eigenschaften erfüllen:
  - Sie sind im Navigationspfad der letzte Ansichtszustand vor dem aktuellen, der einem bestimmten Datentypobjekt als Hauptobjekt zugeordnet ist.
  - Sie sind im Navigationspfad der Vater des letzte Ansichtszustands vor dem aktuellen, der einem bestimmten Datentypobjekt als Hauptobjekt zugeordnet ist.
  - Sie sind im Navigationspfad der Vater des ersten Ansichtszustands,

der einem bestimmten Datentypobjekt als Hauptobjekt zugeordnet ist.

Über diese drei Varianten lassen sich alle üblichen Navigationsmuster abbilden, weitere Bedingungen können leicht ergänzt werden.

- Auswertungszustände (*ReportStates*): Diese Zustände liefern ähnlich zu Ansichtsszuständen ein Ergebnis für den Anwender. Unterschied ist hier, dass diese Zustände keine Bildschirmmaske erzeugen sondern eine Ergebnisliste in einem geeigneten Format (z. B. PDF, Excel oder XML) aufbereiten und dieses Ergebnis an den Client zurückschicken.

Wird ein Knoten mit einem Startzustand verbunden, muss es sich dabei um einen Startzustand eines anderen Prozesses handeln. So wird ein Prozesswechsel modelliert.

Grafische Modellierung Prozessmodelle werden als Graphen modelliert. Wir gehen hier von einer sehr abstrakten Sichtweise mit den beiden folgenden Prinzipien aus:

- Jedem Zustand und jeder Transition wird eine umgangssprachliche und kurze Beschriftung zur Charakterisierung der spezifischen Semantik zugeordnet.
- Weitere Details werden in separaten Editoren angezeigt, wenn der Anwender z. B. über einen Doppelklick im Prozessmodelleditor weitere Informationen zu dem jeweiligen Modellelement anfordert.

Darüber hinaus erhält jeder Zustand eine spezifische grafische Repräsentationen, Transitionen werden als gerichtete Kanten visualisiert und erhalten ein Zusatzsymbol, wenn sie Aktionen auslösen können. Abbildung 8.7 enthält beispielhafte Repräsentationen, Abbildung 8.8 auf Seite 347 zeigt ein kleines Beispielmodell.

Erweiterte Modellfacetten Für Prozessmodelle sind vielfältige Erweiterungen denkbar, dazu gehören u. a.:

- Startzustände können mit einem speziellen *InputInterfaceContext* versehen werden (siehe Abschnitt 8.1.7.3). Dieser beschreibt einen Datenkontext, der Eingabeparameter für den Prozess speichert.
- Startzustände können als Einsprungpunkte für REST-artige Zugriffe (siehe [Wiki09n]) markiert werden und die notwendigen Parameter über ein URL-Muster definieren (siehe [Wolf09] für ein Beispiel, wie URL-Muster gestaltet

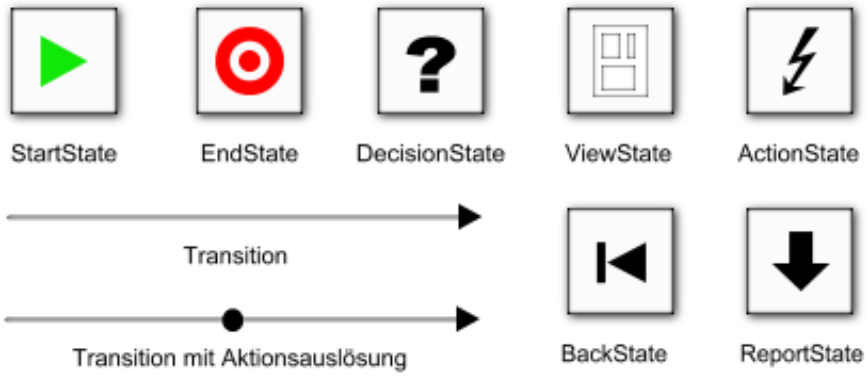


Abbildung 8.7: Grafische Repräsentation von Prozessmodellen

werden könnten – die dort dargestellten Variablennamen würden wir in unserem Fall durch die IDs von Eigenschaftsmodellen aus dem InputInterfaceContext ersetzen)[26].

- Prozesswechsel (also Transitionen mit einem Prozessstartzustand als Zielzustand) könnten mit einem Attribut als Unterprozessaufrufe markiert werden. Diese würden client-seitig durch Anzeige in einem entsprechenden Popup-Fenster abgebildet, serverseitig müssten die erforderlichen Datenkontexte (für Konversationen, Prozesse, Domänenkontexte und Views) gesichert und durch neue Instanzen ersetzt werden, bis ein Rücksprung in den aufrufenden Prozess erfolgt (es müsste also ein Unterprogrammaufruf auf Prozessebene simuliert werden).

Technische Umsetzung Prozessmodelle werden durch den Generator in Spring Webflows konvertiert (siehe [Spri09b]), die die operative Ausführung der Prozesse ermöglichen. Dabei werden die meisten Zustände direkt in die äquivalenten Zustände von Spring Webflow konvertiert. Unterschiede ergeben sich z. B. in der Behandlung von Startzuständen, von denen Spring Webflow nur einen einzigen pro Flow erlaubt, während wir eine beliebige Zahl von Einsprungpunkten erlauben. Dies setzen wir um, in denen wir den einzelne Webflow-Einsprungspunkt so parametrisieren, dass dieser über einen Entscheidungszustand in den von uns gewünschten Prozesszustand verzweigt. Ähnlich gehen wir bei der Umsetzung unserer Entscheidungszustände vor – wir erlauben hier die Definition von Aktionen an den Transitionen, Spring

[26] Analog könnte die Definition eines *OutputInterfaceContext*s erfolgen, um Rückgabewerte zu beschreiben.

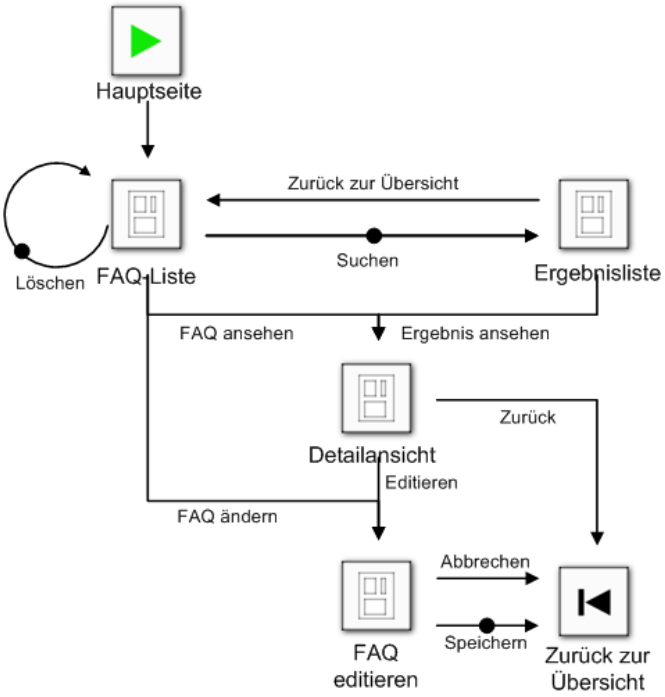


Abbildung 8.8: Prozessmodel am Beispiel des FAQ-Archetypen

Webflow gestattet dies nicht. Diese Einschränkung umgehen wir, indem wir in Webflow weitere Aktionszustände einfügen, auf die zunächst verzweigt wird, um dort die Ausführung der Aktionen zu ermöglichen. Im Wesentlichen lassen sich unsere Prozessmodelle aber 1:1 in Spring-Webflow-Modelle ersetzen, wobei wir in den generierten Flows eine Vielzahl von Service-Aufrufen ergänzen, um unsere semantisch reichhaltigeren Modelle abzubilden. Zudem generieren wir neue Transitionen, um z. B. die vielen impliziten Möglichkeiten zur Zustandsveränderung automatisch zu berücksichtigen, die sich aus Inhaltselementen ergeben: Bei reiner Verwendung von Spring Webflow wäre es z. B. erforderlich, explizit Zustandsübergänge für das Paging von Tabellen oder das Sortieren von Tabellenspalten anzugeben – dies können wir in PIKAS automatisiert abwickeln, da der Generator die entsprechenden Inhaltsmodelle nach erforderlichen impliziten Transitionen und daran hängenden Aktionen befragen und diese selbständig einfügen kann.

8.1.8.1.3 ViewModel

Idee Da Ansichten in einem Portal eines der zentralsten Elemente sind, ordnen wir diesem Konzept einen eigenen primären Modelltyp zu. Dabei spielt auch die Überle-

gung eine Rolle, dass Bildschirm- und Maskendesigner sich intensiv mit Einzelmasken beschäftigen werden, so dass es wiederum aus Sicht von Sperrmechanismen gut ist, diese als eigenständige Aggregate zu betrachten. Wir ordnen Ansichten dabei maximal einem Hauptdatentyp zu, an dem sich die in der Maske dargestellten Inhalte orientieren. Wir zerlegen eine Ansicht außerdem in Einzelteile, um so auf einfache Weise eine Strukturierung der Inhalte sowohl in gestalterischer wie auch in optischer Sicht zu erreichen. Die Einzelelemente können wiederum einem Hauptdatenobjekt zugeordnet werden, indem die relationale Abhängigkeit zum Hauptobjekt der Seite angegeben wird. Damit können in einem Maskenteil zwar nur Daten aus dem Hauptobjekt der Seite oder des Teils sowie aus direkt abhängigen Objekten oder Kindern angezeigt werden, die interne Datenverwaltung vereinfacht sich aber deutlich.

Beschreibung

- Ein View *kann* maximal einem Hauptdomänentypen zugeordnet sein (siehe Abschnitt 8.1.8.1.4 auf Seite 351).
- Ein View *muss* in einen oder mehrere *Subviews* unterteilt werden. Diese partitionieren den Bildschirm horizontal und vertikal. Wir legen Folgendes fest:
  - Die Subview-Liste auf oberster Ebene muss entweder horizontal oder vertikal ausgerichtet sein.
  - Sie enthält Subviews, die einen Teil des verfügbaren Bildschirmplatzes reservieren.
  - Jeder Subview kann wiederum eine Liste von Subviews beinhalten, die in der jeweils anderen Ausrichtung verlaufen (also horizontal, falls der aktuelle Subview ein vertikaler ist und umgekehrt).
  - Diese Verschachtelung kann beliebig tief fortgeführt werden.
  - Jedem Blatt-Subview wird genau ein sogenanntes *PartModel* zugeordnet, dass die eigentlichen Inhalte des Subviews beschreibt.
- Ein View *muss* ein oder mehrere PartModel-Instanzen enthalten, die Inhalte beschreiben.
- Ein View *kann* auf die Angabe eines Subviews verzichten, wenn es genau eine PartModel-Instanz gibt – diese nimmt dann den gesamten zur Verfügung stehenden Raum ein.
- Eine PartModel-Instanz enthält wiederum ein sogenanntes ContentModel als Container für die eigentlichen Inhalte.

- Ein PartModel ist genau einem Domänentyp zugeordnet. Es gilt:
  - Entweder wird der Domänentyp durch Angabe einer Relationskette beschrieben, die vom Hauptdomänentyp des Views ausgehend bis zum gewünschten Domänentyp des PartModels führt *oder*
  - das PartModel übernimmt den Hauptdomänentyp des Views.
- Ein ContentModel beschreibt sowohl das Layout der Seite[27] als auch die konkreten Inhalte[28].

Grafische Modellierung Die grafische Modellierung eines ViewModels liegt sehr nahe: Im Prinzip verwenden wir eine klassische WYSIWYG-Ansicht, wie sie aus vielen Entwicklungsumgebungen zum Design von Masken verwendet werden. Wir nehmen die folgenden Vereinfachungen vor:

- Zunächst wird das ViewModen in SubViews zerlegt (partitioniert).
- Jeder SubView wird dann mit einem Part verbunden, der in klassischer WYSIWYG-Manier entworfen wird.

Wir verzichten auf die Angabe eines WYSIWYG-Editor-Beispiels, da Beispiele hierfür hinlänglich bekannt sind, verweisen aber auf Abschnitt 8.1.8.2.4 auf Seite 366 für weitere Ausführungen zur Typisierung und Positionierung von in ViewModel-Instanzen enthaltenen Elementen.

Erweiterte Modellfacetten Erweiterungen der ViewModelle werden insbesondere hinsichtlich neuer Anzeige- und Eingabeelemente sowie in der expliziten Darstellung von Abhängigkeiten erfolgen:

- Spezielle Domänen erfordern spezielle Eingabeelemente. So sind zum Beispiel die DV-bezogenen Tätigkeiten in der Lebensmittelüberwachung davon geprägt, dass häufig große Datenmengen mit starken Abhängigkeiten zu standardisierten Schüsselkatalogen eingegeben werden müssen. Um eine schnelle und fehlerfreie Eingabe zu unterstützen, werden in diesem Umfeld üblicherweise sogenannte „Katalogauswahlhilfen“ bereitgestellt. Dabei handelt es sich um modale Dialoge, die an einzelne Eingabefelder gebunden sind und die die schnelle Suche

[27] Hier sind verschiedene Formen von Layoutmanagern denkbar, z. B. koordinatenbasierte, rasterbasierte, gruppenbasierte und andere. Wir geben einige als Beispiel vor und erweitern das Layoutsystem bei Bedarf.

[28] Dies sind Eingabe- und Ansichtsfelder, Tabellen, usw. Alle Elemente, die Daten anzeigen, referenzieren entweder DomainTypeModel-Instanzen oder aber in diesen enthaltene PropertyModel-Instanzen.

auf einem sehr großen Katalogdatenbestand mit anschließender Übernahme des gefundenen Ergebnisses in das Eingabefeld unterstützen. Im Webumfeld erweist sich das als besonders schwierig, da es sich häufig um Kataloge mit zehntausenden von Einträgen handelt, die sowohl zum manuellen Durchsuchen als auch zur schlüsselwortbezogenen Suche angeboten werden müssen. Aufgrund heutiger Bandbreiten und den Grenzen der verfügbaren Webtechnologien führt dies zu komplexen Fragestellungen in der Umsetzung[29].

- Um modernen Web-2.0-Möglichkeiten Rechnung zu tragen, ist es sinnvoll, eine Reihe von Abhängigkeiten in zunehmendem Maße explizit zu modellieren (z. B. zwischen Master- und Detaildaten zwecks wechselseitiger Aktualisierung über AJAX).
- Es sind vielfältige Erweiterungen abzusehen, die die ergonomische Steuerung der Oberfläche beeinflussen. So sind z. B. Pflichtfelder je nach Domäne anders zu behandeln: In der Lebensmittelüberwachung werden Pflichtfelder zwar üblicherweise visuell markiert, aber ihre Eingabe wird nicht erzwungen, um eine möglichst flexible Abarbeitung von Erfassungsprozessen zu ermöglichen. In anderen Domänen können Erfassungsprozesse erst dann weitergeführt werden, wenn alle Pflichtfelder eines Prozessschrittes angegeben sind.
- Die stetig zunehmende Komplexität von webbasierten Bedienoberflächen wird zu einer kontinuierlichen Weiterentwicklung einzelner Eingabeelemente führen (z. B. durch Einführung von Autovervollständigung in Eingabefeldern).

Technische Umsetzung View-Modelle werden in JSF-Seiten konvertiert. Diese setzen sich wiederum aus Facelets-basierten Komponenten zusammen, um so die Visualisierung von Seiten zu ermöglichen. Bei der Generierung werden wiederum diverse Optimierungen vorgenommen:

- Das System berücksichtigt, ob im Browser JavaScript aktiv ist und generiert entsprechenden Code.
- Das System berücksichtigt die Yahoo Performance Guidelines (siehe [Yaho09]), indem es z. B. die verwendeten CSS-Stylesheets und JavaScript-Dateien je Seite ermittelt, diese minimiert, komprimiert, zu Bündeln zusammenfasst und CSS-Sprites generiert[30].

[29] Verstärkt wird dies durch die Forderung nach einer SAGA-4-Kompatibilität, da diese auch ein prinzipielles Funktionieren aller wesentlichen Anwendungsbestandteile fordert, selbst wenn kein JavaScript zur Verfügung steht.

[30] Das führt beispielsweise dazu, dass pro Seite maximal vier Zugriffe auf den Server erfolgen (einer

8.1.8.1.4 DomainTypeModel

Idee Die überwiegende Mehrzahl uns bekannter webbasierter Lösungen basiert auf relationalen Datenmodellen, so dass wir in der initialen Version von PIKAS auch nur solche Datenmodelle abbilden werden. Wir verwenden eine etwas speziellere Nomenklatur, um den domänenspezifischen Charakter der ausgeprägten Einzellösungen zu betonen. Dazu bezeichnen wir die Entitäten unseres Datenmodells als Domänentypen (*DomainTye*). Ein Domänentyp wird durch eine Menge von typisierten Eigenschafte repräsentiert – diese werden üblicherweise auf Tabellenspalten in einer Datenbank abgebildet. Die Relationen zwischen Domänentypen werden separat modelliert (siehe dazu Abschnitt 8.1.8.1.5 auf Seite 355).

Beschreibung Ein Domänentyp wird wie folgt beschrieben:

- Er hat einen eindeutigen logischen Namen.
- Er besitzt eine Menge von Eigenschaften. Um sich stark unterscheidende Eigenschaftsarten zu differenzieren, legen wir sogar fest, dass sich die Menge der Eigenschaften aus disjunkten und typisierten Untereigenschaftsmengen zusammensetzt, die wie folgt definiert sind:
  - Es gibt sogenannte *DomainTypeProperties*. Hierbei handelt es sich um atomare Werte, die direkt auf eine physikalische Datenbankspalte abgebildet werden und deren zentrale Aufgabe somit die Speicherung atomarer Werte ist.
  - Es gibt sogenannte *FormulaProperties* (Formel-Eigenschaften). Dies sind Eigenschaften, die sich durch Angabe einer Berechnungsvorschrift aus anderen Eigenschaften eines konkreten Objekts oder seines Kontextes ergeben. Wir schränken hier wie folgt ein: Formeleigenschaften werden nur neu berechnet, wenn sie gespeichert oder in Masken bzw. Geschäftslogik referenziert werden[31]. Formeleigenschaften werden über Programmkon-

für die Seite selbst, einer für die CSS-Daten, einer für die JavaScript-Includes und einer, um alle Grafiken als CSS-Sprites in einer generierten Grafik zu laden).

[31] Wir bilden also keine Datenbankprozeduren und -trigger in ihrer Vollständigkeit ab. Insbesondere werden berechnete Werte dadurch nur teilweise abgebildet: So könnte z. B. das „Alter“ einer Person ausgehend vom Geburtsdatum zwar berechnet werden, es würde aber nicht kontinuierlich in der Datenbank aktualisiert werden. Dies ist bei entsprechenden Abfragen zu berücksichtigen. Umgekehrt lassen sich damit statische Berechnungsabhängigkeiten (z. B. setzt sich der Name einer Person aus dem Vornamen der Person, einem Leerzeichen und ihrem Nachnamen zusammen) einfach abbilden, ohne dass an den relevanten Maskenpositionen Berechnungsvorschriften angegeben werden müssen. So erreichen wir einen Kompromiss aus Ausdrucksstärke und Imple-

strukte einer geeigneten Skriptsprache formuliert[32].

- Es gibt sogenannte *EnumProperties* (Aufzählungseigenschaften). Diese referenzieren Aufzählungen, die durch einen anderen Domänentypen definiert sind und sind durch besondere Anforderungen aus der Lebensmittelüberwachung motiviert, die aber auch auf viele andere Anwendungsfälle übertragbar sind[33]. Sie haben einige Sondereigenschaften:
  - Sie sind über eine implizite Relation mit einem anderen Domänentypen verbunden, der die Aufzählung enthält.
  - Ein oder mehrere Eigenschaften des Bezugsdomänentyps können auch als Eigenschaften des aktuellen Domänentyps angezeigt werden.
  - Die Eingabe eines beliebigen eindeutigen Wertes aus einer Tabellenzeile des Bezugsdomänentyps führt zur korrekten Zuordnung zwischen den beiden Objekten.
  - Die Auflösung dieser Aufzählungseigenschaften wird auf Datenbankseite besonders effizient implementiert, da ein Domänentyp nicht selten fünf, zehn oder mehr Referenzen auf solche Aufzählungstypen enthält[34].

- Ein *PropertyModel* besitzt neben einer eindeutigen ID einen eindeutige Namen, einen logischen Datentypen[35] und ggf. weitere Zusatzinformationen in Abhängigkeit von der konkreten Eigenschaft[36].

Grafische Modellierung Zur grafischen Modellierung von Datentypen gibt es bereits vielfältige Darstellungsformen, so dass wir keine neue Form entwickeln wollen. Wir haben uns für PIKAS für ER-Diagramme unter Verwendung einer modifizierten Martin-Notation entschieden (siehe [Wiki09l, Wiki08n]), da diese unserer Ansicht nach eine bessere und eingängigere Granularität als Klassendiagramme hat und die Notation natürlicher erscheint. Folgende Modifikationen haben wir vorgenommen:

mentierungskomplexität, der viele aber nicht alle Anforderungen abdecken kann. Reichen die Eigenschaften dieser Formeln nicht aus, so ist eine komplexere Implementierung zu planen.

[32] Im konkreten Fall haben wir Groovy (siehe [Code09b]) gewählt und einen Mechanismus vorgesehen, um weitere Skriptsprachen leicht ergänzen zu können.

[33] Konkret geht es darum, dass Datensätze häufig Statusfelder oder andere Referenzen auf dynamische Listen wie z. B. die zuvor erläuterten Kataloge aus der Lebensmittelüberwachung haben. Diese Referenzen müssen in fast allen Anzeigen aufgelöst werden und zudem werden sie häufig nach bestimmten Mustern eingepflegt, z. B. über Auswahllisten oder intelligente Eingabefelder mit Auto-Vervollständigungen.

[34] Konkret bedeutet dies dass z. B. bei einer Implementierung auf Basis von Hibernate (siehe [RedH09]) eingebettete Typen anstelle von Join-Tabellen verwendet werden.

[35] Dies können z. B. “Text“, „Zahl“, „formatierter Text“, „Währung“, „Menge“ sein.

[36] Beispielsweise könnte dies die Anzahl Vor- und Nachkommastellen bei einer Zahl, die Länge eines Textes oder eine konkrete Formel sein.

- Die Bezeichnungen von Relationen stehen an der Ausgangsentität.
- Die Kardinalität wird in Klammern hinter der Relationsbezeichnung angebenen.
- Bei gleicher minimaler und maximaler Kardinalität geben wir diese nur einmal an.
- Es werden per Default keine symbolischen Kardinalitäten abgebildet sondern nur numerische. Die grafische Repräsentation ist eine optionale Gestaltungsform, falls sich aufgrund der Diagrammkomplexität die textuelle Darstellung als zu komplex erweist.

In Kombination ergibt sich daraus eine sehr natürliche Lesart der Diagramme, da man bei sinnvoller Wahl von Relationsbezeichnungen immer in der Folge (Ausgangstyp – Relationsbeschriftung – Zieltyp) lesen kann und dabei nahezu natürlichsprachige Sätze erhält. Abbildung 8.9 stellt zum Beispiel den Sachverhalt „eine Person ist geboren in einem Ort“ sowie „ein Ort ist der Geburtsort von 0 bis N Personen“ dar.

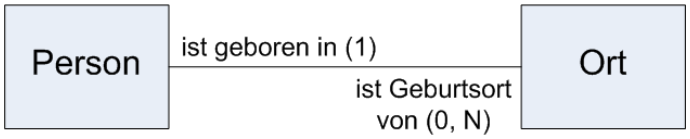


Abbildung 8.9: Beispiel für Domänentypen und Relationen

Sollen einzelne Domänenobjekte genauer betrachtet werden, öffnen unser Modellierungswerkzeug einen modalen tabellenbasierten Dialog, der die einzelnen Eigenschaften auflistet und die übliche Veränderungsoperationen erlaubt[37].

Erweiterte Modellfacetten Sinnvolle Facetten sind u. a.:

- die Möglichkeit, Datenquellen zu definieren. Damit wird es möglich, auch externe Datenquellen einzubinden. Die Zuordnung zu Datenquellen erfolgt über einen symbolischen Namen[38].

[37] Wir weisen darauf hin, dass hier in der Implementierung des Modellierungswerkzeugs Synergie-Effekte zu der Implementierung von Prozessmodellen erzielt werden können, da in beiden Fällen Knoten und Kanten mit Beschriftungen auf einem Zeichenblatt platziert werden müssen. Gelingt es, hier eine geeignet parametrisierbare Form eines Editors zu entwickeln, muss für beide Modellarten im Wesentlichen nur eine Implementierung erfolgen.

[38] Wir verwenden an dieser Stelle symbolische Namen, um Datenquellendefinitionen zu referenzieren. IDs kommen im Gegensatz zu allen anderen Stellen im System nicht in Frage, da diese

- zusätzliche Strukturinformationen bei externen Datenquellen, die das Auffinden von spezifischen Eigenschaftswerten unterstützen[39].
- die Struktur des Primärschlüssels[40].
- Initialisierungsdaten, die nur dann eingefügt werden, wenn sie noch nicht im System vorhanden sind[41].

Technische Umsetzung Zu jedem Domänentypen generiert PIKAS – den Entwurfsmustern von Eric Evans aus Abschnitt 3.2.7 folgend – eine *Factory* zum Erzeugen neuer Objekte sowie ein *Repository* zur Verwaltung einmal erzeugter Objekte (also zum Speichern, Ändern, Löschen und Auffinden). Diese beiden Instanzen abstrahieren vom konkreten Zugriff auf die zugrunde liegende Persistenzschicht (im Sinne des ebenfalls in Kapitel 3 erläuterten DAO-Musters).

Darunter verwaltet das System je Domänentyp einen sogenannten Datenquellentyp, der die technischen Details zur Persistenzanbindung kapselt. Mögliche Datenquellentypen sind z. B. “Hibernate-basierte Datenbanken“, „JDBC-basierte Datenbanken“, „Excel-Tabellen“, „RSS-Feeds“ und viele andere mehr.

Zu den oberen Schichten hin dienen Domänentypen als einheitlicher Abstraktionsmechanismus. Bei der Generierung des Systems analysiert der Generator die Domänentypen, die Datenquellentypen und die Datenquellenkonfigurationen und erzeugt dann dynamisch Spring-Konfigurationen, die die erforderlichen Factories und Repositories enthalten und in einen zusammenhängenden transaktionalen Kontext ein-

global eindeutig sein müssen, wir aber bei Datenquellen ein Staging-Konzept zulassen wollen, d. h. je nach gewählter Stage (z. B. Entwicklung, QA oder Produktion) andere Datenquellen anbinden wollen. Staging ist ein komplexes Konzept, dass in einem eigenen Modell behandelt wird. Jedes Staging-Modell besitzt eine eindeutige ID und ist wiederum einem textuellen Bezeichner eine Stage zugeordnet, so dass an einem Domänentypen jeweils nur ein fachlicher Name für eine Datenquelle referenziert werden kann, ohne dass dem Datenmodellierer das konkrete Staging-Konzept bekannt sein muss. Ein Datenmodellierer könnte also zum Beispiel beschreiben, dass sich die Daten für einen Domänentyp auf eine Datenquelle „SAP-Stammdatensystem“ beziehen.

[39] Bei einer Excel-basierten Datenquelle könnten dies Spaltenzuordnungen zu Eigenschaften sein, bei einer XML-Datenquelle könnten dies XPath-Ausdrücke sein, usw.

[40] In einer ersten Implementierung unseres Prototypen hatten wir zwecks Reduktion der Implementierungskomplexität festgelegt, dass wir ausschließlich technische Primärschlüssel vom Typ „LONG“ verwenden wollen. Wir haben aber schnell festgestellt, dass gerade bei der Fremddatenintegration und Anbindung von Drittsystemen viel zu häufig Daten mit zusammengesetzten (fachlichen) Primärschlüsseln auftreten, so dass wir im aktuellen Prototypen beliebige zusammengesetzte Primärschlüssel erlauben.

[41] Dies erlaubt das Aufsetzen neuer Systeme ohne externe Aktionen, indem z. B. alle Stammdaten in den Modellen vorkonfiguriert werden. Ein Beispiel ist die Definition eines Administrationsusers, der Zugriff auf alle Seiten und Prozesse hat und bei der Systeminitialisierung automatisch angelegt werden soll.

binden. Über ein sogenanntes *DomainTypeRepository* werden zur Laufzeit abstrakte Domänentypen (also *DomainTypeModel*-Instanzen) mit den zuvor beschriebenen Zugriffsklassen in Verbindung gebracht.

8.1.8.1.5 RelationModel

Idee Wir wollen ein vollständiges relationales Datenbankmodell unterstützen. Aus Komplexitätsgründen werden wir in der initalen Version zunächst nur binäre Relationen implementieren und n-äre Relationen erst in späteren Versionen ergänzen.

Beschreibung Relationen haben einen Ausgangstypen und einen Zieltypen. Für jede Seite der Relation ist eine der Kardinalitäten (0,1), (1, 1), (0, N) oder (1, N) anzugeben. Relationen sind gerichtet und werden außer durch ihre ID auch durch einen fachlichen Namen identifiziert.

Grafische Modellierung Relationen sind ein Bestandteil der grafischen Modellierung von Domänentypen (siehe Abschnitt 8.1.8.1.4 auf Seite 351), so dass wir keine zusätzliche Modellierungsform angeben müssen.

Erweiterte Modellfacetten Einzig bisher sinnvoll erscheinende Erweiterung war ein *ConnectionHandler*. Dabei handelt es sich um ein technisches Detail, um spezifische Verknüpfungsformen zwischen internen und externen Datenquellen (oder auch reinen externen Datenquellen) zu ermöglichen[42].

Technische Umsetzung Relationen werden durch PIKAS auf zwei Weisen umgesetzt:

1. Sind die dazugehörigen Domänentypen beide in der zugrunde liegenden relationalen Portaldatenbank (die über Hibernate angesprochen wird) abgebildet, werden die Standardmechanismen von Hibernate zur Relationsbehandlung verwendet.

[42] Üblicherweise werden dazu künstliche Join-Tabellen angelegt und der ConnectionHandler ist dann in der Lage, die Primärschlüssel der beteiligten Entitäten miteinander zu verbinden, um so Abhängigkeiten darzustellen. Er kann also sowohl abhängige Objekte gemäß einer Relation lokalisieren als auch neue Abhängigkeiten aufnehmen bzw. Abhgängigkeiten löschen.

2. Liegt mindestens einer der Domänentypen nicht in der lokalen Portaldatenbank sondern ist über eine andere Datenquelle angebunden, so erzeugt PIKAS eine künstliche Join-Tabelle, die Objektrelationen speichert. Über die resultierenden Performance-Implikationen muss sich der Entwickler jederzeit klar sein, da komplexe Joins zwischen verteilten Objekten kostspielig sein können – dies ist aber ein typisches Problem in verteilten Szenarien, dass wir daher nicht weiter problematisieren.

Die Entscheidung darüber, wie Relationen behandelt werden, trifft die Repository-Abstraktionsausschicht, so dass die darüber liegenden Implementierungsebenen nicht mit diesen Details belastet werden. Lediglich im Entwurf muss man sich mit den Performance-Implikationen verteilter Datenstrukturen auseinandersetzen.

Für spezielle Erweiterungen bietet PIKAS Strategie-Schnittstellen, die sowohl die Behandlung von Relationen als auch den Zugriff auf Objekte individualisierbar machen.

8.1.8.1.6 AgentModel

Idee Agenten sind im Gegensatz zu den üblicherweise anwendergetriebenen Prozessen aus Abschnitt 8.1.8.1.2 auf Seite 342 automatisiert oder ereignisbasiert ablaufende Hintergrundprozesse, die Aktionen auf Daten auslösen.

Beschreibung Ein Agent wird wie folgt definiert:

- Agenten haben einen fachlichen Namen und eine kurze textuelle Beschreibung.
- Ein Agent reagiert auf einen Auslöser. Mögliche Auslöser in der initialen Version von PIKAS sind:
  - Ein Systemereignis findet statt. Beispiele sind „ein Anwender hat sich an- oder abgemeldet“, „das System wurde hochgefahren“, „das System wird heruntergefahren“, usw.
  - Ein Datenereignis findet statt. Beispiele sind „Ein Datensatz wird gleich gelöscht“, „Ein Datensatz wurde aktualisiert“, „Ein Datensatz wurde gelesen“, usw.
  - Ein zeitbasieres Ereignis findet statt. Beispiele sind das Ablaufen eines Timers, das Erreichen eines bestimmten Datums, usw.

  - Ein manuell ausgelöstes Ereignis findet statt. Das typische Beispiel ist hier das Auslösen eines normalerweise zeitbasiert arbeitenden Agenten durch eine explizite Aktion wie z. B. “Das Backup soll nicht erst um Mitternacht sondern durch Klick auf einen entsprechenden Aktionsschalter im Administrationsbereich sofort ausgelöst werden“.

- Ein Agent bewertet den Kontext des Auslösers. Konkret bedeutet dies, dass er auf alle allgemein sichtbaren Daten zum Zeitpunkt des Auslösens Zugriff haben muss und diese als Entscheidungsgrundlage verwenden kann, um zu entscheiden, ob er aktiv wird. Wir lösen dies durch Verbindung der auszuführenden Aktionsliste (siehe Abschnitt 8.1.8.2.1 auf Seite 361) mit einem Filtermodell (siehe Abschnitt 8.1.8.2.3 auf Seite 8.1.8.2.3).
- Ein Agent kann aktiv werden. Konkret wird dies realisiert durch Angabe einer Aktionsliste, die abgearbeitet wird, wenn der Agent dies aufgrund des Kontextes für sinnvoll hält (mit anderen Worten: wenn die bei der Aktionsliste angegebene Filterbedingung den Wert „WAHR“ zurückliefert).

Grafische Modellierung Agenten werden nicht grafisch modelliert. Global definierte Agenten werden einfach in einer tabellenartigen Übersicht mit ihrem Namen und ihrer textuellen Beschreibung angezeigt. Prozessbezogene Agenten werden als Symbol in der Prozessdarstellung mit ihrem Namen angezeigt. Beide Sichten verzweigen auf einen Dialog, der die Konfiguration des Namens, der Bezeichnung und der gefilterten Aktionsliste erlaubt.

Erweiterte Modellfacetten Agentenmodelle werden überwiegend durch neue Auslöser erweitert werden. Dies können triviale Änderungen wie das Ergänzen neuer Systemereignisse sein oder aber komplexe Änderungen wie nachrichtenbasierte Auslöser sein.

Technische Umsetzung Agenten werden je nach Art des Auslösers unterschiedlich behandelt:

- Zeitgesteuerte Agenten werden technisch über einen mit Spring konfigurierten Quartz-Scheduler gesteuert (siehe [Open09]). Dieser ruft die Agenten zum passenden Zeitpunkt auf und führt die konfigurierten Aktionen aus.
- Alle anderen Agenten werden über eine Message-Queue gesteuert. Das System veröffentlicht alle potenziell relevanten Ereignisse in der Queue. Die konfigurierten Agenten werden über neue Ereignisse benachrichtigt und können dann

entscheiden, ob und wie sie reagieren wollen (also ob ihre Aktionslisten ausgeführt werden).

