Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Alexander Dotor Schumann

aus Frankfurt am Main

1. Gutachter: Prof. Dr. Bernhard Westfechtel
2. Gutachter: Prof. Dr. Albert Zündorf

Tag der Einreichung: 24.01.2011

Tag des Kolloquiums: 06.07.2011

Für Honorio Dotor

Zusammenfassung

Es existieren zur Zeit über 70 verschiedene **S**oftware-**K**onfigurations-**M**anagement-**S**ysteme (kurz: SKMS), um die Entwicklung von Software-Anwendungssystemen zu unterstützen. SKMS sind selbst auch Software-Anwendungssysteme, deren Umfang von einigen zehntausend bis zu mehreren Millionen Quelltext-Zeilen reicht. Trotz dieser Größe handelt es sich meist um monolithische Systeme, deren Daten und Funktionen eng verschränkt sind und deren Verfahren lediglich implizit durch den Quellcode beschrieben werden. Eine Identifikation einzelner Methoden ist kaum möglich, ebensowenig wie eine Wiederverwendung bereits implementierter Verfahren. Dies führt dazu, dass für ein neues Verfahren – oder sogar nur für eine neue Kombination existierender Verfahren – ein SKMS von Grund auf neu implementiert wird. Die fehlende Modularität erschwert auch die Wiederverwendung eines SKMS für unterschiedliche Software-Entwicklungsprozesse, da bereits kleine Unterschiede zwischen den Prozessen die Entwicklung eines neuen SKMS erfordern können.

Es besteht daher der Bedarf nach einer neuen Generation von SKMS, die durch ihre modulare Architektur (1) die Wiederverwendung bestehender Verfahren erlauben, (2) die Anpassbarkeit an Entwicklungsprozesse verbessern bzw. überhaupt erst ermöglichen und (3) die Erweiterbarkeit verbessern, um ein SKMS in möglichst vielen Entwicklungsprozessen einsetzen zu können. Die **Mod**ellgetriebene **Mod**ulare Produktlinie für **S**oftware **K**onfigurations **M**anagement Systeme (kurz: **MOD2-SKM**) identifiziert Gemeinsamkeiten von SKMS auf Basis eines Entwicklungsprozesses für **Mod**ellgetriebene Software **P**rodukt**l**inien (MODPL). Der Entwicklungsaufwand eines SKMS wird so reduziert, da gemeinsame Komponenten wiederverwendet und die Produkte aus dem Domänenmodell generiert werden (anstatt sie von Hand zu implementieren). So kann durch Konfiguration des Domänenmodells nach den Anforderungen eines Software-Entwicklungs-Prozesses ein SKMS speziell an den zu unterstützenden Prozess angepasst werden.

In dieser Arbeit wird ein **Merkmals-Modell für SKMS** definiert und ein generisches **Domänenmodell** für die gesamte Systemfamilie der SKMS modelliert und analysiert. Mit Hilfe des Merkmals-Modells lassen sich die generierten SKMS systematisch beschreiben und miteinander vergleichen. Gleichzeitig können damit auch **bestehende SKMS klassifiziert** und so systematisch erfasst und verglichen werden. Durch das MOD2-SKM-Domänenmodell wird die SKM-Domäne mit Hilfe von Komponenten und Klassen beschrieben. **Abhängigkeiten und Kopplungen** werden dabei explizit **identifiziert und reduziert**. So wird das Verständnis der SKM-Domäne vertieft – besonders in Bezug auf **Modularisierbarkeit**, **Erweiterbarkeit** und **Co-Evolution** – und durch eine neue Sicht auf die komplexen Datenabhängigkeiten erweitert.

MOD2-SKM ist gleichzeitig auch ein Prototyp einer neue Generation vom SKMS, denn aus dem Domänenmodell lassen sich vollständig lauffähig SKMS erzeugen. Somit wird

auch die Architektur und Realisierung einer modellgetriebenen Produktlinie für SKMS erforscht. MOD2-SKM zeigt, dass ein einzelnes SKMS nicht mehr von Hand implementiert werden muss. Stattdessen können, mit Hilfe von Konfigurationen, eine Vielzahl von SKMS generiert werden. So lassen sich entweder bestehende SKMS wie CVS oder Subversion nachbilden oder auch vollständig neue SKMS erstellen. Die Erkenntnisse, die beim Entwurf von MOD2-SKM gewonnen wurden, bilden die Grundlagen, die für den Entwurf und die Entwicklung modularer SKMS benötigt werden:

- Sind SKMS überhaupt für MODPL geeignet?
- In welche Komponenten lässt sich ein SKMS zerlegen?
- Welche Kernkomponenten muss jedes SKMS besitzen?
- Welche Kopplungen bestehen zwischen den Komponenten?
- Welche Modellierungsmethoden sind für die lose Kopplung der Komponenten besonders geeignet?

Der MOD2-SKM Entwicklungsprozess folgt dem MODPL-Modellierungsansatz und basiert so letztendlich auf modellgetriebenen Entwicklungsmethoden. MOD2-SKM stellt damit eine nicht-triviale Fallstudie für den Einsatz von modellgetriebenen Produktlinien dar, mit deren Hilfe sich der MODPL-Modellierungsansatz untersuchen lässt. In dieser Arbeit werden somit auch der Modellierungsansatz und die verwendeten modellgetriebenen Entwicklungsmethoden evaluiert und abschließend auch Anforderungen an sie formuliert.

Abstract

Software developers can choose between more than 70 **s**oftware-**c**onfiguration-**m**anagement-**s**ystems (abbr.: SCMS) to support the development of software applications. But SCMS are software applications by themselves and their size varies between a few ten thousand and several million lines of code. Even large systems turn out to be monolithic systems, i.e. their data and functions are tightly coupled and their underlying concepts are implicitly described by source code. Both, identification and reuse of the underlying concepts is nearly impossible. Often, implementation of a new SCM concept means implementation of a new SCMS from scratch. The lack of modularity makes the adaption of SCMS to different development processes difficult. Sometimes even minor changes to a development process require the adoption of a completely new SCMS.

Software development needs a new generation of SCMS with modular architecture to (1) enable the reuse of existing SCM concepts, (2) improve or enable the adaption of SCMS to development processes, and (3) improve extensibility of SCMS, in order to increase the number of development processes a SCMS can be adopted. The **mod**el-driven **mod**ular product line for **s**oftware **c**onfiguration **m**anagement systems (abbr.: **MOD2-SCM**) identifies commonalities of SCMS, based on a model-driven development approach for **mod**el-driven software **p**roduct**l**ines (MODPL). This approach reduces the development effort of a SCMS (1) by reuse of common components, and (2) by replacing manual implementation of a SCMS with code generation from a graphical domain model. Adaption to a development process is achieved by creating a configuration manually and applying it to the domain model automatically.

This thesis defines a **feature-model for SCMS**, and models and analyzes a generic domain model for a whole system family of SCMS. The feature-model provides a schema for a systematic and comparable description of every generated SCMS. The schema can also be used for a systematic **classification** and comparison of **existing SCMS**. The MOD2-SCM **domain model** uses package- and class-diagrams to model the SCM domain. They describe dependencies explicitly and their modular architecture is designed to reduce coupling. This way the graphical models deepen the understanding of the SCM domain – especially about **modularity**, **extensibility** and **co-evolution** – and offer a model-driven view on their complex dependencies.

MOD2-SCM is also a prototype of a new generation of SCMS. MODPL enables the automatic generation of fully executable SCMS out of the domain model. So this thesis explores the design and realization of a model-driven product line for SCMS. MOD2-SCM shows that manual implementation of a single monolithic SCMS is unnecessary. Many different SCMS can be generated instead by creating configurations for a domain model. Both existing SCMS – for example CVS or Subversion – and completely new SCMS can be generated. The insight gained during the design and modeling solves and discusses

fundamental questions about the design and development of modular SCMS:

- Are SCMS eligible for model-driven product lines?
- Which components are SCMS consisting of?
- What are core components, required for any SCMS?
- Which dependencies exist between these components?
- Which modeling methods are most appropriate to reduce coupling?

Development of MOD2-SCM follows the MODPL development approach and is ultimately based on generic model-driven development methods. Therefore, MOD2-SCM is a non-trivial case study for the application of model-driven product lines. Moreover, the model-driven approach in general is evaluated and, finally, the requirements for model-driven product line development and model-driven development in general are derived.

Inhaltsverzeichnis

1. Einleitung **15**
1.1. Software-Konfigurations-Management . . . 15
1.2. Anforderungen an SKMS . . . 19
1.3. SKMS heute . . . 21
1.4. Probleme heutiger SKMS . . . 22
1.5. Verwandte Arbeiten . . . 24
1.6. Modellgetriebene Produktlinien . . . 25
1.7. MOD2-SKM: Eine MODPL für SKMS . . . 26
1.8. Ergebnisse und Aufbau der Arbeit . . . 27
1.8.1. Ergebnisse und Forschungsbeitrag . . . 29
1.8.2. Aufbau der Arbeit . . . 30
1.9. Zusammenfassung . . . 30

2. Verwandte Arbeiten **33**
2.1. Generisches Versionsmodelle . . . 33
2.1.1. ECM/UVM . . . 33
2.1.2. ICE . . . 35
2.1.3. Molhado . . . 38
2.1.4. UEVM . . . 39
2.2. Quellcode-Kombinations-Rahmenwerke . . . 41
2.2.1. Bamboo . . . 41
2.2.2. MCCM . . . 44
2.3. Quellcode-Generatoren . . . 45
2.3.1. VS-Gen . . . 45
2.4. Vergleich und Schlussfolgerungen . . . 47
2.4.1. Generische Versionsmodelle . . . 48
2.4.2. Quellcode-Kombinations-Rahmenwerke . . . 49
2.4.3. Quellcode-Generatoren . . . 49
2.4.4. Schlussfolgerungen . . . 49
2.5. Zusammenfassung . . . 49

3. Modellgetriebene Produktlinien **51**
3.1. Das MODPL-Rahmenwerk . . . 52
3.2. Das Merkmalsmodell . . . 54
3.3. Die Konfiguration . . . 55