8.1.8.1.7 TranslationsModel

Idee Jedes Portal soll internationalisierbar sein. Dazu muss zum einem festgelegt werden, welche Sprachen überhaupt von einer Portalinstanz unterstützt werden. Zum anderen müssen Übersetzungstexte gepflegt werden können. Letzteres werden wir auf zwei Weisen erlauben:

1. In den nachfolgend beschriebenen Übersetzungsmodellen können Default-Übersetzungen für einen spezifischen Kontext definiert werden.
2. In der Datenbank der konkreten Portalinstanz können weitere Übersetzungen programmatisch gepflegt werden, die ggf. Übersetzungen in den Modellen überlagern[43].

Beschreibung Übersetzungen werden mit einem sogenannten „Label“ verbunden. Dieses ist eine textuelle Repräsentation des zu übersetzenden Textes[44]. Diese können über verschiedene Detaillierungsmerkmale mit unterschiedlichen Bedeutungen gefüllt werden. Als Detaillierungsmerkmale verwenden wir den ISO-Sprachcode, den ISO-Landescode, die Prozess-ID, die Seiten-ID und das Label selbst. Einzelne dieser Merkmale können weggelassen werden, der Prototyp verwendet intern die folgende Suchreihenfolge, wobei der erste vollständig qualifizierte Treffer auch angezeigt wird[45]:

1. Sprachcode, Landescode, Prozess-ID, Seiten-ID, Label

[43] Dies erlaubt Anwendern oder Administratoren, vorgegebene Übersetzungen an einen spezifischeren Kontext anzupassen, ohne dass die zugrunde liegenden Modelle geändert werden müssen. Umgekehrt erlauben vorgegebene Übersetzungen in den Übersetzungsmodellen ein sehr schnelles Aufsetzen neuer Portalinstanzen, da der zeitintensive Übersetzungsprozess für alle Begriffe entfällt bzw. auf ein Minimum reduziert wird.

[44] Intern haben wir festgelegt, dass Label ausschließlich aus Buchstaben und Ziffern bestehen und Worte in Camel-Case-Schreibweise aneinander gereiht werden. Gültige Label wären als z. B. “Cancel“, „ListOfConcepts“ und „MyProfile“. Üblicherweise verwenden wir die Kurzschreibweise der englischen Übersetzung als Label für einen zu übersetzenden Text.

[45] Durch die vorgegebene Reihenfolge wollen wir die Definition von Ausnahmen vereinfachen, gleichzeitig die globale Übersetzung möglichst vieler einfacher Begriffe aber so effizient wie möglich gestalten. Operativ verwenden wir fast ausschließlich die Kombination (Sprachcode, Label), da dies für die meisten Fälle ausreichend ist. Unserer Erfahrung nach macht es aber Sinn, sich über die Struktur und Benennung von Label-Texten Gedanken zu machen, um Widersprüchlichkeiten und Doppeldeutigkeiten sowie die damit verbundenen Pflegeaufwände zu reduzieren.

2. Sprachcode, Prozess-ID, Seiten-ID, Label
3. Sprachcode, Landescode, Prozess-ID, Label
4. Sprachcode, Prozess-ID, Label
5. Sprachcode, Landescode, Label
6. Sprachcode, Label
7. Label

Wird kein Treffer gefunden, wird das Label selbst, umgeben von eckigen Klammern, zurückgeliefert, um einen Hinweis auf eine fehlende Übersetzung zu geben[46].

Wir verteilen Übersetzungen auf zwei Modelle:

1. Im *TranslationsModel* werden lediglich Bündel von Übersetzungen gepflegt.
2. Im *TranslationModel* (also einem einzelnen Bündel) werden Übersetzungstexte nach obigem Schema gesammelt.

Die Gesamtmenge der verfügbaren Übersetzungen ergibt sich aus allen in Bündeln gepflegten Übersetzungen, wobei neuere Werte ältere überschreiben.

Grafische Modellierung Für Übersetzungen ist wiederum keine grafische Repräsentation erforderlich. Die vorhandenen Übersetzungen werden in einer tabellarischen Übersicht mit allen oben erwähnten Feldern angezeigt. Filter erlauben die Einschränkung nach bestimmten Werten. Zudem wird ein Import und Export auf Basis von Excel angeboten, um die Vergabe der Übersetzungspflege zu vereinfachen.

Wir haben allerdings festgestellt, dass es sehr hilfreich sein kann, Übersetzungen direkt im operativen Portal pflegen zu können. Hier kann man sich als Erweiterung des Prototyps leicht vorstellen, dass in einem speziellen Übersetzungsmodus entweder zu jeder Seite alle pflegbaren Übersetzungen angeboten werden oder aber sogar neben jeder Übersetzung ein Schalter eingeblendet wird, der die Pflege der spezifischen Übersetzung ermöglicht. Im Prototypen ist dies bislang aber noch nicht realisiert.

Erweiterte Modellfacetten Eine der wichtigsten Erweiterungen ist die Internationalisierung und Übersetzung nicht-textueller Ressourcen (insbesondere also von Grafiken und Bildern). Auch hierfür sind nach obigem Schema beschriebene Ressourcenbündel zu definieren.

[46] Zusätzlich wird dieser Fehler in einer Protokolldatei notiert.

Technische Umsetzung Übersetzungen werden beim Systemstart in einem internen Cache abgelegt. Alle Komponenten, die Texte anzeigen oder verarbeiten, greifen auf diesen Übersetzungscache zu und holen daraus die Texte, die zu den Landes- und Spracheinstellungen des aktuellen Nutzers passen. Zudem gibt es Möglichkeiten, interne Caches neu zu initialisieren, um z. B. eine verzögerungsfreie Pflege von Übersetzungen über eine administrative Oberfläche zu ermöglichen.

8.1.8.1.8 SkinModel

Idee Häufig unterscheiden sich Portale visuell nur in der Form- und Farbgebung, da die Inhalte lediglich an ein spezifisches Corporate Design angepasst werden müssen. Wir wollen einfache optische Variationen dadurch unterstützen, dass wir manuell programmierte Layoutmuster vorgeben, die alle prinzipiellen Strukturentscheidungen für die Optik eines Portals treffen[47]. Initial-Layouts müssen also programmatisch erstellt werden.

Beschreibung Um auf Basis dieser manuell entwickelten Layouts schnell kundenspezifische Ausprägungen zu schaffen, sehen wir den folgenden Skin-Ansatz vor, der durch ein passendes Modell abgebildet wird:

- Jedes Layout beschreibt eine Menge von Variationspunkten durch eine Liste dieser Punkte mit den folgenden Einzelangaben:
  - einem eindeutigen Namen,
  - einer textuelle Erläuterung,
  - einem Wertebereich oder Datentyp (positive Ganzzahl, Bild, Link, Farbe, usw.) und
  - einem Default-Wert.

  Sie beschreiben typisierte Variablen, die befüllt werden können, um das spezifische Aussehen des Layouts zu variieren. Typische Variablen könnten Hintergrundfarben, Textfarben, Logos, Breiten von Bereichen, etc. sein. Zudem könnten Boolesche Felder bestimmte Features aktivieren oder deaktivieren (Pulldown-Menüs, Animationen, Werbebanner, usw.). Letzterer Punkt ist auch der vornehmliche Grund dafür, neben einer rein CSS-basierten Konfiguration ein *SkinModel* zu verwenden.

[47] Dies meint zum Beispiel die Platzierung der Navigationsleisten und der Logos sowie die Art und Anzahl von Untermenüstrukturen sowie die prinzipielle Platzverteilung in einem Browser (volle Breite versus fixierte Breite, etc.).

- Jede Skin belegt einige oder alle dieser Werte mit spezifischeren Inhalten.

Das Laufzeitsystem kombiniert dann die Layout-Implementierung mit der dynamischen Skin-Konfiguration, um so das finale Layout zu generieren.

Grafische Modellierung Die Belegung des Skin-Modells erfolgt wiederum in einer tabellarischen Übersicht mit typspezifischer Editor-Unterstützung[48]. Die Wirkung der Einstellungen wird anhand eines Preview-Modus' verdeutlicht, indem mit einer Beispielseite die aktuellen Einstellungen verifiziert werden können.

Erweiterte Modellfacetten Aktuell sehen wir keinen Bedarf für erweiterte Facetten jenseits der zuvor angedeuteten typisierten Variablen.

Technische Umsetzung Das Layout-Template ist gegenwärtig als Faceletsbasierte Komponenten realisiert, die weitere JSF-Komponenten z. B. zur Visualisierung der Navigation verwendet. Seitentemplates werden zuerst ausgeführt und die fertigen Inhalte werden in das umgebende Gesamtlayout eingefügt. Das programmierte Layouttemplate greift bei der Konstruktion des Seitenlayouts auf das aktive Skin-Modell zu, um die dort definierten Variablen auszulesen und für Layoutentscheidungen zu verwenden (z. B. „befindet sich die Navigation links oder rechts?“, „welches Logo soll für das Portal verwendet werden?“ oder „wie breit ist der Navigationsbalken?“).

8.1.8.2 Sekundäre Modelle

Die folgenden sekundären Modelle beschreiben spezifische Aspekte des Systemverhaltens.

8.1.8.2.1 ActionListModel

Idee An vielen Stellen im System wird die Ausführung von Geschäftslogik erforderlich sein. Dazu definieren wir das Konzept von Aktionslisten als sequentieller Abfolge von Einzelaktionen. Einzelaktionen beschreiben wir in Abschnitt 8.1.8.2.2 ab Seite 362.

[48] Beispiele sind ein Dateidialog für die Auswahl von Grafiken mit Thumbnail-Vorschau odr Dialoge mit Farbpaletten zur Farbselektion.

Beschreibung Eine Aktionsliste enthält eine sequentielle Liste von Aktionen, die als transaktionale Klammer ausgeführt werden[49]. Liefert eine Aktion einen Fehler, wird die gesamte Aktionsliste abgebrochen und die Transaktion zurückgenommen.

Grafische Modellierung Aktionslisten werden in einer einfachen tabellarischen Sicht modelliert, in der die Einzelaktionen untereinander (ggf. angereichert um grafische Icons zwecks leichterer Wiedererkennung) aufgeführt werden. Ein Doppelklick auf eine Aktion führt in einen weiteren Dialog, der die Parametrisierung der Einzelaktion erlaubt.

Erweiterte Modellfacetten Verschiedene Parametrisierungen zur Steuerung der Ausführung einer Aktionsliste sind denkbar:

- Ein Filtermodell (siehe Abschnitt 8.1.8.2.3 auf Seite 365) kann festlegen, ob die Aktionsluste prinzipiell ausgeführt werden soll.
- Über ein Attribut könnte gesteuert werden, ob die Transaktionsgrenze auf der gesamten Liste oder auf den einzelnen Aktionen liegt.
- Über ein Attribut könnte gesteuert werden, ob ein Fehler in einer einzelnen Aktion zum Abbruch der gesamten Aktionsliste führt.

Technische Umsetzung Das *ActionListModel* wird zur Laufzeit interpretiert. Bei Ausführung einer spezifischen Transition (oder eines Agenten, für den eine Aktionsliste definiert ist) wird ein Service aufgerufen, der zur aktuellen Aktionsliste über alle Aktionen iteriert, deren Filterbedingungen prüft und die Aktionen ggf. ausführt. Die Ausführung einer einzelnen Aktion ist im nachfolgenden Abschnitt beschrieben.

Alle Aktionen werden in einer Transaktion geklammert. Zudem prüft das Laufzeitsystem den Rückgabewert einer einzelnen Aktion. Vordefiniert sind die Werte *CONTINUE* (mit der Ausführung fortfahren), *ABORT* (die Abarbeitung der aktuellen Aktionsliste beenden) und *HALT* (die Bearbeitung des aktuellen Requests beenden).

8.1.8.2.2 ActionModel

[49] Konkreter bedeutet dies, dass die in ihr enthaltenen Datenbankoperationen als Transaktion ausgeführt werden. Zudem sehen wir in unserer Java-basierten Implementierung die Möglichkeit verteilter Transaktionen durch Konfiguration eines geeigneten JTA-Session-Managers vor (siehe [SunM09c]). Wir sind uns bewusst, dass manche Operationen (z. B. Veränderungen im Dateisystem) nicht Bestandteil der Transaktion sein werden, so dass inkonsistente Systemzustände entstehen können. Da dies aber in allen gängigen Systemen der Fall ist, sehen wir darin auch keine besondere Einschränkung sondern nehmen diese Tatsache bewusst hin.

Idee Aktionen beschreiben die Geschäftslogik eines Portals. Wir wollen Aktionen – soweit irgend möglich – als deklarative und möglichst abstrakte Aktivitäten verstehen.

Beschreibung Aktionsmodelle existieren in vielfältigen Ausprägungen. Jede spezifische Aktion verfügt über ein eigenes Modell, dass die Aktion konfiguriert. Da Aktionen strukturell sehr vielfältig sein können, wollen wir lediglich einige umgangssprachliche Beispiele für Aktionen geben:

- *Lösche ein Objekt O.* Einziger Parameter ist das zu löschende Objekt (identifiziert über eine eindeutige ID).
- *Verknüpfe ein Objekt O_1 mit einem anderen Objekt O_2 gemäß einer Relation R.* Parameter sind die Objekt-IDs sowie der ID der Relation.
- *Versehe ein Objekt O mit Tags, die sich aus einer Eingabe I ergeben.* Parameter sind
  - die Objekt-ID,
  - die ID des Eingabewertes,
  - eine Regel für das Aufspalten der Eingabe in einzelne Tags[50],
  - die Relations-ID der Relation, die für „ist getagged mit“ steht,
  - die ID der Eigenschaftsspalte, die die Begriffe im Tag-Domänentyp (also dem Ziel der Relation) repräsentiert und
  - eine Angabe darüber, ob existierende Verknüpfungen ersetzt oder ergänzt erden sollen.

  Mag die Menge der Eingaben unfangreich erscheinen, muss man sich klar machen, wie viel Aufwand erforderlich wäre, um diese Aktion klassisch zu programmieren:

  - Das Vorhandensein des Eingabewerts muss geprüft werden.
  - Der Eingabewert muss in Einzelbegriffe aufgeteilt werden.
  - Für jeden Einzelbegriff muss geprüft werden, ob er in der Menge der Tags bereits existiert.
  - Existiert ein Begriff noch nicht, muss er neu angelegt werden.
  - Eine Verknüpfung zwischen dem Tag-Element und dem eigentlichen Objekt muss erstellt werden.

[50] Ein Default könnte sein, alle Eingaben als komma-getrennte Begriffe aufzufassen

    - Diverse Konsistenzprüfungen und Fehlerbehandlungen müssen realisiert werden.

- *Versende eine E-Mail gemäß eines vorgegebenen Templates.*
- *Lade eine Liste von Objekten in einen ScopedDataContext (siehe Abschnitt 8.1.7.3 auf Seite 337).*

Grafische Modellierung Aktionen werden aufgrund ihrer Diversität nicht durch grafische Modelle sondern durch einfache Dialogmasken realisiert. Bei komplexen Operationen werden die verschiedenen Parametrisierungsvarianten ggf. auf verschiedene Registerkarten verteilt.

Erweiterte Modellfacetten Aktionen lassen sich typischerweise durch sehr unterschiedliche Parametrisierungsvarianten unterscheiden. Allgemeine Facetten für fast alle Aktionen sind

- ein Filtermodell (siehe die Abschnitte 8.1.8.2.3 auf Seite 365 sowie 8.1.8.2.3 auf Seite 365), das darüber entscheidet, ob die Aktion tatsächlich ausgeführt werden soll,
- für manche Aktionen eine gefilterte Liste (siehe Abschnitt 8.1.8.2.5 auf Seite 369), so dass die Aktion auf jedem Listenobjekt ausgeführt wird.

Technische Umsetzung Zu jeder vordefinierten Aktion ist ein spezifisches Aktionsmodell definiert. Zur Laufzeitkonfiguration gehört eine Abbildung von Aktionsmodellen auf Aktionsimplementierungen. Wird ein spezifisches Aktionsmodell betrachtet, prüft das System zunächst, ob die Aktion nur unter bestimmten Bedingungen ausgeführt werden soll (ob also ein Filter definiert ist), evaluiert ggf. den Filter und lokalisiert dann (wenn die Aktion ausgeführt werden soll) die dazugehörige Implementierung. Die dazugehörige Implementierung wird über eine Standardschnittstelle aufgerufen und erhält als Parameter u. a. das spezifische Aktionsmodell. Eine Implementierung wird dann das spezifische Aktionsmodell betrachten, Parameter auswerten und die aktionsspezifische Geschäftslogik ausführen.

Um flexibel auf besonders komplexe Anforderungen reagieren und leicht Geschäftslogik integrieren zu können, definieren wir generische Aktionen zum Ausführen von Skripten oder Java-Beans (gemäß einer vorgegebenen Schnittstelle)[51]. Auf diese Wei-

[51] Die besondere Aktion bezeichnen wir als `Execute`-Aktion, die semantisch den Aufruf externer Unterroutinen erlaubt.

se sichern wir, dass unsere Modelle nicht die Ausdrucksfähigkeit unseres Systems beschränken, da wir jederzeit handoptimierte Aktionen integrieren können.

Eine ambitionierte Implementierung der Transition-, ActionList- und Action-Modelle könnte die Ideen von Subtext (siehe [Edwa07] bzw. unsere Ausführungen auf Seite 72) aufgreifen – hier ist die Bewertung des Aufwands der relevante Faktor. Da zu Zeitpunkt des Schreibens Subtext leider (immer) noch nicht als Open Source verfügbar war, haben wir diese Idee bislang nicht weiter vertieft.

8.1.8.2.3 FilterModel

Idee Ein Filtermodell ist eine abstrakte Repräsentation für einen Booleschen Ausdruck. Ausdrücke können sowohl Primitive (also z. B. Vergleiche von Eigenschaftswerten) als auch komplexe Funktionen sein[52].

Beschreibung Ein Filtermodell beschreibt auf abstrakte Weise einen logischen Ausdruck. Wir stellen die folgenden Untermodelle bereit, um allgemeine Boolesche Ausdrücke und grundlegende Operationen abzubilden:

- Es werden die n-ären Booleschen Verknüpfungen UND und ODER bereitgestellt.
- Es wird der unäre Negationsoperator NOT bereitgestellt.
- Es werden die üblichen Vergleichsoperatoren bereitgestellt (gleich, ungleich, kleiner, größer, kleiner oder gleich, größer oder gleich). Vergleichsoperatoren können Konstanten und Eigenschaften miteinander vergleichen. Die Operanden müssen den gleichen Typ besitzen.
- Es werden typische String-Operationen wie „beginnt mit“, „endet mit“, „enthält“ bereitgestellt.
- Es werden höherwertige Funktionen bereitgestellt wie „Objekt hat einen bestimmten Typ“, „der User hat sich authentifiziert“, „der User besitzt eine bestimmte Rolle“, „beim Binden von Werten ist es zu einem Validierungsfehler gekommen“ u. a. m.

Wir erlauben die beliebige Schachtelung solcher Ausdrücke.

[52] Beispiele für komplexe Funktionen sind „User ist angemeldet“, „es gibt Validierungsfehler“ oder „aktueller User kann auf Ressource R zugreifen“.

Grafische Modellierung Filtermodelle visualisieren wir auf folgende Weise:

1. Der Ausdruck wird als Baum dargestellt. An den inneren Knoten finden sich die Verknüpfungsoperatoren, an den Blättern die Operanden oder die höherwertigen Funktionen.
2. Der Ausdruck wird zudem umgangssprachlich als Text angezeigt.
3. Die Parametrisierung eines Operanden bzw. einer höherwertigen Funktion erfolgt über einen Dialog, in dem alle Parameter auf kontextspezifische Weise eingestellt werden können.

Erweiterte Modellfacetten Typischer erweitere Modellfacetten sind Detailparametrierungen allgemeiner Operationen (z. B. Hinweise darauf, ob Groß- und Kleinschreibung bei Stringoperationen berücksichtigt werden soll).

Technische Umsetzung Filtermodelle werden von kontextspezifischen Konvertern in logische Ausdrücke für den jeweiligen Kontext konvertiert. So existieren z. B. Konverter, um aus einem Filtermodell Code zu generieren, der das Filtermodell zur Laufzeit auf Java-Ebene gegen den aktuellen Systemkontext prüft ebenso wie Konverter, die aus einem Filtermodell eine Ergänzung für die `WHERE`-Klausel eines SQL-basierten Datenbankzugriffs erzeugen. Auf diese Weise kann ein Filtermodell kontextspezifisch ausgewertet werden.

Wichtig ist dabei, dass je nach Kontext Abfragen auf Werte subtil anders zu interpretieren sind: So sind z. B. Wertabfragen aus einem der aktuellen Kontexte (z. B. gegen den Session-Kontext) bei der Laufzeitauswertung auf Java-Ebene jedes Mal aufzulösen, während sie bei der Konvertierung des Filtermodells in einen SQL-Teilausdruck quasi als Konstanten anzusehen sind, die einmal zum Zeitpunkt der Konvertierung ausgewertet werden, nicht aber für jede betrachtete Ergebniszeile einer SQL-Abfrage.

8.1.8.2.4 ContentElementModel

Idee Benutzerschnittstellen weisen je nach Szenario und Anwenderkreis vielfältige Ein- und Ausgabeelemente auf. Diese Unterschiede beginnen bei der Typisierung der Elemente (reine Anzeige, reine Ausgabe, kombinierte Varianten), gehen über die Komplexität der Elemente (einfache formatierte Eingabemöglichkeiten, Auswahllisten, komplexe Graphstrukturen) und reichen bis zur Anbindung an darunterliegende

Datenstrukturen (1:1-Abbildungen von Inhalten, interpretierte Visualisierung, aggregierte Visualisierung). Wir wollen für unsere webbbasierten Lösungen eine Vielzahl auch erweiterbarer Ein- und Ausgabelemente vorsehen, daher definieren wir als gemeinsame Basisklasse das *ContentElementModel*.

Beschreibung *ContentElementModel*-Instanzen werden in sechs Typen differenziert:

1. *Konstante Elemente:* Diese Elemente zeigen z. B. konstante (aber übersetzbare) Texte oder Grafiken an. Typische Beispiele sind Überschriften, Bilder, usw.
2. *Wert-lesende Elemente:* Diese Elemente zeigen einen einzelnen Objektwert (also eine Objekteigenschaft) an. Dabei handelt es sich üblicherweise um reine Anzeigefelder.
3. *Wert-schreibende Elemente:* Diese Elemente sind üblicherweise auch *wert-lesende Elemente*, erlauben aber zusätzlich die Veränderung des Wertes. Typische Beispiele sind ein- oder mehrzeilige Eingabefelder, Auswahllisten, Checkboxen, usw.
4. *Listen-lesende Elemente:* Diese Elemente lesen eine Liste von Elementen (mglw. gefiltert) und zeigen diese an. Beispiele sind Tabellen, Listen, usw.
5. *Listen-schreibende Elemente:* Diese Elemente sind analog zu *wert-schreibenden Elementen* auch *Listen-lesende Elemente* und erlauben die Veränderung der Listenwerte. Ein typisches Beispiel ist eine editierbare Tabelle.
6. *Aktionsauslösende Elemente:* Diese Elemente lösen Aktionen explizit (durch Angabe einer Transition) oder implizit (durch Zuordnung zu einem anderen komplexeren Element) aus. Beispiele für explizite Aktionsauslöser sind Schalter, für implizite Aktionsauslöser sortierbare Tabellenspalten oder Paging-Elemente einer Tabelle.

ContentElementModel-Instanzen besitzen weitere Eigenschaften besitzen, z. B. einen Filter, der angibt, ob das Element sichtbar ist oder einen Filter, der angibt, ob der oder die Werte in einem schreibenden Element tatsächlich veränderbar sind sowie Positionierungsangaben.

Es ist zu beachten, dass viele Elemente kombinierte Typen sein werden – so ist eine Dropdownbox ein Wert-schreibendes und Listen-lesendes Element, dass ggf. sogar noch als impliziter Aktionsauslöser fungiert (wenn z. B. abhängige Listen bei Auswahl eines Wertes gefiltert werden sollen).

Grafische Modellierung Wir werden zur Modellierung von Sichten klassische WYSIWYG-Vorgehensweisen schaffen, bei denen Masken mit ihren Eingabeelementen annähernd so gestaltet werden können, wie sie dann zur Laufzeit auch aussehen. Um die Gestaltung von Maske zu vereinfachen, werden wir diverse Standardlayouts vorsehen, die ein besonders schnelles Zusammenklicken von Sichten erlauben:

- *Zweispaltiges Basislayout:* Alle Elemente der Maske (bis auf Überschriften und Schalter) werden in nur zwei Spalten angeordnet. Die erste Spalte enthält üblicherweise die Beschriftung eines Ein- oder Ausgabefeldes, die zweite Spalte das eigentliche Feld. Dieses Layout ist insbesondere für einfache Detailsichten, simple Stammdatenerfassungen, usw. geeignet. Großer Vorteil ist, dass solche Layouts z. B. automatisiert zu Datentypen erzeugt werden können, insbesondere um CRUD-Szenarien abzubilden[53].
- *n-spaltiges Basislayout:* Alle Elemente werden in n Spalten angeordnet, wobei jede Spalte eine feste Breite hat. Dieses Layout eignet sich für etwas komplexere Erfassungsmasken, bei denen Elemente in mehr als zwei Spalten angeordnet werden sollen. Aufgrund der festen Spaltenbreiten ist die Anordnung im Vergleich zu freien Positionierung immer noch sehr einfach und erfordert keine fortgeschritteneren Kenntnisse in der ergonomischen Gestaltung von Oberflächen.
- *Freies Layout:* Diese Verallgemeinerung des n-spaltigen Layouts[54] erlaubt die freie rasterbasierte Platzierung von Elementen. Damit können – unter Berücksichtigung der Layoutbeschränkungen in Webbrowsern – nahezu beliebige Layouts realisiert werden.

Insbesondere ist zu beachten, dass diese Layoutmodelle aufeinander aufbauen, so dass man beim Design einer Maske mit der einfachsten Varianten beginnen und bei steigenden Anforderungen automatisiert auf die nächst komplexere Layoutvariante migrieren kann. Bei diesen Hinweisen wollen wir es belassen und verweisen zudem auf Abschnitt 8.1.8.1.3 auf Seite 347 für weitere Ausführungen zu Sichten.

Erweiterte Modellfacetten Typische erweiterte Modellfacetten betreffen Details von Ein- und Ausgabeelementen (z. B. spezielle Formatierungen für die Ausgabe

[53] CRUD steht für *Create, Read, Update, Delete*, also die elementaren Datenoperationen auf einem Objekt und bezeichnet üblicherweise den minimal notwendigen Workflow, um Objekte pflegen zu können.

[54] Es handelt sich deswegen um eine Verallgemeinerung, da sich ein pixelgenaues Layout mit x Pixeln Breite und y Pixeln Höhe als x-spaltiges Layout ausdrücken lässt, bei dem jede Spalte die Breite „1 Pixel“ hat.

oder reguläre Ausdrücke oder Zeichenmengen, die zulässige Eingaben filtern).

Technische Umsetzung Für jedes *ContentElementModel* existiert ein Template, dass dieses Modell in JSF-Code konvertiert. Dabei werden vielfältige technische Parametrierungen ergänzt, die sich aus den deklarativen Modellinstanzen ableiten lassen (z. B. implizite Rechteprüfungen, kontextabhängige CSS-Klassen oder konkrete Variablenbezüge, die sich aus abstrakten Modellreferenzen ergeben).

8.1.8.2.5 FilteredListModel

Idee In vielen Fällen müssen zur Implementierung von Geschäftslogik Teilmengen des Datenbestandes selektiert und dann bearbeitet werden. Ein *FilteredListModel* erlaubt die Selektion einer solchen Teilmenge.

Beschreibung Ein *FilteredListModel* setzt sich aus bereits bekannten Bestandteilen zusammen: Es wird ein *DomainTypeModel* (siehe Abschnitt 8.1.8.1.4) festgelegt, aus dessen Inhalten eine Teilmenge selektiert werden soll. Es wird außerdem ein *FilterModel* (siehe Abschnitt 8.1.8.2.3) definiert, dass die zu selektierende Teilmenge charakterisiert.

Grafische Modellierung Die Auswahl des *DomainTypeModel*s kann über einen einfachen Dialog (z. B. eine Dropdown-Box) erfolgen, zur Darstellung des Filters verwenden wir die in Abschnitt 8.1.8.2.3 beschriebene Vorgehensweise.

Erweiterte Modellfacetten Erweiterte Modellfacetten können die zu selektierende Teilmenge weiter einschränken, z. B. durch die Festlegung, dass gleiche Objekte nur einmal selektiert werden sollen (DISTINCT auf SQL-Ebene) oder dass nur eine feste Anzahl von Elementen („die ersten n Objekte“, „die letzten n Objekte“) selektiert werden sollen.

Technische Umsetzung Eine gefilterte Liste ist nichts anderes als eine spezielle Parametrisierung einer Aktion: Die Aktion wird nur für ein bestimmtes Aktionsmodell ausgeführt sondern sie wird für das spezifische Aktionsmodell einmal je gefiltertem Objekt ausgeführt. Die gefilterten Objekte können trivial ermittelt werden, indem der Filter als Parametrisierung der allgemeinen `findBy`-Methode des

für den angegebenen Domänentype zuständigen Repositories verwendet wird (siehe Abschnitt 8.1.8.1.4 ab Seite 351).

8.1.8.3 Tertiäre Modelle

Die folgenden beispielhaften tertiären Elemente werden allesamt in einfachen tabellarischen Dialogen („Eigenschaftslisten“) dargestellt.

8.1.8.3.1 PortalModel Das Portalmodell dient als Wurzelknoten aller miteinander verknüpfter Modelle. Insofern ist es insbesondere ein Container, der die anderen verfügbaren Modelle auflistet, z. B. die Menge der verfügbaren Prozesse, der verfügbaren Domänentypen und der darstellbaren Navigationen. Zudem enthält es eine Vielzahl interner technischer Informationen, die das Laufzeitsystem zur Generierung der Anwendung benötigt. Dazu werden allgemeine Modelle mit zusätzlicher Semantik versehen. Beispiele sind „Domänentyp x repräsentiert einen Anwender, der sich anmelden kann“ oder „Prozess y repräsentiert den Anmeldeprozess, der auszulösen ist, wenn ein Anwender auf nicht authorisierte Seiten zugreift“.

Technische Umsetzung Das Portalmodell wird nicht direkt in ein Laufzeitartefakt übersetzt sondern dient dem Generator als Wurzelknoten für die verschiedenen untergeordneten XML-Modelle sowie für Meta-Informationen (z. B. “welcher Domänentyp hat die Semantik *Benutzer*“ oder „welche Benutzer-Eigenschaft hat die Semantik *Loginname*“).

8.1.8.3.2 DataSourceModel Um die Integration von externen Datenquellen und Schnittstellen zu modellieren, erweitern wir die in Abschnitt 8.1.8.1.4 ab Seite 351 beschriebenen Domänentypen derart, dass wir jedem Domänentypen eine Datenquelle zuordnen. Geschieht dies nicht explizit, wird als Default die interne Portaldatenbankinstanz benutzt. Ein *DataSourceModel* ist sehr einfach strukturiert, indem es einen *Datenquellentypen* sowie eine Menge von (Name, Wert)-Paaren zur Parametrisierung des Datenquellentyps festlegt. Identifiziert wird eine Datenquelle über einen logischen Namen[55]. Ein Datenquellentyp wiederum wird auf eine spezifische und

[55] Eine Datenquelle besitzt wie alle unsere Modelle auch eine GUID, diese kann aber nicht zu Identifikation verwendet werden, da wir je Stage andere Parametrisierungen von Datenquellen verwenden werden. Daher identifizieren wir sie über einen zusätzlichen Namen, der im Gegensatz zu GUIDs gerade nicht global eindeutig sein muss. Siehe dazu auch unsere Ausführungen auf Seite 354.

systeminterne Implementierung abgebildet, die über spezielle Strategie-Interfaces die üblichen CRUD-Operationen auf einem Domänentypen ermöglicht. Durch diese Abstraktion wird es für alle anderen Elemente des Systems transparent möglich, auf verschiedene Datenquellen zuzugreifen.

Eine weitere Komplexität tritt bei miteinander verknüpften Datentypen auf (z. B. “Kunde hat Ansprechpartner“). Hierzu führen wir das Konzept sogenannter *ConnectionHandler* ein, die als Modellfacette einer Relation (siehe Abschnitt 8.1.8.1.5 auf Seite 355) angegeben werden können und bei Vorhandensein die Verantwortung für das Bereitstellen verknüpfter Daten delegiert bekommen. Per Default werden als Relationen über eine entsprechende direkte Repräsentation auf Datenbankebene abgebildet. Sollen aber zum Beispiel Daten aus einem Wiki mit Daten aus einem ERP verknpüft werden, würden wir auf der semantischen Relation (definiert durch ein *RelationModel*) zwischen den Domänentypen einen speziellen *ConnectionHandler* definieren, der die Zuordnung verwaltet[56] (siehe wiederum Abschnitt 8.1.8.1.5 auf Seite 355).

Technische Umsetzung Ein Datenquellenmodell wird von dem spezifischen Datenquellentreiber frei interpretiert. Die intiale Version erlaubt (Name, Wert)-Tupel, die dazu dienen, gängige Konfigurationen zu beschreiben. Die Interpretation erfolgt durch manuell erzeugten Code in den spezifischen Treibern. Abhängig vom Treiber kann dies bereits beim Systemstart oder erst im Verlauf des Zugriffs auf eine spezifische Datenquelle der Fall sein.

8.2 CDPM am Beispiel „Portalsysteme“

Für allgemeine Portalprojekte verwenden wir ein iteratives, evolutionäres, rekursives und inkrementelles Story-basiertes Vorgehensmodell. Gemeinsam mit dem Anwender werden zu Beginn die prinzipiellen Anforderungen des Projekts in einem möglichst kurzen und präzisen Pflichtenheft formuliert, Details der Implementierung werden konsequent nach dem in Kapitel 6 beschriebenen Rapid-Development-Vorgehen (siehe Abschnitt 6.4.3.4) inkrementell erarbeitet und in iterativen Prototypen zur finalen Lösung hin entwickelt. Häufig verwenden wir dabei Feature Driven Development als

[56] Ein typisches Beispiel aus unserer Praxis ist hier z. B. die Anbindung von kundenspezifischen Besuchsberichten als in einem Außendienstportal zu pflegendes Element, die dann mit Kundenstammdaten aus einem ERP-System verknüpft werden, um das ERP-System mit CRM-Elementen aufzubohren. Auf diese Weise kann die häufig wesentlich kostspieligere Erweiterung des ERP-Systems einfach umgangen werden.

grundlegendes Vorgehensmodell. Das Pflichtenheft verwendet Mindmaps für Navigationsmodelle, Prozessmodelle und Mockups für Masken, so dass eine Umsetzung der Anforderungen sehr direkt möglich ist.