3.4. Das konfigurierbare Domänenmodell . . . 57
3.4.1. Verhaltensmodellierung . . . 58
3.4.2. Merkmalsmarkierungen . . . 58
3.5. Das konfigurierte System . . . 61
3.6. Kernprobleme beim Entwurf konfigurierbarer Domänenmodelle . . . 64
3.6.1. Kernproblem: Syntaktisch korrekte konfigurierte Modelle . . . 64
3.6.2. Kernproblem: Verwaiste Modellelemente . . . 65
3.6.3. Kernproblem: Verwaiste Merkmale . . . 65
3.6.4. Kernproblem: Modulare Architektur . . . 66
3.6.5. Kernproblem: Semantische Konsistenz mit Merkmalsmodell . . . 66
3.7. Zusammenfassung . . . 67

4. Software-Konfigurations-Management 69
4.1. Definition von SKM und SKMS . . . 69
4.1.1. Was ist SKM? . . . 69
4.1.2. Was ist ein SKMS? . . . 71
4.1.3. Welche Aufgaben unterstützt ein SKMS? . . . 73
4.1.4. Wer verwendet ein SKMS? . . . 73
4.1.5. SKM-Umgebung oder SKM-Werkzeug? . . . 74
4.1.6. MOD2-SKM oder MOD2-VKS? . . . 75
4.2. Konzepte im Anforderungsbereich „Struktur“ . . . 75
4.2.1. Produktraum . . . 76
4.2.2. Die Objektkennung . . . 76
4.2.3. Beziehungen im Produktraum . . . 76
4.2.4. Produktmodelle . . . 77
4.3. Konzepte im Anforderungsbereich „Komponenten“ . . . 78
4.3.1. Versionsraum . . . 79
4.3.2. Die Versionskennung . . . 79
4.3.3. Beziehungen im Versionsraum . . . 80
4.3.4. Integration von Produkt- und Versionsraum . . . 81
4.3.5. Versionsmodelle . . . 85
4.3.6. Datenspeicherung . . . 88
4.4. Konzepte im Anforderungsbereich „Team“ . . . 91
4.4.1. Repositorien . . . 91
4.4.2. Arbeitsbereiche . . . 93
4.5. Zusammenfassung . . . 94

5. MODPL in der SKM-Domäne 95
5.1. Ablauf der MOD2-SKM Domänenanalyse . . . 96
5.1.1. Generelle Eignung von SKM für MODPL . . . 96
5.1.2. Gliederung des Merkmals-Modells nach FORM . . . 97
5.1.3. Auswahl der Quellen . . . 98
5.1.4. Identifikation und Integration von Merkmalen . . . 100

5.2. Aufbau des MOD2-SKM Merkmals-Modell . . . 100
5.2.1. Erstellung des MOD2-SKM Merkmalsmodells . . . 101
5.2.2. Beschreibung der Merkmale . . . 103
5.3. Beschreibung der Spitzenmerkmale . . . 104
5.3.1. Merkmale der Befähigungs-Ebene . . . 104
5.3.2. Merkmale der Betriebsumgebungs-Ebene . . . 111
5.3.3. Merkmale der Domänentechnologie-Ebene . . . 113
5.3.4. Merkmale der Implementierungstechniken-Ebene . . . 121
5.3.5. Probleme und Grenzen . . . 123
5.4. Validierung des MOD2-SKM Merkmals-Modells . . . 125
5.4.1. Concurrent Versioning System . . . 125
5.4.2. GIT . . . 126
5.4.3. Mercurial . . . 127
5.4.4. Subversion . . . 128
5.5. Stand der Umsetzung . . . 129
5.5.1. Umsetzung der Spitzenmerkmale . . . 130
5.6. Merkmalsmarkierungen im Domänenmodell . . . 133
5.6.1. Untersuchung der Merkmalsmarkierungen . . . 134
5.6.2. Untersuchung der aussagenlogischen Einschränkungen . . . 137
5.7. Zusammenfassung . . . 139

6. Die MOD2-SKM Produktlinie 141
6.1. Aufbau des MOD2-SKM Domänenmodells . . . 142
6.1.1. Modellierung von Variabilität . . . 149
6.1.2. Probleme und Grenzen . . . 154
6.2. Architektur des MOD2-SKM Domänenmodells . . . 155
6.2.1. Modularität: Architektur von Produktlinien . . . 155
6.2.2. Zentral vs. Dezentral: Architektur von SKMS . . . 157
6.2.3. Umsetzung im MOD2-SKM Domänenmodell . . . 157
6.2.4. Module der MOD2-SKM-Produktlinie . . . 163
6.3. Die MOD2-SKM Kernkomponenten . . . 171
6.3.1. Kernmodul: core.communication . . . 173
6.3.2. Kernmodul: core.delta . . . 175
6.3.3. Kernmodul: core.exceptions . . . 177
6.3.4. Kernmodul: core.history . . . 178
6.3.5. Kernmodul: core.id . . . 182
6.3.6. Kernmodul: core.merge . . . 184
6.3.7. Kernmodul: core.persistency . . . 186
6.3.8. Kernmodul: core.product . . . 188
6.3.9. Kernmodul: core.repository . . . 191
6.3.10. Kernmodul: core.server . . . 195
6.3.11. Kernmodul: core.storage . . . 200
6.3.12. Kernmodul: core.synchronisation . . . 202
6.3.13. Kernmodul: core.user . . . 203

6.3.14. Kernmodul: core.workspace 206
6.4. Die MOD2-SKM Modulbibliothek 211
6.4.1. Modul: history.base 213
6.4.2. Modul: history.flat 214
6.4.3. Modul: history.tree 216
6.4.4. Modul: id.hash 222
6.4.5. Modul: id.hierarchic 223
6.4.6. Modul: id.latest 224
6.4.7. Modul: id.primarykey 225
6.4.8. Modul: id.qualifiedname 226
6.4.9. Modul: id.tag 227
6.4.10. Modul: merge 229
6.4.11. Modul: persistency.coobra 235
6.4.12. Modul: persistency.hibernate 237
6.4.13. Modul: product.ecore 239
6.4.14. Modul: product.filesystem 240
6.4.15. Modul: product.usecase 247
6.4.16. Modul: repository.complexitem 251
6.4.17. Modul: repository.singleitem 253
6.4.18. Modul: server.base 254
6.4.19. Modul: server.rmi 271
6.4.20. Modul: server.runtimeconfig 273
6.4.21. Modul: storage.base 279
6.4.22. Modul: storage.complex 280
6.4.23. Modul: storage.delta 283
6.4.24. Modul: synchronisation.lockitem 287
6.4.25. Modul: synchronisation.lockserver 289
6.4.26. Modul: user.flat 290
6.4.27. Modul: workspace.base 292
6.4.28. Modul: workspace.runtimeconfig 295
6.5. Untersuchung der Architektur 296
6.5.1. Existierende Metriken für Kopplung 296
6.5.2. Kopplung von Paketen 297
6.5.3. Module und ihre Abhängigkeiten 300
6.6. Zusammenfassung 312

7. MOD2-SKM für Eclipse 315
7.1. Konfigurieren des SKMS 316
7.2. Architektur der MOD2-SKM-Erweiterung 319
7.3. Funktionalität der MOD2-SKM-Erweiterung 322
7.4. Modulare Benutzeroberflächen 324
7.5. Zusammenfassung 325

8. Ergebnis und Ausblick **327**
8.1. Prüfung der Hypothesen . . . 327
8.2. Modularisierung von SKMS . . . 328
8.3. Konzepte modellgetriebener Produktlinien . . . 329
8.3.1. Entwicklungsprozess . . . 330
8.3.2. Konzeption Merkmalsmodell . . . 334
8.3.3. Konzeption Domänenmodell . . . 335
8.3.4. Korrektheit und Testen . . . 338
8.3.5. Konzeption der Auslieferung . . . 339
8.4. Werkzeuge und Modelle des MODPL-Werkzeugkastens . . . 339
8.4.1. Merkmalsmodell und Konfigurationen . . . 339
8.4.2. Feature Plugin . . . 341
8.4.3. Paketdiagramm . . . 341
8.4.4. Paketdiagramm-Editor . . . 342
8.4.5. Domänenmodell . . . 344
8.4.6. Fujaba . . . 345
8.4.7. MODPL-Editor . . . 348
8.4.8. Konfigurierte SKMS . . . 349
8.4.9. MODPL-Konfigurator . . . 349
8.5. Modellgetrieben Entwickeln . . . 349
8.5.1. Verwendung eines Architekturmodells . . . 350
8.5.2. Große Modelle . . . 350
8.5.3. Mehrere Metamodelle . . . 350
8.5.4. Modellierungstechniken . . . 351
8.6. Zukünftige Entwicklungen . . . 352
8.7. Zusammenfassung . . . 353

A. Fachbegriffe: Deutsch – Englisch **355**

B. MOD2-SKM Merkmals-Analyse der SKM-Domäne **359**

C. MOD2-SKM Merkmalsmodell der SKM-Domäne **435**

D. Konfigurationen des MOD2-SKM Merkmalsmodells **439**

E. Stand der Umsetzung **447**

1. Einleitung

1.1. Software-Konfigurations-Management

Software-**K**onfigurations-**M**anagement (SKM) ist eine Methode zur Unterstützung von Softwareentwicklungsprozessen [Kru03]. Ziel eines Prozesses ist die Entwicklung eines Software-Anwendungssystems, das aus ein oder mehreren Software-Elementen[1] besteht. SKM unterstützt die Entwickler durch die „Identifikation von Konfigurationen (zusammengehörende Menge von Software-Elementen) zum Zweck der systematischen Steuerung von Konfigurationsänderungen und der Aufrechterhaltung der Vollständigkeit und Verfolgbarkeit der Konfiguration während des gesamten Software-Lebenszyklus“ [KMLO98]. Folgende Aktivitäten sind zentral für SKM [Kru03]:

- Software-Elemente identifizieren und lokalisieren,
- Entwicklungshistorie von Software-Elementen speichern,
- Versionen aus der Historie betrachten und wiederherstellen,
- Änderungen an den Software-Elementen nachvollziehen,
- Verantwortliche Entwickler den Software-Elementen und Änderungen zuordnen

Bei diesen Aktivitäten werden die Entwickler durch ein **S**oftware-**K**onfigurations-**M**anagement-**S**ystem (SKMS) unterstützt. Das SKMS ist ein Werkzeug, das diese Aktivitäten und das SKM automatisiert und den Entwickler durch die einzelnen Arbeitsschritte leitet. Abbildung 1.1 zeigt die funktionalen Anforderungen an SKMS [Dar90]. Jeder Kasten stellt einen von acht funktionalen Anforderungsbereichen dar, an Hand derer sich die Anforderungen an SKMS klassifizieren lassen:

- „Buchhaltung“ sammelt statistische Daten über Prozess und Produkt.
- „Controlling“ kontrolliert die Art und Weise sowie den Zeitpunkt von Änderungen.
- „Komponenten“ identifiziert, klassifiziert und speichert die Komponenten, aus denen das Produkt besteht und verwaltet auch den Zugriff auf sie.