Für den spezielleren Bereich der öffentlichen Lebensmittelüberwachung, für den die Firma BALVI das PIKAS-CMDSD-System einsetzt, wird ein mit dem jeweiligen Auftraggeber und den Rahmenbedingungen der spezifischen Ausschreibungen abgestimmtes (und ggf. modifiziertes) Projektmanagementmodell basierend auf dem V-Modell XT (siehe Abschnitt 3.4.3) gewählt. Die von uns vorgestellten CDPM-Prinzipien können unverändert verwendet werden, die Methoden sind nur insoweit Bestandteil des jeweiligen konkreten Vorgehensmodells, als dass der Kontext dies erlaubt. Andernfalls dienen sie als Hilfsmethoden während der operativen Abwicklung des Projekts mit einem kontextspezifischen Gewichtungsgrad. Insbesondere eignen sie sich dabei zur effizienten Organisation von Workshops und Abstimmungsterminen.

Es zeigt sich in beiden Fällen, dass bestehende Ansätze sehr einfach mit CDPM kombiniert werden können, das die Mehrzahl unserer Vorgaben das Verhalten der Projektteilnehmer zu steuern versucht und keine formalen Prozesse verändert. Einzig Rapid Development ist eine spezielle Vorgehensweise, die sich nicht in jedes Vorgehensmodell einfügt. Sind agile Vorgehensmodelle prinzipiell mit dem jeweiligen Kunden vereinbar, kann auch diese Methode unverändert übernommen werden. Ansonsten raten wir dazu, sie als Arbeitsmittel in operativen Abstimmungsgesprächen zu nutzen und nicht als formalen Bestandteil des spezifisch ausgeprägten Vorgehensmodells zu definieren. In letzterem Fall wird sich unserer Erfahrung nach die Gesamteffizienz des Projekts gefühlt verringern, aber immer noch mindestens die Qualität rein klassischer Ansätze besitzen. Uns liegen gegenwärtig aber noch keine umfassenden Daten zu diesen Sachverhalten vor, so dass wir dazu auf Abschnitt 10.3.1 auf Seite 413 verweisen.

8.3 Zusammenfassung

Wir haben anhand des Beispiels von PIKAS gezeigt, wie sich die Methoden CMDSD und CDPM auf die Domäne „Mittelständische Portallösungen“ anwenden lassen. Dazu haben wir in Abschnitt 8.1 auf Grundlage unserer Definitionen aus Abschnitt 5.3 die Anwendung der Schritte 1 bis 10 des CMDSD-Vorgehensmodells beschrieben. Wesentliche Schritte dabei sind die Domänenabgrenzung, die Entwicklung des Architekturstils, die Herleitung von Archetypen und Anforderungen, die Modellbildung

und die Umsetzung der Modelle in technische Artefakte. Schritt 11 sowie CDPM haben wir nur kurz umrissen, da die detaillierte Evaluierung hier noch aussteht und bislang lediglich erste Erfahrungswerte vorliegen. Es hat sich dabei gezeigt, dass alle Schritte gut durchführbar sind und in ihrer Reihenfolge zu einer strukturierten Planung und Umsetzung des CP-Ansatzes beitragen.

Darauf aufbauend werden wir in Kapitel 9 anhand konkret gemessener Aufwandswerte Erkenntnisse zur allgemeinen Leistungsfähigkeit unseres CP-Ansatzes ableiten und ein Formelwerk erstellen, das bei der Planung weiterer CP-Vorgehensmodelle für andere Domänen wertvolle Dienste leisten kann.

Kapitel 9

Bewertung der Ergebnisse

„Für das Können gibt es nur einen Beweis: Das Tun."
— Marie von Ebner-Eschenbach (1830–1916), Österreichische Schriftstellerin

In Kapitel 8 haben wir unseren CP-Ansatz exemplarisch am Beispiel der Domäne „*mittelständische Portalsysteme*" umgesetzt und die operativen Erfahrungswerte und Erkenntnisse zusammengestellt. In diesem Kapitel analysieren wir die vorliegenden Erfahrungen bei Aufbau und Nutzung des CP-Ansatzes im mittelständischen Kontext und gehen dabei insbesondere auf das Kosten-/Nutzenverhältnis ein. Die Erfahrungen wurden im Zeitraum Oktober 2007 bis Juli 2009 von im Schnitt zwei bis drei Personen während der Erstellung des Systems gesammelt. Gleichzeitig wurden erste Projekte gestartet, um die Brauchbarkeit und Leistungsfähigkeit des Ansatzes möglichst frühzeitig an Hand realer Anwendungsfälle zu prüfen.

Die Implementierung des resultierenden CP-Ansatzes ist nicht abgeschlossen und wird projektbegleitend weiter fortgeführt, so dass zukünftig weitere Erkenntnisse zu erwarten sind. Alle in Kapitel 8 beschriebenen Modelle sind auf XML-Ebene umgesetzt, die Entwicklung der grafischen Editoren hat gerade erst begonnen, so dass hier noch keine Evaluierung möglich ist. Wir gehen auf die Auswirkungen der fehlenden Editoren sowie die daraus resultierenden Schlussfolgerungen insbesondere in Abschnitt 9.2 ein.

Bei der Aufarbeitung der Ergebnisse fokussieren wir uns auf zwei der drei Hauptmethoden des Conceptual Programming (CP): Conceptual Model Driven Software Development (CMDSD) und Concept Driven Project Management (CDPM). Die dritte Methode „Produktgremium" lassen wir außen vor, da Sie im Wesentlichen der Theorie der Softwareproduktlinien und dort dem Managementregelkreis entliehen ist und hier keine neuen Erkenntnisse gewonnen wurden. Zu CMDSD stellen wir in

Abschnitt 9.1 eine Reihe von Erfahrungswerten vor und leiten dort auch eine Reihe von Bewertungsformeln her, die bei Machbarkeitsprüfungen für andere CP-basierte Vorhaben genutzt werden können. Zu CDPM beschreiben wir in Abschnitt 9.2 einige operative Erfahrungswerte, die wir allerdings bislang nicht weiter untersuchen konnten, da uns wie bereits erläutert noch die notwendigen grafischen Werkzeuge fehlen. Wir gehen aber auf übertragbare Experimente ein, die den potenziellen Nutzwert illustrieren. Grundlage für alle Ergebnisse ist das von uns entwickelte CMDSD-Werkzeug PIKAS (siehe Seite 310) sowie die mit dem Werkzeug gesammelten Erfahrungswerte. In Abschnitt 9.3 skizzieren wir weitere auf dem CP-Ansatz basierende Projekte und dort gesammelte Erfahrungen. Abschnitt 9.4 deutet andere mögliche Einsatzfelder an, die zum Teil in ihrer Struktur deutlich von dem von uns gewählten prototypischen Beispiel abweichen. Abschnitt 9.5 fasst die Ergebnisse dieses Kapitels kurz zusammen und leitet in das Fazit der Arbeit über.

9.1 Ergebnisse der CMDSD-Nutzung

Die CMDSD-Methode gliedert sich in ihrer Bewertung in zwei Hauptbereiche. Zum einen steht vor ihrer Anwendung die Entwicklung eines CMDSD-Werkzeugs durch einen IT-Dienstleister, das es erlaubt, die Ideen der Methode domänenspezifisch anzuwenden. Diese Phase ist im Wesentlichen eine Vorinvestitionsphase. Zum Anderen wird das CMDSD-Werkzeug dann in konkreten Projekten angewendet, um eine wesentlich schnellere Lösungsentwicklung zu ermöglichen. In dieser Phase geht es also darum, die getätigten Investitionen in Erlöse umzusetzen. Dabei ist natürlich zu berücksichtigen, dass auch während der Nutzung in Projekten weitere Investitionen erforderlich sein werden, um das Werkzeug weiter zu entwickeln, Fehlern zu beheben, usw. Damit die Methode als zielführend angesehen werden kann, sollten die Erlöse durch die Einführung und Nutzung des CMDSD-Werkzeugs die zuvor getätigten Investitionen schnell wieder einspielen. Wir sehen hier als typischen Zeitraum zur Erzielung des „Return on Invest“ einen Maximalzeitraum von ca. zwei Jahren.

Wie bereits in der Einleitung erläutert, ist unser prototypisches CMDSD-Werkzeug noch nicht vollständig realisiert, da die grafischen Editoren noch nicht vorhanden sind. Darauf gehen wir in Abschnitt 9.2 ausführlicher ein, wollen aber hier bereits begründen, warum dies keine wesentliche Einschränkung der Relevanz der nachfolgend vorgestellten Ergebnisse darstellt: Wie in Abschnitt 8.1.3 erläutert, verwenden wir intern ein XML-basiertes Format zur Speicherung der Modelle. Dieses XML-Format ist dahingehend optimiert, dass auch Menschen sehr gut damit arbeiten

können. Dies war eine wichtige Anforderung, um möglichst früh Projekte auf Basis von PIKAS umsetzen zu können. Wir planen, dieses XML-Format auch in Zukunft zu unterstützen und benutzbar zu halten, so dass jederzeit die Möglichkeit bestehen wird, die Ebene der grafischen Editoren zu verlassen und auf XML-Ebene zu arbeiten. Dies hat zwei wesentliche Vorteile: Zum einen müssen die grafischen Editoren nicht jede neue Eigenschaft der XML-Modelle auch grafisch unterstützen sondern die Modelle nur einlesen und zurückschreiben können. Diese Vorgehensweise reduziert die Entwicklungsabhängigkeiten in der Werkzeugkette und beschleunigt die Gesamtevolution von PIKAS. Zum anderen stellen die auf Basis der XML-Modelle erzielten und nachfolgend beschriebenen Ergebnisse eine Untergrenze für den Geschwindigkeitsgewinn in der Entwicklung dar, da man davon ausgehen kann, dass eine grafische Unterstützung eher eine Beschleunigung in der Entwicklung erlaubt und – sollte dies ausnahmsweise nicht der Fall sein – man auch jederzeit auf die XML-Ebene ausweichen kann, wenn es um die Effizienz geht.

Zwei weitere Einschränkungen sind bei den nachfolgend beschriebenen experimentellen Ergebnissen zu berücksichtigen:

- Da die Umsetzung des Prototypen im realen Geschäftsumfeld durch die QuinScape GmbH und nicht im Laborumfeld erfolgt, waren keine Budgets vorhanden, um parallel eine Umsetzung mit klassischen Ansätzen zu evaluieren. Wir können uns daher im Folgenden lediglich auf vom CP-Projekt unabhängigen Mitarbeitern getroffene Aufwandsabschätzungen berufen, die auf deren subjektiven Erfahrungen beruhen und somit lediglich Anhaltspunkte für das Kosten-/Nutzenverhältnis liefern. Da dies aber sehr wirklichkeitsgetreu das Umfeld der meisten mittelständischen Dienstleister widerspiegeln wird, die sich für einen CP-basierten Ansatz interessieren, halten wir dies nur bedingt für eine Schwäche, da subjektive Entscheidungen bei beschränkten Budget- und Erfahrungsrahmen immer eine Rolle spielen werden.
- Die Vergleichexperimente wurden nur in kleinem Rahmen durchgeführt. Um sie repräsentativer zu gestalten, wären größere Zahlen von Wiederholungen mit unterschiedlichen Entwicklern notwendig. Aus Kostengründen waren solche Experimente bislang nicht durchführbar. Wir sind diesem Problem begegnet, indem wir mit den gleichen Entwicklern eine Reihe von Messungen vorgenommen haben, bei denen die betroffenen Entwickler schätzen mussten, welchen Aufwand sie bei klassischer Herangehensweise und bei der Nutzung von PIKAS erwarten würden. Dann haben wir die Anforderungen mit PIKAS umgesetzt und die Aufwände gemessen. Die Aufwandsschätzungen für den

klassischen Ansatz wurden gemittelt, wobei QuinScape-interne Korrekturfaktoren für die individuellen Entwickler aufgerechnet wurden, die sich aus den Abweichungen zwischen vergangenen Schätzungen und dann real angefallenen Aufwänden ergeben. Diese Vorgehensweise entspricht der QuinScape-internen Vorgehensweise bei der Aufwandskalkulation für reale Angebote und hat sich in der Vergangenheit gut bewährt, um gewinnbringende Projekte zu realisieren, so dass sie uns auch hier für die Evaluierung angemessen erscheint.

9.1.1 Kosten der Erstellung des CMDSD-Werkzeugs

Es gibt eine Reihe komplexer und von relativ schwer messbaren Faktoren abhängigen Berechnungsmodellen zur Überprüfung des Kosten-Nutzen-Verhältnis und zur Berechnung des „Return On Invest“ (ROI) beim Einsatz von Software-Produktlinien, die sich eigentlich als Grundlage anbieten würden, um zu versuchen, mit Ihnen auch den Nutzen des CP-Einsatzes zu bewerten (siehe z. B. [ClMC05, BMMS04, BCMM03, Cohe03, With96]). Es wird von der Verwendung dieser Verfahren abgesehen, weil uns zum einen nicht alle Kenngrößen vorliegen, die zur Anwendung dieser Verfahren erforderlich sind und weil es zum anderen den Rahmen dieser Arbeit gesprengt hätte, diese Verfahren auf ihre Anwendbarkeit und Sinnhaftigkeit für eine CP-basierte Vorgehensweise zu untersuchen. Insbesondere fehlt uns gegenwärtig aber eine signifikante Stichprobenmenge von Projekten, die bereits mit PIKAS umgesetzt wurden, um einen letztendlichen Vergleich vorzunehmen.

Stattdessen wollen wir die bislang angelaufenen Kosten untersuchen, da diese als Grundlage dienen, um zu ermitteln, wie viel Zeit erforderlich war, um zu einem produktiv nutzbaren System zu kommen.

Im konkreten Fall können wir von einem internen Kostensatz von K_M EUR pro Mitarbeitertag ausgehen. Außerdem wurden T_G Arbeitstage investiert, so dass sich als einfache Formel für die bislang angelaufenen *vergangenheitsbezogenen Erstellungskosten* K_{E_V} ergibt:

$$K_{E_V} = K_M \times T_G \tag{9.1}$$

Alternativ muss man bei einer zukunftsbezogenen Betrachtungsweise die zu erwartenden Kosten abschätzen. Hier bieten sich im Wesentlichen zwei Vorgehensweisen an: Man kann versuchen, eine mittleren Aufwandsabschätzung vorzunehmen, die von Durchschnittswerten ausgeht oder man kann eine detaillierte leistungsbezogene Schätzung durchführen. Wir geben für beide Vorgehensweisen Formeln an.

Bei einer mittelnden Aufwandsabschätzung gehen wir davon aus, dass im Mittel M_E Mitarbeiter eingesetzt werden. Die Arbeiten erstrecken sich über einen Zeitraum von D_E Monaten, im Durchschnitt rechnen wir mit T_m Arbeitstagen pro Monat. Allgemein ergeben sich *durchschnittliche Gesamterstellungskosten* K_{E_D} in Höhe von

$$K_{E_D} = K_M \times M_E \times D_E \times T_m \tag{9.2}$$

Leistungsbezogene Abschätzungen können wie folgt berechnet werden: K_M lässt sich aus den internen Controlling-Daten eines Unternehmens zuverlässig ermittelt. D_E kann auf Grundlage eines Lasten- und Pflichtenheftes abgeschätzt werden. Dazu kann es sinnvoll sein, von Durchschnittswerten auf konkrete Leistungspositionen $L_1, \ldots, L_n$ überzugehen, wobei sich dann für eine Aufwandsbewertungsfunktion $f_a : L \rightarrow \mathbb{N}$ (die die Zahl der geschätzten Personentage zur Realisierung einer spezifischen Leistungsposition liefert[1]) folgende modifizierte Formel zur Berechnung der *leistungsbezogenen Gesamterstellungskosten* K_{E_L} ergibt:

$$K_{E_L} = K_M \times (\sum_{i=1}^{n} f_a(L_i)) \tag{9.3}$$

Wir ermitteln die initialen Erstellungskosten für PIKAS mit den folgenden Parametern für die vereinfachte Formel 9.1, da wir rückwirkend Evaluierungsergebnisse bewerten:

- Wir setzen interne Kosten pro Mitarbeiter und Tag in Höhe von $K_M = 350$ EUR an.
- Laut unserer Zeiterfassung haben wir $T_G = 990$ Personentage Aufwand in die bisherige Entwicklung von PIKAS investiert.

Eingesetzt in Gleichung 9.1 ergibt sich damit:

$$\begin{aligned} K_{E_V} &= K_M \times T_G \\ &= 350 \times 990 \\ &= 346500 \end{aligned}$$

Die initiale Entwicklung des PIKAS-Systems war also eine Investition in Höhe von 346.500 EUR. Wir werden im weiteren Verlauf auf diesen Wert zurückkommen, ihn

[1] Wir gehen aus kaufmännischer Vorsicht ausschließlich von ganzzahligen Personentagen aus und empfehlen, bei kleineren Leistungspositionen diese nach Möglichkeit geeignet zusammenzufassen und im Zweifel in vernünftigem Maße aufzurunden, um zu einer konservativen Abschätzung zu gelangen.

fortan aber als K_E (für *Erstellungskosten*) bezeichnen, da alle nachfolgenden Formeln mit jeder der Gleichungen 9.1 bis 9.3 arbeiten können. Es gilt also $K_E = 346500$ für unseren PIKAS-Prototypen.

Zu beachten ist noch die Frage, wie Urlaub und Krankheit zu berücksichtigen sind. Wir haben uns dazu entschieden, diese Faktoren gänzlich aus den Formeln heraus zu lassen, da sie ebenso bei einer alternativen Vorgehensweise wirken würden und somit für unsere Betrachtungen einen konstanten Faktor darstellen. Somit wirken sich diese Faktoren faktisch nur auf den Zeitraum aus, der bis zur Zielerreichung verstreicht.

9.1.2 Nutzen der Erstellung des CMDSD-Werkzeugs

Der Nutzen des CMDSD-Werkzeugs PIKAS ergibt sich insbesondere aus zwei konkreten Kenngrößen: Zum einen stellen wir in Abschnitt 9.1.2.1 den Einfluss auf die Einarbeitungs- und Weiterbildungskosten von Mitarbeitern dar, zum anderen beschreiben wir in Abschnitt 9.1.2.2 die Auswirkungen auf die Umsetzungsgeschwindigkeit von Anforderungen. Zudem beschreiben wir in Abschnitt 9.1.2.3 die Konsequenzen bei Technologiemigrationen und in Abschnitt 9.1.2.4 die Auswirkungen des Einsatzes von PIKAS auf die Lösungsqualität. Auch diese beiden Faktoren werden sich positiv auswirken, in die Berechnung des Kosten-Nutzen-Verhältnisses in Abschnitt 9.1.3 nehmen wir aber nur die Ergebnisse zur gesteigerten Umsetzungsgeschwindigkeit aus Abschnitt 9.1.2.2 auf, da sich zu den anderen Punkten auch Kritik finden lässt, durch die wir unsere Ergebnisse aber nicht verwässern wollen[2].

9.1.2.1 Reduzierung von Einarbeitungs- und Weiterbildungskosten

Direkt aus der Verfügbarkeit des CMDSD-Werkzeugs ergibt sich bereits ein zumindest grob abschätzbarer Nutzen, da die Einarbeitungskosten für neue Mitarbeiter sowie die Weiterbildungskosten in das Web- und Portalumfeld deutlich reduziert werden konnten. Die Reduktion ergibt sich daraus, dass die Mitarbeiter zum einen

[2]So könnte man argumentieren, dass Einarbeitungs- und Weiterbildungskosten genauer zu untersuchen sind und entsprechende Tätigkeiten in der heutigen IT-Welt häufig „on the job“ erfolgen. Die konkreten Auswirkungen auf die Umsetzungsqualität haben wir nicht unter Laborbedingungen gemessen, so dass wir Sie hier auch nicht weiter anführen wollen. Die bei Technologiemigrationen erzielten Effizienzgewinne erfordern ebenfalls eine breitere Stichprobenbasis und sind zudem sehr schlecht planbar bei einer strategischen Kostenbetrachtung. Daher erscheint es uns angemessen, alle drei Punkte außen vor zu lassen, da dies unsere Bewertung äußerst konservativ und vorsichtig gestaltet.

wesentlich weniger technische Details erfassen, verstehen und anwenden können müssen, um zukünftig eine reichhaltige Webanwendung entwickeln zu können und zu anderen daraus, dass viele technologische Weiterentwicklungen „unter der Haube“ des Generators vorgenommen werden können. Wir haben dazu historisch erfasste Einarbeitungsaufwände aus dem Umfeld der QuinScape GmbH mit den Einarbeitungsaufwänden verglichen, die im Zusammenhang mit PIKAS gemessen werden konnten.

Konkret ergeben sich die folgenden Zahlen:

- Vor der Umstellung auf PIKAS hat ein Mitarbeiter im Mittel $E_{M_{\text{alt}}}$ Monate Einarbeitungszeit benötigt, um alle Technologien hinreichend detailliert anwenden zu können, die zur Umsetzung moderner JavaEE-basierter Portale und Webanwendungen erforderlich sind.

- Nach der Umstellung benötigt ein Mitarbeiter im Schnitt $E_{M_{\text{neu}}}$ Monate Einarbeitungszeit, um alle Aspekte des PIKAS-Systems so weit verinnerlicht zu haben, dass er selbstständig komplexe Portal- und Webprojekte umsetzen kann. Annahme ist in beiden Fällen, dass der Mitarbeiter mit guten Erfahrungen als Programmierer vorgebildet ist und nicht das Programmieren von Grund auf erlernen muss. Müssen gänzlich unerfahrene Entwickler in klassische oder PIKAS-basierte Entwicklung eingearbeitet werden, gehen wir (basierend auf Erfahrungswerten für den klassischen Fall) von einer Verdopplung bis Verdreifachung der Einarbeitungsaufwände aus. Diesen Aspekt wollen wir hier aber nicht weiter betrachten, da wir bereits in Kapitel 5 ausgeführt haben, dass CP-basierte Ansätze nur dann Sinn machen, wenn umfassende Vorerfahrungen in der Domäne existieren, so dass man sinnvollerweise nicht die unerfahrensten Mitarbeiter initial mit solchen Projekten beschäftigen wird.

- Die Zeit, die für Weiterbildungsmaßnahmen benötigt wird, reduziert sich von geschätzten $W_{M_{\text{alt}}}$ auf $W_{M_{\text{neu}}}$ Tage (man beachte die unterschiedlichen Zeiteinheiten zwischen Einarbeitung und Weiterbildung!)[3].

- A_{NM} ist die Zahl der Neueinstellungen pro Jahr und A_{AM} die Zahl der weiterzubildenden Altmitarbeiter pro Jahr.

Die jährliche Ersparnis N_{KM} durch reduzierte Einarbeitungs- und Weiterbildungs-

[3] Wir haben uns dazu entschlossen, die unterschiedlichen Einheiten zu wählen, da es real normalerweise ein deutlich unterschiedliches Verhältnis gibt und die Werte leicht umzurechnen sind.

aufwände ergibt sich als (Nutzen für das Wissensmanagement[4]):

$$\begin{aligned} N_{\mathrm{KM}} &= (A_{\mathrm{NM}} \times (E_{M_{\mathrm{alt}}} - E_{M_{\mathrm{neu}}}) \times T_m \times K_M) + \\ &\quad (A_{\mathrm{AM}} \times (W_{M_{\mathrm{alt}}} - W_{M_{\mathrm{neu}}}) \times K_M) \\ &= ((A_{\mathrm{NM}} \times (E_{M_{\mathrm{alt}}} - E_{M_{\mathrm{neu}}}) \times T_m) + \\ &\quad (A_{\mathrm{AM}} \times (W_{M_{\mathrm{alt}}} - W_{M_{\mathrm{neu}}}))) \times K_M \end{aligned} \tag{9.4}$$

Wir setzen Werte ein, die sich aus konkreten Werten bei der QuinScape GmbH ergeben:

- Im Schnitt werden pro Jahr $A_{\mathrm{NM}} = 2$ neue Mitarbeiter eingestellt.
- Diese haben bislang $E_{M_{\mathrm{alt}}} = 2$ Monate Einarbeitungszeit benötigt.
- Basierend auf vorläufigen Ergebnissen scheinen neue Mitarbeiter nur noch $E_{M_{\mathrm{neu}}} = 0,5$ Monate Einarbeitung mit PIKAS zu benötigen.
- Es existieren A_{AM} 12 alte Mitarbeiter, die zukünftig mit PIKAS arbeiten werden.
- Diese haben in der Vergangenheit $W_{M_{\mathrm{alt}}} = 15$ Tage pro Jahr auf Weiterbildung in Webtechnologien investiert.
- Nach vorläufigen Erfahrungen scheinen mit Einführung von PIKAS nur noch $W_{M_{\mathrm{neu}}} = 5$ Tage Weiterbildung pro Jahr (im Wesentlichen, um neue Features von PIKAS kennenzulernen[5]) erforderlich zu sein.
- Wir gehen von $T_m = 18$ produktiven Tagen pro Monat aus.
- Wir gehen außerdem von einem internen Kostensatz von $K_M = 350$ EUR pro Mitarbeiter und Tag aus, der dem QuinScape-Satz ähnelt.

Setzen wir diese Zahlen in Gleichung 9.4 ein, erhalten wir:

$$\begin{aligned} N_{\mathrm{KM}} &= ((A_{\mathrm{NM}} \times (E_{M_{\mathrm{alt}}} - E_{M_{\mathrm{neu}}}) \times T_m) + (A_{\mathrm{AM}} \times (W_{M_{\mathrm{alt}}} - W_{M_{\mathrm{neu}}}))) \times K_M \\ &= ((2 \times (2 - 0,5) \times 18) + (12 \times (15 - 5))) \times 350 \\ &= (54 + 120) \times 350 \\ &= 60900 \end{aligned}$$

[4] Wir bezeichnen die Einarbeitungs- und Weiterbildungsaufwände für Mitarbeiter hier vereinfachend als „Wissensmanagement“ und verwenden daher den Index KM für „Knowledge Management“.

[5] Die neuen Features sind das Ergebnis projektbegleitender Aufwände, die auch bei Nutzung eines CMDSD-Werkzeugs anfallen. Diese werden wir im Rahmen unserer Kosten-Nutzen-Analyse ab Seite 391 als $K_{\mathrm{PB}_{\mathrm{Jahr}}}$ berücksichtigen, greifen hier durch ihre Auswirkungen auf die Notwendigkeit von Weiterbildung aber bereits vor.

Für das oben definierte Szenario ergeben sich durch die Einführung von PIKAS also Kosteneinsparungen im Weiterbildungsbereich in Höhe von ca. 60.900 EUR pro Jahr. Diese Zahl ist insbesondere dann sehr interessant, wenn im Unternehmen eine steigende Zahl von Mitarbeitern in PIKAS (bzw. in einem vergleichbaren CP-Ansatz) geschult würden und sich in solchen Szenarien ähnliche Einsparungen pro Mitarbeiter ergeben, da sich hier wieder Skaleneffekte sehr schnell zum Positiven auswirken können (siehe auch unsere Ausführungen dazu auf Seite 39). Wir wollen diesen Punkt zusammenfassend abschließen mit der Bemerkung, dass wir hier bereits einen Indikator dafür sehen, dass eine hohe Abstraktion tatsächlich signifikant Kosten sparen helfen kann, insbesondere da Weiterbildungskosten in unseren vorausgegangenen Ausführungen bislang kein dominantes Thema waren.

9.1.2.2 Reduzierung von Umsetzungsaufwänden

Um die Auswirkungen auf die Umsetzungsgeschwindigkeit in Projekten zu messen, haben wir eine Reihe von Vergleichsexperimenten durchgeführt, bei denen sowohl bereits bekannte als auch neue Anforderungen mit PIKAS umgesetzt wurden. Für bereits bekannte Anforderungen haben wir Aufgabenstellungen aus vergangenen Projekten ausgewählt und den dafür tatsächlich angefallenen Aufwand in Verhältnis gesetzt zu der Zeit, die für eine Umsetzung auf Basis von PIKAS benötigt wurde. Für neue Anforderungen wurde auf die bereits zuvor geschilderte Weise eine Abschätzung für die traditionell vornehmlich manuelle Programmierung vorgenommen und dann in Verhältnis gesetzt zu der für die Umsetzung mit PIKAS benötigten Zeit.

Wir haben dazu drei verschiedene Anforderungen untersucht, die wir kurz in aufsteigender Komplexität vorstellen wollen:

1. *Umsetzung einer Übersetzungspflegemaske*: Für alle im System pflegbaren Übersetzungstexte (siehe dazu Abschnitt 8.1.8.1.7 auf Seite 358) sollen die Übersetzungstexte zu den unterstützten Portalsprachen auch über die Datenbank pflegbar sein, so dass Anwender mit ausreichenden Rechten die Texte der Anwendungen anpassen können. Übersetzungen werden durch einen logischen Bezeichner identifiziert, der einem Tupel aus (Sprache, Land, Prozess, Seite) zugeordnet ist. Land, Prozess und Seite sind dabei optionale Angaben und das Laufzeitsystem sucht die konkrete Übersetzung für ein vorgegebenes Label gemäß entsprechender Defaulting-Regeln. Die Anwendung zur Übersetzungspflege soll aus einer Übersichtsmaske bestehen, in der alle existierenden Übersetzungen tabellarisch aufgelistet werden (inklusive Paging) und die Fil-

termöglichkeiten bereitstellt, um nach Label plus dem oben definierten Identifikationstupel über „enthalten in“-Suchen zu recherchieren. Durch Klick auf einen einzelnen Eintrag gelangt der Anwender auf eine Pflegemaske für den Eintrag, von der aus er seine Änderungen speichern, den Änderungsvorgang abbrechen und zur Übersichtsmaske zurückkehren kann. Zudem gibt es ausgehend von der Übersichtsmaske die Möglichkeit, neue Einträge anzulegen und bestehende Einträge zu löschen.

2. *Umsetzung eines Rechteadministrationsbereichs*: Es sind Administrationsmasken zu erstellen. Diese setzen sich aus einer Reihe von Master-Detail-Masken nach analogem Vorbild der Übersetzungspflege zusammen, um die folgenden Entitäten zu verwalten:

   a) User
   b) Rollen (Sammlungen von Rechten)
   c) Rechte

   Zudem gibt es eine Reihe von Zuordnungsprozessen, um diese Entitäten mit dem Laufzeitsystem zu verbinden:

   - Ein User kann beliebig viele Rollen zugeordnet bekommen. Analog können einer Rolle User zugeordnet werden.
   - Eine Rolle kann beliebig viele Rechte erhalten. Analog kann ein Recht auf Rollen verteilt werden.
   - Rechte können an Seiten des Portals vergeben werden (d. h. dass man das Recht hat, auf die entsprechende Seite zuzugreifen).
   - Rechte können an Prozessen des Portals vergeben werden (d. h. .dass man das Recht hat, auf den entsprechenden Prozess zuzugreifen).
   - Rechte können an Navigationseinträgen vergeben werden (d. h. dass der Rechteinhaber den Navigationseintrag sehen darf).
   - Rechte an Transitionen können gestrichen werden (d. h. dass es möglich ist, bestimmte Aktionen zu sperren).

   Außerdem ist ein Registrierungsprozess für Anwender zu implementieren, bei dem anonyme Anwender einen Systemzugang beantragen können, der dann durch den Administrator gewährt oder abgelehnt werden kann. Bei Gewährung wird automatisch ein neuer Account angelegt und dass dazugehörige Kennwort per E-Mail verschickt, bei Ablehnung erfolgt ebenfalls eine Benachrichtigung. Zudem erhält der anonyme Anwender eine E-Mail-Benachrichtigung darüber, dass sein Antrag eingegangen ist.

3. *Entwicklung eines Auktionsportals*: Hier geht es darum, eine komplexe Auktionssoftware nach Vorbild von Systemen wie `www.swoopo.de` oder `www.hammerdeal.de` zu implementieren. Wir verzichten auf die Angabe des (mehr als 60 Seiten starken) Pflichtenhefts und verweisen auf die entsprechenden Portale, um einen Eindruck von der Komplexität zu erlangen.

Für jedes der drei Szenarien wurden Aufwandswerte für eine klassische Implementierung (basierend auf technischen Frameworks) ermittelt. Im ersten Szenario liegen Erfahrungswerte aus einem älteren Projekt vor, bei dem ein Struts-ähnliches Controller-Framework basierend auf Servlets, JSPs und Tag Libraries eingesetzt wurde. Im zweiten Fall wurde ebenfalls eine Abschätzung basierend auf diesem Framework durchgeführt. Dazu wurden drei Entwickler herangezogen, einer davon hatte eine solche Anforderung bereits mit dem Framework selbst umgesetzt. Im dritten Szenario wurden Vergleichswerte herangezogen, die wir von einem Entwickler einer vergleichbaren Auktionssoftware erhalten haben und der eine technische Umsetzung basierend auf Spring Web MVC und Hibernate vorgenommen hat.

Zu kritisieren sind bei dieser Versuchsstellung die folgenden Faktoren:

- Bei der Umsetzung von bekannten Anforderungen auf Basis von PIKAS konnte nicht sichergestellt werden, dass der mit PIKAS arbeitende Entwickler wirklich die gleichen Vorkenntnisse und Vorerfahrungen wie der ursprüngliche Umsetzer hatte. Wir haben versucht, dem zu begegnen, indem wir Entwickler mit einer möglichst ähnlichen Vorgeschichte ausgewählt haben, sind uns aber des subjektiven Charakters dieser Entscheidung bewusst.
- Bei der Umsetzung neuer Anforderungen auf der Basis von PIKAS konnte keine tatsächliche Vergleichsimplementierung vorgenommen werden, so dass die Vergleichswerte tatsächlich subjektive Schätzwerte sind. Wir haben versucht, diesem Problem zu begegnen, indem wir zum einen den umsetzenden Entwickler selbst haben schätzen lassen und zum anderen zwischen drei und fünf Schätzwerte gemittelt haben (die zudem noch mit bekannten internen Aufwandsfaktoren multipliziert wurden, um die Abweichung zwischen Schätzung und tatsächlicher Umsetzungszeit basierend auf historischen Daten zu berücksichtigen).
- Die Erhebung der historischen Daten erfolgte auf vorliegenden Controlling-Daten, die geltenden Rahmenfaktoren wurden nicht im wissenschaftlichen Sinne präzisiert. Dies resultiert aus der Erhebung der Zahlen unabhängig von der Zielsetzung dieser Arbeit. Wir halten diese Einschränkung aber für akzeptabel, da auch alle anderen Wirtschaftlichkeitsberechnungen bei der QuinScape

GmbH auf diesen Zahlen basieren und die Zahlen somit eine konsistente Basis für alle Wirtschaftlichkeitsuntersuchungen bilden.

- Drei Szenarien sind eigentlich zu wenig für eine repräsentative Schätzung der Effizienz. Wir sehen aber eine gewisse Aussagekraft in den vorliegenden Schätzungen, da die drei Szenarien eine sehr unterschiedliche Komplexität (von 2 über 17 bis zu mehr als 100 Webseiten) haben und so sehr verschiedene Anforderungsprofile abdecken.
- Die Vergleichsaufwände beziehen sich auf bestimmte Frameworks und Technologien und die Frage, ob diese Technologien die beste Wahl darstellten, bleibt unbeantwortet. Da hinter allen drei Anforderungen kommerzielle Interessen standen, ist aber davon auszugehen, dass die jeweiligen Technologien nach bestem Wissen und Gewissen ausgewählt wurden.