[1] Der Begriff „Software-Element“ stammt aus dem Informatik-Begriffsnetz der *Gesellschaft für Informatik e. V.*, und bezeichnet einen kleinsten, als unteilbar behandelten, Bestandteil eines Software-Systems [Ie10]. In der Literatur werden unterschiedliche Begriffe synonym verwendet, z.B. Software-Objekt [CW98], Entität [ABCM99], Komponente [ABCM99], Artefakt [Hoe00] oder Software-Artefakt [Kov05].

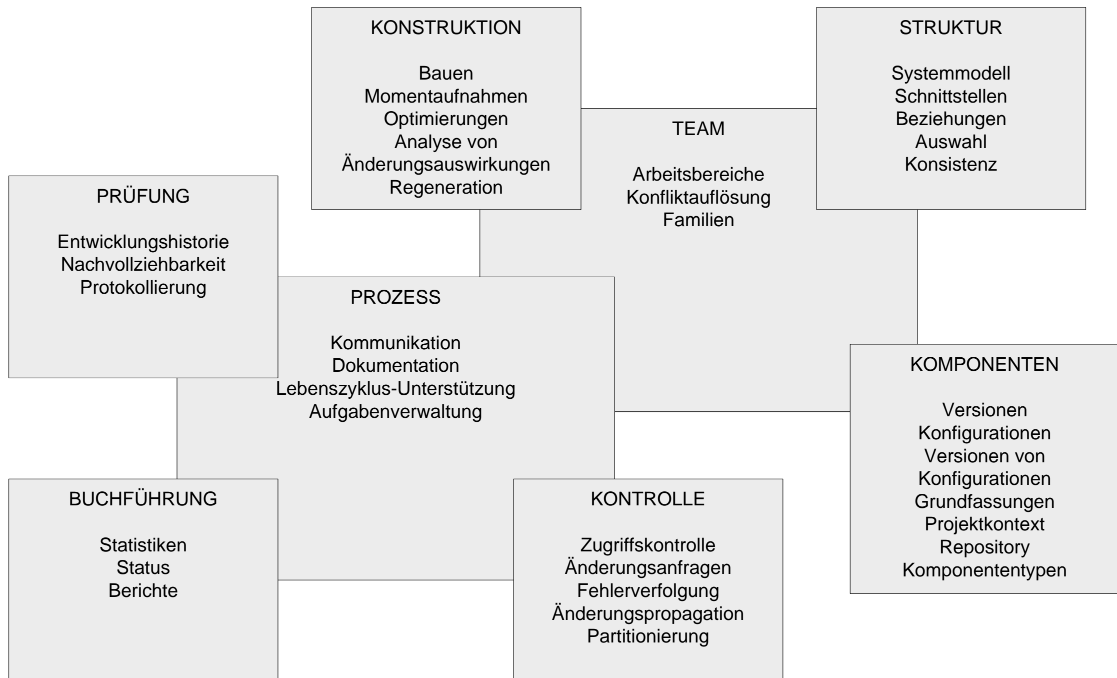
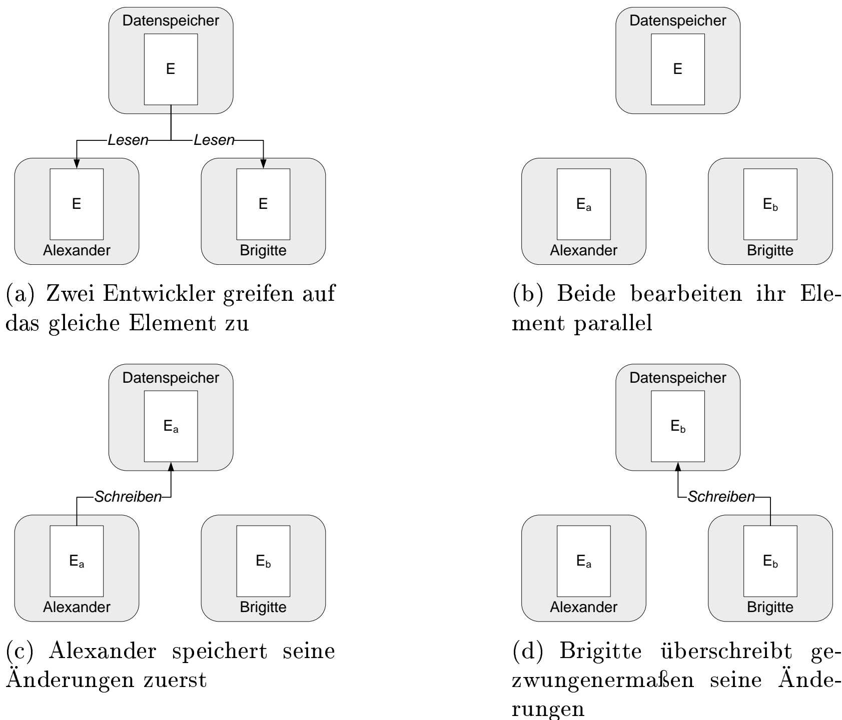


Abbildung 1.1.: Funktionale Anforderungen an SKMS (nach [Dar90])

- „Konstruktion“ unterstützt die Erstellung des fertigen Produktes und der Elemente, aus denen es besteht.
- „Prozess“ unterstützt die Planung und Verwaltung des weiteren Entwicklungsablaufs.
- „Prüfung“ verwaltet die Entstehungsgeschichte der Komponenten und den Ablauf des Prozesses.
- „Struktur“ stellt die Architektur des Produktes dar, d.h. er verwaltet die Beziehungen zwischen den Komponenten und stellt ihre Konsistenz sicher.
- „Team“ unterstützt ein Projektteam bei der gemeinsamen Entwicklung mehrerer Produkte.

Heutzutage ermöglicht erst die Unterstützung eines Entwicklungsprozesses für ein Entwicklerteam die Entwicklung komplexer Software-Anwendungssysteme. Daher bildet auch die Unterstützung des Entwicklungsprozesses („Prozess“) sowie für verteiltes Arbeiten („Team“) den Schwerpunkt in SKMS – dargestellt durch die beiden zentralen Kästen in Abbildung 1.1. Die sechs äußeren Kästen, „Buchführung“, „Controlling“, „Komponenten“, „Konstruktion“, „Prüfung“, „Struktur“ repräsentieren weitere Aufgabengebiete, die in Entwicklungsprozessen vorkommen, aber nicht als zentraler Bestandteil gelten und nur im Bedarfsfall verwendet werden. Daher werden sie oft auch als eigenständige Aufgabenbereiche betrachtet und durch eigene Werkzeuge unterstützt. z.B. werden Werkzeuge,



(a) Zwei Entwickler greifen auf das gleiche Element zu

(b) Beide bearbeiten ihr Element parallel

(c) Alexander speichert seine Änderungen zuerst

(d) Brigitte überschreibt gezwungenermaßen seine Änderungen

Abbildung 1.2.: Problem: Konkurrierende Änderungen (nach [CSFP08])

die sich nur auf die Bereiche „Komponenten“ und „Team“ konzentrieren, als **V**ersions-**K**ontroll-**S**ysteme (VKS), oder auch Revisions-Kontroll-Systeme, bezeichnet. Im Falle einer ganzheitliche Prozessunterstützung vereinen jedoch die beiden zentralen Bereiche die sechs äußeren, d.h. die Aktivitäten aller acht Bereiche werden durch ein ganzheitliches SKMS unterstützt. Die folgenden Abschnitte geben einen kurzen Überblick über die acht Bereiche und stellen zentrale Probleme und Aufgaben vor.

Team

Die Unterstützung von Entwicklerteams ist einer der beiden zentralen Aufgabenbereiche eines SKMS. Sobald mehrere Personen gemeinsam an einem Anwendungssystem arbeiten, kann es zu konkurrierenden Änderungen kommen: Z.B. werden die Software-Elemente zentralisiert gespeichert, damit alle Entwickler darauf zugreifen können (vgl. Abbildung 1.2(a)). Nun verändern zwei Mitarbeiter parallel ein Element und speichern ihre Änderungen (vgl. Abbildung 1.2(b) und Abbildung 1.2(c)). Dies führt dazu, dass die zweite Änderung die erste überschreibt (so dass effektiv nur die zweite Änderung durchgeführt wurde, vgl. Abbildung 1.2(d)).

SKMS koordinieren die Zusammenarbeit, z.B. indem sie Verfahren zur Synchronisation implementieren und so den Zugriff auf die zentral gespeicherten Elemente steuern. Diese Verfahren verhindern das unabsichtliche Überschreiben von Änderungen, indem sie entweder parallele Änderungen vermeiden oder mögliche Konflikte erkennen. Bei Letz-

terem unterstützt das SKMS die Entwickler durch Verfahren für die automatische oder manuelle Auflösung der Konflikte.

Prozess

Die Unterstützung des Entwicklungsprozesses, in dessen Rahmen die Entwickler das Produkt entwickeln, ist einer der beiden zentralen Aufgabenbereiche eines SKMS. Die Aufgaben und Regeln des Prozesses stellen Anforderungen an das SKMS in Form von **Funktionen**, **Metadaten** und **Beschränkungen**, die das SKMS anbieten und speichern bzw. einhalten soll. Das Vorgehensmodell des **R**ational **U**nified **P**rocess (RUP) erfordert z.B., dass Elemente bestimmten Phasen zugeordnet werden können, während E**x**treme **P**rogramming (XP) für jede Klasse einen Testfall verlangt. Insbesondere die Anforderungen des Vorgehensmodells und des Entwicklungsprozesses bestimmen, welches SKMS überhaupt geeignet ist. Sollte kein geeignetes vorhanden sein, so muss zunächst ein bestehendes SKMS erweitert bzw. ein neues speziell für diese Anforderungen implementiert werden.