Nach Umsetzung der drei Szenarien ergeben sich die in Abbildung 9.1 dargestellten Aufwände (alle Aufwände sind als Personentage dargestellt).

Szenario	Aufwand klassisch	Aufwand mit PIKAS	Aufwandsersparnis
1	ca. 4 PT	ca. 2,5 PT	ca. 62,5%
2	ca. 22 PT	ca. 13,5 PT	ca. 61,36%
3	ca. 475 PT	ca. 180 PT	ca. 62,1%

Abbildung 9.1: Aufwandsvergleich PIKAS zu klassischen Ansätzen

Um möglichst konservativ vorzugehen, verwenden wir die kleinste experimentell ermittelte Aufwandsersparnis und runden diese auf 60% ab. Das heißt im Umkehrschluss, dass wir erwarten, in PIKAS-basierten Projekten mit 40% weniger Aufwand als bei vergleichsweise klassischer (manueller) Programmierung auszukommen. Wir definieren den Rationalisierungsgrad für die weiteren Berechnungen also als $R_G = 0,6$.

9.1.2.3 Reduzierung von Migrationsaufwänden

Eins der Argumente für eine stark abstrahierende Beschreibung der domänenspezifischen Sachverhalte ist die Fähigkeit zur transparenten Evolution der darunterliegenden Technologien (siehe dazu auch unsere Ausführungen auf Seite 179). Die Stichhaltigkeit dieser Aussage konnten wir bereits sehr früh exemplarisch verifizieren, da sich durch frühes Kundenfeedback Zweifel an der sinnhaften Implementierung der View-Technologie in den Browser-Clients ergab. Diese Zweifel mündeten in einem vollständigen Strategiewechsel bei der Handhabung der View-Technologie.

Konkret haben wir die Präsentationsschicht von PIKAS vollständig ausgetauscht. Die initiale Version der Präsentationsschicht basierte auf Spring Web MVC (siehe [Spri09c]) und wurde im Zeitraum von Oktober 2007 bis April 2008 implementiert. Sie enthielt bereits funktionsreiche Eigenschaften wie vollständigen AJAX-Support, modulare Kompositions- und Austauschmöglichkeiten für Seitenelemente, usw. Auf Basis dieser Technologie sowie der zu dem Zeitpunkt vorhanden ViewModel- und ContentElementModel-Definitionen wurden eine Reihe beispielhafter Portalanwendungen mit insgesamt 18 ProcessModel-Instanzen und 102 ViewModel-Instanzen realisiert, die beispielhafte Datenpflegeanwendungen, eine Nutzerverwaltung und ähnliche Szenarien implementierten (des Weiteren gab es zu dem Zeitpunkt natürlich eine Vielzahl weiterer Modelle für die Domäne, die Navigation, usw.)[6]. Durch Kundenfeedback gewannen wir den Eindruck, dass eine Realisierung auf Basis von JavaServer Faces (JSF, siehe [JSFC09]) vorteilhafter wäre, da JSF ein wesentlich standardisierteres Komponentenmodell als Spring Web MVC bietet und hier zudem reichhaltigere Client-Bibliotheken verfügbar sind.

Innerhalb von sechs Wochen wurde daher im Zeitraum April bis Juni 2008 die Implementierung der Präsentationsschicht von Spring-Web-MVC auf JSF portiert. Beteiligt waren daran im Mittel 2,5 Entwickler, so dass ein Gesamtaufwand von ca. 75 Personentagen angefallen ist (6 Wochen $\times$ 5 Arbeitstage $\times$ 2,5 Personen). Die Schätzung für die händische Migration einer Anwendung im Umfang von 102 Seiten und 18 Prozessmodellen lag bei ca. 87 Personentagen ($102 \times 0,5$ Tage für die Migration der Masken plus 18×2 für die Migration der Workflows)[7]. Interessant sind zwei Beobachtungen:

1. Der Aufwand für die Migration der Generatortechnologie von der Spring-Web-MVC-Basis auf JSF hat keinen Zusammenhang zu der Menge der verfügbaren Anwendungen bzw. Anwendungsseiten, sondern basiert (in einem noch nicht genauer untersuchten Zusammenhang) auf den internen Artefakten des Generators[8]. Der Aufwand für die Migration der konkreten Anwendungsseiten verhält sich linear zur Menge der Seiten und Workflows. Selbst wenn die konkreten Faktoren je nach zu migrierenden Technologien variieren, wird sofort

[6] Wir haben uns zur Angabe der Seiten und Workflows als Komplexitätsmaß entschieden, da wir bei QuinScape diese Kennzahlen auch für interne Aufwandsabschätzungen verwenden und sich hier aus Erfahrungswerten ein starker und zuverlässiger Zusammenhang zu den Aufwänden für die Realisierung von Webanwendungen ergeben hat.

[7] Diese Faktoren basieren auf Erfahrungswerten aus der Firmengeschichte der QuinScape GmbH, also aus früheren Migrationsprojekten.

[8] Eine wesentliche Rolle wird hier die Zahl der Templates spielen, die angepasst werden müssen, um den Technologiewechsel zu realisieren. Zudem haben die internen Hilfsmethoden und -services zur Integration der technologiespezifischen Abhängigkeiten sicher einen großen Einfluss.

klar, dass der Aufwand mit der Anwendungsgröße steigt, vermutlich sogar überproportional, da sich die Testkomplexität mit zunehmender Anwendungsgröße deutlich erhöht.

2. Bei der Migration der Generator-Templates auf die neue Technologie musste kein einziges XML-Modell für die Anwendungen verändert werden. Alle Modelle konnten unverändert weiter verwendet werden.

Damit hat sich für uns gezeigt, dass die These von der Fähigkeit zur transparenten Technologie-Evolution verifiziert werden kann. Der Aufwand für die Migration wäre nur marginal gestiegen, wenn wesentlich mehr Anwendungsmodule zu portieren gewesen wären (aufgrund der Notwendigkeit, jede Anwendung zumindest einmal durchzutesten). Der klassische manuelle Ansatz skaliert sehr schlecht und führt insbesondere zu signifikant höherem Testaufwand. Letzteres begründet sich darin, dass bei der Migration der Generierungsstrategie an zentraler Stelle Änderungen vorgenommen werden, die sich überall gleichförmig auswirken, so dass Fehler sehr schnell beobachtbar sind, da sie überall wirken. Bei der manuellen Anpassung ist jedes einzelne Artefakt besonders intensiv zu testen, da jede Änderung lokal vorgenommen wird und hier große Unterschiede auftreten können.

Zusammenfassend zeigt sich also exemplarisch, dass die Verwendung technologieunabhängiger Modelle selbst bei kleineren Anwendungsszenarien bereits signifikante Vorteile mit sich bringt, die mit zunehmender Zahl oder Komplexität der betroffenen Anwendungen drastisch an Mehrwert gewinnen.

9.1.2.4 Steigerung der Umsetzungsqualität

In diesem Abschnitt wollen wir auf subjektive Erfahrungen in der beobachteten Umsetzungsqualität von PIKAS-basierten Lösungen im Vergleich zu klassisch entwickelten Anwendungen berichten. Wir haben diese nicht wissenschaftlich ausgewertet, da solche Untersuchungen den Rahmen der Arbeit gesprengt hätten – sie sind aber ein sehr interessantes und potenzialträchtiges Thema für weitere Forschungsarbeiten, auf die wir in Kapitel 10.3 zurückkommen werden. Wir konnten die folgenden positiven Auswirkungen auf Grundlage des von uns entwickelten Prototypen beobachten, die perspektivisch genauer untersucht werden müssen:

- CP-basierte Anwendungen haben eine durchgängig hohe Qualität. Dies liegt darin begründet, dass der Generator die domänenspezifischen typischen Komfortmerkmale einer Anwendung berücksichtigen und automatisch generieren

kann. Betrachtet man das Beispiel „Webanwendungen“, so sieht der klassische Entwicklungszyklus üblicherweise so aus, dass ein Entwickler zunächst die grundlegende Maske realisiert, mit realen Daten füllt und dann die auslösbaren Aktionen implementiert. Erst dann werden normalerweise als weniger wichtig erachtete Aspekte wie Layout, Usability, Komfortfunktionen, Zugriffsgeschwindigkeiten, etc. angegangen. Da sich dieser Zyklus bei klassischer Vorgehensweise normalerweise für jeden Workflow oder sogar jede Maske wiederholt, besteht die hohe Wahrscheinlichkeit, dass sich fast jeder Workflow bzw. jede Maske einer größeren Anwendung in Details von anderen Workflows und Masken unterscheidet. Teilweise liegt dies darin begründet, dass unterschiedliche Entwickler unterschiedliche Herangehensweisen pflegen (also Workflows oder Usability-Fragen durch unterschiedliche Systematiken gelöst werden), teilweise liegt dies in der menschlichen Natur begründet, die Fehler zulässt (so könnten zum Beispiel Tooltips nur in einigen Masken vorhanden sein, da sie in anderen schlicht vergessen wurden). Ein Generator-basierter Ansatz vermeidet viele dieser Fehlerquellen, da der Generator die immer gleichen Regeln und Templates zur Erzeugung der Lösung verwendet, was eine gleichbleibende Qualität sicherstellt.

- CP-basierte Anwendungen haben vom ersten Augenblick des Entstehens an eine sehr hohe Qualität. Auch dieser Punkt ergibt sich aus den Ausführungen des vorausgegangenen Punktes. Dort haben wir den typischen Entwicklungszyklus bei manueller Programmierung beschrieben, der iterativ zu einer vollständig implementierten und reichhaltigen Maske bzw. entsprechenden Workflows führt. Bei einem Generator-basierten Ansatz werden immer sofort auch alle „Komfort-Features“ in die generierte Lösung integriert, so dass sich selbst prototypisch realisierte Masken bereits mit sehr hohem Komfort ausgestattet zeigen und die Außenwirkung bereits in frühem Stadium sehr beeindruckend ist.

- CP-basierte Anwendungen erfordern wesentlich weniger Testaktivitäten. Diese konzentrieren sich wesentlich stärker auf die individuelle Businesslogik als auf die Standardinfrastruktur, die bereits durch Tests des Generatorwerkzeugs abgesichert werden.

Diese Punkte haben wir aber nicht weiter formell untersucht und sehen hier wichtige Ansatzpunkte für nachfolgende Forschungsarbeiten.

9.1.3 Bewertung des Kosten-Nutzen-Verhältnisses

Zur Berechnung des tatsächlichen Kosten-Nutzenverhältnisses treffen wir die folgenden Annahmen:

- Wie bereits erläutert verwenden wir ausschließlich die Zahlen zur Steigerung der Umsetzungsgeschwindigkeit, da diese am präzisesten gemessen werden konnten.
- Alle neu eingestellten Mitarbeiter arbeiten zukünftig mit dem CMDSD-Werkzeug. Andernfalls erscheint die strategische Entwicklung eines solchen Werkzeugs fragwürdig. Seltene Ausnahmen zwecks Bestandspflege, etc. vernachlässigen wir hier.
- Die Altmitarbeiter werden nur $W_{P_A} = 0,75$ (100% Arbeitszeit = 1) ihrer Arbeitszeit damit verbringen, Lösungen mit dem neuen Werkzeug umzusetzen. Die verbleibende Zeit ziehen wir für anderweitige Weiterbildungen, die Pflege von Altsystemen, usw. ab.
- Wir gehen von einem durchschnittlichen Auslastungsgrad von $W_P = 0,75$ für alle Mitarbeiter aus.
- Wir vernachlässigen die Einsparung für Weiterbildung, weil wir zum einen eine möglichst konservative Einschätzung erhalten wollen und zum anderen dadurch direkt der Kritik begegnen, dass hohe Teile an Weiterbildung heutzutage direkt in IT-Projekten erfolgen, also als „Training on the Job“.

Zunächst einmal ergeben sich T_P produktive Arbeitstage pro Jahr als Produkt aus 12 Monaten, T_m Arbeitstagen pro Monat und einer Auslastung von W_P zu:

$$\begin{aligned} T_P &= 12 \times T_m \times W_P && (9.5) \\ &= 12 \times 18 \times 0,75 \\ &= 162 && (9.6) \end{aligned}$$

An 162 Arbeitstagen pro Jahr können die Mitarbeiter also projektbezogene Umsätze generieren. Die Jahresgesamtkosten $K_{M_{\text{Jahr}}}$ (Arbeitgeberbrutto) eines Mitarbeiters lassen sich abschätzen über

$$\begin{aligned} K_{M_{\text{Jahr, klassisch}}} &= K_M \times T_m \times 12 \\ &= 350 \times 18 \times 12 \\ &= 75600 \end{aligned}$$

Ein Mitarbeiter erzeugt im Durchschnitt also jährliche Kosten (Arbeitgeberbrutto) in Höhe von ca. 75.600 EUR.

Diesen Kosten sind nun die Einnahmen durch Projekte gegenüberzustellen. Sei N_{T_P} der durchschnittliche Tagessatz pro fakturiertem Projekttag. Den *Erlös pro Mitarbeiter und Jahr* $N_{M_{\text{Jahr}}}$ erhalten wir somit zusammen mit T_P aus Gleichung 9.5 als:

$$N_{M_{\text{Jahr, klassisch}}} = T_P \times N_{T_P} - K_{M_{\text{Jahr}}} \tag{9.7}$$

Setzen wir Umsätze in Höhe von $N_{T_P} = 780$ EUR pro Tag[9] sowie den errechneten konkreten Wert für T_P aus Berechnung 9.6 an, so ergibt sich

$$\begin{aligned} N_{M_{\text{Jahr, klassisch}}} &= T_P \times N_{T_P} - K_{M_{\text{Jahr}}} \\ &= 162 \times 780 - 75600 \\ &= 50760 \end{aligned}$$

Bei einfacher Betrachtung würden sich also Erlöse in Höhe von 50.760 EUR pro Mitarbeiter und Jahr ergeben. Wir wollen aber genauer hinschauen und berücksichtigen, dass Projekte auf Abschätzungen beruhen und Abschätzungen eine gewisse Ungenauigkeit aufweisen, so dass der interne Kostensatz mit einem Aufschlag zu versehen ist. Diesen Risikofaktor R_{T_P} wollen wir in Gleichung 9.7 berücksichtigen – faktisch reduziert er die Zahl der fakturierbaren Projekttage aufgrund real existierender Abschätzungsfehler zu Ungunsten des Dienstleister, so dass wir den *risikobereinigten Erlös pro Mitarbeiter und Jahr* $N^!_{M_{\text{Jahr, klassisch}}}$ erhalten als

$$N^!_{M_{\text{Jahr, klassisch}}} = \frac{T_P \times N_{T_P}}{R_{T_P}} - K_{M_{\text{Jahr}}} \tag{9.8}$$

Nehmen wir an, dass die durchschnittliche Abschätzung eines Projektaufwandes 20% zu Ungunsten des Dienstleisters ausfällt und gehen wir ausschließlich von Festpreisprojekten aus, so erhalten wir $R_{T_P} = 1,2$ und damit einen risikobereinigten

[9] Dieser angenommene Tagessatz ergibt sich aus Erfahrungswerten der QuinScape. Für „Standardaufgaben“ (z. B. einfache Entwicklungsarbeiten scheinen gegenwärtig ca. 600,00 EUR Tagessatz zu erzielen zu sein, für Spezialaufgaben und Experten, die über nachhaltiges Produktwissen verfügen, sind häufig 960,00 EUR erzielbar. Natürlich weichen konkrete Tagessätze in alle Richtungen ab, diese Grenzwerte haben sich aber bewährt. Daher haben wir den Mittelwert $(600 + 960)/2 = 780$ angenommen.

Erlös pro Mitarbeiter und Jahr in Höhe von[10]:

$$\begin{aligned} N^{!}_{M_{\text{Jahr, klassisch}}} &= \frac{T_P \times N_{T_P}}{R_{T_P}} - K_{M_{\text{Jahr}}} \\ &= \frac{162 \times 780}{1,2} - 75600 \\ &= 29700 \end{aligned} \tag{9.9}$$

Eine 20%-ige Ungenauigkeit zu Lasten des Dienstleisters in den Abschätzungen führt also bereits fast zu einer Halbierung der Erlöse pro Mitarbeiter und Jahr. Wir wollen diesen (konservativ abgeschätzten) Erlöswert $N^{!}_{M_{\text{Jahr, klassisch}}} = 29.700$ für unsere weiteren Betrachtungen verwenden.

Der *risikobereinigte Jahreserlös für den klassischen Ansatz* ergibt sich damit in Abhängigkeit von der Zahl der in Projekten arbeitenden Mitarbeiter M_P zu:

$$N^{!}_{\text{Jahr, klassisch}} = N^{!}_{M_{\text{Jahr, klassisch}}} \times M_P \tag{9.10}$$

Betrachten wir nun die Veränderung der Zahlen, wenn Projekte fortan mit PIKAS durchgeführt werden. Zunächst einmal müssen wir berücksichtigen, dass die Verwendung von PIKAS auch fortlaufende Zusatzkosten für die Pflege und Weiterentwicklung des Systems erzeugen wird. Wir gehen davon aus, dass zukünftig M_{PB} Personen projektbegleitend damit beschäftigt sein werden, PIKAS weiterzuentwickeln. Daraus ergeben sich *projektbegleitende Kosten* $K_{\text{PB}_{\text{Jahr}}}$ pro Jahr in Höhe von:

$$K_{\text{PB}_{\text{Jahr}}} = M_{\text{PB}} \times K_{M_{\text{Jahr}}} \tag{9.11}$$

Für unseren konkreten Fall schätzen wir, dass fortlaufend zwei Entwickler mit der Weiterentwicklung von PIKAS beschäftigt sein werden (also $M_{\text{PB}} = 2$ gilt). Somit ergeben sich projektbegleitende jährliche Kosten in Höhe von

$$\begin{aligned} K_{\text{PB}_{\text{Jahr}}} &= 2 \times 75600 \\ &= 151200 \end{aligned}$$

Diese Kosten in Höhe von 151.200 EUR sind also vom Jahresertrag auf Basis von PIKAS abzuziehen, um den Gesamterlös auf Basis von PIKAS zu berechnen.

[10] Wir kommen außerdem zu der Einschätzung, dass bei einer solchen *durchschnittlichen* Ungenauigkeit *zu Lasten* des Dienstleisters dringend ein besseres Schätzverfahren anzuwenden ist.

Greifen wir nun auf den auf Seite 385 ermittelten Rationalisierungsfaktor R_G durch die Einführung von PIKAS in Höhe von $R_G = 0,6$ auf. Ferner nehmen wir an, dass dieser Faktor mit einem bestimmten prozentualen Wirkgrad W_{R_G} auch in Mehrgeschäft umgesetzt werden kann. Damit ergibt sich als Erweiterung von Gleichung 9.8 der risikobereinigte Erlös pro Jahr und Mitarbeiter auf Basis von PIKAS als:

$$N^{!}_{M_{\text{Jahr, CP}}} = \frac{T_P \times N_{T_P}}{R_{T_P} \times (1 - (R_G \times \frac{W_{R_G}}{100}))} - K_{M_{\text{Jahr}}} \tag{9.12}$$

Mit den oben genannten Zahlen ergibt sich damit konkret (zunächst bei einer Annahme von $W_{R_G} = 100$):

$$\begin{aligned} N^{!}_{M_{\text{Jahr, CP}}} &= \frac{T_P \times N_{T_P}}{R_{T_P} \times (1 - (R_G \times \frac{W_{R_G}}{100}))} - 75600 \\ &= \frac{162 \times 780}{1,2 \times (1 - (0,6 \times \frac{100}{100}))} - 75600 \\ &= \frac{126360}{0,48} - 75600 \\ &= 187650 \end{aligned}$$

Der plötzliche Erlösanstieg pro Mitarbeiter (immerhin das 6,3-fache des Ergebnisses aus Berechnung 9.9!) ist sicher mit großer Vorsicht zu genießen, sind wir doch dabei davon ausgegangen, dass die Mitarbeiter weiterhin zu 100% ausgelastet werden können und zudem die 2,5fache Menge an Beauftragungen eingeworben werden konnte. Insbesondere Letzteres ist – ohne Zunahme von Kosten aufgrund erhöhter Vertriebsaufwändungen – eine hochgradig fragwürdige Annahme. Gehen wir von einem eher pessimistischen Szenario aus, bei dem sich der Wirkgrad gerade mal bei $W_{R_G} = 30\%$ bewegt, erhalten wir

$$\begin{aligned} N^{!}_{M_{\text{Jahr, CP}}} &= \frac{162 \times 780}{1,2 \times (1 - (0,6 \times \frac{30}{100}))} - 75600 \\ &= \frac{126360}{0,984} - 75600 \\ &\approx 52815 \end{aligned}$$

Auch hier haben wir immer noch einen Erlöszuwachs auf das ca. 1,8-fache. Wir erachten trotz der inhärenten Unsicherheit bei dieser Gesamtabschätzung die Annahme eines Wirkgrades von 30% bei ca. 60% mehr zur Verfügung stehender Zeit aufgrund

gewonnener Effizienz als plausibel.

Betrachten wir nun wieder die Jahressicht. Der *risikobereinigte Jahreserlös für den CP-basierten Ansatz* ergibt sich damit in Abhängigkeit von der Zahl der in Projekten arbeitenden Mitarbeiter M_P analog zu Gleichung 9.10 als:

$$N^{!}_{\text{Jahr, CP}} = N^{!}_{M_{\text{Jahr, CP}}} \times M_P - K_{\text{PB}_{\text{Jahr}}} \tag{9.13}$$

Um schlussendlich den Mehrerlös $N_{\text{Jahr, Umstieg auf CP}}$ zu berechnen, kombinieren wir die Gleichungen 9.10 und 9.13 zu:

$$\begin{aligned} N^{!}_{\text{Jahr, Umstieg auf CP}} &= N^{!}_{\text{Jahr, CP}} - N^{!}_{\text{Jahr, klassisch}} \\ &= (N^{!}_{M_{\text{Jahr, CP}}} - N^{!}_{M_{\text{Jahr, klassisch}}}) \times M_P - K_{\text{PB}_{\text{Jahr}}} \qquad (9.14) \\ & \qquad (9.15) \end{aligned}$$

Konkret ergibt sich für unseren PIKAS-Prototypen:

$$\begin{aligned} N^{!}_{\text{Jahr, Umstieg auf CP}} &= (N^{!}_{M_{\text{Jahr, CP}}} - N^{!}_{M_{\text{Jahr, klassisch}}}) \times M_P - K_{\text{PB}_{\text{Jahr}}} \\ &= (52815 - 29700) \times M_P - 151200 \\ &= 23115 \times M_P - 151200 \qquad (9.16) \\ & \qquad (9.17) \end{aligned}$$

Der Zeitraum für die Erreichung des Return on Invest D_{ROI} ergibt sich damit als:

$$D_{\text{ROI}} = \frac{K_E}{N^{!}_{\text{Jahr, Umstieg auf CP}}} \tag{9.18}$$

Für PIKAS bedeutet dies konkret bei $M_P = 14$:

$$\begin{aligned} D_{\text{ROI}} &= \frac{K_E}{N^{!}_{\text{Jahr, Umstieg auf CP}}} \\ &= \frac{346500}{23115 \times 14 - 151200} \\ &\approx 2,01 \qquad (9.19) \end{aligned}$$

Nach knapp zwei Jahren können die Investitionen also wieder eingeholt werden und ab diesem für IT-Investitionen sehr häufig angesetzten Zeitraum zur Erreichung des ROI stellen sich nachhaltige Gewinne ein. Zudem hat sich der Unternehmenswert des IT-Dienstleisters auch noch dadurch erhöht, dass nachhaltige Investitionen (bis zu diesem Zeitpunkt in Höhe von ca. $K_E + 2 \times 151200 = 346500 + 2 \times 151200 = 648900$)

in ein hauseigenes Produkt durchgeführt und mit dem durch die auf dem Produkt basierenden Projekte einhergehenden langsamen aber stetigen Markenaufbau aller Wahrscheinlichkeit nach zusätzliche Werte geschaffen wurden.

Zuguterletzt kann man natürlich durch triviale Umformungen auch noch eine Formel angeben, um zu ermitteln, wie viele Mitarbeiter ein mittelständische IT-Unternehmen auf Basis des CP-basierten Ansatzes zu beschäftigen muss, damit sich der Ansatz rechnet. Dazu kombinieren wir die Gleichungen 9.14 und 9.18 und definieren als Konstante $D_{\text{ROI}} = 2$, um binnen zwei Jahren alle Investitionen wieder einspielen zu können. Es ergibt sich:

$$\begin{aligned} & D_{\text{ROI}} && \stackrel{!}{\geq} \frac{K_E}{N^!_{\text{Jahr, Umstieg auf CP}}} \\ \Leftrightarrow \quad & 2 && \stackrel{!}{\geq} \frac{K_E}{(N^!_{M_{\text{Jahr, CP}}} - N^!_{M_{\text{Jahr, klassisch}}}) \times M_P - K_{\text{PB}_{\text{Jahr}}}} \\ \Leftrightarrow \quad & M_P && \stackrel{!}{\geq} \left(\frac{1}{(N^!_{M_{\text{Jahr, CP}}} - N^!_{M_{\text{Jahr, klassisch}}})} \right) \times \left(\frac{K_E}{2} + K_{\text{PB}_{\text{Jahr}}} \right) \end{aligned} \tag{9.20}$$

Abschließend sei angemerkt, dass wir zudem eine Reihe von wesentlichen Faktoren außer Acht gelassen haben, die sich durchweg positiv auf das Kosten-Nutzen-Verhältnis auswirken könnten:

- Eine Annahme für den Wunsch nach der Einführung eines CP-basierten Systems war, dass es auf Kundenseite einen großen Bedarf an entsprechenden Lösungen gibt. Somit ist es bei korrekter Einschätzung durch den Ersteller der CP-Lösung sogar möglich, dass nicht nur 100% der Rationalisierungseffekte konkret umgesetzt und ein entsprechender Mehrerlös erzielt werden kann, sondern es besteht sogar je nach Branche und Umfeld die Möglichkeit, eine wesentlich höhere Zahl von Neukunden als bei klassischer Umsetzung des Geschäftsmodells zu finden, was sogar noch zu einem Wachstum in der Mitarbeiterzahl und damit in Umsatz und Erlösen führen kann. M. a. W.: In diesem Szenario würden sich die initialen Umsetzungskosten sogar noch viel schneller einspielen. Wir lassen diesen Faktor aber gänzlich außer acht, da neuen Kunden auch in den meisten Fällen zusätzliche Vertriebsaufwände entgegenstehen, die eine stichhaltige detailliertere Bewertung dieses Faktors sehr schwierig machen.
- Wir haben in Teil I der Arbeit ausgeführt, dass CP-basierte Lösungsansätze voraussetzen, dass der Ersteller einer solchen Lösung umfangreiche Vorerfahrungen benötigt, um mit angemessenem Aufwand zu einer nutzbringenden

Lösung zu gelangen. Umfangreiche Vorerfahrungen können üblicherweise aber nur durch praktische Projekte erlangt werden, was wiederum impliziert, dass der Dienstleister die Möglichkeit hat, die CP-Entwicklungsarbeiten durch laufende und neue Projekte querzusubventionieren[11]. Durch das von uns aufgestellte Formelwerk wird es anderen IT-Dienstleistern in vergleichbaren Situationen möglich, abzuwägen, welche Voraussetzungen zu schaffen sind, um die CP-Entwicklungsarbeiten möglich zu machen.

- Bei der Alternative zur Nutzung von CP sind hausinterne Frameworks zu berücksichtigen. Diese entstehen üblicherweise, um die Entwicklungsgeschwindigkeit zu steigern und Projekte nach Best Practices umzusetzen. Somit könnte man auf den ersten Blick annehmen, dass unsere Betrachtungsweise dahingehend „unfair“ ist, dass wir die Effizienzsteigerung durch solche Frameworks ignoriert haben und somit den CP-Nutzen mit einem zu geringen klassischen Nutzen vergleichen. Wir nehmen eine andere Position aus und halten unsere Form der Abschätzung sogar für konservativ. Dies begründet sich wie folgt:
  - Auch die Entwicklung von hausinternen Frameworks erzeugt Kosten. Diese Kosten haben wir beim klassischen Ansatz mit 0 angesetzt, so dass wir hier von einem maximal ungünstigen Verhältnis zu Lasten von CP ausgehen.
  - Auch CP-basierte Ansätze können durch Best Practices, etc. erweitert werden. So ist beispielsweise bei unserem PIKAS-Prototypen absehbar, dass hausinterne Sammlungen von Aktionsmodellen, usw. entstehen, die projektspezifische Anforderungen und typische Aufgaben in domänenzentrierten Portalen effizienter abbilden. Somit kann man annehmen, dass CP und klassische Ansätze in etwa die gleiche Leistungsfähigkeit besitzen, soweit die Optimierung durch hausinterne Frameworks betrachtet wird.
  - Hausinterne Frameworks sind meist sehr direkt mit bestimmten Basistechnologien verbunden. In allen uns bekannten Fällen sind sie Erweiterungen

[11] Tatsächlich ist dies zum Beispiel der Fall für den bei QuinScape entwickelten Prototypen. Diese wurde erst aufgrund einer dafür geschaffenen Partnerschaft mit der BALVI GmbH möglich, die umsatzwirksame Unterstützungsleistungen zur Entwicklung beigetragen hat. Die Umsätze konnten nur aufgrund der geplanten Entwicklung realisiert werden, sind also tatsächlich unmittelbar auf das geplante CP-Werkzeug zurückzuführen und wären ohne selbiges nicht erzielt worden. Real bedeutet dies, dass die tatsächlich angefallenen Entwicklungskosten K_E weniger als ein Drittel der oben berechneten Summe betrugen und sich die Kosten somit sehr viel schneller als binnen zwei Jahren und insbesondere mit deutlich geringerem Risiko für alle Beteiligten wieder einspielen ließen.

einer spezifischen Technologie[12]. Damit fehlt aber gerade die Technologie-Abstraktion, die eine wesentliche Stärke von CP ist, so dass nicht nur die Weiterbildungskosten aufgrund der erforderlichen technologischen Detailkenntnisse (wie bereits in Abschnitt 9.1.2.1 erläutert) höher sind, sondern auch regelmäßig zusätzliche Aufwände für die Migration auf neue Technologiestufen zu berücksichtigen sind. Diese werden üblicherweise drastisch höher sein, wenn auch Projekte mitmigriert werden, da aufgrund der mangelnden Abstraktion jede einzelne Benutzungsstelle migriert werden muss. Faktisch führt dies unserer Erfahrung nach dazu, dass Inhouse-Frameworks sehr schnell altern, da entsprechende technologische Sprünge kaum möglich sind und daher nach einer initialen Beschleunigung der Entwicklungsgeschwindigkeit mit steigenden Kundenanforderungen sogar eher ein Effizienzverlust einhergeht, da die Frameworks sich sehr schwer damit tun, neueste technologische Errungenschaften angemessen abzubilden[13].

Auch einige weitere Kritikpunkte wollen wir nicht außer Acht lassen: So sind die Werte für die Altmitarbeiter anhand historischer Daten aus der QuinScape-Geschichte geschätzt, die neuen Werte wurden anhand drei betroffener Neueinstellungen sowie dreier Altmitarbeiter, die über den Zeitraum bereits mit PIKAS arbeiten konnten, ermittelt. Natürlich ist diese Stichprobe sehr klein und theoretisch von vielen Einflussfaktoren (Qualität der Mitarbeiter, Vorerfahrungen, Spezialkenntnisse, Lernbereitschaft, usw.) abhängig. Andererseits werden auch andere Unternehmen hier nur in seltensten Fällen genauere Vorinformationen für eine Abschätzung haben, so dass uns diese Werte für eine Abschätzung ausreichend scheinen. Zudem beinhalten die Berechnungen eine sehr große Zahl von subjektiven Abschätzungen, was die Stichhaltigkeit fragwürdig erscheinen lässt. Andererseits sind die Berechnungen immer noch vergleichsweise einfach und geradlinig, so dass es einem Entscheider möglich ist, an jeder Stelle die Auswirkungen unsicherer Abschätzungen zu bewerten und eine (wenn auch subjektive) Risikoeinschätzung vorzunehmen.

[12] Dies muss natürlich nicht so zu sein. Da viele hausinterne Frameworks aber ohne explizite Budgets projektbegleitend und damit initial projekterfolgsmindernd entwickelt werden, sind die Entwickler dieser Frameworks aber fast immer gezwungen, den direktesten Weg zu wählen, d. h. es verbleibt kein Raum für die Abstraktionen, die CP vorschreibt.

[13] Dies scheint häufig dazu zu führen, dass alternative Ansätze in die Frameworks aufgenommen werden, um den Alterungsprozess zu verlangsamen. Auch dies wirkt aber nur kurzfristig, da umgekehrt die Weiterentwicklungs- und Einarbeitungsaufwände für das Framework weiter steigen, was bei einer konstanten Ressourcenverfügbarkeit wieder zu einer Verlangsamung der Entwicklungsgeschwindigkeit und auch damit zu einem Alterungsprozess führt.

Insgesamt ergeben sich aus unserer Sicht aus den vorausgegangenen Ausführungen und Beschreibungen zwei wesentliche Erkenntnisse:

1. Es lässt sich ein Formelwerk angeben, um Kosten und Nutzen geplanter CP-Entwicklungen abzuschätzen. Wesentlich sind hier die Gleichung 9.14 zur Bewertung des zu erwartenden Nutzens, Gleichung 9.18 zur Berechnung des Zeitraums, nach dem die Vorleistungen wieder eingespielt sind und schließlich Gleichung 9.20, um zu prüfen, ob der IT-Dienstleister überhaupt die notwendige Größe besitzt, die eine entsprechende CP-Entwicklung erst wirtschaftlich möglich macht. Viele der Einflussfaktoren beruhen zwar auf Schätzwerten, positiv ist aber, dass diese Schätzwerte in vielen Fällen systematisch gefestigt werden können durch Sammlung und Auswertung von Vergangenheitswerten. Insbesondere bei einer perspektivisch geplanten CP-Entwicklung kann so eine möglichst solide Grundlage für die Entscheidung über die Entwicklungsarbeiten getroffen werden.
2. Eine kostendeckende und sogar gewinnbringende Entwicklung von CP-Entwicklungsmethoden ist möglich. Die von uns aus der Entwicklung unseres Prototypen abgeleiteten Zahlen belegen dies eindrucksvoll, insbesondere da die realen Kosten deutlich geringer sind als die exemplarisch durchgerechneten Werte (40% Effizienzgewinn in Projekten, ROI nach maximal zwei Jahren).

9.2 Ergebnisse der CDPM-Nutzung

Aufgrund der noch nicht implementierten grafischen Editoren konnten wir die Auswirkungen unserer CDPM-Methode noch nicht unter Laborbedingungen messen. Wir haben aber für die drei in Abschnitt 9.1.2.2 beschriebenen Szenarien versucht, einen realen Versuchsaufbau nachzustellen. Dazu wurden die erforderlichen grafischen Editoren durch existierende und vergleichbare Werkzeuge ersetzt, die annähernd die Funktionalität liefern, die wir uns für die PIKAS-Editoren vorstellen:

- Navigationsmodelle wurden aufgrund des Open-Source-Werkzeugs *FreeMind* bzw. der kommerziellen Anwendung *MindManager* (siehe [Free09b, Mind06]) erstellt.
- Prozessmodelle wurden über *Microsoft Visio* und vereinfachte UML-Aktivitätsdiagramme visualisiert (siehe [Micro09b]).
- Viewmodelle wurden über den Anwendungsdesigner von *Intrexx* erstellt (siehe [Intr09]).

- Das Domänenmodell wurde wieder über *Microsoft Visio* und vereinfachte Klassendiagramme visualisiert.

Damit nähern wir die grafische Erstellung an die von uns in Kapitel 8 beschriebene Darstellungsweise an und können zumindest vergleichbare Effekte während des Designs erzielen. Als nachteilig erweist sich natürlich, dass die von uns geforderte Rapid-Prototyping-Arbeitsweise nicht generativ möglich ist. Die so erzeugten Modelle können zwar 1:1 von einem Entwickler direkt in XML-Modelle übersetzt werden, aus denen dann auch eine lauffähige Anwendung generiert wird – der Vorgang funktioniert aber nicht auf Knopfdruck. Eine Ausnahme bilden die Viewmodelle, aus denen mit Intrexx bereits auf Knopfdruck (allerdings in der Intrexx-Implementierung) lauffähige Anwendungen erzeugt werden können. Trotzdem ließen sich bereits von uns angestrebte Effekte in der Zusammenarbeit mit Domänenexperten auf Kundenseite beobachten: Der Kunde konnte gemeinsam mit einem CMDSD-Experten in extrem kurzen und iterativen Zyklen seine Anforderungen an die Bedienoberfläche von Systemen spezifizieren und das schnelle Feedback sowie die Möglichkeit zum Ausprobieren spezifizierter Anforderungen halfen in hohem Maße dabei, schnell zu einer guten Lösung zu kommen. Kunden merkten außerdem an, dass es ein großer Vorteil ist, mit vergleichsweise geringem Aufwand verschiedene Alternativen ausprobieren zu können, um sich dann für die Sinnvollste zu entscheiden – eine Vorgehensweise, die bei manueller Programmierung kaum möglich ist. Zudem wurde als positiv empfunden, dass die Modelle in technischer Hinsicht sehr abstrakt sind und daher kaum technische Fragen entstehen. Stattdessen könne man sich sehr auf die Fachlichkeit fokussieren, was als durchweg positiv wahrgenommen wurde.

Insgesamt zeigt sich hier aber noch weiterer Bedarf an einer ausführlicheren Evaluierung von CDPM – die oben beschriebenen ersten Erfahrungswerte deuten an, dass sich der Prozess der Anforderungserhebung tatsächlich in dem von uns angestrebten Maß beschleunigen und in seiner Qualität verbessern lässt, uns liegen aber noch keine verlässlichen Zahlenwerte vor.

9.3 Weitere Projekte auf Basis des CP-Konzepts

In den folgenden Abschnitten werden eine Reihe von Projekten kurz vorgestellt, die aufgrund ihres CP-basierten Potenzials als förderungswürdig eingestuft wurden bzw. mit Industriepartnern als Kooperationsprojekte durchgeführt werden und aktuell laufen.

9.3.1 WODOMI

WODOMI ist ein Tempus-gefördertes EU-Projekt (Referenzkennzeichen *Tempus CARDS JEP-40045-2005*). Der Name steht für *Workflow and Document Management Infrastructure towards Integrated University* und das Ziel des Projekts ist (siehe [Wodo09]) *„to establish a sustainable web based network system to be used as an interoperable environment for University workflow and document management and will lead towards implementation of "integrated University" concept and goals recommended by Bologna process. The sustainable web based network system will enable equal opportunities and multilingual approach. The appropriate ICT infrastructure will enable transparent and easy access to all University workflow activities and documents."* Die folgenden Partner sind an dem Projekt beteiligt:

- Technische Universität Dortmund, Dortmund, Deutschland (Contractor institution)
- University Sts Cyril and Methodius, Faculty of Natural Sciences & Math., Skopje, Mazedonien (Coordinating institution)
- Technische Universität Graz, Graz, Österreich (Partner institution)
- QuinScape GmbH, Dortmund, Deutschland (Partner institution, Projektverantwortung und Mitglied des Steering Committees: Thomas Biskup)
- University „St Clement Ohridski", Faculty of Technical Sciences, Bitola, Mazedonien (Partner institution)
- South Eastern Europe University of Tetovo, Communication Sciences and Technologies, Tetovo, Mazedonien (Partner institution)

Das Projekt hat eine Laufzeit von September 2006 bis August 2009, Anfang Mai 2009 wurde aber eine Verlängerung um zwei Monate beschlossen, um aufgrund politisch bedingter Verzögerungen auf mazedonischer Seite den Projekterfolg doch noch herbeiführen zu können.

Im Rahmen des Projekts war eine Reihe von Workflows zu modellieren, die von zentraler Bedeutung sind, um ein einheitliches Verwaltungsmodell für die mazedonischen Universitäten zu schaffen. Das Verwaltungsmodell ist die Grundlage für die Umsetzung der von der EU im Vertrag von Bologna geforderten Strukturanpassungen, die wiederum Voraussetzung sind, damit Mazedonien die Aussicht auf eine EU-Mitgliedschaft bewahren kann. Bereits früh im Projekt wurde der Entschluss gefasst, das Projekt auf Basis von CP umzusetzen. Der besondere Schwerpunkt lag dabei auf

der Anwendung der CMDSD-Methoden, aber auch der CDPM-Ansatz konnte eingebracht werden. Grundlage für die technische Abbildung der Anforderungen ist ein PIKAS-ähnliches System, dass sich aber stärker auf die Bedürfnisse des universitären Kontextes fokussiert. PIKAS selbst kam für das Projekt aus mehreren Gründen nicht in Frage:

- Das Projekt hat früher begonnen und die rechtzeitige Verfügbarkeit von PIKAS konnte nicht sichergestellt werden.
- Das WODOMI-Projekt benötigt einen eingeschränkteren Funktionsumfang, dafür ist die Verfügbarkeit grafischer Editoren von essentieller Bedeutung, um gemäß unserer CDPM-Vorgehensweise frühzeitig in die gemeinsame Modellierung der Workflows mit den Fachanwendern einzusteigen.
- Die mazedonischen Universitäten sind auf Microsoft-Technologien festgelegt. QuinScape setzt mit PIKAS auf Java-basierte Technologien.

Dies führte dazu, dass die Ideen und Erfahrungswerte des CP-Ansatzes als Leistung von QuinScape in das Projekt eingeflossen sind und die mazedonischen Partner auf dieser Grundlage eine eigene Lösung umgesetzt haben mit dem Ziel, ein CMDSD-Werkzeug (den sogenannten PortalBuilder) zu schaffen, das von allgemeiner Relevanz für die Optimierung universitärer Prozesse sein kann. Eine grundlegende Version des Systems wurde von einem beauftragten mazedonischen Unternehmen entwickelt und könnte für einen hohen fünfstelligen Euro-Betrag realisiert werden, inklusive grafischer Pflegewerkzeuge (siehe [Obje09]). Das Projekt selbst ist noch nicht abgeschlossen. Stand Mai 2009 hat die mazedonische Regierung einen Kurswechsel beschlossen, der nun eine landesweite neue Lösung basierend auf einem Dokumentenmanagementsystem vorsieht. Die mit dem Portal Builder konzipierten Workflows und Prozesse konnten in sehr kurzer Zeit mit den beteiligten Universitäten gemeinsam erstellt werden, was um so erstaunlicher ist, da hier eine sehr schwierige Interessenslage herrschte – denn die bislang autarken Lehrstühle werden durch die Änderungen in ein zentralisiertes und vom Dekanat geleitetes System gezwungen, was zunächst einmal auf Aufgabe von Privilegien und Macht bedeutet. Die zukünftigen Anwender waren durch die ergebnisorientierte Vorgehensweise hochmotiviert. Die eigentlichen Projektrisiken sind durch die nicht immer konsequente Unterstützung durch leitende Autoritäten und die Regierung selbst (inklusive eines Regierungswechsels) bedingt. Alle Projektpartner werten das Projekt bislang aber als Erfolg, eine für den Oktober 2009 in Vorbereitung befindliche Konferenz wird die Ergebnisse der breiteren Öffentlichkeit vorstellen.

WODOMI ist somit ein Referenzprojekt für die Einsetzbarkeit des CP-Konzeptes für Dritte und hat im Rahmen seiner Arbeiten wichtige Erkenntnisse bei der Entwicklung von PIKAS beigetragen. Die prinzipielle CP-Vorgehensweise hat sich – soweit sie in der jeweiligen Projektphase bereits formuliert war – nachhaltig zum Projekterfolg beigetragen und konnte zudem aufgrund der vom Autor im Rahmen des Projekts gewonnenen Erkenntnisse weiter verfeinert werden.

9.3.2 BALVI iPi

Wie bereits auf Seite 307 ausgeführt kooperieren die Unternehmen BALVI und QuinScape bei der Erstellung einer Software-Plattform für den behördlichen Verbraucherschutz. Grundlage der Plattform sind die Ergebnisse dieser Arbeit, ein Teil der prototypischen Erfahrungswerte aus diesem Kapitel stammt auch aus der Erfahrung, die der Autor als Projektleiter und Chefarchitekt in dieser Kooperation sammeln konnte.

Wir zitieren unsere Einleitung zu Teil III der Arbeit: *„BALVI ist deutscher Marktführer für Software zur Überwachung der Tiergesundheit, der Lebensmittel- und der Futtermittelsicherheit für öffentliche Auftraggeber und verfügt hier über einen Marktanteil von 100% – alle deutschen Bundesländer setzen mittlerweile die Software BALVI iP ein. Konsequenterweise sind die nächsten anstehenden Schritte eine Ausweitung der Tätigkeiten auf internationale Märkte sowie eine Migration der existierenden auf Delphi basierenden Plattform auf neue (Java-)Technologien. Letzterer Schritt ist ein langfristiges Ziel: In den nächsten Jahren werden zunächst webbasierte Zusatzprodukte um das Standardprodukt herum aufgebaut und zu einem noch zu planenden Zeitpunkt wird das alte Produkt durch das dann existierende neue Produkt abgelöst.“*

2009 haben die ersten konkreten Kundenprojekte begonnen und wir gehen gegenwärtig davon aus, bis Ende des Jahres erste PIKAS-basierte Installationen ausliefern zu können. Bislang bewährt sich das PIKAS-Konzept in technischer Hinsicht ganz gemäß der Erwartungen. Folgendes Zitat von Dr. Rainer Fehling, Zweigstellenleiter BALVI Dortmund, belegt dies: *„Aus Sicht von BALVI ist der mit PIKAS eingeschlagene Weg genau richtig – die modellbasierte Vorgehensweise erlaubt es unseren Entwicklern und Fachkonzeptionierern schnell Ergebnisse zu produzieren, insbesondere viel schneller als bei der klassischen manuellen Programmierung. Zudem sehen wir als Produkthersteller einen sehr großen Wert in der gewählten Abstraktionsebene, da zukünftige technische Migrationen drastisch vereinfacht werden.“* Die aktuelle Planung fußt auf dem Aufbau einer BALVI-Integrationsplattform, die auf dem PIKAS-

Ansatz aufsetzt und dann Fachschalen für Integrationsszenarien, Intranetportale und eGovernment-Anwendungen für den behördlichen Verbraucherschutz ergänzt. Diese Fachschalen werden wiederum durch spezifische Module erweitert, die Einzelprozesse wie die Behandlung von Ordnungswidrigkeiten, die Verbindung zu Systemen wie HIT (Herkunfsinformationssystem Tier, siehe [BSLF09]) oder komplexe Abläufe wie z. B. die Tierzählung unterstützen. Die Zusammenarbeit ist strategisch ausgelegt und wird u. a. durch das im folgenden Abschnitt 9.3.3 beschriebene Förderprojekt im Rahmen des IKT.NRW-Programms unterstützt.

Zusammenfassend liefert die Kooperation mit BALVI wichtige Hinweise und Anwendungsfälle für die Validierung von PIKAS und demonstriert zudem die wirtschaftliche Bedeutung des CP-Ansatzes, da weitere Unternehmen bereit sind, signifikante Mittel in die Umsetzung eines CP-basierten Ansatzes zu investieren.

9.3.3 IKT.NRW

QuinScape konnte zusammen mit dem Lehrstuhl Dienstleistungsinformatik der Technischen Universität Dortmund (siehe [TUDO09]) sowie der BALVI GmbH (siehe [Balv09]) im Rahmen des IKT.NRW-Clusters zur Stärkung der Informations- und Kommunikationstechnologien in Nordrhein-Westfalen (siehe [IKTN09]) ein Förderprojekt zum Thema „Entwicklung eines Systems für die integrative Lebensmittelüberwachung im Verbund amtlicher und privatwirtschaftlicher Institutionen“ positionieren (siehe [MWME08]). In diesem Projekt wird der in dieser Arbeit vorgestellte PIKAS-Prototyp um Funktionen und Modelle zur Abbildung von serviceorientierten Architekturen (SOA) und zur Integration von Drittsystemen erweitert, um so einen modellbasierten und generativen Middleware-Ansatz für den mittelständischen Kontext am Beispiel der Lebensmittelüberwachung zu realisieren.

Das Bewertungsgremium lobte das hohe Innovationspotenzial des Ansatzes im Hinblick auf seine Auswirkungen auf die mittelständische Leistungs- und Anpassungsfähigkeit. Die Arbeiten für das Projekt haben im April 2009 begonnen, bis Ende Mai 2009 wurde in Lastenheft erstellt, dass nun bis zum Herbst 2009 in ein Pflichtenheft überführt wird.

Die Förderung zeigt, dass auch Dritte dem prinzipiellen CP-Ansatz nachhaltige Bedeutung zur Förderung mittelständischer Leistungsfähigkeit bestätigen und illustriert zudem, dass sich auch ein spezifischer CP-Entwurf auf angrenzende Themenfelder ausweiten lässt.

9.4 Weitere Einsatzfelder

Nachfolgend werden weitere Einsatzfelder skizziert, die sich auch für CP-Ansätze eignen. Es sei betont, dass diese Aufzählung nicht ansatzweise vollständig ist. Wir wollen stattdessen die Bandbreite der möglichen Einsatzfelder andeuten.

9.4.1 Massive Multiplayer Online Role-Playing Games

Massive Multiplayer Online Role-Playing Games (MMORPG, siehe [Wiki09o]) haben in den letzten Jahren große Erfolge gefeiert. Besitzen die meisten Unternehmen bereits ausgefeilte interne Entwicklungssysteme, so bietet sich gerade dieses Genre für Meta-Engines basierend auf dem CP-Ansatz an, da sich viele Konzepte sehr anschaulich abbilden lassen:

- Karten könnten auf äußerst natürliche Weise mit einem Karteneditor entworfen. Viele moderne Computerspiele bieten bereits entsprechende Level-Editoren.
- Spielregeln und spezielle Gegenstände, Monster und Ereignisse könnten über eine domänenspezifische *Rules Engine* abgebildet werden, die eine umgangssprachliche Formulierung spieltechnischer Sachverhalte ermöglicht.
- Grafiken könnten über einen 3D-Grafikeditor entworfen oder aus existierenden Komponenten neu komponiert werden.

Allerdings muss man sehen, dass es hier einen deutlichen Unterschied zu dem bislang von uns fokussierten mittelständischem Umfeld gibt, da hier weder mit den Skaleneffekten zu rechnen ist, die sich für mittelständische Anwendungen ergeben noch die technische Einstiegshürde ganz so leicht zu bewältigen ist, zumal die initialen Investitionen für ein modernes MMORPG viele Millionen Euro betragen werden. Interessant könnte der CP-Ansatz aber für etablierte MMORPG-Anbieter sein: Bislang werden Erweiterungen zumeist nur von den Herstellern selbst vertrieben. Ansätze wie *Second Life* (siehe [Wiki09p]) haben das große Potenzial von Systemen bewiesen, die es den Spielern erlauben, ihre virtuellen Welten selbst aufzubauen und zu erweitern. Würde man MMORPG-Prinzipien mit einem CP-basierten Erweiterungsmechanismus kombinieren, könnte man mglw. auch in diesem Kontext ähnliche Erfolge feiern, indem man Spieler in die Lage versetzt, eigene Quests, Karten, Monster, Regionen, usw. zu erzeugen.

9.4.2 Semantic-Web-Framework-Ansätze

In [BiMa04a] haben wir auf die Bedeutung des Semantic Web für die Zukunft einer internet-basierten Wissensgesellschaft hingewiesen. Wir haben erstmals ein Framework zur Modellierung Semantic-Web-basierter Systeme vorgestellt, dass wir in weiteren Arbeiten wie [BiMa04b, BiMa04c, BiMa04d, BiMa04e, BiMa05, BiMR05] zunehmend detailliert und unter der Bezeichnung WASP[14] zur Blaupause für eine Standardarchitektur für Semantic-Web-Lösungen ausgebaut haben. In Artikeln wie [BiMa07] und [BiHM07] haben wir die Brücke zu unserem Conceptual-Programming-Ansatz geschlagen und gezeigt, wie man mit diesen Standardkomponenten zu einem allgemeineren Entwicklungsansatz kommt.

Darauf aufbauend besteht zusätzlicher Forschungsbedarf, um gemäß der in Abschnitt 5.3 beschriebenen Vorgehensweise diese Grundlagen zu einem vollwertigen CP-Ansatz zu entwickeln. Für weitere Details verweisen wir auf unsere oben zitierten Veröffentlichungen.

9.4.3 Datenbasiertes Publizieren (Databased Publishing)

Die Erstellung komplexer Dokumente ist heute eine der Hauptaufgaben von Agenturen, die mit Layout- und Design-Experten Corporate Designs erstellen und dann für spezifische Dokumente umsetzen. Große Herausforderungen stellen sich auch hier im mittelständischen Umfeld, da z. B. sehr viele produzierende Unternehmen umfangreiche technische Kataloge produzieren und zu Marketing- wie Vertriebszwecken in Papierform für ihre Kunden nutzen. Die Erstellung dieser Kataloge ist in vielen Fällen eine sehr komplexe Aufgabe, da sich insbesondere die folgenden Aufgaben stellen:

- Das Material ist extrem umfangreich. Nicht selten müssen mehrere tausend – wenn nicht sogar zehntausend – Katalogseiten produziert werden.
- Das Material muss häufig mehrsprachig erstellt werden. Insbesondere asiatische Sprachen erzeugen durch drastisch abweichende Layoutvorgaben hohe Aufwände.
- Das Material muss stetig parallel zur Produktentwicklung aktuell gehalten werden.

[14] WASP ist ein Akronym für die zentralen Komponenten des WASP-Frameworks: Web Services, Agenten, Semantic Web und Personalisierung.

- Katalogseiten sind selten vollkommen schematisch sondern es gibt vielfältige Sonderfälle und Ausnahmeregeln.

Klassische Vorgehensweisen sehen vor, das gesamte zu erzeugende Material entweder händisch unter Verwendung von DTP-Systemen wie Adobe Indesign oder Quark Express durch entsprechende Fachkräfte erstellen zu lassen oder es wird auf halbautomatische Systeme mit sogenannten Template-basierten Ansätzen zurückgegriffen, die Seitenmuster vorgeben und diese dynamisch mit Daten füllen. QuinScape hat hier auf Basis seines Produkts DocScape (siehe [Docs06]) einen alternativen Ansatz entwickelt, bei dem das Wissen, dass Layouter und Designer besitzen und das eventuell in Corporate-Design-Richtlinien formalisiert ist, durch XML-Regeln festgehalten werden kann, um dann auf Grundlage von vollständigen Katalogdaten (also Texten, Grafiken, usw.) vollautomatisiert gemäß der formulierten Regeln ein druckfähiges Layout zu erzeugen.

Auch in diesem Ansatz finden sich viele der Ideen, die wir mit CP formalisiert und in eine Methodensammlung gegossen haben:

- Durch menschenlesbare XML-Regeln wird auf sehr hoher Abstraktionsebene beschrieben, was das Layout der zu erstellenden Publikation ausmacht. Die Details der Erstellung sind dem automatisierten System vorbehalten.
- Durch iterative Verfeinerung können Domänenexperten (also Setzer, Layouter, Gestalter, usw.) gemeinsam mit Werkzeugexperten ein Druckwerk erstellen.
- Details der technischen Implementierung werden dabei weitgehend abstrahiert, da auch die Produktion von druckfähigen PDF-Dateien weitgehend standardisiert ist.

Der grundlegende (gegenwärtig eher technisch orientierte) DocScape-Ansatz könnte sich dafür anbieten, zu einem vollwertigen CP-Ansatz ausgearbeitet zu werden, indem die zugrunde liegende Technologie um weitere Abstraktionen angereichert wird

9.4.4 Föderierte ERP-Systeme (FERP)

Die Idee hinter förderierten ERP-Systemen (siehe [Breh09]) ist die Schaffung einer Standardarchitektur, die es erlaubt, ERP-Komponenten von verschiedensten Herstellen miteinander zu kombinieren, um so leicht maßgeschneiderte und bedarfsgerechte ERP-Lösungen für spezifische Branchen und Bedürfnisse umsetzen zu können.

Sind diese Ideen im Augenblick noch sehr neu, bieten sie dennoch einen idealen Ansatzpunkt, um ein CP-basiertes Vorgehensmodell auf ihrer Grundlage aufzusetzen:

- FERP-Einzelkomponenten können als hochabstrakte Bausteine einer ERP-Gesamtlösung angesehen werden.
- Durch im Rahmen des CP-Entwurfs zu konzipierende Modelle könnten diese Bausteine Bestandteile eines modularen Bausatzes zur iterativen Planung und Umsetzung maßgeschneiderter und dennoch hochgradig individueller ERP-Lösungen insbesondere für den Mittelstand werden.
- Ein durch betriebswirtschaftliche Notwendigkeiten geprägtes Vorgehensmodell könnte als Grundlage dienen, um die Auswahl, Konfiguration und Integration der Bausteine iterativ (nach Prozessen, Abteilungen oder anderen Kriterien) voranzutreiben.
- Die Verfügbarkeit von Standardkomponenten kann insbesondere bei Vorhandensein eines Vorgehensmodells zur gezielten und schnellen Einführung darauf basierender fertiger Lösungen zu Skaleneffekten sowohl in der Verbreitung der Komponenten als auch ihrer Preisstruktur führen, was signifikante Vorteile sowohl für Anbieter als auch Nutzer der Komponenten bietet (siehe wiederum unsere Ausführungen ab Seite 39).

Voraussetzung dafür aber ist, dass der in [Breh09] beschriebene (oder ein vergleichbarer) Ansatz zur Integration verteilter ERP-Funktionalitäten sich etabliert und eine hinreichende Reife erlangt. Heute ist dies noch nicht gegeben, dennoch ist die prinzipielle Eignung des Themas für einen CP-Ansatz zweifelsohne eine interessante Perspektive für mittelständische IT-Dienstleister im ERP-Umfeld, die nach neuen Marktchancen suchen und bereit sind, in eine entsprechende Evaluierung zu investieren.

9.5 Zusammenfassung

In diesem Kapitel haben wir die uns vorliegenden praktischen Erfahrungen mit dem CP-Ansatz evaluiert. Für die CMDSD-Methode konnten wir in Abschnitt 9.1 belegen, dass es möglich ist, effizient und bezahlbar CMDSD-Werkzeuge zu entwickeln, die dann in konkreten Projekten auch einen schnellen Return-On-Invest ermöglichen. Wir konnten als Nutzwerte Effizienzsteigerungen von 40% sowie die Möglichkeit zu einem ROI nach maximal zwei Jahren für unser prototypisches System nachweisen. Außerdem haben wir dort eine Formelsammlung entwickelt, die es erlaubt, die Machbarkeit des CP-Ansatzes abzuschätzen. Besonders hervorzuheben sind hier die Gleichungen 9.18 und 9.20, die es ermöglichen, die Amortisationsrate von Inves-

titionen sowie die sinnvolle Mindestgröße eines mittelständischen IT-Dienstleisters abzuschätzen, für den sich eine Investition in ein CMDSD-Werkzeug rechnen könnte. Für die CDPM-Methode haben wir in Abschnitt 9.2 zumindest Indizien beschrieben, die ebenfalls darauf hindeuten, dass hier weitere Effizienz- und Qualitätsgewinne erzielbar sind:

- Schnelles Feedback und Prototypisierung erhöhten gefühlt die Lösungsqualität.
- Unklare Alternativen konnten schnell prototypisiert und so geklärt werden.
- Fachlicher Fragestellungen überwiegen im Gegensatz zu technischen Fragen.

In zukünftigen Arbeiten muss die CDPM-Methode aber wesentlich detaillierter untersucht werden. In Abschnitt 9.3 haben wir weitere Erfahrungen aus anderen noch nicht abgeschlossenen Projekten beschrieben, die auf unseren CP-Ideen aufbauen. Zudem haben wir in Abschnitt 9.4 die Übertragbarkeit des CP-Ansatzes auf beispielhafte weitere Anwendungsfelder skizziert, darunter Massive Multiplayer Online Role-Playing Games, Semantic-Web-Frameworks, vollautomatisiertes und datenbasiertes Publizieren von Dokumenten sowie föderierte ERP-Systeme.

Zusammenfassend lässt sich sagen, dass Conceptual Programming (CP) ein Ansatz ist, der tatsächlich dazu führt, dass mittelständische Unternehmen Technologien und Ansätze effektiv einsetzen können, die ansonsten Großprojekten und -unternehmen vorbehalten sind. Insbesondere ist es mittelständischen IT-Dienstleistern möglich, solche Ansätze auch innerhalb ihrer typischen Budgets umzusetzen. Somit erweist sich CP als wichtiger Beitrag zur Förderung der Wettbewerbsfähigkeit mittelständischer Unternehmen.

Kapitel 10

Schlussbetrachtung

„Eine mächtige Flamme entsteht aus einem winzigen Funken.“
— Dante Alighieri (1265-1321), italienischer Dichter

Diese Arbeit stellt die Frage, ob es gelingen kann, Technologien und Methoden aus dem Großprojektumfeld auf den mittelständischen Kontext so zu übertragen, dass es bei ausreichender Flexibilität gelingt, unter mittelständischen Rahmen- (und insbesondere Budget-)Beschränkungen leistungsstarke und zukunftssichere IT-Lösungen zur Steigerung der globalen Wettbewerbsfähigkeit zu entwickeln. Dazu haben wir ein synergetisches Vorgehensmodell vorgeschlagen, dass auf den vier Komponenten *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität* aufbaut und diese Einzelbausteine auf neue Weise zu einer flexiblen und mittelstandskonformen Gesamtmethodik kombiniert.

In Abschnitt 10.1 zeigen wir den Weg zur Beantwortung der Eingangs gestellten zentralen Frage überblicksartig auf, bevor wir in Abschnitt 10.2 noch einmal die besonderen Leistungen unseres Ansatzes diskutieren. In Abschnitt 10.3 beschließen wir diese Arbeit mit einem Ausblick auf sich aus der Arbeit ergebende zukünftige Forschungsthemen.

10.1 Abschließende Zusammenfassung

Die zunehmende Globalisierung der Wirtschaft zwingt Unternehmen aller Größenordnungen, sich dem daraus resultierenden kontinuierlichen Wandel zu stellen und die eigenen Prozesse auf wechselnde Rahmenbedingungen abzustimmen. Gelingt es Großunternehmen noch, sich diesen Herausforderungen durch hohen Ressourcenein-

satz zu stellen, stellt die Globalisierung mittelständische Unternehmen vor zunehmend schwerer zu bewältigende Herausforderungen. Denn der Mittelstand verfügt insbesondere im IT-Umfeld (als zentralem Motor für die Anpassungsfähigkeit von Unternehmen) weder über hinreichend tiefgehende Expertise noch über die Ressourcen, um Lösungsansätze aus dem Großkonzernumfeld erfolgreich auf den eigenen Kontext zu übertragen. Daraus resultiert eine zunehmend sinkende Konkurrenzfähigkeit mit entsprechend drastischen Risiken für eine Exportnation wie Deutschland, in der ca. 70% aller Arbeitsplätze, 80% aller Ausbildungsplätze und immerhin ca. 40% der Umsatzsteuereinnahmen des Staates durch mittelständische Unternehmen beigesteuert werden. Überraschenderweise beruht ein erheblicher Teil der im Mittelstand eingesetzten Software auf Individualentwicklung – häufig deswegen, weil mittelständische Unternehmen Nischen besetzen, für die sich die Entwicklung von Standardprodukten nicht rentiert. Gerade im Individualentwicklungsumfeld wird es bei stark begrenzten Ressourcen nahezu unmöglich, in einem globalen Kontext zu konkurrieren.

Um diesen Herausforderungen zu begegnen, etablieren wir in dieser Arbeit ein neues IT-Vorgehensmodell, das auf einer synergetischen Kombination existierender und bewährter Ideen beruht, die *modellbasierte und generative Entwicklung*, *domänenfokussierte Ansätze* sowie *agile Entwicklungsmethoden* kombinieren und diese um eine *kundenorientierte Sicht* auf die Entwicklungsprozesse erweitert. So werden schnelle und pragmatische Problemlösungsprozesse mit starker IT-Unterstützung für ausgewählte Domänen ermöglicht. Dadurch wollen wir einen Paradigmenwechsel in der Betrachtung der Entwicklungsmethodik erreichen. Dieser entfernt sich von technologisch orientierten Ansätzen und regt zu einer domänen- und aufgabenfokussierten Herangehensweise an.

In Kapitel 2 haben wir die oben genannten Teilaspekte auf ihre prinzipielle Nützlichkeit hin untersucht. Da wir mithilfe ausgewählte Studien ein hohes Eignungspotenzial (durch Reduzierung von Kommunikationsfehlern, der Steigerung der Entwicklungsgeschwindigkeit und -qualität sowie der Vermeidung früher Fehler) nachweisen konnten, haben wir dann in Kapitel 3 mehr als 30 existierende Ansätze und Methoden zu den oben genannten vier Teilaspekten bezüglich ihrer spezifischen Qualitäten untersucht. Dabei haben wir das *Domain Driven Design* (Abschnitt 3.2.7), die Theorie der *Software-Produktlinien* (Abschnitt 3.2.6), *generative Programmierung* (Abschnitt 3.3.4) und das *Feature Driven Development* (Seite 131) als besonders potenzialträchtig identifiziert. Zudem haben wir insgesamt 53 Konzepte aus den mehr als 30 betrachteten Methoden und Vorgehensmodellen abgeleitet, in denen sich die

Stärken der betrachteten Verfahren widerspiegeln (siehe die Übersicht auf Seite xvii). Auf Grundlage dieser 53 Konzepte haben wir in Kapitel 4 ausgehend von einer von uns auf Grundlage der 53 Konzepte entwickelten Struktur aus Werten, Prinzipien und Methoden ein integriertes Vorgehensmodell in Form des *Conceptual Programming* (CP) hergeleitet und dessen zwei Hauptmethoden *CMDSD* (Conceptual Model Driven Software Development) und *CDPM* (Concept Driven Project Management) in den Kapiteln 5 und 6 detailliert beschrieben. Diese beiden Hauptmethoden nutzen die von uns identifizierten vier Hauptoptimierungsrichtungen *Kundenorientierung*, *Domänenfokussierung*, *Automatisierung* und *Agilität*, um die besonderen Stärken der zugrunde liegenden Ansätze für eine ausgewählte Domäne im mittelständischen Kontext nutzbar zu machen. CP erlaubt es mittelständischen IT-Dienstleistern, neue Geschäftsmodelle und -felder zu erschließen. CP ermöglicht es aber auch mittelständischen Kunden dieser Dienstleister, die Vorteile von in Großprojekten bewährten Ansätzen mit wesentlich geringerem Ressourceneinsatz für ihren eingeschränkteren Kontext zu nutzen. Von zentraler Bedeutung sind in unserem modellbasierten Ansatz dabei die konsequente Archetypisierung von Domänenkonzepten, die durch Reduktion auf aufgabenspezifische Fachmodelle eine weitgehende generative Automatisierung der Implementierung erlaubt. Wir haben den wesentlichen Unterschied im Vergleich zu den existierenden Verfahren – nämlich eine synergetische Herangehensweise im Gegensatz zur isolierten Betrachtung einzelner Teilaspekte – noch einmal in Kapitel 7 herausgearbeitet, um dann anhand eines CMDSD-Prototyps zur modellbasierten Generierung von Portalsystemen und Webapplikationen namens PIKAS die Machbarkeit des von uns entwickelten Ansatzes zu belegen. PIKAS haben wir in Kapitel 8 beschrieben. In Kapitel 9 haben wir die dabei gewonnenen Erfahrungen mit Zahlen untermauert und dabei ein Formelwerk beschrieben, mit dem für andere Domänen die Wirtschaftlichkeit und Machbarkeit eines CP-basierten Ansatzes abgeschätzt werden kann – ausgehend von der Größe des sich mit dem Thema beschäftigenden IT-Dienstleisters. Die konkreten Zahlen zeigen exemplarisch eine nachhaltige wirtschaftliche Eignung unseres CP-Ansatzes und belegen unsere initialen Thesen: Ein Return-on-Invest kann binnen zwei Jahren erzielt werden (im Fall von PIKAS sogar deutlich früher aufgrund günstigerer Rahmenbedingungen). In Folge kann eine signifikante Leistungssteigerung in der Entwicklung erreicht werden: Wir konnten für unseren prototypischen Ansatz Effizienzgewinne von bis zu 40% nachweisen.

Es hat sich somit gezeigt, dass mittelständische Unternehmen mihilfe der hier vorgestellten Methodologie nicht nur in die Lage versetzt werden, ihre IT-Prozesse nachhaltig zur Unterstützung der Kerngeschäftsprozesse an eine sich ständig verän-

dernde globale Wirtschaft anzupassen. Ihnen bietet sich sogar die Chance, erfolgreich mit Großkonzernen zu konkurrieren, da kleinere Unternehmen mit einer agilen IT-Infrastruktur schneller die notwendigen organisatorischen Änderungen nachvollziehen können: Ein besonderer Wettbewerbsvorteil und eine Marktchance für kleine und mittlere Unternehmen, die durch Großkonzerne aufgrund ihrer inhärenten Komplexität nicht durch den strategischen Ressourcenvorteil ausgeglichen werden können. Das betont noch einmal die Bedeutung der Ergebnisse, die wir in den vorausgehenden Kapiteln vorgestellt und belegt haben: Die zu erwartenden Prozess- und Ergebnisverbesserungen machen eine Umstellung der Software-Entwicklungsprozesse auf CP, CMDSD und CDPM sowohl bei mittelständischen IT-Dienstleistern als auch bei deren Kunden äußerst erstrebenswert.