Buchführung

Die verwalteten Software-Elemente sind sowohl in aktuellen wie auch in vergangenen Zuständen im SKMS vorhanden. Gleichzeitig lässt sich auch ihre Historie verfolgen, so dass sich das SKMS als Datenbasis für den aktuellen Status des Projektes oder Statistiken über die Entwicklungsgeschichte nutzen lässt. So kann z.B. der Umfang von Änderungen einzelnen Entwicklern zugeordnet werden, um ihre Leistungen einzuschätzen.

Controlling

Die Anzahl der Entwickler und ihre Organisationsform stellen unterschiedliche Anforderungen an ein SKMS. Eine kleine Gruppe oder ein einzelner Entwickler benötigt Unterstützung für konkurrierende Änderungen, um von mehreren Arbeitsplätzen aus zu arbeiten, während bei firmenübergreifenden Projekten mehrere Repositorien synchronisiert werden müssen. Im Falle von Subunternehmern und Unterprojekten ist auch eine hierarchische Verwaltung und Synchronisation erforderlich, da sich z.B. die Zugriffsrechte der Entwickler des Subunternehmens von denen der internen Entwickler unterscheiden. Während des Betriebs eines Anwendungssystems liefern seine Benutzer Fehlerberichte und Änderungswünsche, die durch SKMS protokolliert und mit Entwicklern und Software-Elementen in Beziehung gesetzt werden, z.B. indem einem Entwickler ein Fehlerbericht und die betroffenen Elemente zwecks Fehlerbehebung zugewiesen werden.

Komponenten

Anwendungssysteme bestehen aus einer Menge von Software-Elementen. Änderungen und Erweiterungen am System bedeuten Änderungen an bestehenden Elementen (bzw. das Anlegen neuer Elemente). Während der Entwicklung eines Anwendungssystems – und auch während seines Betriebs – werden immer wieder Änderungen durchgeführt: Es

wird neue Funktionalität hinzugefügt, bestehende Funktionalität verbessert oder auch Fehler behoben. Aufgabe des SKMS ist es die Entwicklungshistorie explizit zu verwalten, um z.B. fehlerhafte Änderungen Rückgängig zu machen. Im Falle von Produktfamilien sollen bestehende Software-Elemente erneut verwendet werden, so dass SKMS auch die Verwaltung von Varianten unterstützen, z.B. in dem für eine Schnittstelle mehrere Realisierungen verwaltet werden.

Konstruktion

Die Software-Elemente, welche die Entwickler bearbeiten, unterscheiden sich meist von den Elementen, die an die Benutzer des Anwendungssystems ausliefern, z.B. bearbeiten die Entwickler in der Regel Quellcode, dessen Kompilat ausgeliefert wird. Der Kompilierprozess muss nach jeder Änderung wiederholt werden und ist – gerade bei vielen Software-Elementen – zeitaufwändig, so dass er weitestgehend automatisiert wird. Dies ist Aufgabe eines SKMS, in dem z.B. nach einer Änderung nur betroffene Elemente neu kompiliert werden.

Prüfung

Das entwickelte Anwendungssystem soll möglichst vollständig und korrekt ausgeliefert werden. Um das sicherzustellen, sind Prüfungen auf Vollständigkeit und Korrektheit nötig. Dies wird z.B. durch zusätzliche Überprüfungen von Änderungen durch weitere Entwickler realisiert: Sobald eine Änderung an Software-Elementen abgeschlossen ist, werden die Änderungen von einem zweiten Entwickler überprüft, der die Änderungen bestätigt oder ablehnt. Ein SKMS unterstützt die Überprüfung, in dem es z.B. den Status der Elemente verwaltet und ungeprüfte Elemente nur den beteiligten Entwicklern zugänglich macht.

Struktur

Eine Aufgabe eines SKMS ist es, die Entwickler bei der konsistenten Auswahl voneinander abhängiger Elemente zu unterstützen. Zwischen den Software-Elementen eines Anwendungssystems existieren verschiedene Abhängigkeiten, z.B. unterscheidet sich die Abhängigkeit zwischen Quellcode und kompiliertem Code von einer Import-Beziehung zwischen zwei Modulen. Da in einem SKMS die Elemente in unterschiedlichen Entwicklunsgzuständen vorhanden sind, ist es wichtig, immer eine konsistente Menge an Elementen auszuwählen. Soll z.B. der frühere Entwicklungszustand eines Moduls wiederhergestellt werden, so muss sichergestellt sein, dass auch die geeignete Schnittstelle gewählt wird.

1.2. Anforderungen an SKMS

Unterschiedliche Entwicklungsprozesse stellen unterschiedliche Anforderungen an das unterstützende SKMS. Je nachdem, welche der acht Bereiche aus Abbildung 1.1 Teil des Prozesses sind, können die Anforderungen den Bereichen zugeordnet werden. Ein SKMS

unterstützt einen Prozess letztendlich nur dann, wenn es dessen Anforderungen erfüllt. Es ist daher notwendig, die Funktionalität und Beschränkungen existierender SKMS zu kennen, um festzustellen, ob ein SKMS zur Unterstützung eines Entwicklungsprozesses geeignet ist oder nicht. Daraus lässt sich folgende Anforderung ableiten:

> **Anforderung 1:** Existierende SKMS sollten systematisch beschrieben werden, um sie vergleichbar zu machen und die Auswahl an Hand der Anforderungen des Entwicklungsprozesses zu erleichtern.

Wurde ein SKMS zur Unterstützung eines Entwicklungsprozesses verwendet, so ist es wünschenswert, es für alle folgenden Prozesse einzusetzen. Dies ist möglich, solange es die Anforderungen der Folgeprozesse erfüllt. Verändern sich die Anforderungen, so ist das SKMS im Idealfall konfigurierbar oder zumindest erweiterbar und kann so an die veränderten Anforderungen angepasst werden. Ist das nicht möglich, so muss – wie beim ersten Prozess – das SKMS neu gewählt oder sogar implementiert werden.

Die Software-Elemente, die durch das SKMS verwaltet werden, stellen u.A. Anforderungen bezüglich: **Speicherplatz**, **Zugriffsdauer** und Verwaltung von **Abhängigkeiten**. So erfordert die Verwendung von Videodaten deutlich mehr Speicherplatz als Quellcode und die Verwaltung vieler kleiner Elemente schnellere Zugriffszeiten als die weniger großer. Insbesondere neue Entwicklungsverfahren verwenden neue Typen von Elementen, die ebenfalls mittels SKMS verwalten werden sollen. Dies führt zu neuen Anforderungen an Speicherverfahren, Darstellungsformen und Algorithmen eines SKMS. Ein Beispiel ist die Modellgetriebene Software-Entwicklung: Statt Quellcode-Elementen werden nun grafische Modelle verwendet, so dass nun ein grafisches Werkzeug zur Anzeige eines Modells erforderlich ist. Insbesondere beim Vergleich zweier grafischer Modelle sind diese neuen Anforderungen deutlich zu erkennen: Die grafische Darstellung der Differenz ist erwünscht, jedoch selbst noch Gegenstand grundlegender Forschung [Uhr08].

> **Anforderung 2:** SKMS sollten sich an die Software-Elemente anpassen lassen, die sie verwalten.

Durch den Einsatz neuer Entwicklungsmethoden werden auch neue Anforderungen an die Organisationsformen und somit auch das SKMS gestellt. Ein Beispiel sind agile Software-Entwicklungsmethoden, wie z.B.bei Paar-Programmierung: In diesem Fall sind zwei Entwickler für Änderungen an Quellcode-Elementen verantwortlich und nicht nur der Entwickler, der die Änderungen speichert.

> **Anforderung 3:** SKMS sollten sich an die Entwicklungsprozesse anpassen lassen, die sie unterstützen.

Vorgehensmodelle, wie z.B. der Rational Unified Process, sind Schablonen, die an die speziellen Bedürfnisse einer Firma oder sogar eines einzelnen Projektes angepasst werden [Kru03]. Weiterhin werden die Entwicklungsprozesse regelmäßig analysiert und die so gewonnen Erkenntnisse zu Verbesserung – und damit Änderung – des Entwicklungsprozesses verwendet. **Änderungen sind somit die Regel – und *nicht* die Ausnahme**. Das bedeutet, dass die Software, die zur Unterstützung eines Entwicklungsprozesses

verwendet wird, regelmäßig an die Änderungen angepasst werden sollte. SKM ist als Unterstützungsprozess ebenfalls von Änderungen betroffen, so dass auch SKMS von regelmäßigen Änderungen betroffen sind [Est95] [ZS97].

> **Anforderung 4:** SKMS sollten möglichst konfigurierbar und erweiterbar sein, um bei einer Änderung angepasst – und nicht ersetzt – zu werden.

Außer den systematischen Anforderungen stellen die Entwickler zusätzlich individuelle Anforderungen an die Bedienbarkeit und Darstellung eines SKMS. Insbesondere die Benutzerschnittstelle ist in diesen Fällen von zentraler Bedeutung. Interessanterweise erhält dieser Faktor nur wenig Aufmerksamkeit. Ein Großteil der SKMS-Werkzeuge arbeitet z.B. nur auf der Kommandozeile, so dass sie auf textuelle Darstellungsformen beschränkt sind. Oft müssen die Entwickler die Konsequenzen komplexer Operationen ohne Werkzeugunterstützung erfassen, d.h. sie müssen die Operationen mental „ausführen“ um ihr Resultat zu erhalten. Ein Beispiel ist das übliche Vorgehen zur Aktualisierung des lokalen Arbeitsbereichs: Will der Entwickler die neusten Änderungen aus dem zentralen Repository einspielen, aktualisiert er seinen Arbeitsbereich. Im Falle konkurrierender Änderungen werden die betroffenen Elemente automatisch verschmolzen. Ist dies nicht möglich, so muss der Entwickler den Konflikt manuell auflösen. Bei vielen SKMS-Werkzeugen muss der Entwickler die Konflikte herbeiführen, um sie zu erkennen, da es kaum Werkzeuge gibt, die das Resultat der Aktualisierung anzeigen, ohne sie auszuführen. Dieser Mangel an Darstellungsformen erschwert den Zugang und die Arbeit in der SKM-Domäne.

1.3. SKMS heute

Aktuell existieren mindestens 70 verschiedene SKMS [Rev10a, Rev10b]. Meist handelt es sich dabei um monolithische Systeme, bei denen – im Gegensatz zu einem modularen System – nur wenige Schnittstellen und ein geringer Grad an Kapselung existieren: Die Funktionen sind daher stark von den internen Datenstrukturen abhängig, was dazu führt, dass Änderungen verschiedene Positionen im Quellcode des SKMS betreffen. Dadurch wird die Planung, Durchführung und Validierung von Änderungen sehr aufwändig, so dass sich die Struktur und Funktionalität eines monolithischen Systems nur wenig oder überhaupt nicht anpassen lässt [BD09b].

Die monolithische Architektur und die enge Verschränkung von Daten und Funktionen erschweren (bzw. verhindern) die Wiederverwendung von bestehenden Systemen. Dadurch erfordert die Weiterentwicklung einzelner Funktionalitäten eines SKMS oft eine vollständige Neuimplementierung des kompletten Systems, was bei einem Umfang von mehreren hunderttausend bzw. Millionen Quellcodezeilen einen hohen Entwicklungsaufwand bedeutet. Dies lässt sich gut am Beispiel der Entwicklungsgeschichte des **C**oncurrent **V**ersions **S**ystem (CVS) und **S**ub**v**ersio**n** (SVN) illustrieren: CVS wird seit 1986 entwickelt. Ursprünglich eine Sammlung von Shell-Skripten, wurde es einige Jahre später sowohl in C als auch C++ reimplementiert [Ced05]. Beide Implementierungen besitzen die gleiche Kommandozeilen-Schnittstelle und sogar die gleiche Struktur: Die C++-Implementierung verwendet *keine* Klassen, um CVS zu beschreiben. SVN wird seit 2000

entwickelt und ist als Sammlung gemeinsam nutzbarer C-Bibliotheken entworfen. Es besitzt das Projektziel „ein neues SKMS zu implementieren, das die grundlegenden Konzepte von CVS übernimmt, ohne seine Fehler zu kopieren.“ [CSFP08].

Alle drei Projekte stellen ihren Quellcode als freie Software zur Verfügung, und dennoch wird kaum Quellcode in den Projekten wiederverwendet. Gerade zwischen den beiden C-Implementierungen von CVS und SVN hätte wiederverwendbarer Quellcode den Entwicklungsaufwand deutlich reduzieren können, da beide Projekte einen Umfang von mehreren hunderttausend Zeilen Quellcode besitzen. Abbildung 1.3(a) zeigt die zeitliche Entwicklung des Umfangs in Codezeilen der C-Implementierung von CVS, während Abbildung 1.3(b) die von SVN zeigt.

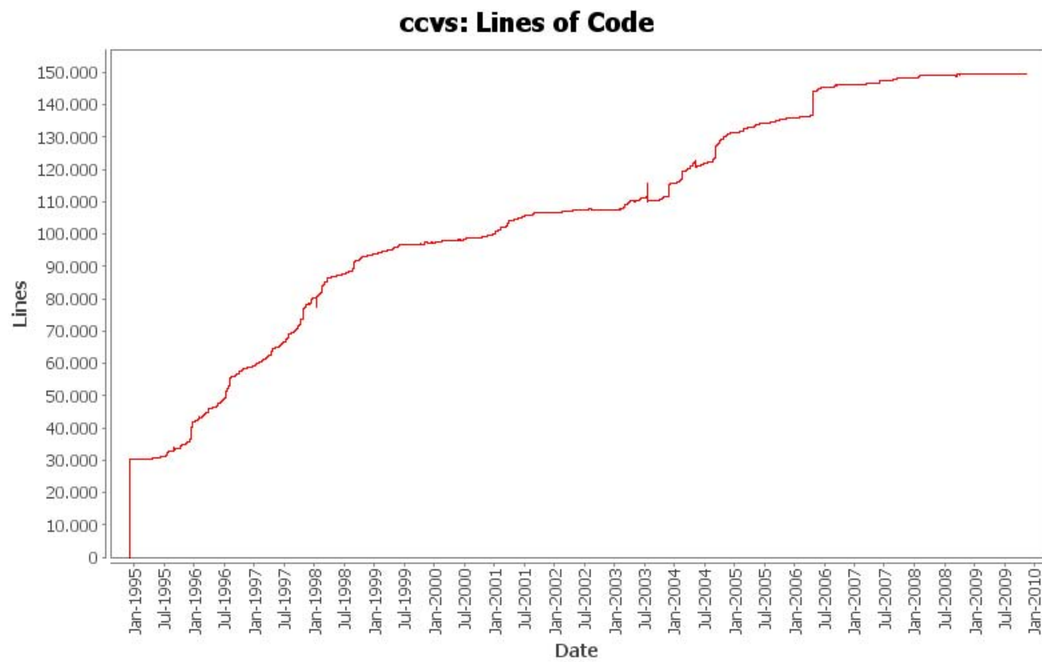
1.4. Probleme heutiger SKMS

Die meisten SKMS sind seit mehreren Jahren in Entwicklung und auch für Benutzer gut dokumentiert. Fast zu jedem System existiert ein Handbuch mit ausführlichen Beispielen [Ced05] [CSFP08] [Ca09] [O'S09]. Dadurch werden zwar Techniken zum effektiven Einsatz des jeweiligen SKMS beschrieben, doch die zugrunde liegenden Methoden und Verfahren werden nur als Black-Box präsentiert. So bleibt, im Falle eines Open-Source-SKMS, lediglich die implizite Beschreibung durch den Quellcode, um Rückschlüsse auf die Verfahren und Methoden zu ziehen. Bei einem proprietären SKMS steht selbst dieser nicht mehr zur Verfügung, so dass lediglich aufwändige Black-Box-Tests bleiben, um überhaupt Vermutungen über die eingesetzten Verfahren anstellen zu können.

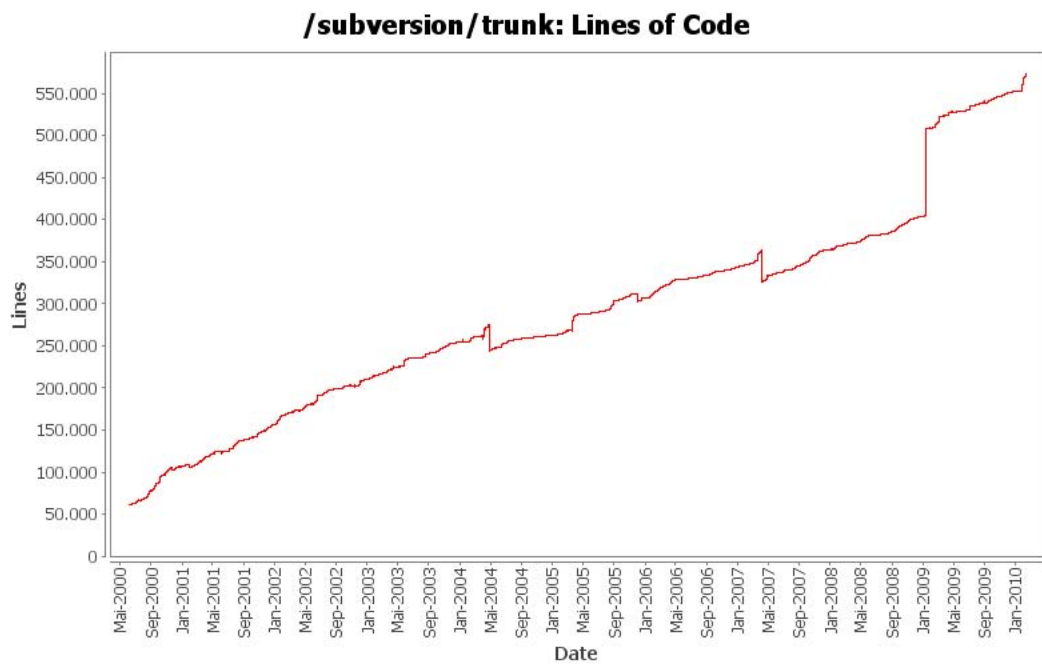
> **Problem 1 (Implizite Beschreibung):** Die 1. Anforderung (s. S. 20) ist nicht erfüllt: Die implizite Beschreibung durch Quellcode erschwert die Untersuchung, den Vergleich und die Diskussion unterschiedlicher Verfahren und Methoden. Eine systematische Auswahl bzw. Anpassung eines SKMS an ein Projekt oder einen Entwicklungsprozess ist nur mit hohem Aufwand möglich.

Bestehende SKMS sind in den meisten Fällen über das Dateisystem miteinander gekoppelt, d.h. es existiert ein Arbeitsbereich, in dem eine konsistente Version des Anwendungssystems abgelegt wird. Existierende Entwicklungswerkzeuge werden nun so eingesetzt, als ob im Arbeitsbereiche ein unversioniertes Anwendungssystem existiert d.h. der Arbeitsbereich bildet die Schnittstelle zwischen dem SKMS und den restlichen Werkzeugen, die im Entwicklungsprozess eingesetzt werden. Dadurch sind SKMS auf Dateien als Software-Elemente beschränkt, was sich insbesondere bei Modellgetriebener Software-Entwicklung als Hindernis erweist, da einzelne Modellelemente bzw. Teilmodelle nicht versioniert werden können [AKK+08].

> **Problem 2 (Dateisystem-fixiert):** Die 2. Anforderung (s. S. 20) ist nicht erfüllt: Der Fokus auf das Dateisystem als Schnittstelle beschränkt existierende SKMS auf bestimmte Entwicklungsmethoden. Durch die monolithische Struktur ist es nicht möglich, den Typ der versionierten Software-Elemente auszutauschen, so dass eine Neuimplementierung eines vollständigen SKMS angestrebt wird [AKK+08].



(a) CVS in C



(b) Subversion

Abbildung 1.3.: Veränderung des Umfangs in Quellcodezeilen

Aufgrund ihrer monolithischen Struktur sind die Daten und Funktionen heutiger SKMS eng gekoppelt und daher nur schwer zu erweitern. Es existieren wenig austauschbare Module oder Komponenten. Dies ist eine Erklärung für die große Zahl an SKMS, die meist auf den gleichen Methoden und Verfahren basieren und sich lediglich in den eingesetzten Techniken und Programmiersprachen unterscheiden: Anstatt ein neues Verfahren in ein bestehendes SKMS zu integrieren, wird ein vollständig neues SKMS entwickelt. Auch dieses System wird monolithisch um das neue Verfahren herum aufgebaut, so dass es ebenfalls schwer erweiterbar wird.