Aufgrund der zuvor beschriebenen Leistungen lassen sich das Thema und seine Bedeutung für die Wirtschaftsinformatik ([Wiki06e]) wie folgt einordnen:

- Die definierten Methoden für CMDSD und CDPM finden insbesondere Anwendung bei der Entwicklung, Planung und Pflege von betrieblichen Informationssystemen (also dem Teilgebiet „Informations- und Kommunikationssysteme“) für mittelständische Unternehmen. In diesem Umfeld definieren sie auch neue Konzepte für das Informations- und Prozessmanagement.
- Die Weiterentwicklung bzw. Verfeinerung von heute existierenden Modellierungsverfahren basiert auf praktischen Erfahrungen der Softwaretechnik bzw. des Software Engineering, hier insbesondere im Themenfeld agiler Methoden.
- Aus operativer Sicht kommen neue Erkenntnisse zum Projektmanagement hinzu.
- In strategischer Hinsicht werden die Organisationsberatung und die Analysephase des Software Engineering sowie das Service Engineering ([Wiki06a]) bei mittelständischen Unternehmen um agile modellgetriebene Aspekte angereichert.

10.2 Diskussion der Ergebnisse

Die Frage, ob es gelingen kann, erfolgreiche Ansätze aus Großprojekten auf den Mittelstand zu übertragen, beantwortet diese Arbeit mit einem klaren „Ja“. Die hier vorgestellte Methodik eignet sich ausdrücklich, um die individuelle Software-Entwicklung im mittelständischen Kontext für eingeschränkte Domänen auch in ei-

nem globalisierten Anforderungsumfeld flexibel und konkurrenzfähig zu halten. Diesen Anspruch haben wir bereits in Kapitel 1 wie folgt als Kernleistung der Arbeit ausgedrückt: die „*Entwicklung eines auf ausführbaren Fachmodellen basierenden Paradigmas (inklusive eines Vorgehensmodells), aufgrund dessen komplexe IT-Lösungen im mittelständischen Umfeld zuverlässig und anforderungsgerecht unter Berücksichtigung der existierenden Ressourcenbeschränkungen realisiert werden können*“. Die CP-Methodik besteht aus vier synergetisch kombinierten Paradigmenwechseln:

1. dem Wechsel von *technologie- und architekturzentrierten* zu *kundenorientierten* Werkzeugen und Methoden,
2. dem Wechsel von *allgemeinen* zu *domänenspezifischen* Modellen,
3. dem Wechsel von *manueller Codezentrierung* zu *modellbasierter Generierung bzw. Automatisierung* und
4. dem Wechsel von *präskriptiven* zu *integrierten agilen* Projektmanagementmethoden.

Wir haben dabei gezeigt, wie durch die Kreuzung einer horizontalen mit einer vertikalen Domäne und aus deren Kombination mit existierenden Entwicklungsansätzen ein neues, leistungsfähigeres Vorgehensmodell für die Erstellung von Softwaresystemen entsteht:

1. Wir haben eine *horizontale Domäne* ausgewählt („Mittelstand“) und daraus Anforderungen an den Software-Erstellungsprozess abgeleitet (Wirtschaftlichkeit, Flexibilität, Zugänglichkeit).
2. Wir haben die Betrachtung auf eine einzelne *vertikale Domäne* („Portale“) reduziert, um den Software-Erstellungsprozess und die zugrunde liegenden, bereits existierenden Ansätze durch sinnvolle Beschränkungen zu vereinfachen und zu beschleunigen.

Der resultierende CP-Ansatz ermöglicht es, Folgekosten durch Wartung und Betrieb von IT-Lösungen drastisch zu verringern, indem die Entwicklungssysteme darauf ausgelegt werden, auf einem höheren Abstraktionsniveau zu arbeiten, das technische Implementierungsdetails weitestgehend vor den Entwicklern und Nutzern verbirgt. Dadurch wird eine Technologie-Evolution auf Generatorebene möglich. Einmal entwickelte Anwendungen werden langfristig konkurrenzfähig und können auf dem Stand der Technik gehalten werden.

Durch unseren Prototypen PIKAS konnten wir die Möglichkeit belegen, mit mittelständischen Beschränkungen Lösungen zu entwickeln, die bislang Großunternehmen

vorbehalten waren. Konkrete Zahlen illustrieren das: Wir konnten durch unseren Prototypen die Entwicklungsaufwände um mehr als 40% reduzieren und dabei gleichzeitig die Qualität der entstandenen Lösung (die zudem eine weitreichend transparente Technologie-Migration ermöglicht) steigern. Zudem haben wir für den Gesamtansatz am Beispiel einen ROI von weniger als zwei Jahren nachgewiesen und für weitere Anwendungsgebiete ein Formelwerk erstellt, das auch zukünftig Machbarkeitsprüfungen ermöglicht.

Somit eröffnet *Conceptual Programming* einen erfolgsträchtigen Weg, um mittelständischen Unternehmen auch langfristig wichtige Vorteile im globalen Wettbewerb zu verschaffen.

10.3 Ausblick

In diesem Abschnitt wollen wir kurz auf zukünftige Forschungs- und Weiterentwicklungsmöglichkeiten des hier ausgearbeiteten Ansatzes eingehen.

10.3.1 Formale Evaluierung von CDPM

Wie bereits in Abschnitt 8.2 ausgeführt, steht eine wissenschaftliche Bewertung der CDPM-Methode noch aus. Wir haben einige Indizien für zusätzliche Nutzeffekte des CDPM-Vorgehens gefunden. Diese sind in sorgfältig abgestimmten Experimenten zu prüfen und hinsichtlich ihres Wirkgrades zu bewerten. Zudem könnten sich dabei wichtige Erkenntnisse für das von uns im nachfolgenden Abschnitt 10.3.6 beschriebene Arbeitsfeld der strukturierteren und domänenspezifischen Methodenauswahl für das Projektmanagement ergeben.

10.3.2 Katalogunterstützung für domänenspezifische CP-Instanzen

Wie in Abschnitt 4.3.3 erläutert ist es erforderlich, das CP-Vorgehensmodell für spezifische Domänen ggf. durch Ausprägung weiterer Werte, Prinzipien und Methoden zu detaillieren und auf den spezifischen Domänenkontext hin zu optimieren. Hier eröffnet sich ein ergiebiges Forschungsgebiet, um Klassen von domänenspezifischen Detaillierungen näher einzugrenzen (z. B. in formale, organisatorische oder kommunikative Kategorien) und für jede Kategorie sinnvolle Strukturen von Werten,

Prinzipien und Methoden zu sammeln.
So würde die zukünftige Ausprägung von CP-Instanzen vereinfacht und der gegenwärtig sehr hohe „kreative“ Anteil dieser Tätigkeiten durch einen formalen Prozess unterstützt. Zudem können so durch einen definierten Prozess die Einstiegshürden und -kosten bei der Ausprägung von CP-Ansätzen weiter reduziert werden.

10.3.3 Ergonomische Optimierung von CMDSD

Beim Studium modellbasierter Verfahren ist uns immer wieder aufgefallen, dass sehr wenig über kognitive und psychologische Aspekte der Modellbildung diskutiert wird. Stattdessen scheinen Modelle häufig „vom Himmel zu fallen“ – zumindest werden sie selten strukturell begründet. Hier bietet sich ein Brückenschlag zur Psychologie an, um formale IT-Modelle hinsichtlich ihrer Zugänglichkeit und Verständlichkeit zu optimieren. Vor allem wäre es interessant, Qualitätskriterien und Best Practices für die Herleitung eigener Modelle zu entwickeln, da die Verzahnung von Verhaltens- und Wahrnehmungspsychologie mit mathematischen und formalen Anforderungen potenziell vielfältige neue Fragestellungen aufwirft.

10.3.4 Test von Fachmodellen

Es stellt sich die Frage, wie Fachmodelle (also Modelle des CMDSD-Systems) effizient getestet werden können. Interessant sind sie aufgrund ihrer Erweiterungspunkte, der Modellfacetten sowie der Laufzeitein- und ausgaben. Zum einen ist zu untersuchen, ob sich entsprechende Testfälle automatisiert aus Fachmodellen erzeugen lassen. Zum anderen ist zu prüfen, wie effizient Testfälle durch Domänenexperten spezifiziert werden können, da Domänenexperten besonders qualifiziert sind, um die fachliche Korrektheit eines instanziierten Systems zu verifizieren. Besondere Schwierigkeit ist hier, dass bei der Testfalldefinition nach gegenwärtigem Erkenntnisstand sowohl Wissen über die Fachdomäne als auch von der Implementierungs- und Arbeitsweise des CMDSD-Systems erforderlich ist.

Aus der Arbeit mit dem PIKAS-Prototypen sind bereits einige Ideen entstanden, wobei zwischen dem Test von Modellen, die aus Standardelementen zusammengesetzt sind, und dem Test von Erweiterungspunkten zu unterscheiden ist:

1. Für den Test von Modellen, die sich ausschließlich aus Standardkomponenten zusammensetzen, benötigt man einen speziellen Aufzeichnungsmodus in der Laufzeitplattform. Dieser unterstützt die folgenden Eigenschaften:

- Der Modus kann zu Beginn einer Transaktion oder eines Workflows aktiviert werden.
- Der Aufzeichnungsmodus kann zu einem wählbaren Zeitpunkt gestoppt werden.
- Das Laufzeitsystem zeichnet alle externen Eingaben in das System und alle erzeugten Ausgaben des Systems auf.
- Domänenexperten können die Ein- und Ausgaben betrachten und entscheiden, ob der Aufzeichnungslauf einen erfolgreichen Test darstellen.
- Die Testläufe müssen durch CMDSD-Experten insofern erweiterbar sein, dass bestimmte Ausgangssituationen (also Vorbedingungen) in dem System erfüllt sind. Dies ermöglicht die wiederholte Durchführung von Tests. Optimalerweise arbeiten CMDSD-Systeme datenbankbasiert. In diesem Fall sollte vor Beginn eines Testlaufs ein Abzug der vollständigen Testdatenbank erstellt werden, der bei späteren Läufen vor dem Test wieder eingespielt wird.
- Das Laufzeitsystem verwaltet die erfolgreichen Tests in einer Testbibliothek.

Existiert ein solcher Aufzeichnungsmodus, können bereits bei den Abnahme- und Akzeptanztests des Systems Transaktionen bzw. Prozess- und Workflowdurchläufe aufgezeichnet und – wenn erfolgreich – in der Testbibliothek gespeichert werden. Bei späteren Läufen werden die aufgezeichneten Testläufe automatisiert wiederholt.

2. Für den Test von Erweiterungspunkten muss das CMDSD-System automatisierte Tests enthalten, welche die Schnittstelle beschreiben. Diese automatisierten Tests können natürlich auch für projektspezifische Implementierungen von Erweiterungspunkten verwendet werden.

10.3.5 Rapid Prototyping

Rapid Prototyping liefert exzellente Ergebnisse für kleine Projektteams, hat sich in der Vergangenheit aber nicht für große Entwicklungsteams bewährt (siehe dazu [Tayl95], S. 146–147). Wir haben die praktische Erfahrung gemacht, dass dieser Ansatz für CMDSD dennoch in eingeschränkten Maße funktioniert, solange größere Projektteams in kleineren Teilteams weitgehend unabhängig voneinander arbeiten können. Eine Komplexitäts- oder Größenbeschränkung in der horizontalen Domäne scheint hier maßgeblich zu sein.

In der Betrachtung der vertikalen Domäne sind die zugrunde liegenden Modelle jedoch meist sehr eng miteinander verknüpft, so dass eine saubere Partitionierung der Projektverantwortlichkeiten nur schwer vorzunehmen ist. Hier kommt es im Projektverlauf oft zu konfliktären Änderungen und Ergänzungen an Modellen, für die eine Konfliktresolution erforderlich ist. Aufgrund der nicht-trivialen Abhängigkeiten zwischen Modellen und der potenziell weitreichenden Auswirkungen auch begrenzter Änderungen an einem zentralen Modell ergeben sich äußerst komplexe Fragestellungen bei Versionskonflikten zwischen Modellen. Zudem stellt sich die Frage, wie CMDSD-Nutzern überhaupt Konflikte in Modellen benutzerfreundlich und ergonomisch dargestellt werden können, um eine manuelle Konfliktauflösung für nicht automatisch entscheidbare Konflikte zu ermöglichen. Diesem Thema können zukünftige Forschungsaktivitäten verstärkte Aufmerksamkeit widmen, da unsere stetig zunehmende Erfahrung in CMDS-Projekten zeigt, dass die generelle Methodik auch bei großen Kunden und Projekten auf starkes Interesse stößt und dort ein immenser, auch wirtschaftlich motivierter Handlungsbedarf besteht.

10.3.6 Domänenspezifische Auswahl von Projektmanagementmodellen

In Abschnitt 3.4 haben wir auf einige präskriptive und agile Vorgehensmodellen hingewiesen. Für unseren Ansatz haben wir eine agile, iterative und evolutionäre Herangehensweise definiert. Davon gibt es viele Varianten. Um eine geeignete auszuwählen ist die Ableitung eines domänenspezifischen Kriterienkatalogs elementar. Umgekehrt wäre zu untersuchen, welche Kriterien die einzelnen Vorgehensmodelle aufweisen, die sie für spezielle Kontexte entweder besonders gut oder schlecht erscheinen lassen.

10.3.7 Übertragung von CP auf Projekte außerhalb des Mittelstands

In [BiMa09] haben wir als Grundlage für zukünftige Arbeiten eine Verbindung zwischen den Arbeiten von Zachmann zum Thema *Enterprise Architecture* und der Anwendbarkeit von CP, CMDSD und CDPM zur Herstellung einer geradlinigen Kohärenz von IT-Systemlandschaften und -Architekturen aufgezeigt. Da Enterprise Architecture eher ein Thema in Großkonzernen ist, deutet sich hier bereits die Möglichkeit an, CP auch auf den Kontext von Großkonzernen, Großprojekten und

langlaufenden IT-Systemen zu übertragen, um auch hier eventuell signifikante Prozessverbesserungen zu erreichen. Damit würde sich der Kreis schließen, denn wir haben zu Beginn der Arbeit unseren Ansatz dadurch motiviert, dass wir bewährte und erfolgreiche Ansätze aus dem Umfeld von Großprojekten auf den mittelständischen Kontext übertragen wollen. Zukünftige Arbeiten werden sich der Frage widmen, ob und wie die von uns in dieser Arbeit vorgestellten Effizienz- und Qualitätsverbesserungen sich bei einer Rückübertragung auf den Ursprungskontext auswirken.

„This is the end, beautiful friend,
This is the end, my only friend, the end.“
— The Doors (1965–1973), US-amerikanische Rockband

Teil IV

Anhang

Literaturverzeichnis

[AAGI06] Agile Alliance (Hrsg.): *Agile Alliance :: What is Agile Software Development?*, `http://www.agilealliance.org/intro`, Abruf am 15.10.2006

[Aden09] adenin TECHNOLOGIES AG (Hrsg.): *adenin TECHNOLOGIES Inc - adenin Intranet Technologies*, `http://www.adenin.de`, 2009, Abruf am 12.06.2009

[Adob09] Adobe (Hrsg.): *publishing software | Adobe Indesign CS 4*, `http://www.adobe.com/products/indesign/`, Abruf am 10.01.2009

[AgAl06] Agile Alliance (Hrsg.): *Agile Alliance*, `http://www.agilealliance.org/home`, Abruf am 27.07.2009

[AhKW88] Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J.: *The AWK Programming Language.* Addison-Wesley, 1988

[Ambl02a] Ambler, Scott W.: *Agile Modeling*, John Wiley & Sons, 2002

[Ambl02b] Ambler, Scott W.: *Bridging the Distance*, `http://www.ddj.com/dept/architect/184414899`, Dr. Dobb's, 12.08.2002, Abruf am 15.10.2006

[Ambl04a] Ambler, Scott W.: *Skinnier RUP*, `http://www.ddj.com/dept/architect/184415143`, 12.04.2004, Abruf am 16.09.2006

[Ambl04b] Ambler, Scott W.: *The Object Primer. Agile Model-Driven Developement with UML 2.0*, Cambridge University Press, 2004

[Ambl06a] Ambler, Scott W.: *Agile Modeling (AM) Homepage: Effective Practices for Modeling and Documentation*, `http://www.agilemodeling.com/`, 16.09.2006, Abruf am 16.09.2006

[Ambl06b] Ambler, Scott W.: *Supersize me*, `http://www.ddj.com/dept/architect/184415491`, Dr. Dobb's, 15.02.2006, Abruf am 15.10.2006

[Andr05] Andres, Thomas: *Vom Geschäftsprozess zum Anwendungssystem: Modellgetriebene Entwicklung Betriebswirtschaftlicher Software*, In: OBJEKTspektrum Online 2005 – die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1275`, 2005

[ApSF06] Apache Software Foundation, `http://www.apache.org`, Abruf am 06.10.2006

[ASFC06] Apache Software Foundation: *Commons Project*, `http://www.apache.org/commons`, Abruf am 06.10.2006

[ASFJ06] Apache Software Foundation: *Jakarta Project*, `http://www.apache.org/jakarta`, Abruf am 06.10.2006

[ASFS09] Apache Software Foundation: *Struts*, `http://struts.apache.org/`, Abruf am 12.06.2009

[Balv09] BALVI GmbH (Hrsg.): *BALVI GmbH*, `http://www.balvi.de/`, Abruf am 20.03.2009

[BaTZ05] Bartelt, Christian; Ternité, Thomas; Zieger, Matthias: *Modellbasierte Entwicklung mit dem V-Modell XT*, In: OBJEKTspektrum 5/05, `http://www.sigs.de/publications/os/2005/05/bartelt_ternite_OS_05_05.pdf`, Abruf am 16.09.2006

[BBBC01] Beck, Kent; Beedle, Mike; Bennekum, Arie van; Cockburn, Alistair et al.: *Manifesto for Agile Software Development*, `http://www.agilemanifesto.org/`, 2001, Abruf am 03.12.2006

[BCMM03] Böckle, Günter; Clements, Paul; McGregor, John D.; Muthig, Dirk; Schmid, Klaus: *A Cost Model for Software Product Lines*, Fifth International Workshop on Product Family Engineering (PFE-5), Siena, Italy, November 4–6, 2003

[BDUF06] C2 Wiki (Hrsg.): *Big Design Up Front*, `http://c2.com/cgi/wiki?BigDesignUpFront`, 11.10.2006, Abruf am 21.10.2006

[BeAn05] Beck, Kent; Andres, Cynthia: *Extreme Programming Explained. Embrace Change. Second Edition*, Addison-Wesley, Boston, 2005

[BeBG05] Beydeda, S.; Brook, M.; Gruhn, V.: *Model Driven Software Development*, Springer, 2005

[Beed02] Beedle, Mike: *Agile Software Development with Scrum*, Prentice Hall, 2002

[BeHo98] Beyer, Hugh; Holtzblatt, Karen: *Contextual Design*, Morgan Kauffmann, San Francisco, 1998

[Bers96] Berson, Alex: *Client/Server Architecture*, McGraw-Hill, 1996

[Bigg02] Biggerstaff, Ted: *A New Architecture for Transformation-Based Generators*, `http://softwaregenerators.com/NewControlStructure3.PDF`, Abruf am 31.12.2008

[BiHM07] Biskup, Thomas; Heyer, Nils; Marx Gómez, Jorge: *Building Sound Semantic Web Frameworks for Scalable and Fault-Tolerant Systems*; in: Sugumaran, V. (Hrsg.): *Advanced Topics in Intelligent Information Technologies.* Idea Group Publishing Hershey (PA), London 2007, S. 153–181

[BiMa07] Biskup, Thomas; Marx Gómez, Jorge: *Conceptual Model Driven Software Development (CMDSD) as a Catalyst Methodology for Building*

Sound Semantic Web Frameworks; in: Radaideh, M. (Hrsg.): *Architecture of Reliable Web Applications Software.* Idea Group Publishing Hershey (PA), London 2007, S. 194–221

[BiMa04a] Biskup, Thomas; Marx Gómez, Jorge: *Building Blocks of a Semantic Web Framework – Requirements Analysis and Architectural Implications*, Proceedings of 3rd International Workshop on Web Semantics – WebS'04 in conjunction with the 14th International Conference on Database and Expert Systems Applications DEXA 2004 —- Zaragoza (Spain), Database and Expert Systems Applications, 15th International Workshop on (DEXA'04), 2004, S. 214–218

[BiMa04b] Biskup, Thomas; Marx Gómez, Jorge: *The WASP Framework – Bridging the Gap between the Web of Systems, The Web of Services and the Web of Semantics with Agent Technology*, Proceedings of the 1st Workshop on Intelligent Mobile Agents in Peer-To-Peer Networks (IMAP2P) in conjunction with the 4th International Symposium on Engineering of Intelligent Systems, (EIS'2004), Madeira (Portugal), CD Proceedings, 2004

[BiMa04c] Biskup, Thomas; Marx Gómez, Jorge: *Component Requirements for a Universal Semantic Web Framework*, In: AIS SIGSEMIS Bulletin, Volume 1, Number 3, October 2004, S. 25–28

[BiMa04d] Biskup, Thomas; Marx Gómez, Jorge: *Requirements for a Semantic Web Framework Architecture*, In: Proceedings of XII Latin-Ibero-American Congress on Operations Research and Systems CLAIO'2004 — Havana (Cuba), 2004

[BiMa04e] Biskup, Thomas; Marx Gómez, Jorge: *Approach to Bridge the Gap between the Web of Systems, the Web of Services and the Web of Semantics using Agent Technology and the WASP Framework*, Proceedings of 15th IRMA International Conference – Innovations Through Information Technology, (IRMA'2004), New Orleans, (USA), CD Proceedings

[BiMa06] Biskup, Thomas; Marx Gómez, Jorge: *How to Simplify Software Projects with Conceptual Model Driven Software Development (CMDSD)*, Proceedings of International Conference on Cost Effectiveness and Efficiency in Industry (OIE'2006), Poznan (Poland), S. 25–32

[BiMR05] Biskup, Thomas; Marx Gómez, Jorge; Rautenstrauch, Claus: *The WASP Framework – Bridging the Gap between the Web of Systems, the Web of Services and the Web of Semantics with Agent Technology*, In: International Journal of Intelligent Information Technologies (IJIIT), Vol. 1, No. 2, April - June 2005, S. 68–82

[BiMa05] Biskup, Thomas; Marx Gómez, Jorge: *Building a Semantic Web Framework with Reusable and Configurable Core Technologies*, In: Proceedings of 16th IRMA International Conference – Managing Modern Organizations With Information Technology, (IRMA'2005), San Diego, USA, 2005, S. 859–862

[BiMa09] Biskup, Thomas; Marx Gómez, Jorge: *Bridging the gap between EA goals and technology requirements with Conceptual Programming*; in: Doucet, G., Gøtze, J., Saha, P., Bernard, S. (Hrsg.): *Coherency Management: Architecting the Enterprise for Alignment, Agility, and Assurance*, Idea Group Publishing Hershey (PA), erscheint 2009

[Bisk01] Biskup, Thomas: *Realisierung natürlichsprachiger Web-Agenten in Java*, Tagungsband der Net.ObjectDays, 2001

[Bitp02] Bitplan (Hrsg.): *Bitplan: Erfolgsfaktoren in der Softwareentwicklung, Auswertung der Agility Days 2002*, `http://www.bitplan.de/download/AuswertungErfolgsfaktorenAgilityDays2002.pdf`, 2005, Abruf am 30.09.2005

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

[BMBF00] BMBF (Hrsg.): *Analyse und Evaluation der Softwareentwicklung in Deutschland. Eine Studie für das Bundesministerium für Bildung und Forschung*, `http://www.isi.fhg.de/iuk/dokumente/evasoft_abschlussbericht.pdf`, Dezember 2000, Abruf am 10.10.2006

[BMMS04] Böckle, G.; Clements, P.; McGregor, J. D.; Muthig, D.; Schmid, K.: *Calculating ROI for Software Product Lines*, IEEE Software, Volume 21, Issue 3, May–June 2004, Seite 23–31

[BMRS96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996

[Boeh81] Boehm, Barry W.: *Software Engineering Economics*, Englewood Cliffs, Prentice-Hall PTR, 1981

[Boeh88] Boehm, Barry W.: *A Spiral Model of Software Development and Enhancement*, IEEE Computer 21, 1988

[BoTu03a] Boehm, Barry; Turner, Richard: *Observations on Balancing Discipline and Agility*, `http://agile2003.agilealliance.org/files/P4Paper.pdf`, 23.12.2008, Abruf am 23.12.2008

[BoTu03b] Boehm, Barry; Turner, Richard: *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley Longman, Amsterdam, Auflage: 2004

[Breh09] Brehm, Nico: *Föderierte ERP-Systeme auf Basis von Web Services*, Dissertation, Carl von Ossietzky Universität Oldenburg, Fakultät II (Informatik-, Wirtschafts- und Rechtswissenschaften), 2009

[Broo95] Brooks, Frederick P.: *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*, Addison-Wesley, 1995

[BSLF09] Bayerisches Staatsministerium für Landwirtschaft und Forsten (Hrsg.): *Informationen*, `http://www.hi-tier.de/info99.html`, 22.02.2008, Abruf am 21.06.2009

[BSSW09] Biskup, Thomas; Stalitza, Rafael; Steiger, Sascha; Wloka, Nils: *Spring Praxishandbuch 2: Dynamisierung, Verteilung und Sicherheit*, Entwickler.Press, erscheint 2009

[BuKn02] Bunse, Christian; von Knethen, Antje: *Vorgehensmodelle kompakt*, Spektrum Akademischer Verlag GmbH, Heidelberg / Berlin 2002

[Bühl05] Bühler, Christian: *Ausführbare UML-Spezifikation: Cassandra/XUML*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1276`, 2005

[Bund08] Bundesministerium des Inneren (Hrsg.): *SAGA 4.0 – Standards und Architekturen für E-Government-Anwendungen*, `http://gsb.download.bva.bund.de/KBSt/SAGA/SAGA_v4.0.pdf`, März 2008, Abruf am 25.03.2009

[BWHS08] Biskup, Thomas; Wloka, Nils; Helmberger, Sven; Steiger, Sascha; Spielmann, Holger; Stalitza, Rafael: *Spring Praxishandbuch 1: Integration und Testing*, Entwickler.Press, 2008

[Cawl05] Cawley, Piers: *Fluent Interfaces*, `http://www.bofh.org.uk/articles/2005/12/21/fluent-interfaces`, 21.12.2005, Abruf am 29.12.2008

[ClMC05] Clements, Paul C.; McGregor, John D.; Cohen, Sholom G.: *The Structured Intuitive Model for Product Line Economics (SIMPLE)*, CMU/SEI-2005-TR-003, 2005

[ClNo02] Clements, Paul C.; Northrop, Linda M.: *Software Product Lines. Practices and Patterns*, Addison-Wesley, 2002

[ClNo07] Clements, Paul; Northrop, Linda; mit: Bachmann, Felix; Bergey, John; Chastek, Gary; Cohen, Sholom; Donohoe, Patrick; Jones, Lawrence; Krut, Robert; Little, Reed; McGregor, John; O'Brien, Liam : *A Framework for Software Product Line Practice, Version 5.0*, 2007, `http://www.sei.cmu.edu/productlines/framework.html`, Abruf am 06.01.2009

[Code09a] Codehaus.org (Hrsg.): *XStream – About XStream*, `http://xstream.codehaus.org/`, Abruf am 10.01.2009

[Code09b] Codehaus.org (Hrsg.): *Groovy - Home*, `http://groovy.codehaus.org/`, 01.06.2009, Abruf am 12.06.2009

[CoHa05] Coplien, James O.; Harrison, Neil B.: *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall, 2005

[Cohe03] Cohen, Sholom: *Predicting When Product Line Investment Pays*, CMU/SEI-2003-TN-017, 2003

[CoHM93] Coplien, James O.; Hutz, Susanna; Marykuca, Brent: *Iterative development / OO: The bottom line*, In: Archibald, Jerry L. (Editor): *OOPS Messenger 2 (April 1993) 4*, S. 101–108

[Coll01] Collins, Jim: *Good to Great: Why Some Companies Make the Leap... and Others Don't*, HarperCollins, 2001

[Cont08] Contentmanager.de (Hrsg.): *Produktübersicht - Content Management: Content Management Systeme*, `http://www.contentmanager.de/itguide/marktuebersicht_produkte_cms.html`, Abruf am 29.12.2008

[Crinn91] Crinnion, John: *Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology*, Plenum Press, New York, 1991.

[Crmn06] CRMnext (Hrsg.): *CRM Challenges – Building n Effective Strategy*, `http://www.crmnext.com/us/Academy/crmchallenges.pdf`, 2006, Abruf am 18.03.2009

[CuNo98] Cusumano, M. A.; Nobeoka, K.: *Thinking Beyond Lean – How Multi-Project Management is Transforming Product Development at Toyota and Other Companies*, The Free Press, New York, 1998

[CzEi00] Czarnecki, Krzysztof; Eisenecker, Ulrich W.: *Generative Programming – Methods, Tools, and Applications*, Addison-Wesley, 2000

[Davi98] Davis, Alan M.: *Operational Prototyping: A new Development Approach*. IEEE Software, September 1992.

[Deri06] Deridder, Dirk: *A Concept-Centric Environment for Software Evolution in an Agile Context*, Ph. D. Thesis, Vrije Universiteit Brussel, 2006

[DHMi98] Dröschel, Wolfgang; Heuser, Walter; Midderhoff, Rainer: *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*, Oldenbourg, München 1998

[Dine07] Dinechin, Christophe de: *Concept Programming The Art of Turning Ideas into Code*, `http://xlr.sourceforge.net/Concept\%20Programming\%20Presentation.pdf`, 2007, Abruf am 21.01.2009

[Dine09] Dinechin, Christophe de (Hrsg.): *XLR: Extensible Language and Runtime - Scope Injection / Context-oriented programming*, `http://xlr.sourceforge.net/`, Abruf am 21.01.2009

[Dmit06] Dmitriev, Sergey: *Language Oriented Programming: The Next Programming Paradigm*, `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`, Abruf am 09.10.2006

[Docs06] QuinScape GmbH (Hrsg.): *DocScape: Datenbasiertes Pubizieren mit regelbasierter Vollautomatisierung*, `http://www.docscape.de`, 07.08.2006, Abruf am 07.08.2006

[Dona06] Donald, Keith: *Spring's Architecture - Not a Single Dependency Cycle*, `http://www.springsource.org/node/310`, 18.07.2006, Abruf am 11.03.2009

[DrHM98] Dröschel, Wolfgang; Heuser, Walter; Midderhoff, Rainer (Hrsg.): *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*, Oldenbourg, München, 1998

[DrWi99] Dröschel, Wolfgang; Wiemers, Manuela: *Das V-Modell 97. Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*, Oldenbourg, München 1999

[DrWi99] Dröschel, Wolfgang; Wiemers, Manuela: *Das V-Modell 97. Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*, Oldenbourg, München 1999

[DSSW97] Drosdowski, Günther (Hrsg.); Scholze-Stubenrecht, Werner (Hrsg.); Wermke, Matthias (Hrsg.): *Duden – Das Fremdwörterbuch, 6., auf Grundlage der amtlichen Neuregelung der deutschen Rechtschreibung überarbeitete und erweitere Auflage*, Bibliographisches Institut & F. A. Brockhaus AG, 1997

[Ecke05] Eckel, Bruce: *Computing Thoughts – Object Design*, `http://www.artima.com/weblogs/viewpost.jsp?thread=115101`, 15.07.2005, Abruf am 06.10.2006

[Ecks04] Eckstein, Jutta: *Agile Softwareentwicklung im Großen. Ein Eintauchen in die Untiefen erfolgreicher Projekte*, Dpunkt Verlag, 2004

[Edwa07] Edwards, Jonathan: *No Ifs, Ands, or Buts – Uncovering the Simplicity of Conditionals*, OOPSLA'07, 21.–25. Oktober 2007, Montréal, Québec, Canada, `http://subtextual.org/OOPSLA07.pdf`, Video verfügbar (siehe [Edwa07b], Abruf am 21.01.2009

[Edwa07b] Video zu [Edwa07], `http://subtextual.org/subtext2.html`, Abruf am 21.01.2009

[Edwa08] Edwards, Jonathan: *Modular Generation and Customization*, MIT-CSAIL-TR-2008-061, Technical Report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, USA, 10. Oktober 2008, `http://hdl.handle.net/1721.1/42895`, Abruf am 21.01.2009

[Edwa09a] Edwards, Jonathan: *subtext – Uncovering the simplicity of programming*, Subtext Homepage, `http://subtextual.org/`, Abruf am 21.01.2009

[Ehle05] Ehlert, Martin: *MDD – Rechnet sich das in meinem Projekt?*, In: OBJEKTspektrum Online 2005 – die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1278`, 2005

[ErFa02] Erdogmus, Hakan; Favaro, John: *Keep your Options Open: Extreme Programming and the Economics of Flexibility*, In: *Extreme Programming Perspectives*, Addison-Wesley, 2002

[EuCo03] European Comission: *The new SME definition – User guide and model declaration*, European Comission, 2003

[EuKo03] Europäische Kommission (Hrsg.): *Empfehlung 2003/361/EG: KMU-Definition*, `http://europa.eu.int/comm/enterprise/enterprise_policy/sme_definition/index_de.htm`, 2003, Abruf am 17.09.2006

[Evan04a] Evans, Eric: *Domain Driven Design – Tackling Complexity in the Heart of Software*, Pearson Education, Inc., New Jersey, 2004

[Evan04b] Evans, Eric: *Domain-Driven / Model-Driven*, `http://www.domaindrivendesign.org/discussion/blog/evans_eric_ddd_and_mdd.html`, 04.02.2004, Abruf am 09.01.2009

[Eysh06] Eysholdt, Moritz: *Vergleich der Leistungsfähigkeit zweier Ansätze zur generativen Softwareentwicklung unter Verwendung der MDA und ARIS*, Individuelles Projekt, Universität Oldenburg, Fachbereich Wirtschaftsinformatik, 2006

[FCEl04] Filman, Robert E. (Hrsg.); Clarke, Siobhan (Hrsg.); Elrad, Tzilla (Hrsg.): *Aspect-Oriented Software Development*, Addison-Wesley Longman, 2004

[Fisc05a] Fischer, Frank: *Die Fabrik für Programmcode kommt langsam in Sicht*, In: Computer Zeitung (2005) 46, S. 19

[Fisc05b] Fischer, Frank: *Experten streiten um UML*, In: Computer Zeitung 46, 2005, S. 19

[Fowl03] Fowler, Martin: *Patterns of enterprise application architecture*, Addison-Wesley, 2003

[Fowl05a] Fowler, Martin: *FluentInterface*, `http://martinfowler.com/bliki/FluentInterface.html`, 20.12.2005, Abruf am 29.12.2008

[Fowl05b] Fowler, Martin: *Generating Code for DSLs*, `http://www.martinfowler.com/articles/codeGenDsl.html`, 12.06.2005, Abruf am 30.01.2009

[Fowl06a] Fowler, Martin: *Using an Agile Software Process with Offshore Development*, `http://www.martinfowler.com/articles/agileOffshore.html`, 18.06.2007, Abruf am 15.10.2006

[Fowl06b] Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?*, `http://www.martinfowler.com/articles/languageWorkbench.html`, 12.06.2006, Abruf am 20.01.2009

[Free09a] The Freemarker Project (Hrsg.): *FreeMarker: Java Template Engine Library - Overview*, `http://freemarker.org/`, 2009, Abruf am 12.06.2009

[Free09b] Freemind (Hrsg.): *Main Page - FreeMind - free mind mapping software*, `http://freemind.sourceforge.net/wiki/index.php/Main_Page`, 30.05.2009, Abruf am 19.06.2009

[FrWe02] Freese, Tammo; Westphal, Frank: *Agile Software-Entwicklung mit Xtreme Programming*, Oldenburger Software Forum (OSF), 30.01.2002

[GHJV96] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Entwurfsmuster. Elementer wiederverwendbarer objektorientierter Software*, Addison-Wesley, 1996

[Gilb94] Gilb, R.: *Software: chronisch mangelhaft*, In: Spektrum der Wissenschaft, Dezember 1994, S. 56–63

[Glad01] Gladwell, Malcom: *The Tipping Point*, Abacus, Time Warner Book Group UK, 2001

[Glas06] Glas, Christian: *Mittelstand findet wieder zur IT*, In: CIO Online, `http://www.cio.de/markt/analysen/825552/index.html`, 02.08.2006, Abruf am 19.11.2006

[Glec93] Gleckmann et al.: *The Technology Payoff*, Business Week, Juni 1993, S. 57–68

[Govo99] Govoni, Darren: *Java Application Frameworks*, John Wiley & Sons, 1999

[Gran78] Granovetter, Mark: *Threshold Models of Collective Behaviour*, In: American Journal of Sociology, Volume 83, 1978, S. 1420–1443

[Gran83] Granovetter, Mark: *Threshold Models of Diffusion and Collective Behaviour*, In: Journal of Mathematical Sociology, Volume 9, 1983, S. 143–186

[Gran95] Granovetter, Mark: *Getting a Job*, Chicago University Press, 1995

[Gron06] Gronau, Norbert: *Wandlungsfähigkeit als Richtschnur*, In: Mittelstandsmagazin „Lösung“, Ausgabe 3/2006, Steeb, S. 8–9, `http://www.steeb.de/infocenter/loesung/03_2006/loesung_03_2006_08.pdf`, Abruf am 19.11.2006

[Gruh05] Gruhn, Volker: *Auf dem Weg in neue technologische Welten: Retten Sie das Anwendungswissen mit MDA!*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1280`, Abruf am 29.09.2005

[GSCK04] Greenfield, Jack; Short, Keith; Cook, Steve; Kent, Stuart: *Software Factories – Assembling Applications with Patterns, Models, Frameworks and Tools*, Wiley Publishing Inc., 2004

[Gurz08] Gurzki, Thorsten: *Portale zwischen Unternehmenskommunikation, Prozessen und IT – Aktueller Stand und strategisches Vorgehen*, In: Tagungsband: 9. Jahresforum Mitarbeiterportale 2008 (Online verfügbar unter: `http://www.gurzki.de/index.php/artikel.html?file=files/vortraege/Gurzki_JFMAP08_Portal_UK_IT.pdf`, Abruf am 19.01.2009), Marcus Evans, Berlin, 2008

[GüKa04] Günterberg, Brigitte; Kayer, Gunter: *SMEs in Germany, Facts and Figures 2004*, Statistisches Bundesamt Bonn, 2004

[HaKn08] Hampp, Tilmann; Knauß, Markus: *Eine Untersuchung über Korrekturkosten von Software-Fehlern*, In: Softwaretechnik-Trends 28:2, Mai 2008.