> **Problem 3 (Monolithische Architektur):** Die 3. Anforderung (s. S. 20) ist nicht erfüllt: Durch die monolithische Struktur können SKMS entweder nur unsystematisch und mit großem Aufwand an die Anforderungen angepasst werden, oder es ist sogar notwendig, ein neues SKMS zu wählen bzw. zu implementieren.

Eine Folge davon ist die parallele Entwicklung einer großen Anzahl an SKMS, über die die grundlegenden Methoden und Verfahren der SKM-Domäne unsystematisch verteilt sind. Die bisherigen Vergleiche von SKMS bestätigen, dass mehr SKMS als Methoden existieren [CW98]. Die Systeme unterscheiden sich lediglich in ihrer Kombination der Methoden und deren Implementierung. Durch diese unsystematische Kombination wird die Analyse und Bewertung der SKM-Methoden erschwert: Es ist kaum nachvollziehbar, aus welchen Gründen eine Methode gewählt wurde. Und die enge Kopplung innerhalb eines monolithischen SKMS verbirgt die tatsächlichen Abhängigkeiten zwischen den Methoden und Verfahren.

> **Problem 4 (Mangelnde Kombinationsmöglichkeiten):** Die 4. Anforderung (s. S. 21) ist nicht erfüllt: Die systematische Kombination mehrerer SKM-Methoden und -Verfahren ist nur durch zeitaufwändige Neuimplementierung eines SKMS möglich. Dies erschwert die Erforschung der SKM-Domäne, da neue Methoden und Verfahren nur mit großem Zeitaufwand untersucht werden können.

1.5. Verwandte Arbeiten

Es existieren mehrere Verfahren, die ein oder mehrere der oben beschriebenen Probleme lösen sollen. Diese Verfahren lassen sich in drei Methoden einteilen:

1. Generisches Versionsmodell
2. Quellcode-Kombinations-Rahmenwerk
3. Quellcode-Generator

Generischen Versionsmodelle verwenden ein einheitliches Meta-Datenmodell, das als Schema für die Implementierung unterschiedlicher Datenmodelle eingesetzt wird. Ziel

dieser Methode ist es, mit einem einzigen Verfahren möglichst viele Datenmodelle mit Hilfe elementarer Datentypen zu beschreiben. Und auch die Funktionalität des SKMS wird durch Elementaroperationen auf dem einheitlichen Datenmodell beschrieben. Wiederverwendung wird in dieser Methode durch die Wiederverwendung der Elementaroperationen erreicht. Das **U**nified **V**ersion **M**odel (UVM) [WMC01], das **I**ncremental **C**onfiguration **E**nvironment (ICE) [ZS97], Molhado [NNMB05] und das **U**nified **E**xtensional **V**ersioning **M**odel (UEVM) [ABCM99] gehören zu dieser Methode.

Bei Quellcode-Kombinations-Rahmenwerken wird die Funktionalität eines SKMS manuell implementiert und anschließend der Quellcode in Fragmente zerlegt. Jedes Quellcode-Fragment wird nun einer oder mehreren SKMS-Implementierungen zugeordnet, in denen es verwendet wird. Durch Auswahl einer bestimmten Implementierung werden die benötigten Quellcode-Fragmente selektiert und zu einem SKMS zusammengesetzt. Wiederverwendung wird in dieser Methode durch Wiederverwendung von Quellcode-Fragmenten in mehreren Implementierungen erreicht. Die Rahmenwerke *Bamboo* [Guo08] und *MCCM* [Hoe00] gehören zu dieser Methode.

Quellcode-Generatoren erzeugen Quellcode aus grafischen Modellen, indem sie die Modellelemente auf Quellcode-Vorlagen abbilden. Durch Annotation der Modellelemente kann die Abbildung zusätzlich noch parameterisiert werden, um die Quellcode-Generation zu steuern. Meist lässt sich jedoch nur der Quellcode für die Struktur – und nicht für das Verhalten – generieren, so dass das Verhalten mit Hilfe der Quellcode-Vorlagen spezifiziert wird. Mit Hilfe der Annotationen können dann die benötigten Quellcode-Fragmente in der Vorlage gewählt werden, um so – ähnlich wie bei der bedingten Übersetzung – das gewünschte Verhalten zu erzeugen. Wiederverwendung wird in dieser Methode durch die Wiederverwendung der Quellcode-Vorlagen beim Generieren eines SKMS erreicht. Durch Annotation der Modellelemente mit den Merkmalen einer Produktlinie (s. Abschnitt 1.6) ist diese Methode ein Schritt in Richtung modellgetriebener Produktlinien. Der **V**ersioning **S**ystems **Gen**erator (VS-Gen) [Kov05] gehört zu dieser Methode.

1.6. Modellgetriebene Produktlinien

Eine Produktlinie versucht, die Gemeinsamkeiten von Produkten auszunutzen, die für ein und dieselbe Domäne entwickelt werden. Im Falle einer Software-Produktlinie ist ein Produkt ein Software-Anwendungssystem. Zunächst werden gemeinsame und unterschiedliche Merkmale bestehender – und insbesondere zu entwickelnder – Systeme identifiziert. Für jedes gemeinsam verwendete Merkmal wird nun eine Software-Komponente implementiert und einer domänenspezifischen Software-Bibliothek hinzugefügt. Anschließend kann die Komponente in allen Produkten (d.h. Software-Systemen) wiederverwendet werden, die dieses Merkmal besitzen. Bei Entwicklung eines neuen Software-Anwendungssystems werden, an Hand seiner Merkmale, nur noch die bestehenden Komponenten ausgewählt. Dies reduziert den Entwicklungsaufwand eines Produktes durch Wiederverwendung, da lediglich für einzigartige Merkmale neue Komponenten entwickelt werden müssen [WL99] [CN01] [PBL05].

Eine **Mod**ellgetriebene Software **P**rodukt**l**inie (MODPL) ersetzt die Bibliothek domä-

nenspezifischer Software-Komponenten durch ein domänenspezifisches Modell. Ein Produkt entsteht nun nicht mehr durch manuelle Implementierung, sondern durch Modelltransformation: Zunächst wird an Hand der Merkmale eine Konfiguration des Produktes erstellt. Diese wird in der nun folgenden Modelltransformation dazu verwendet, um das Domänenmodell in ein Modell des gewünschten Produktes umzuwandeln. Aus diesem Modell wird letztendlich ein ausführbares Software-System generiert, das der gewünschten Konfiguration entspricht.

1.7. MOD2-SKM: Eine MODPL für SKMS

Die große Zahl an bereits existierenden SKMS und die geringe Zahl an grundlegenden Methoden [CW98] lassen vermuten, dass eine Produktlinie für die SKM-Domäne entwickelt werden kann.

> **Hypothese 1:** Es ist möglich eine MODPL für SKMS zu entwickeln, d.h. es existiert eine ausreichende Anzahl gemeinsamer Merkmale für bestehende SKMS. Gleichzeitig gibt es auch ausreichend Unterschiede, so dass eine Systemfamilie entstehen kann.

Dazu ist es notwendig, die Verfahren der SKM-Domäne zu entkoppeln, um sie möglichst unabhängig voneinander wiederverwenden zu können. Nur so kann ein Domänenmodell entstehen, das durch eine Konfiguration in das gewünschte SKMS transformiert werden kann.

> **Hypothese 2:** Die Verfahren der SKM-Domäne lassen sich lose koppeln, so dass sie modularisiert werden können. Somit bietet sie einen Lösungsansatz für Problem 3: Monolithische Architektur (s. S. 24).

Eine MODPL für SKMS besteht aus einer Merkmals- und einem Domänenmodell. Im Merkmalsmodell wird die SKM-Domäne systematisch analysiert und dokumentiert, während das Domänenmodell die einzelnen Merkmale und ihre Abhängigkeiten untereinander modelliert. In [BDW08a] werden ein Merkmals- und ein Domänenmodell für VKS vorgestellt, was vermuten lässt, dass sich der Produktlinien-Ansatz auch auf SKMS allgemein anwenden lässt.

> **Hypothese 3:** Eine MODPL für SKMS erfüllt die 1. Anforderung (s. S. 20), und erlaubt es, SKMS systematisch zu beschreiben und zu vergleichen, da SKMS sich anhand der Merkmale des Merkmalsmodells kategorisieren lassen. Somit bietet sie einen Lösungsansatz für Problem 1: Implizite Beschreibung (s. S. 22).

Aus dem Domänenmodell kann – mit Hilfe einer Konfiguration – ein Produkt, d.h. ein konkretes SKMS generiert werden, das der Konfiguration entspricht.

Hypothese 4: Eine MODPL für SKMS erfüllt die 4. Anforderung (s. S. 21), und erlaubt es, SKMS schnell an veränderte Anforderung anzupassen, da sich der Entwicklungsaufwand eines SKMS im Idealfall auf das Erstellen einer Konfiguration beschränkt. Bei neuen Merkmalen beschränkt sich der Aufwand auf die Modellierung der neuen Funktionalität, da bestehende Merkmale wiederverwendet werden können. Somit bietet sie einen Lösungsansatz für Problem 4: Monolithische Architektur (s. S. 24).

Durch Definition des Datenmodells des Produkts als optionales Merkmal wird die Entwicklung von SKMS, die nicht auf einem Dateisystem-Datenmodell basieren, Aufgabe der Domänenentwicklung. Damit bietet MOD2-SKM einen Lösungsansatz für Problem 2: Dateisystem-fixiert (s. S. 22).

1.8. Ergebnisse und Aufbau der Arbeit

Die Untersuchung der aufgestellten Hypothesen erfolgt auf Grundlage einer selbst entwickelten **Mod**ellgetriebenen **Mod**ularen Produktlinie für **S**oftware-**K**onfigurations-**M**anagement Systeme (MOD2-SKM). In dieser Arbeit wird MOD2-SKM beschrieben, um sie anschließend zum Belegen oder Widerlegen der Hypothesen zu verwenden.

Die Umsetzung von MOD2-SKM erfolgt mit Hilfe des MODPL-Rahmenwerks und der zugehörigen Entwicklungsumgebung. Abbildung 1.4 gibt sowohl einen Überblick über die Werkzeuge des MODPL-Rahmenwerks als auch die Bestandteile von MOD2-SKM und damit über die Ergebnisse dieser Arbeit. Sie zeigt:

1. die eingesetzten Werkzeuge (*grünes Rechteck*).
2. die Aufgaben, für die ein Werkzeug eingesetzt wird (*abgerundetes Rechteck*).
3. die Modelle, für die ein Werkzeug eingesetzt wird (*normales Rechteck*).
4. welches Modell das Ergebnis einer Aufgabe ist (*Pfeil von Aufgabe zu Modell*).
5. welche Modelle in einer Aufgabe verwendet werden (*Pfeil von Modell zu Aufgabe*).
6. die von mir durchgeführten Aufgaben und erstellten Modelle, d.h. die Ergebnisse dieser Arbeit (*gelber Hintergrund*)
7. die automatisiert durchgeführten Aufgaben (*grau-blauer Hintergrund*)

Grundlage von MOD2-SKM ist eine Domänenanalyse, die die gemeinsamen und unterschiedlichen Merkmale für eine Produktlinie für SKMS identifiziert und beschreibt. Die Merkmale werden durch die Analyse bereits bestehender SKMS, deren Dokumentation sowie wissenschaftlicher Artikel über SKM-Methoden gewonnen und – mittels des *Feature Plugins* [AC04] – als Merkmalsmodell beschrieben. Aus dem Merkmalsmodell werden dann, durch Selektion von Merkmalen, Konfigurationen erstellt. Dabei wird der Auswahlprozess durch Abhängigkeiten im Merkmalsmodell unterstützt, indem bei Auswahl eines Merkmals alle benötigten Merkmale automatisch ausgewählt werden.

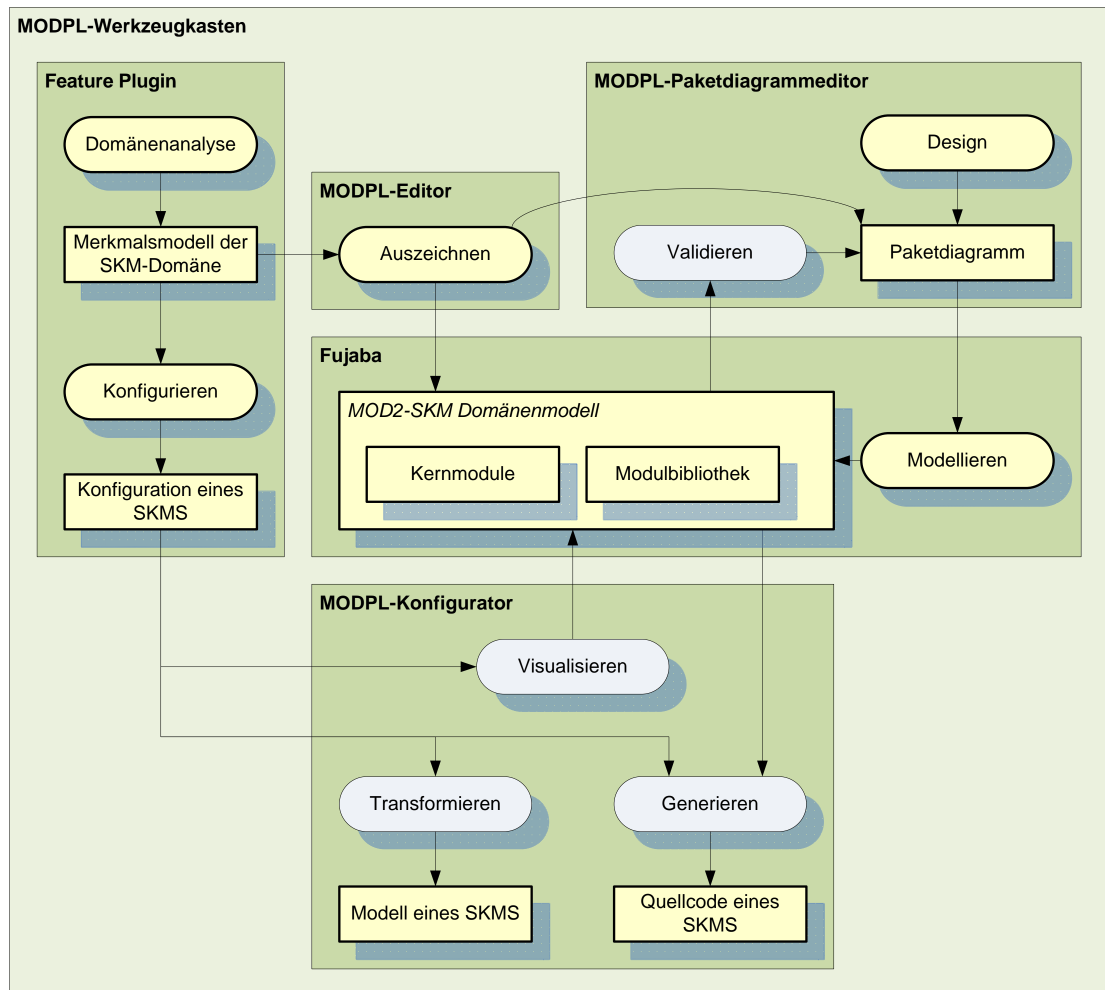


Abbildung 1.4.: Überblick über die MOD2-SKM Produktlinie

Die Datenstrukturen und Operationen von MOD2-SKM werden mit Hilfe von mehreren Modellen mit unterschiedlichen Detailgraden beschrieben. Die grobe Architektur von MOD2-SKM wird mit Hilfe von Paketen in der Aufgabe Design entworfen, und – mittels des MODPL-Paketdiagrammeditors – als UML-Paketdiagramm modelliert. Auf Basis des Paketdiagramms sind dann in Fujaba die Datentypen (als UML-Klassendiagramme) und Operationen (als Fujaba-Storydiagramme) des MOD2-SKM Domänenmodells modelliert. Das Domänenmodell kann gegen das Paketdiagramm validiert werden, um sicherzustellen, dass die Datentypen gemäß ihrer Sichtbarkeiten im Paketdiagramm verwendet werden.

Der MODPL-Editor wird zum Auszeichnen des Paketdiagramms und des Domänenmodells verwendet. Dabei werden Modellelemente (z.B. Pakete, Klassen oder Methoden) mit Merkmalen aus dem Merkmalsmodell annotiert. Ein annotiertes Element ist nur dann in einem generierten SKMS vorhanden, wenn alle annotierten Merkmale Teil seiner Kon-

figuration sind. Innerhalb des Domänenmodells selbst wird zwischen Kernmodulen und der Modulbibliothek unterschieden. Die Kernmodule sind nicht annotiert und in jedem generierten SKMS vorhanden, während die Module in der Modulbibliothek annotiert und von den ausgewählten Merkmalen einer Konfiguration abhängig sind.

Der MODPL-Konfigurator erzeugt nun aus dem MOD2-SKM Domänenmodell und einer Konfiguration das entsprechende SKMS. Es existieren drei unterschiedliche Verfahren, die Konfiguration auf das Domänenmodell anzuwenden:

1. Visualisieren
2. Transformieren
3. Generieren

Beim Visualisieren werden die Elemente des Domänenmodells hervorgehoben, die Teil des konfigurierten SKMS sind, d.h. sie sind entweder immer Teil eines SKMS oder mit den Merkmalen aus der Konfiguration ausgezeichnet. Dieses Verfahren hat eine Vorschaufunktion für die anderen beiden. Beim Transformieren wird das Domänenmodell in ein Modell eines SKMS überführt, das den Merkmalen in der Konfiguration entspricht. Dieses SKMS-Modell kann nun weiter bearbeitet und um einzigartige Funktionalität erweitert werden. Beim Generieren wird sofort der Quellcode des konfigurierten SKMS erzeugt – so, als ob der Quellcode aus dem unveränderten, transformierten Modell generiert worden wäre.

1.8.1. Ergebnisse und Forschungsbeitrag

Zentraler Teil von MOD2-SKM ist zum einen die Domänenanalyse. Ergebnis ist ein Merkmalsmodell für die SKM-Domäne. Mit Hilfe des Merkmalsmodells lassen sich die SKMS, die mit MOD2-SKM generiert werden, auf einer einheitlichen Basis vergleichen. Zusätzlich lassen sich bestehende SKMS mit Hilfe des Merkmalsmodells klassifizieren und so untereinander und mit den MOD2-SKM-Systemen vergleichen. Das Merkmalsmodell leistet damit einen Beitrag zu einer einheitlichen Klassifikation von SKMS und erleichtert so die Auswahl eines geeigneten SKMS für Entwicklungsprozesse oder Vorgehensmodelle.

Zwei weitere Schwerpunkte liegen auf der Modellierung der Architektur und der Domäne: Ergebnis der Architekturmodellierung ist das Paketdiagramm, das, mit Hilfe von Namensräumen und Sichtbarkeitsregeln, den Aufbau von MOD2-SKM beschreibt. Das andere Ergebnis ist das Domänenmodell mit seinen Kernmodulen und der Modulbibliothek. Sie modellieren Methoden und Verfahren aus der SKM-Domäne mit Hilfe ausführbarer grafischer Modelle. Zusätzlich analysiert und dokumentiert das Domänenmodell auch die Kopplung zwischen den einzelnen Aufgabenbereichen und Methoden von SKMS. Das Domänenmodell leistet so einen Beitrag für eine neue Generation von SKMS, die durch den Produktlinienansatz und ihre modulare Architektur die Wiederverwendbarkeit von SKMS erhöhen. Zusätzlich leistet diese Arbeit auch einen Beitrag zur Analyse und Modellierung von SKMS-Methoden sowie der Architektur von SKMS.

MOD2-SKM ist auch eine nicht-triviale[2] Fallstudie eines MODPL-Projekts, d.h. der MODPL-Prozess und die verwendeten Modellierungstechniken und -werkzeuge werden analysiert und bewertet. Letztendlich ist MOD2-SKM auch eine Fallstudie für Modellgetriebene Software-Entwicklung im Allgemeinen, so dass auch hier eine abschließende Analyse und Bewertung erfolgt. Somit leistet diese Arbeit auch einen Beitrag zur Untersuchung und Verbesserung von MODPL im Speziellen und Modellgetriebener Entwicklung im Allgemeinen.