[Hart06] Hartmann, Deborah: *Interview: Jim Johnson of the Standish Group*, `http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS`, 25.08.2006, Abruf am 16.09.2006

[Have05] Havenstein, Heather: *Fast-moving development: ARAD tools can cut costs – The architected rapid application development tools also boost productivity*, `http://www.computerworld.com/industrytopics/financial/story/0,10801,98762,00.html`, 10.01.2005, Abruf am 19.01.2009

[HeCo01] Heineman, George T.; Councill, William T.: *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Longman, 2001

[Heim02] Heim, Harald: *Plugins Unplugged – Where Have All the Plugins Gone?*, `http://www.graphics.com/modules.php?name=Sections\&op=printpage\&artid=23`, 22.08.2002, Abruf am 18.03.2009

[HeMe02] Heering, Jan; Mernik, Marjan: *Domain-Specific Languages for Software Engineering*, Proceedings of the 35th Hawaii International Conference on System Sciences, 2002, `http://csdl.computer.org/comp/proceedings/hicss/2002/1435/09/14350279.pdf\&ei=pgcWQ6CwKsKYYMfF9OAI`, Abruf am 26.01.2009

[Hirs02] Hirsch, Martin: *Making RUP Agile*, OOPSLA 2002 Practitioner Report, `http://agilealliancebeta.org/system/article/file/1135/file.pdf`, Abruf am 16.08.2006.

[Hons06] Honsel, Gregor: *Aufmerksamkeitskurven – Die Hype-Zyklen neuer Technologien*, `http://www.spiegel.de/netzwelt/tech/0,1518,443717,00.html`, 21.10.2006, Abruf am 01.03.2009

[Hube02] Hubert, R.: *Convergent Architecture. Building Model-Driven J2EE Systems with UML*, John Wiley & Sons, 2002

[IKTN09] IKT.NRW – Cluster Informations- und Kommunikationstechnologie (Hrsg.): *Startseite | IKT.NRW*, `http://www.ikt-nrw.de/`, 2009, Abruf am 21.06.2009

[Info09] Fachbereich Informationswissenschaft der Universität des Saarlandes (Hrsg.): *Open Source und Freie Software*, `http://server02.is.uni-sb.de/courses/wiki/Open_Source_und_Freie_Software`, 12.02.2009, Abruf am 01.03.2009

[InMB06a] Institut für Mittelstandsforschung Bonn (Hrsg.): *Mittelstand – Definition und Schlüsselzahlen*, `http://www.ifm-bonn.org/dienste/definition.htm`, 2006, Abruf am 17.09.2006

[InMB06b] Institut für Mittelstandsforschung Bonn (Hrsg.): *Institut für Mittelstandsforschung Bonn*, `http://www.ifm-bonn.org/`, 2006, Abruf am 17.09.2006

[InSo09] Intentional Software (Hrsg.): *Clippings*, Online verfügbar unter: `http://www.intentsoft.com/news/clippings.html`, Abruf am 20.01.2009

[Intr06] United Planet (Hrsg.), *United Planet - Serienmäßige Vorlagen*, `http://www.unitedplanet.com/de/intrexx-xtreme/vorlagen`, Abruf am 10.01.2009

[Intr09] United Planet (Hrsg.), *United Planet - Portal*, `http://www.intrexx.de`, 2009, Abruf am 20.03.2009

[Iseg05] Iseger, Martijn: *Domain-specific modeling for generative software development*, `http://www.itarchitect.co.uk/articles/display.asp?id=161`, 17.05.2005, Abruf am 25.01.2009

[ISO9241] Wikipedia (Hrsg.): *EN ISO 9241*, `http://de.wikipedia.org/wiki/ISO_9241`, 04.12.2008, Abruf am 03.01.2009

[ITPR07] IT & Production — das Online-Magazin für erfolgreiche Produktion (Hrsg.): *Herausforderung Mittelstand: Harmonisierte, globale Prozesse im Idealfall (Interview mit Andreas Naunin)*, `http://www.it-production.com/index.php?seite=einzel_artikel_ansicht&id=36976`, IT & Production, Ausgabe September 2007

[Jack99] Jackson, M.: *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley, 1999

[Jame50] James, William: *The Principles of Psychology*, Volume 1 & 2, Dover Publications, 1950

[Jone02] Jones, Derek M.: *The 7±2 Urban Legend*, MISRA C Conference, 2002, `http://www.knosof.co.uk/cbook/misart.pdf`, Abruf am 30.12.2008

[Jone03] Jones, Joel: *Abstract Syntax Tree Implementation Idioms*, The 10th Conference on Pattern Languages of Programs 2003, `http://jerry.cs.uiuc.edu/\~plop/plop2003/Papers/Jones-ImplementingASTs.pdf`, 2003, Abruf am 20.01.2009

[JSFC09] JavaServer Faces Community (Hrsg.): *javaserverfaces: JavaServer Faces Community*, `https://javaserverfaces.dev.java.net/`, Abruf am 10.01.2009

[JSR222] Java Community Process (Hrsg.): *JSR 222: Java™ Architecture for XML Binding (JAXB) 2.0*, Abruf am 10.01.2009

[JSR315] Java Community Process (Hrsg.): *JSR 315: Java^{TM} Servlet 3.0 Specification*, Abruf am 10.01.2009

[Kand09] Kandelberg, Alexander: *Entwicklung einer Eclipse-Erweiterung zur Unterstützung der Pflege einer domänenspezifischen Sprache für Portalsysteme*, Diplomarbeit am Lehrstuhl 13: Dienstleistungsinformatik der TU Dortmund, erscheint 2009

[Kech06] Kecher, Christoph: *UML 2.0. Das umfassende Handbuch*, Galileo Press, Bonn, 2006.

[Kent02] Kent, Stuart: *Model driven engineering*, In: IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods. London, UK: Springer-Verlag, S. 286–298, 2002

[KGHV06] Kirchhof, Anja; Gurzki, Thorsten; Hinderer, Henning; Vlachakis, Joannis: *„Was ist ein Portal" – Definition und Einsatz von Unternehmensportalen*, `http://www.gurzki.de/index.php/artikel?file=files/publikationen/Whitepaper+Was+ist+ein+Portal+Gurzki.pdf`, Fraunhofer-Institut für Arbeitswirtschaft und Organisation (IAO), 06/2006

[KiRS04] Kittlaus, Hans-Bernd; Rau, Christoph; Schulz, Jürgen: *Software-Produkt-Management*, Springer, 2004

[KLLe09] Ko, Ryan K. L.; Lee, Stephen S. G.; Lee, Eng Wah: *Business Process Management (BPM) Standards: A Survey*, In: Business Process Management Journal, Emerald Group Publishing Limited. Angenommen am 02.12.2008. Erscheint in Volume 15 Issue 5 (2009).

[KlPi04] Klett, Christian; Pivernetz, Michael: *Controlling in kleinen und mittleren Unternehmen. Ein Handbuch mit Auswertung auf der Basis der Finanzbuchhaltung*, 3. Auflage, NWB Verlag, 2004

[KMAu04] Ko, Andrew J.; Myers, Brad A.; Aung, Htet Htet: *Six Learning Barriers in End-User Programming Systems*, In: Proceedings of IEEE Symposium On VLHCC, 2004, S. 199–206, `http://www.cs.cmu.edu/\~ajko/LearningBarriers.pdf`

[Knut92] Donald E. Knuth: *Literate Programming*, Stanford, California: Center for the Study of Language and Information, CSLI Lecture Notes, no. 27, 1992

[Krau05] Krauss, Timo: *Durchgängige Modellorientierung macht Anwendungsentwicklung produktiver*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1283`, 2005

[Kris98] Kristen, G.: *Kennis is Macht (2nd Edition)*, Academic Service, 1998.

[Kruc04] Kruchten, Philippe: *The Rational Unified Process. An Introduction*, Addison-Wesley, 2004

[Lau04] Lau, Kung-Kiu (Hrsg.): *Component-Based Software Development: Case Studies (Component-Based Software Development)*, World Scientific Pub Co, 2004

[Lebo82] Le Bon, Gustave: *Psychologie der Massen* (im Original von 1895: *Psychologie des foules*), Kröner, 1982

[Lehm08] Lehmann, Frank R.: *Integrierte Prozessmodellieurng mit ARIS*, dpunkt.verlag GmbH, 2008

[Leie06] Leierseder, Wolfgang: *Zurückhaltender Mittelstand*, `http://www.channelpartner.de/produkteundtechnologien/projekte_und_loesungen/642456/index.html`, 15.09.2006, Abruf am 19.11.2006

[LeTu93] Leveson, Nancy G.; Turner, Clark S.: *An investigation of the Therac-25 accidents*, Computer, July 1993, S. 18–41.

[Lien86] Lien, David A.: *The Basic Handbook: Encyclopedia of the BASIC Computer Language (3rd ed. ed.)*, Compusoft Publishing, 1986

[Lim98] Lim, Wayne C.: *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*, Prentice-Hall, 1998

[Lind05] Lindberg, Peter: *Architected Rapid Application Development (ARAD)*, Online verfügbar unter: `http://tesugen.com/archives/05/01/arad`, 26.01.2005, Abruf am 19.01.2009

[LiRa08] Linssen, Oliver; Rachmann, Alexander: *PRINCE2 - ein prozessorientierter Projektmanagementansatz*, In: HMD 260, Praxis der Wirtschaftsinformatik, S. 69–77, dpunkt.verlag, Heidelberg, 2008

[Livs05] Sun Microsystems (Hrsg.): *Envisioning a New Language: A Conversation With Sun Microsystems' Victoria Livschitz*, `http://java.sun.com/developer/technicalArticles/Interviews/livschitz2_qa.html`, 2005, Abruf am 17.09.2005.

[LLSW00] Levine, Rick; Locke, Christopher; Searls, Doc; Weinberger, David: *Das Cluetrain Manifest. 95 Thesen für die neue Unternehmenskultur im digitalen Zeitalter*, Econ, 2000

[Löff08] Löffelmann, Klaus: *Visual Basic 2008 – Das Entwicklerbuch*, Microsoft Press, 2008

[Lübe06] Lübeck, Jürgen: *Alle Jahre wieder: IT Hype Cycle*, online verfügbar unter: `http://blog.juergen-luebeck.de/archives/653-Alle-Jahre-wieder-IT-Hype-Cycle.html`, 10.08.2006, Abruf am 01.03.2009

[LüMS05] Lübke, Daniel; Marx Gómez, Jorge; Schneider, Kurt: *Serviceorientierte Architekturen und Prozessmanagement. Chancen durch die Verbindung zweier Welten*, ERP Management 3/2005, S. 19–22, Homepage: `http://www.erp-management.de`

[MaBu87] Macro, A.; Buxton, J.: *The Craft of Software Engineering*, Addison Wesley, Reading, MA, 1987

[Manc08] Manchester, Phil: *Dip into concept programming – Do what it says on the tin*, http://www.theregister.co.uk/2008/01/16/concept_programming/, 16.01.2008, Abruf am 21.01.2009

[Marg08] Marguerie, Fabrice: *Great fluent interface sample: the fluent repeater*, http://weblogs.asp.net/fmarguerie/archive/2008/04/24/great-fluent-interface-sample-the-fluent-repeater.aspx, 24.04.2008, Abruf am 19.02.2009

[Mart81] Martin, James: *Application Development Without Programmers*, Prentice Hall, 1981

[Mart01] Martin, Robert Cecil: *Preliminary chapter of Object Oriented Analysis and Design with Applications. Chapter 4: The Process*, http://www.objectmentor.com/resources/articles/RUPvsXP.pdf, Abruf am 16.09.2009

[Mart03] Martin, Robert Cecil, *Agile Software Development. Principles, Patterns, and Practices*, Pearson US Imports & PHIPEs, 2003

[McGi96] McGibbon, T.: *Software Reliability Data Summary*, Technical Report, Data Analysis Center for Software, 1996

[Meta09] MetaCase (Hrsg.): *MetaCase – Domain-Specific Modeling with MetaEdit+*, Homepage von MetaCase, http://www.metacase.com/, Abruf am 24.01.2009

[Min68] Minsky, M.: *Semantic Information Processing*, Cambridge, Mass., MIT Press, 1968

[Micr06] Microsoft (Hrsg.): *Mehr Planungssicherheit vonnöten*, http://www.microsoft.com/germany/mittelstand/leaderfellow.mspx?id=500076, Abruf am 19.11.2006

[Micr08] Microsoft (Hrsg.): *Visual Studio 2008 Developer Center*, http://msdn.microsoft.com/de-de/vs2008/default.aspx, Abruf am 10.01.2009

[Micr09a] Microsoft (Hrsg.): *Vorlagen für Arbeitswelt, Privates und Spiele*, http://office.microsoft.com/de-de/templates/FX100595491031.aspx, Abruf am 10.01.2009

[Micro09b] Microsoft (Hrsg.): *Visio-Homepage – Microsoft Office Online*, http://office.microsoft.com/de-de/visio/default.aspx, Abruf am 19.06.2009

[Mill56] Miller, G. A.: *The magical number seven, plus or minus two: Some limits on our capacity for processing information.* Psychological Review, 63, No. 2, 1956, S. 81–97, http://www.musanim.com/miller1956/, Abruf am 30.12.2008

[Mind06] Mindjet (Hrsg.): *Mindjet: Software for Brainstorming and Planning*, http://www.mindjet.com/de/, 2006, Abruf am 07.08.2006

[Misc07] Mischke, Robert: *Fluent Interface vs. Method Chaining vs. DSL*, http://www.speak-friend.com/archive/2007/09/12/90.aspx, 12.09.2007, Abruf am 22.01.2009

[MWME08] Ministerium für Wirtschaft, Mittelstand und Energie des Landes Nordrhein-Westfalen (Hrsg.): *Zur Förderung vorgeschlagene Projekte im Rahmen des Wettbewerbs IKT.NRW*, http://www.ziel2-nrw.de/0_2_Aktuelles/00_Newsmedia/IKT_Siegerliste_2008_11_14.pdf, 14.11.2008, Abruf am 21.06.2009

[Myer86] Myers, Ware: *Can Software for the Strategic Defense Initiative ever be error-free?*, In: Computer Volume 19, Issue 11, November 1986, S. 61–67

[Navr96] Návrat, P.: *A closer look at prgramming expertise: critical survey of some methodological issues*, in: *Information and Software Technology*, no. 38, 1996, Elsevier, S. 37–46

[Nebu09] Nebulon Pty. Ltd. (Hrsg.): *The Latest FDD Process*, Print Version 1.3, http://www.nebulon.com/articles/fdd/download/fddprocessesA4.pdf, Abruf am 29.01.2009

[Newe90] Newell, A.: *Unified Theory of Cognition*, Harvard University Press, Cambridge, MA, 1990

[Nöll06] Nöllke, Matthias: *Kreativitätstechniken*, Haufe, 2006

[OAW] http://www.openarchitectureware.org/, Abruf am 06.10.2006

[Obje09] ObjectX Software Development (Hrsg.): *Portal Builder*, http://www.objectx.com.mk/Portal_Builder.html, 2009, Abruf am 21.06.2009

[OBri05] O'Brien, Timothy M.: *Jakarta Commons Cookbook*, O'Reilly Media, Inc., 2005

[OMG01] OMG Architecture Board ORMSC (Hrsg.): *Model driven architecture (MDA)*, OMG document number ormsc/2001-07-01, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, Juli 2001, Abruf am 28.01.2009

[OMG06a] Object Management Group (Hrsg.): *MDA*, http://www.omg.org/mda, 2006, Abruf am 02.09.2006

[OMG06b] Object Management Group (Hrsg.): *UML*, http://www.uml.org/, 2006, Abruf am 02.09.2006

[OMG06c] Object Management Group (Hrsg.): *OMG's MetaObject Facility (MOF) Home Page* http://www.omg.org/mof/, 2006, Abruf am 2006-09-17

[OMG09] Object Management Group (Hrsg.): *Catalog of OMG Modeling and Metadata Specifications*, http://www.omg.org/technology/documents/modeling_spec_catalog.htm, 2009, Abruf am 28.01.2009

[Open09] OpenSymphony (Hrsg.): *Quartz – Quartz Overview*, `http://www.opensymphony.com/quartz/`, 06.03.2009, Abruf am 27.07.2009

[OrHa98] Orfali, Robert; Harkey, Dan: *Client/Server Programming with Java and CORBA*, John Wiley & Sons, 1998

[PaFe02] Palmer, Stephen R.; Felsing, John M.: *A Practical Guide to the Feature-Driven Development*, Prentice Hall International, 2002

[Parn72] Parnas, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules*, In: Communications of the ACM 15 (1972) 12, S. 1052–1058

[PCHa93] Poulin, Jeffrey S.; Caruso, Joseph M.; Hancock, Debera R.: *The business case for software reuse*, IBM Systems Journal, Volume 32, Issue 4 (October 1993), S. 567–594

[Pete87] Peters, Tom: *Thriving on Chaos: Handbook for a Management Revolution*, Knopf, New York, 1987

[PfHa91] Pfeiffer, H.D.; Hartley, R.T.: *The conceptual programming environment, CP: reasoning representation using graph structures and operations*, In: Proceedings of IEEE Workshop on Visual Languages, Kobe, Japan, 1991.

[Pont07] Pontin, Jason: *Awaiting the Day When Everyone Writes Software*, New York Times, `http://www.nytimes.com/2007/01/28/business/yourmoney/28slip.html?ex=1327640400&en=d2d098d765b27104&ei=5088&partner=rssnyt&emc=rs`, 28.01.2007, Abruf am 19.01.2009

[Popp03] Poppendieck, Mary: *Lean Development Software. An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003

[Pößn05] Pößneck, Lutz: *Projektmanagement. IT-Projekte: Wege aus dem Chaos*, `http://www.silicon.de/enid/it_services/13078`, 02.05.2005, Abruf am 19.11.2006

[Pres09] Presser, Gero: *Business-to-Business Portale in der Praxis. Planung und Umsetzung von Kunden- und Partnerportalen*, United Planet Books, Freiburg, 2009

[Pusc04] Puschmann, Thomas: *Prozessportale – Architektur zur Vernetzung mit Kunden und Lieferanten*, Springer Verlag, Berlin, 2004

[Qi4j08] Qi4j Team (Hrsg.): *www.qi4j.org*, `http://www.qi4j.org`, Abruf am 29.12.2008

[Quin09] QuinScape GmbH (Hrsg.): *Homepage der QuinScape GmbH*, `http://www.quinscape.de`, Abruf am 20.03.2009

[RaBr06a] Rausch, Andreas; Broy, Manfred: *Das V-Modell XT - Grundlagen, Erfahrungen und Werkzeuge*, dpunkt.verlag, Heidelberg, 2006

[RaBr06b] Rausch, Andreas; Broy, Manfred: *Das V-Modell XT. Grundlagen, Methodik und Anwendungen*. Springer, Heidelberg, 2006

[RedH09] Red Hat Middleware (Hrsg.): *hibernate.org - Hibernate*, `https://www.hibernate.org/`, 2009, Abruf am 12.06.2009

[Reis87] Reiss, Steven P.: *A conceptual programming environment*, Proceedings of the 9th international conference on Software Engineering, März 1987, Monterey, California, United States, S. 225–235

[Reis89] Reiss, Steven P.: *Conceptual programming*, Proceedings of the 5th international software process workshop on Experience with software process models, Kennebunkport, Maine, United States, 10.–13. Oktober 1989, S. 121–123

[Reis09] Reiss, Steven P.: *Steven P. Reiss Home Page*, `http://www.cs.brown.edu/~spr/`, Abruf am 11.01.2009

[Resi09] Resig, John: *jQuery: The Write Less, Do More, JavaScript Library*, `http://jquery.com`, 2009, Abruf am 12.06.2009

[Reyn08] Reynolds, Garr: *presentation zen – Simple Ideas on Presentation Design and Delivery*, New Riders, 2008, S. 130

[Rich06] Richter, Boris: *Vorbereitung und Durchführung einer Umfrage zur Untersuchung des ERP-KMU-Dilemmas*, Individuelles Projekt, Department für Informatik, Carl-von-Ossietzky-Universität, Oldenburg, 2006

[Ried97] Riedemann, Eike: *Testmethoden für sequentielle und nebenläufige Softwaresysteme*, B. G. Teubner. Stuttgart, 1997

[RiVa03] Richards, Lauren (Hrsg.); Varia, Kewal (Hrsg.): *Compuware: Landmark Study Reveals 35 per cent Productivity Gains for Businesses Using Model-Driven Architecture*, `http://www.compuware.co.uk/pressroom/news/21072003_02.htm`, 21.07.2003, Abruf am 24.01.2009

[RoMo02] Rosenfeld, Louis; Morville, Peter: *Information Architecture for the World Wide Web, Second Edition*, O'Reilly & Associates, 2002

[RoRo99] Robertson, S.; Roberston J.: *Mastering the Requirements Process*, Addison-Wesley, New York, 1999

[Rose07] Rosenberg, Scott: *Part I: Anything You Can Do, I Can Do Meta*, Online verfügbar unter: `http://www.technologyreview.com/computing/17969/page1/`, 08.01.2007, Abruf am 19.01.2009

[Royc70] Royce, W. W.: *Managing the Development of Large Software Systems: Concepts and Techniques*, Proc. IEEE WESCON, 1970, S. 1–9

[Rupp07] Rupp, Chris: *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*, Hanser Fachbuchverlag, 2007

[SaCu03] Sauer, Chris; Cuthbertson, Christine: *The State of IT Project Management in the UK 2002–2003*, `http://www.computerweeklyms.com/pmsurveyresults/surveyresults.pdf`, Abruf am 17.09.2006

[Samm69] Sammet, Jean E.: *Programming Languages: History and Fundamentals*, Englewood Cliffs, N. J., Prentice-Hall, 1969

[SBBB02] Shull, Forrest; Basili, Vic; Boehm, Barry A.; Brown, Winsor; Costa, Patricia; Lindvall, Mikael; Port, Dan; Rus, Ioana; Tesoriero, Roseanne; Zelkowitz, Marvin: *What We Have Learned about Fighting Defects*, Proceedings, Metrics 2002, Juni 2002, S. 249–258

[ScJo02] Scheer, August-Wilhelm; Jost, Wolfram: *ARIS in der Praxis*, Springer, 2002

[Sche94] Schenk, Jeffrey D.: *Novell's Guide to Implementing Client Applications and Architecture*, Sybex Inc., 1994

[Schi02] Schienmann, B.: *Kontinuierliches Anforderungsmanagement*, Addison-Wesley, München, 2002

[Schm05] Schmid, Alexander: *Programmiersprachenbasierte DSLs*, In: OBJEKTspektrum Online 2005 – die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1284`, 2005

[Schm06] Schmidt, Douglas C.: *Model-Driven Engineering*, IEEE Computer 39 (2), Februar 2006, `http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf`, Abruf am 27.01.2009

[ScLK05] Schmitz, Volker; Leukel, Jörg; Kelkar, Oliver: *Spezifikation BMEcat® 2005*, online verfügbar im anmeldepflichtigen Downloadbereich unter `http://www.bmecat.org`, BME e. V., 2005

[Seeg05] Seegmüller, Kirsten: *SAP-Modulwissen allein reicht nicht mehr aus*, In: Computer Zeitung (2005) 44, S. 14

[SEI09] Software Engineering Institute: *Software Product Line Case Studies*, Carnegie Mellon, `http://www.sei.cmu.edu/productlines/spl_case_studies.html`, Abruf am 06.01.2009

[Shal01] Shalloway, Allan, *Xtreme RUP: Lightening Up the Rational Unified Process*, `http://www.netobjectives.com/download/xrup.pdf`, 2001-02-09, Abruf am 16.09.2006

[Simo95] Simonyi, Charles: *The Death of Computer Languages, The Birth of Intentional Programming*, Technical Report MSR-TR-95-52, Online verfügbar unter: `ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc`, Microsoft Research, 1995

[Simo96] Simonyi, Charles: *Intentional Programming – Innovation in the Legacy Age*, Position Paper, IFIP WG 2.1 meeting, 1996

[Simo97] Simonyi, Charles: *Intentional Programming*, Usenix Conference on Domain Specific Languages, Santa Barbara, CA, USA, 1997

[SmMe81] Smith, E. E.; Medin, D. L.: *Categories and Concepts*, Harvard University Press, Cambridge, MA, 1981

[Soft08] Softline: *ERP - Enterprise Resource Planing Marktübersicht*, `http://www.softguide.de/software/erp.htm`, 2008, Abruf am 29.12.2008

[Soft08] Softwaregenerators.com: *The Anticipatory Optimization Generator (AOG): Recent Work*, `http://softwaregenerators.com/RecentWork.html`, Abruf am 31.12.2008

[Spri09a] SpringSource (Hrsg.): *SpringSource Community*, `http://www.springsource.org/`, Abruf am 10.01.2009

[Spri09b] SpringSource (Hrsg.): *Spring Web Flow 2*, `http://www.springsource.org/webflow`, Abruf am 10.01.2009

[Spri09c] SpringSource (Hrsg.): *Spring Web MVC*, `http://static.springframework.org/spring/docs/2.0.x/reference/mvc.html`, 2009, Abruf am 12.06.2009

[SPC97] Software Productivity Consortium: *Evolutionary Rapid Development. SPC document SPC-97057-CMC*, version 01.00.04, Juni 1997

[Stan03a] The Standish Group (Hrsg.): *Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved By 50%*, `http://www.standishgroup.com/press/article.php?id=2`, Abruf am 30.09.2005

[Stan03b] The Standish Group (Hrsg.): *Chaos Chronicles Version 3*, `http://www.standishgroup.com/chaos/index.php`, Abruf am 30.09.2005

[Stap02] Stapleton, Jennifer: *DSDM: Business Focused Development*, Pearson Addison-Wesley, 2002

[Stee02] Steeb (Hrsg.): *Internationale Projekte erfolgreich meistern — Auswirkungen der Globalisierung auf die IT im Mittelstand*, In: Mittelstandsmagazin „Lösung“, Ausgabe 3/2002, Steeb, S. 8–9, `http://www.steeb.de/_global/popup../infocenter/loesung/03_2002/loesung_03_2002_08.pdf`, Abruf am 19.11.2006

[StVö05] Stahl, Thomas; Völter, Markus: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*, Dpunkt Verlag, 2005

[SunM09a] Sun Microsystems (Hrsg.): *JavaServer Pages Technology*, `http://java.sun.com/products/jsp/`, 2009, Abruf am 12.06.2009

[SunM09b] Sun Microsystems (Hrsg.): *Developer Resources for Java Technology*, `http://java.sun.com/`, 2009, Abruf am 12.06.2009

[SunM09c] Sun Microsystems (Hrsg.): *Java Transaction API (JTA)*, `http://java.sun.com/javaee/technologies/jta/index.jsp`, 2009, Abruf am 12.06.2009

[SWC&95] Stillings, N. A.; Weisler, S. E.; Chase, C. H.; Feinstein, M. H.; Garfield, J. L.; Rissland, E. L.: *Cognitive Science: An Introduction*, Second Edition, The MIT Press, Cambridge, MA, 1995

[Sysm09] SysML Forum (Hrsg.): *SysML Forum - SysML Tools, Methods, Training & Mailing Lists*, `http://www.sysml.org`, Abruf am 22.02.2009

[Tayl82] Taylor, David A.: *Mind: The Human Difference*, Simon & Schuster, New York, 1982

[Tayl95] Taylor, David A.: *Business Engineering with Object Technology*, John Wiley & Sons, Inc., 1995

[Tecd09] TecDoc Informations System GmbH (Hrsg.): *Tecdoc Portal*, `http://www.tecdoc.de`, Abruf am 13.03.2009

[Tech05] TechConsult GmbH (Hrsg.): *IT im Mittelstand. Sicherung von Marktanteilen unter gestiegenem Wettbewerbsdruck. Deutschland 2005–2007*, `http://www.techconsult.de/download/pdf/inhaltsverzeichnis_mittelstandsstudie_2005.pdf`, Veröffentlicht im Februar 2006, Abruf am 15.10.2006

[Ther05] Therré, Peter: *Agile Softwarefabrik mittels Model Driven Development*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1285`, 2005

[Tolv05] Tolvanen, Juha-Pekka: *Domain-Specific Modeling for Full Code Generation*, In: METHODS & TOOLS, Global knowledge source for software development professionals, Fall 2005, Volume 13, Number 3, `http://www.methodsandtools.com/archive/archive.php?id=26`, 2005, Abruf am 24.01.2009

[TKLM03] Theis, Fabian (Hrsg.); Kuhn, Stefan; Langham, Matthew; Müller, Jürgen; Wang, Dapeng; Ziegeler, Carsten: *Portale und Webapplikationen mit Apache Frameworks*, Software & Support Verlag GmbH, 2003

[Trat06] Tratar, Erwin: *Anforderungen erfassen, aber mit System: Modellbasiertes Requirements Engineering*, In: OBJEKTspektrum – die Zeitschrift für Software-Engineering und -Management, November/Dezember 2006, Nr. 6, S. 29–31

[TUDO09] Lehrstuhl für Dienstleistungsinformatik der Technischen Universität Dortmund (Hrsg.): *Lehrstuhl Dienstleistungsinformatik / e-Services Research Group / Prof. Dietmar Jannach*, `http://ls13-www.cs.uni-dortmund.de/homepage/index.shtml`, 03.05.2009, Abruf am 10.07.2009

[Tuft03] Tufte, Edward T.: *The magical number seven, plus or minus two: Not relevant for design*, `http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0000U6&topic_id=1`, 20.04.2003, Abruf am 30.12.2008

[Uocd97] University of California, Davis (Hrsg.): *Rapid Application Development (RAD)*, `http://sysdev.ucdavis.edu/WEBADM/document/rad_toc.htm`, 16.04.1997, Abruf am 22.01.2009

[Vers00] Versteegen, Gerhard: *Projektmanagement mit dem Rational Unified Process*, Springer, Berlin, 2000.

[VMF09] Visual Modeling Forum (Hrsg.): *Domain-Specific Modeling*, `http://www.visualmodeling.com/DSM.htm`, Abruf am 24.01.2009

[VöSt05] Völter, Markus; Stahl, Thomas: *OBJEKTspektrum Online-Ausgabe. Thema: Model-Driven Development. Editorial*, In: OBJEKTspektrum Online 2005 – die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1277`, 2005

[Ward94] Ward, Martin: *Language Oriented Programming*, Software – Concepts and Tools, Vol. 15, No. 4, S. 147–161, 1994

[Wass96] Wassermann, Anthony I.: *Toward a discipline of software engineering*, IEEE Software, November 1996, Volume 13, Issue 6, S. 23–31

[Wein65] Weinberg, Gerald M.: *Experiments in Problem Solving*, Doctoral Thesis, Ann Arbor, University Microfilms, 1965, `http://www.satisfice.com/articles/weinberg.pdf` (Abruf am 30.12.2008)

[Wein71] Weinberg, Gerald M.: *The Psychology of Computer Programming*, Dorset Ho. Pub. Co., 1971

[Wein98] Weinberg, Gerald M.: *The Psychology of Computer Programming (Silver Anniversary)*, Dorset Ho. Pub. Co., 1998

[Wege99] Wegener, Ingo: *Theoretische Informatik: eine algorithmenorientierte Einführung*, Teubner, Stuttgart, 1999

[WeKS01] Wermke, Matthias (Hrsg.); Kunkel-Razum, Kathrin (Hrsg.); Scholze-Stubenrecht, Werner (Hrsg.): *Duden: Herkunftswörterbuch, Etymologie der deutschen Sprache, 3., völlig neu bearbeitete und erweiterte Auflage*, Dudenverlag, 2001, Mannheim

[WeWe05] Welz, Stephan; Weise, Dirk: *Fünf Best Practices für Model Driven Architecture*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1286`, 2005

[Whit95] Whitenack, Bruce G.: *RAPPeL: A Requirements Analysis Process Pattern Language for Object-Oriented Development*, In: Coplien, James; Schmidt, Doug: *Pattern Languages of Program Design*, 1995, S. 259–291

[Wiki06a] Wikipedia (Hrsg.): *Service Engineering*, `http://de.wikipedia.org/wiki/Service_Engineering`, 23.07.2006, Abruf am 07.08.2006

[Wiki06b] Wikipedia (Hrsg.): *Model Driven Architecture*, `http://de.wikipedia.org/wiki/Model_Driven_Architecture`, 2006, Abruf am 02.09.2006

[Wiki06c] Wikipedia (Hrsg.): *Unified Modeling Language*, `http://de.wikipedia.org/wiki/Unified_Modeling_Language`, 2006, Abruf am 02.09.2006

[Wiki06d] Wikipedia (Hrsg.), *Portal: Unified Modeling Language (UML)*, `http://de.wikipedia.org/wiki/Portal:Unified_Modeling_Language`, 2006, Abruf am 02.09.2006

[Wiki06e] Wikipedia (Hrsg.): *Wirtschaftsinformatik*, `http://de.wikipedia.org/wiki/Wirtschaftsinformatik`, 04.08.2006, Abruf am 07.08.2006

[Wiki06f] Wikipedia (Hrsg.): *Meta-Object Facility*, `http://de.wikipedia.org/wiki/Meta-Object_Facility`, 22.06.2006, Abruf am 2006-09-17

[Wiki06g] Wikipedia (Hrsg.): *Skaleneffekt*, `http://de.wikipedia.org/wiki/Skaleneffekt`, 30.07.2006, Abruf am 2006-09-17

[Wiki06h] Wikipedia (Hrsg.): *Verbundeffekt*, `http://de.wikipedia.org/wiki/Verbundeffekt`, 12.07.2006, Abruf am 2006-09-17

[Wiki06i] Wikipedia (Hrsg.): *Economies of scope*, `http://en.wikipedia.org/wiki/Economies_of_scope`, 24.07.2006, Abruf am 2006-09-17

[Wiki06j] Wikipedia (Hrsg.): *Returns to scale*, `http://en.wikipedia.org/wiki/Returns_to_scale`, 15.09.2006, Abruf am 2006-09-17

[Wiki06k] Wikipedia (Hrsg.): *Agile software development*, `http://en.wikipedia.org/wiki/Agile_software_development`, 08.10.2006, Abruf am 15.10.2006

[Wiki06l] Wikipedia (Hrsg.): *Wirtschaftlichkeit*, `http://de.wikipedia.org/wiki/Wirtschaftlichkeit`, 12.10.2006, Abruf am 21.10.2006

[Wiki06m] Wikipedia (Hrsg.): *Return on Investment*, `http://de.wikipedia.org/wiki/Return_on_Investment`, 20.10.2006, Abruf am 22.10.2006

[Wiki06n] Wikipedia (Hrsg.): *HTML*, `http://en.wikipedia.org/wiki/HTML`, 01.12.2006, Abruf am 02.12.2006

[Wiki07] Wikipedia (Hrsg.): *Prozessportal*, `http://de.wikipedia.org/wiki/Prozessportal`, 02.10.2007, Abruf am 19.01.2009

[Wiki08a] Wikipedia (Hrsg.): *V-Modell*, `http://de.wikipedia.org/wiki/V-Modell`, 12.12.2008, Abruf am 30.12.2008

[Wiki08b] Wikipedia (Hrsg.): *Software prototyping*, `http://en.wikipedia.org/wiki/Software_prototyping`, 28.12.208, Abruf am 02.01.2009

[Wiki08c] Wikipedia (Hrsg.): *Pareto-Verteilung*, `http://de.wikipedia.org/wiki/Pareto-Prinzip`, 23.12.2008, Abruf am 06.01.2009

[Wiki08d] Wikipedia (Hrsg.): *Scope creep*, `http://en.wikipedia.org/wiki/Feature_Creep`, 23.11.2008, Abruf am 06.01.2009

[Wiki08e] Wikipedia (Hrsg.): *Archetyp*, `http://de.wikipedia.org/wiki/Archetyp`, 22.12.2008, Abruf am 06.01.2009

[Wiki08f] Wikipedia (Hrsg.): *Lotus 1-2-3*, `http://en.wikipedia.org/wiki/Lotus_123`, 18.11.2008, Abruf am 10.01.2009

[Wiki08g] Wikipedia (Hrsg.): *Drill-Down*, http://de.wikipedia.org/wiki/Drilldown, 12.04.2008, Abruf am 17.01.2009

[Wiki08h] Wikipedia (Hrsg.): *Portal (Informatik)*, http://de.wikipedia.org/wiki/Unternehmensportal, 27.10.2008, Abruf am 19.01.2009

[Wiki08i] Wikipedia (Hrsg.): *Semantic-oriented programming*, http://en.wikipedia.org/wiki/Semantic-oriented_programming, 11.11.2008, Abruf am 21.01.2009

[Wiki08j] Wikipedia (Hrsg.): *4GL*, http://de.wikipedia.org/wiki/4GL, 29.10.2008, Abruf am 26.01.2009

[Wiki08k] Wikipedia (Hrsg.): *Computer-aided software engineering*, http://en.wikipedia.org/wiki/Computer-aided_software_engineering, 30.12.2008, Abruf am 27.01.2009

[Wiki08l] Wikipedia (Hrsg.): *Feature Driven Development*, http://de.wikipedia.org/wiki/Feature_Driven_Development, 10.11.2008, Abruf am 29.01.2009

[Wiki08m] Wikipedia (Hrsg.): *Hype-Zyklus*, http://de.wikipedia.org/wiki/Hype-Zyklus, 29.10.2008, Abruf am 01.03.2009

[Wiki08n] Wikipedia (Hrsg.): *Martin-Notation*, http://de.wikipedia.org/wiki/Martin-Notation, 13.11.2008, Abruf am 06.04.2009

[Wiki09a] Wikipedia (Hrsg.): *Microsoft Access*, http://en.wikipedia.org/wiki/Microsoft_Access, 10.01.2009, Abruf am 10.01.2009

[Wiki09b] Wikipedia (Hrsg.): *Microsoft Excel*, http://en.wikipedia.org/wiki/Microsoft_excel, 09.01.2009, Abruf am 10.01.2009

[Wiki09c] Wikipedia (Hrsg.): *Brotkrümelnavigation*, http://de.wikipedia.org/wiki/Breadcrumb, 03.01.2009, Abruf am 17.01.2009

[Wiki09d] Wikipedia (Hrsg.): *Visual programming languages*, http://en.wikipedia.org/wiki/Visual_programming_language, 20.01.2009, Abruf am 22.01.2009

[Wiki09e] Wikipedia (Hrsg.): *Computer-Aided Software Engineering*, http://de.wikipedia.org/wiki/Computer-Aided_Software_Engineering, 15.01.2009, Abruf am 27.01.2009

[Wiki09f] Wikipedia (Hrsg.): *Business process modeling*, http://en.wikipedia.org/wiki/Business_process_modelling, 24.01.2009, Abruf am 27.07.2009

[Wiki09g] Wikipedia (Hrsg.): *ASP.NET*, http://de.wikipedia.org/wiki/ASP.NET, 11.03.2009, Abruf am 13.03.2009

[Wiki09h] Wikipedia (Hrsg.): *Wiki*, http://en.wikipedia.org/wiki/Wiki, 17.03.2009, Abruf am 18.03.2009

[Wiki09i] Wikipedia (Hrsg.): *Borland Delphi*, http://de.wikipedia.org/wiki/Borland_Delphi, 20.03.2009, Abruf am 20.03.2009

[Wiki09j] Wikipedia (Hrsg.): *Data Access Object*, `http://de.wikipedia.org/wiki/Data_Access_Object`, 03.02.2009, Abruf am 24.03.2009

[Wiki09k] Wikipedia (Hrsg.): *Blog*, `http://de.wikipedia.org/wiki/Blog`, 23.03.2009, Abruf am 25.03.2009

[Wiki09l] Wikipedia (Hrsg.): *Entity-Relationship-Modell*, `http://de.wikipedia.org/wiki/Entity-Relationship-Modell`, 03.03.2009, Abruf am 06.04.2009

[Wiki09m] Wikipedia (Hrsg.): *Assoziogramm*, `http://de.wikipedia.org/wiki/Assoziogramm`, 06.06.2009, Abruf am 12.06.2009

[Wiki09n] Wikipedia (Hrsg.): *Representational State Transfer*, `http://en.wikipedia.org/wiki/Representational_State_Transfer`, 10.06.2009, Abruf am 12.06.2009

[Wiki09o] Wikipedia (Hrsg.): *Massively Multiplayer Online Role-Playing Game – Wikipedia*, `http://de.wikipedia.org/wiki/Mmorpg`, 22.04.2009, Abruf am 19.06.2009

[Wiki09p] Wikipedia (Hrsg.): *Second Life – Wikipedia*, `http://de.wikipedia.org/wiki/Second_Life`, 18.06.2009, Abruf am 19.06.2009

[Wiki09q] Wikipedia (Hrsg.): *RSS – Wikipedia*, `http://en.wikipedia.org/wiki/RSS`, 06.07.2009, Abruf am 27.07.2009

[Wiki09r] Wikipedia (Hrsg.): *Konzept* `http://de.wikipedia.org/wiki/Konzept`, 30.10.2009, Abruf am 15.11.2009

[Wikt06] Wiktionary (Hrsg.): *Archetyp*, `http://de.wiktionary.org/wiki/Archetyp`, 01.12.2006, Abruf am 06.01.2009

[Wikt09] Wiktionary (Hrsg.): *Konzept*, `http://de.wiktionary.org/wiki/Konzept`, 12.09.2009, Abruf am 15.11.2009

[With96] Withey, James: *Investment Analysis of Software Assets for Product Lines*, CMU/SEI-96-TR-010, 1996

[Wodo09] WODOMI Project Team (Hrsg.): *Workflow and Document Management Infrastructure towards Integrated University*, `http://wodomi.ii.edu.mk/`, 2009, Abruf am 12.06.2009

[Wolf09] Wolff, Eberhard: *REST mit Spring 3*, `http://jandiandme.blogspot.com/2009/01/rest-mit-spring-3.html`, 04.01.2009, Abruf am 12.06.2009

[W3C06] World Wide Web Consortium (Hrsg.): *Extensible Markup Language (XML) 1.1 (Second Edition)*, `http://www.w3.org/TR/xml11/`, 16.08.2006, Abruf am 10.01.2009

[W3C07] World Wide Web Consortium (Hrsg.): *XSL Transformations (XSLT) Version 2.0*, `http://www.w3.org/TR/xslt20/`, 23.01.2007, Abruf am 10.01.2009

[W3C08a] World Wide Web Consortium (Hrsg.): *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, `http://www.w3.org/TR/xmlschema11-1/`, 20.06.2008, Abruf am 10.01.2009

[W3C08b] World Wide Web Consortium (Hrsg.): *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*, `http://www.w3.org/TR/xmlschema11-2/`, 20.06.2008, Abruf am 10.01.2009

[Yaho09] Yahoo (Hrsg.): *Yahoo Developer Network: Exceptional Performance*, `http://developer.yahoo.com/performance/`, 2009, Abruf am 25.05.2009

[Zipf49] Zipf, G. K.: *Human Behaviour and the Principle of Least Effort*, Cambridge, Mass., Addison-Wesley, 1949

[Zoll05] Zollinger, Philip: *Embedded Systems: Produktivitätssteigerung mit Model Driven Development*, In: OBJEKTspektrum Online 2005 - die Zeitschrift für Software-Engineering und -Management, `http://www.sigs-datacom.de/sd/news/document.php?PID=1287`, 2005, Abruf am 29.09.2005

Index

4GL, 76, 77

Abstraktion, 73
AC-MDSD, 104
Adaption, 237, 289
Aggregat, 92
Agile Alliance, 29, 40
Agile Modeling, 118, 120, 138, 208, 246, 271
Agile Vorgehensmodelle, 129
Agilität, 40, 47, 71, 88, 121, 138, 155, 203, 243, 244, 246
AM, *siehe* Agile Modeling
Anforderungsmanagement, 25
Angemessenheit des Entwicklungsprozesses, 74
Anwendungshürden, 75
Application Engineering, 84
ARAD, 82, 292
Archetyp, 203
  PIKAS, 323
    Aufgabenverwaltung, 328
    Blog, 327
    Dokumentenliste, 326
    FAQ, 325
    Formularbasierte Anwendung, 324
    Forum, 326
    Kalender, 328
    Kombinationsanwendung, 330
    Ressourcen, 329
    Wiki, 329
Archetypisierung, 203
Architectured Rapid Application Development, *siehe* ARAD
Architekturstil, 60, 61, 74, 135, 169, 177, 199
ARIS, 116
Ausführbare Prototypen, 125, 137, 196, 233, 262
Auswahlhürden, 75
Automatisierung, 36, 47
Automatisierung von Infrastruktur, 82, 137, 154, 164, 229, 230

BALVI GmbH, 315
BALVI iPi, 401
Bandwidth, 58
Booch, 129
BPM, *siehe* Business Process Modeling
Business Process Modeling, 116

CA, *siehe* Convergent Architecture
Capability Maturity Model for Software, 128
Capability Maturity Model Integration, 128
CASE, 77
Catalysis, 129
CDPM, 8, 40, 43, 135, 241
  Überblick, 245
  Adaption, 289
  Anforderungsmanagement, 286
  Beispiel, 371
  Entwurfssystematik, 158
  Ergebnisse, 397
  Makroebene, 264
  Methoden
    Agiles Glossar, 277
    Analyse durch Design, 269
    Conceptual Pair Programming, 273
    CP-Coaching, 267
    Lebendes Angebot, 276
    Philosophischer Projekt-Kickoff, 265

Rapid Development, 271
Refactoring, 276
Mikroebene, 262
Prinzipien
Bedarfsorientierung, 258
CMDSD-basiertes Vorgehen, 256
Direktheit, 257
Ehrlichkeit und Offenheit, 252
Explizivität und Transparenz, 257
Fairness, 252
Kompetenz statt Komplexität, 255
Konsequenz, 256
Min-Max, 261
Respekt, 250
Start small, then grow, 261
Uniformität, 254
Rollen, 279
CMDSD-Experte, 281
Domänenexperte, 280
Mediator, 282
Protokollführer, 282
Teamorganisation, 279
CE, *siehe* Convergent Engineering
Chefarchitekt, 131
Chefprogrammierer, 131
Chief Architect, 131
Chief Programmer, 80, 131, 171
CIM, 96
Cleanroom, 128
CMDSD, 8, 40, 43, 135, 163
Überblick, 168
Adaption, 237
Beispiel, 311
Entwurfssystematik, 158
Ergebnisse, 375
Methoden
Archetypisierung, 203
Dekomposition von Komplexität, 224
Differenziere deine Lösung, 231
Domänenabgrenzung und Modellbildung durch Domain Driven Design, 211
Entwicklung eines Architekturstils, 198
Entwicklungsleitung durch einen Chief Programmer, 198
Erhaltung der Migrationsfähigkeit, 231
Intention statt Imperativ, 228
Kapselung von Technologien, 230
Komplexitätsbewertung durch das Pareto-Prinzip, 209
MDSD-basierte Software-Produktlinien, 199
Realisiere wissenskonforme Zugänglichkeit, 232
Strukturierte Modellbildung, 213
Unternehmenskonformes Projektmanagementmodell, 202
Werkzeuginhärente Archetypen, 227
Nutzen, 379
PIKAS, 310
Prinzipien
Abstraktion, 181
Inkrementelle Evolution, 195
Kompaktheit, 180
Lokalität und Linearität, 190
Problemorientierung, 175
Rapid Prototyping, 195
Standardkonformität, 178
Tradition und Innovation, 191
Uniformität, 177
Vereinfachung, 174
Variabilität, 235
Werkzeugkosten, 377
Ziele, 164
CMMI, 128
Common Warehouse Metamodel, *siehe* CWM
Computation Independent Model, *siehe* CIM
Computer Aided Software Engineering, *siehe* CASE
Concept Driven Project Management, *siehe* CDPM
Concept Programming, 58, 299

Conceptual Model Driven Software Development, *siehe* CMDSD
Conceptual Programming, *siehe* CP
Conceptual Programming (CRL), 305
Conceptual Programming (Reiss), 302
Convergent Architecture, 60, 295
Convergent Engineering, 49, 108, 216, 220, 293
ConversationContext, 337
Core Asset Development, 84
CP, 8, 40, 140
  Adaption, 160
  Einordnung in die Wirtschaftsinformatik, 411
  Herleitung, 135
  Methoden, 143, 155
    Concept Driven Project Management (CDPM), 157
    Conceptual Model Driven Software Development (CMDSD), 156
    Produktgremium, 157
  Prinzipien, 142, 152
    Agilität, 154
    Automatisierung, 153
    Domänenfokussierung, 152
    Kundenorientierung, 152
  Validierung, 159
  Werte, 142, 145
    Flexibilität, 148
    Wirtschaftlichkeit, 146
    Zugänglichkeit, 150
CWM, 95

Databased Publishing, 404
Datenbasiertes Publizieren, 404
Datenkontext, 338
  ConversationContext, 337
  DomainContext, 337
  ProcessContext, 337
  SessionContext, 337
  ViewContext, 337
DDD, 87, 165, 297
  Aggregat, 92
  Entity, 91
  Factory, 92
  Layered Architecture, 91
  Modul, 91
  Package, 91
  Repository, 92
  Service, 91
  Smart UI, 91
  Strategic Design, 92
  Value Object, 91
Designhürden, 74
Domänenanalyse, 90
Domänenfokussierung, 9, 28, 31, 47, 54, 75, 77
Domänenmodell, 88
Domänenmodell als Handwerkszeug, 90
Domain Driven Design, 78, *siehe* DDD
Domain Driven Design als Konstruktionsschema, 93, 111, 120, 136, 170, 201, 211, 213, 217, 276
Domain Engineering, 84
Domain Engineering und Application Engineering, 84
Domain Specific Languages, *siehe* DSL
Domain-Specific Modeling, 38, 102, 112
DomainContext, 337
DomainType, 351
DomainTypePropertyModel, 351
DSDM, 131
DSL, 69, 79, 94
DSM, *siehe* Domain-Specific Modeling
Durchgängiges Domänenmodell, 89
Dynamic Systems Development Method, 131

Economies of Scale, 24, 39
Economies of Scope, 24, 39, 84, 86
Eng fokussierte Software-Produktlinien, 87, 136, 140, 155, 165, 199
Enge Kommunikation, 50
Enger Rahmen, 115, 136, 214, 233, 248, 287
Entity, 91
Entropie, 65
Entwicklungsprozess, 64

Entwicklungsstrukturen, 63
Entwurfsmuster, 86
Entwurfsparadoxon, 65
Entwurfssystematik, 158
EnumPropertyModel, 352
Erfolgsfaktoren für IT-Projekte, 20
ERP-KMU-Dilemma, 19
Erweiterbarkeit, 55, 137, 171, 207, 218, 234
Erweiterungspunkt, 207, **220**
Exklusive Domänenmodellierung, 114, 137, 140, 207, 209, 212, 261
Extreme Programming, 41, 119, 130

Föderierte ERP-Systeme, 405
Förderung kreativer Organisationen, 66, 138, 196, 212, 244, 274, 280, 282
Factory, 92
FDD, *siehe* Feature Driven Development
Feature Driven Development, 87, 110, 131
Feature Modeling, 110
Feature-Modellierung zur Domänenabgrenzung, 110, 136, 158, 169, 200, 246
FERP, 405
Flexibilisierung von Fachlichkeit, 65, 137, 207
Fluent Interface, 67, 104
Fokussierung zwecks Realisierbarkeit, 57, 136, 189, 208, 259
FormulaPropertyModel, 351
Fusion, 129

Ganzheitliche Betrachtungsweise, 62, 75, 135, 177, 199
Generative Programmierung, 109, 111, 120, 136, 171, 201
Generierung, 10, 38
Geschlossene Menge sinnvoller Alternativen, 62, 75, 135, 179, 199, 212
Gleichförmige Abstraktion, 99, 137, 178, 194, 214
GP, *siehe* Generative Programmierung

Höherwertige Repräsentationsformen, 55, 137, 164, 175, 225, 230

Identifikation von Konzepten, 48, 135
IKT.NRW, 402
Information Hiding, 73
Informationshürden, 75
Inkrementelles, evolutionäres, rekursives und iteratives Vorgehen, 127, 138, 195, 202, 203, 233, 246, 247, 262, 272
Intentional Programming, 54, 298
IP, *siehe* Intentional Programming
IT-Projekt
  Erfolgsfaktoren, 20
  Gründe für das Scheitern, 26
  Qualitätsprobleme, 25
  Qualitätsverbesserungen, 27
  Terminüberschreitungen, 31
Iteratives und inkrementelles Vorgehen, 66, 138, 195, 203, 212, 272

Kernleistung, 8, 11
Kernthese, 7
KMU, 17
KobrA, 129
Kognitive Zugänglichkeit, 101, 137, 164, 224, 233, 255
Kommunikationsfehler, 28
Kommunikationsprobleme, 25
Komponenten, 32, 36
Konzept, 48
  Abstraktion, 73
  Agile Modeling, 120, 138, 208, 246, 271
  Agilität, 88, 121, 138, 155, 203, 243, 244, 246
  Angemessenheit des Entwicklungsprozesses, 74
  Architekturstil, 61, 74, 135, 169, 177, 199
  Ausführbare Prototypen, 125, 137, 196, 233, 262
  Automatisierung von Infrastruktur, 82, 137, 154, 164, 229, 230
  Chief Programmer, 80

Domänenmodell als Handwerkszeug, 90
Domain Driven Design als Konstruktionsschema, 93, 111, 120, 136, 170, 201, 211, 213, 217, 276
Domain Engineering und Application Engineering, 84
Durchgängiges Domänenmodell, 89
Eng fokussierte Software-Produktlinien, 87, 136, 140, 155, 165, 199
Enge Kommunikation, 50
Enger Rahmen, 115, 136, 214, 233, 248, 287
Entwicklungsprozess, 64
Entwicklungsstrukturen, 63
Erweiterbarkeit, 55, 137, 171, 207, 218, 234
Exklusive Domänenmodellierung, 114, 137, 140, 207, 209, 212, 261
Förderung kreativer Organisationen, 66, 138, 196, 212, 244, 274, 280, 282
Feature-Modellierung zur Domänenabgrenzung, 110, 136, 158, 169, 200, 246
Flexibilisierung von Fachlichkeit, 65, 137, 207
Fokussierung zwecks Realisierbarkeit, 57, 136, 189, 208, 259
Ganzheitliche Betrachtungsweise, 62, 75, 135, 177, 199
Generative Programmierung, 111, 120, 136, 171, 201
Geschlossene Menge sinnvoller Alternativen, 62, 75, 135, 179, 199, 212
Gleichförmige Abstraktion, 99, 137, 178, 194, 214
Höherwertige Repräsentationsformen, 55, 137, 164, 175, 225, 230
Identifikation von Konzepten, 48, 135
Inkrementelles, evolutionäres, rekursives und iteratives Vorgehen, 127, 138, 195, 202, 203, 233, 246, 247, 262, 272
Iteratives und inkrementelles Vorgehen, 66, 138, 195, 203, 212, 272
Kognitive Zugänglichkeit, 101, 137, 164, 224, 233, 255
Kooperative, inkrementelle Vorgehensmodelle, 71
Kurze Kommunikationswege, 109, 138, 196, 233, 257
Makro- und Mikroprozesse der Modellbildung, 114, 120, 135, 262
Massentaugliche Werkzeuge, 106, 136, 155, 171, 192
Metrische Bewertung, 74
Modellgetriebene Vorgehensweise, 105, 137, 154, 164, 165, 171, 197, 201
Nebenläufigkeit, 78, 137, 215
Organisationsstruktur, 64
Portierbarkeit durch Abstraktion, 80
Prinzipien- und methodenbasierte Definition, 61, 63, 135, 141
Psychologisch vorteilhafte Strukturbildung, 70
Rapid Application Development, 81, 137, 165, 196, 233, 244, 246, 262, 270, 273
Rapid Prototyping von Nutzerschnittstellen, 73
Spezifizität, 65, 73, 75, 137, 180
Strukturierte Werkzeugnutzung, 73
Systematische Modellbildung, 59, 136, 168, 247
Technologieprojektionen, 64
Visuelle Modellierung, 107, 137, 171, 174, 196, 229
Werkzeugunterstützung, 64
Werte, Prinzipien und Methoden, 118

Wiederverwendung, 63, 74, 136, 178, 199, 218
Zusammenarbeit von Fach- und IT-Experten, 91
Kooperative, inkrementelle Vorgehensmodelle, 71
Koordinationshürden, 75
Kundenorientierung, 9, 21, 24, 28, 47, 49
Kurze Kommunikationswege, 109, 138, 196, 233, 257

Language Oriented Programming, 68, 79
Language Workbench, 68, 106, 299
Layered Architecture, 91
Literate Programming, 104
LOP, *siehe* Language Oriented Programming

Makro- und Mikroprozesse der Modellbildung, 114, 120, 135, 262
Makroebene des CDPM, 264
Management, 84
Massentaugliche Werkzeuge, 106, 136, 155, 171, 192
MDA, 35, 38, 95
MDE, *siehe* Model Driven Engineering
MDSD, 38, 103
Metacase, 112
Metaprogrammierung, 54
Methoden, 143, 155
Agiles Glossar, 277
Analyse durch Design, 269
Archetypisierung, 203
Concept Driven Project Management (CDPM), 157
Conceptual Model Driven Software Development (CMDSD), 156
Conceptual Pair Programming, 273
CP-Coaching, 267
Dekomposition von Komplexität, 224
Differenziere deine Lösung, 231
Domänenabgrenzung und Modellbildung durch Domain Driven Design, 211
Entwicklung eines Architekturstils, 198
Entwicklungsleitung durch einen Chief Programmer, 198
Erhaltung der Migrationsfähigkeit, 231
Intention statt Imperativ, 228
Kapselung von Technologien, 230
Komplexitätsbewertung durch das Pareto-Prinzip, 209
Lebendes Angebot, 276
MDSD-basierte Software-Produktlinien, 199
Philosophischer Projekt-Kickoff, 265
Produktgremium, 157
Rapid Development, 271
Realisiere wissenskonforme Zugänglichkeit, 232
Refactoring, 276
Strukturierte Modellbildung, 213
Unternehmenskonformes Projektmanagementmodell, 202
Werkzeuginhärente Archetypen, 227
Metrische Bewertung, 74
Mikroebene des CDPM, 262
Mittelstand, 3, 16
Definition, 16
ERP-KMU-Dilemma, 19
Herausforderungen, 3
Architekturzentrierung, 4
Code-Zentrierung, 6
Generalisierung, 5
Präskriptive Vorgehensmodelle, 6
Statische Vorgehensmodelle, 6
Technologiezentrierung, 4
Individualsoftware, 18
Investitionskraft, 17
KMU, 17
Wirtschaftskraft, 17
MMORPG, 403

Model Driven Architecture, *siehe* MDA
Model Driven Engineering, 114
Model Driven Software Development, *siehe* MDSD
Model-Driven Engineering, 302
Modell
  Ausführbarkeit, 233
  Erweiterungspunkt, 207, **220**
  Facette, **218**, 220
  Gruppe, 217
  Komposition, 223
  primär, 215
  sekundär, 215
  tertiär, 217
  Transformation, 234
Modellfacette, **218**, 220
Modellgetrieben, 94
Modellgetriebene Vorgehensweise, 105, 137, 154, 164, 165, 171, 197, 201
Modellgruppe, 217
Modellhierarchie, 337
Modellkomposition, 223
Modellrepräsentation, 319
Modelltransformation, 234
Modul, 91
MOF, 38, 95, 103

Navigationsmodell, 341
Nebenläufigkeit, 78, 137, 215

Object Management Group, *siehe* OMG
Objectory, 129
OMG, 38, 95
OMT, 129
openArchitectureWare, 104
Optionspreistheorie, 41
Organisationsstruktur, 64

Package, 91
Pair Programming, 130
Paradigmenwechsel, 8
  Automatisierung, 10
  Domänenfokussierung, 9
  Generierung, 10
  Kundenorientierung, 9
Personal Software Process, 128
PIKAS, 189, 308, 310
  Anforderungen, 330
  ContentModel, 348
  Datenkontext, 338
    ConversationContext, 337
    DomainContext, 337
    ProcessContext, 337
    SessionContext, 337
    ViewContext, 337
  DomainTypeModel, 349, 351
  DomainTypePropertyModel, 351
  EnumPropertyModel, 352
  FormulaPropertyModel, 351
  Kosten/Nutzen, 389
  Modell
    Navigation, 341
    Prozess, 346
  Modelle
    Übersetzungen, 358
    Agenten, 356
    Aktionen, 362
    Aktionslisten, 361
    Datenquellen, 370
    Domänentypen, 351
    Filter, 365
    Gefilterte Listen, 369
    Inhalte, 366
    Navigation, 339
    Portal, 370
    Prozesse, 342
    Relationen, 355
    Skins, 360
    Views, 347
  Modellhierarchie, 335, 337
  Modellrepräsentation, 319
  Navigationsmodell, 341
  PartModel, 348
  PropertyModel, 349, 352
  Prozessmodell, 346
  Schichtenmodell, 316
  Scopes, 337
  SubViewModel, 348
  View, *siehe* ViewModel
  ViewModel

Hauptdomänentyp, 348
PIM, 96, 114
Planbasierte Vorgehensmodelle, 127
Platform Independent Model, *siehe* PIM
Platform Specific Model, *siehe* PSM
Portierbarkeit durch Abstraktion, 80
Präskriptive Verfahren, 42
Präskriptive Vorgehensmodelle, 127
Primäre Modelle, 215
PRINCE2, 128
Prinzipien, 142, 152
Abstraktion, 181
Agilität, 154
Automatisierung, 153
Bedarfsorientierung, 258
CMDSD-basiertes Vorgehen, 256
Direktheit, 257
Domänenfokussierung, 152
Ehrlichkeit und Offenheit, 252
Explizivität und Transparenz, 257
Fairness, 252
Inkrementelle Evolution, 195
Kompaktheit, 180
Kompetenz statt Komplexität, 255
Konsequenz, 256
Kundenorientierung, 152
Lokalität und Linearität, 190
Min-Max, 261
Problemorientierung, 175
Rapid Prototyping, 195
Respekt, 250
Standardkonformität, 178
Start small, then grow, 261
Tradition und Innovation, 191
Uniformität, 177, 254
Vereinfachung, 174
Prinzipien- und methodenbasierte Definition, 61, 63, 135, 141
ProcessContext, 337
Product Development, 84
PropertyModel, 352
Prototyping, 124
Prototypisierung, 73
Prozess, 336
vs. Workflow, 336
Prozessmodell, 346
PSM, 96, 114
PSP, 128
Psychologisch vorteilhafte Strukturbildung, 70

QuinScape GmbH, 313

RAD, *siehe* Rapid Application Development
Rapid Application Development, 80, 81, 137, 165, 196, 233, 244, 246, 262, 270, 273, 292
Rapid Prototyping, 165
Rapid Prototyping von Nutzerschnittstellen, 73
Refactoring, 58, **92**, 130, 276
Repository, 92
Retrospektive, 130
Reverse Engineering, 113
RUP, 42, 129

Schichtenmodell, 316
Scrum, 131
Sekundäre Modelle, 215
Semantic Noise, 58
Semantic Web, 404
Semantic-oriented programming, 75
Service, 91
SessionContext, 337
Signal/Noise Ratio, 59
Skaleneffekte, *siehe* Eonomies of Scale39
Smart UI, 91
SME, 17
Software Factories, 54
Software Factory, 105, 165
Software Process Model, *siehe* Vorgehensmodell
Software-Produktlinie, 83, 106, 165, 295
Spezifizität, 65, 73, 75, 137, 180
SPL, *siehe* Software-Produktlinie
Standardisierung, 32
Strukturierte Werkzeugnutzung, 73
Subtext, 72
SW-CMM, 128

Syntactic Noise, 58
SysML, 97
Systematische Modellbildung, 59, 136, 168, 247

T-Technik, 276, **287**, 288
Tagging, 332
Team Software Process, 128
Teamorganisation, 279
Technologieprojektionen, 64
Tertiäre Modelle, 217
TSP, 128

UML, 35, 42, 95
User-Driven Computing, 71

V-Modell, 128
Value Object, 91
Variabilität, 86, 235
Verbundeffekte, *siehe* Economies of Scope
Verständnishürden, 75
ViewContext, 337
Visual Programming, 75
Visuelle Modellierung, 107, 137, 171, 174, 196, 229
Vorgehensmodell, 121
  agil, 129
  evolutionär, 125
  inkrementell, 126
  planbasiert, 71, 127
  präskriptiv, 71, 127
  Prototyping, 124
  Wasserfallmodell, 122

Wasserfallmodell, 53, 122
Werkzeugunterstützung, 64
Werte, 142, 145
  Flexibilität, 148
  Wirtschaftlichkeit, 146
  Zugänglichkeit, 150
Werte, Prinzipien und Methoden, 118
Wiederverwendung, 63, 74, 136, 178, 199, 218
WODOMI, 399
Workflow, 336
  vs. Prozess, 336

XMI, 95
XML Metadata Interchange, *siehe* XMI
XP, 130

YAGNI, 130

Zusammenarbeit von Fach- und IT-Experten, 91

Lebenslauf

Persönliche Daten:	Thomas Biskup Steinstraße 8 58452 Witten Geboren am 02. Juli 1971 in Gelsenkirchen verheiratet
Schulausbildung:	
1977–1981	Grundschule in Gelsenkirchen-Hassel
1981–1990	Max-Planck-Gymnasium in Gelsenkirchen-Buer Abitur: Mai 1990
Hochschulausbildung:	
10/1990–4/1997	Universität Dortmund Studium der Kerninformatik mit Nebenfach Betriebswirtschaft Schwerpunkte: Softwaretechnik und Datenbanken Abschluss: Diplom, Note: Mit Auszeichnung Thema der Diplomarbeit: Entwicklung eines Werkzeugs zur Generierung von Testmustern aus einer Programmspezifikation unter Verwendung von Ursache/Wirkungsgraphen
Zivildienst:	
5/1997–5/1998	Zivildienst in der Psychologischen Beratungsstelle der Stadt Marl
Auszeichnungen:	
3/1998	Preisträger des „Hans-Uhde-Preises zur Förderung der Wissenschaft“
9/1998	Diplomarbeitspreis der Ernst-Denert-Stiftung für Software Engineering
Berufliche Laufbahn:	
6/1998–7/2000	Software-Entwickler bei der PRO DV Software AG www.prodv.de
8/2000–12/2002	Wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik I der Universität Dortmund
3/2001–heute	Geschäftsführender Gesellschafter der QuinScape GmbH (seit 1/2003 Vollzeitbeschäftigung) www.quinscape.de