1.8.2. Aufbau der Arbeit

Die ersten drei Kapitel erläutern die Grundlagen, auf denen MOD2-SKM aufbaut: Kapitel 2 beschreibt vorausgegangene und verwandte Arbeiten, der Erkenntnisse die Entwicklung von MOD2-SKM beeinflusst haben. Kapitel 3 stellt den MODPL-Prozess vor und erläutert die modellgetriebene Entwicklung von Produktlinien. Anschließend wird die SKM-Domäne in Kapitel 4 vorgestellt und in ihre grundlegenden Methoden eingeführt. Die weiteren Kapitel beschreiben und analysieren MOD2-SKM und folgen dabei ihrer Entwicklung und damit den Schritten des MODPL-Prozesses: Zunächst wird in Kapitel 5 ein Merkmals-Modell der SKM-Domäne definiert und zur Erstellung eines konfigurierbaren Domänenmodells herangezogen: Dazu wird in Kapitel 6 die Architektur des Domänenmodells, der Kern von MOD2-SKM und die Modulbibliothek mit ihren Verbindungen zum Merkmalsmodell vorgestellt. Dabei sind die Kopplungen zwischen den einzelnen Komponenten des Domänenmodells von zentraler Bedeutung: Eine Analyse der Kopplungen erfolgt am Ende des Kapitels in Abschnitt 6.5. Anschließend wird in Kapitel 8 MOD2-SKM analysiert, um die obigen 4 Hypothesen zu belegen oder zu widerlegen. Abschließend werden aus den nun explizit vorhandenen Modellen die Abhängigkeiten zwischen SKM-Methoden abgeleitet, Schlussfolgerungen für die Entwicklung zukünftiger SKMS gezogen und ein Ausblick auf zukünftige Forschungsmöglichkeiten gegeben.

1.9. Zusammenfassung

Heutige Software-Konfigurations-Management-Systeme (SKMS) sind monolithische und schwer erweiterbare Systeme, die mit großem Aufwand an die modernen Entwicklungsprozesse angepasst oder sogar extra neu implementiert werden. Daher werden sie unsystematisch modifiziert und neu entwickelt. So ist es nahezu unmöglich, systematische Vergleiche zwischen bestehenden SKMS anzustellen. Zusätzlich sind die meisten SKM-Verfahren nur implizit durch den Quellcode beschrieben, bzw. bei proprietären Systemen nur aus Black-Box-Tests ableitbar. Dies führt letztendlich dazu, dass die Methoden der SKM-Domäne nur aufwändig verglichen und erforscht werden können. Insbesondere die Abhängigkeiten unterschiedlicher Verfahren werden durch die enge Kopplung der monolithischen Systeme verschleiert.

[2] Die Domänenanalyse in dieser Arbeit umfasst über 100 konfigurierbare Merkmale, während z.B. die Analyse in [Kov05] aus ca. 20 konfigurierbaren Merkmalen besteht.

Eine **Mod**ellgetriebene **Mod**ulare Produktlinie für **S**oftware-**K**onfigurations-**M**anagement Systeme (MOD2-SKM) bietet eine Lösung für diese Probleme:

1. MOD2-SKM definiert ein Merkmalsmodell, mit dessen Hilfe die generierten und auch die bestehenden SKMS klassifiziert und die eingesetzten Verfahren systematisch verglichen werden können.

2. MOD2-SKM ist ein Prototyp für eine neue Generation von SKMS. Durch den Produktlinienansatz und die modulare Architektur wird die Erweiterbarkeit erhöht und der Entwicklungsaufwand für neue SKMS reduziert, da konkrete SKMS nicht mehr implementiert, sondern gemäß des Merkmalsmodells konfiguriert und anschließend aus dem Domänenmodell generiert werden.

3. MOD2-SKM erleichtert die Erforschung neuer Verfahren und Methoden aus der SKM-Domäne, da nur das neue Verfahren modelliert werden muss, während die bereist existierenden Verfahren im Domänenmodell wiederverwendet werden. Zusätzlich wird der Zugang zu existierenden Methoden durch die Verwendung grafischer Modelle erleichtert.

2. Verwandte Arbeiten

Die große Anzahl an parallel existierenden SKMS und der hohe Zeitaufwand für ihre Implementierung führten zu der Beobachtung, dass die existierenden SKMS eine kleine Menge von Methoden immer wieder neu – jedoch in unterschiedlichen Kombinationen – implementieren [Fei91] [CW98]. Aus dieser Beobachtung wird geschlossen, dass durch Wiederverwendung bereits implementierter Verfahren der Implementierungsaufwand für neue SKMS gesenkt werden kann. Die fehlende Modularisierung bereits existierender SKMS verhindert jedoch effektiv die Wiederverwendung bereits implementierter Verfahren.

Es existieren bereits mehrere Forschungsarbeiten, die die Wiederverwendbarkeit von SKMS-Verfahren untersuchen. Dabei werden auch ein oder mehrere Komponenten von SKMS identifiziert, die wiederverwendet werden sollen. Die Forschungsarbeiten lassen sich dabei in drei unterschiedliche Kategorien einteilen: Generische Versionsmodelle, Quellcode-Kombinations-Rahmenwerke und Quellcode-Generatoren.

2.1. Generisches Versionsmodelle

Generische Versionsmodelle verwenden ein einheitliches Meta-Datenmodell, das als Schema für die Implementierung unterschiedlicher Datenmodelle eingesetzt wird. Ziel dieser Methode ist es, mit einem einzigen Verfahren möglichst viele Datenmodelle mit Hilfe elementarer Datentypen zu beschreiben. Und auch die Funktionalität des SKMS wird durch Elementaroperationen auf dem einheitlichen Datenmodell beschrieben. Wiederverwendung wird in dieser Methode durch die Wiederverwendung der Elementaroperationen erreicht.

2.1.1. ECM/UVM

Das **U**nified **V**ersion **M**odel (UVM) von Bernhard Westfechtel, Reidar Conradi und Bjørn P. Munch ist ein einheitliches Rahmenwerk für Versionsmodelle. Eine Referenzimplementierung von UVM wurde im Rahmen des SKMS **E**POS **C**onfiguration **M**anagement (ECM) System entwickelt. Kern von UVM ist eine instrumentierbare Versionierungsmaschine (engl. instrumentable version engine), die sich an die Anforderungen eines Versionsmodells anpassen lässt. Dabei werden keine Annahmen über die versionierten Elemente getroffen, so dass beliebige Software-Elemente (Dateien, Objektgraphen, etc.) gespeichert werden können [WMC01].

Abbildung 2.1 zeigt ein Schichtenmodell des UVM. Die unterste Schicht bildet ein *Deltaspeicher* mit selektiven intensionalen Deltas, d.h. alle Versionen eines Elements werden

Abbildung 2.1.: UVM: Idealisiertes Schichtenmodell (nach [WMC01])

überlappend gespeichert. Dazu wird das Element in Fragmente zerlegt und die Fragmente mit Sichtbarkeiten markiert. Um ein Element zusammenzusetzen wird anhand der Sichtbarkeiten bestimmt, ob das Fragment Teil des entsprechenden Elements ist oder nicht. Das Eintragen der Sichtbarkeiten in den Speicher und ihre Auswertung findet auf der Ebene der *Versionierungsregeln* statt. Sichtbarkeiten werden in einer 3-wertigen Logik durch Konjunktionen über einer Menge boolescher Variablen (sog. Optionen) ausgedrückt. Eine Option o_i kann entweder vorhanden (o_i), nicht vorhanden ($\neg o_i$) oder ungesetzt ($true$[1]) sein (vgl. [WMC01]). Eine mögliche Sichtbarkeitsregel über den Optionen $O = \{32bit, Linux, Mac\}$ wäre z.B. $s = \neg 32bit \wedge Linux\ Mac$ ist ungesetzt und kann als ($\wedge true$) eliminiert werden.

Mit Hilfe der Optionen und der Sichtbarkeiten können die Fragmente nun zum Lesen und Schreiben gefiltert werden. Während beim Lesen alle Optionen gesetzt sein müssen (d.h. es wird genau eine Version ausgewählt), reicht beim Schreiben aus, die Optionen nur teilweise zu binden (d.h. es wird eine Versionsmenge ausgewählt). Sind z.B. beim Schreiben alle Optionen undefiniert, so werden alle Fragmente im Deltaspeicher verändert. Die Sichtbarkeitsregeln werden in einer Regeldatenbank verwaltet, die sich um zusätzliche Bedingungen erweitern lässt. So können z.B. zusätzliche Abhängigkeiten eingeführt oder Optionen zum schnelleren Zugriff gruppiert werden.

Die Versionsmodelle werden nun mit Hilfe der Regeldatenbank definiert, d.h. für Revisionen werden Optionen eingeführt und ihre Abhängigkeiten untereinander durch Sichtbarkeitsregeln ausgedrückt. Ein Versionsmodell, bestehend aus Revisionen und einer linearen Historie, führt z.B. zur Einführung der Optionen Δ_j, die angibt, ob ein Fragment zum jeweiligen Delta gehört. Eine Revision r_i, die aus den Deltas $\{\Delta_x | x = 1, \ldots, i\}$

[1] Durch Abbilden einer ungesetzten Option auf den Wert *true* können diese aus den Konjunktionen eliminiert werden, so dass es genügt, die gesetzten Variablen eines Terms anzugeben.

besteht lässt sich so durch Aufzählen aller Δ_j-Optionen beschreiben:

$$r_i \;=\; \Delta_1 \wedge \ldots \wedge \Delta_i \wedge \neg\Delta_{i+1} \wedge \ldots \wedge \neg\Delta_n$$

Die nächsten drei Schichten bilden das SKM-Repositorium. Die unterste Schicht ist das Datenmodell, mit dem die zu speichernden Daten modelliert werden. Die Versionierungsmaschine behandelt die Fragmente und Elemente lediglich als Bytesequenzen, erst durch das Datenmodell werden die Sequenzen typisiert und so zu den Elementen, mit denen der Benutzer des SKMS letztendlich interagiert, z.B. Dateien oder Objekte. Zur Modellierung des Datenmodells wird ein Entitäts-/Beziehungsmodell verwendet, aber es können auch andere Datenmodelle verwendet werden. Die Transaktionsschicht koordiniert schließlich das gemeinsame Bearbeiten von Elementen, und erweitert die Regeldatenbank um neue Optionen zur Verwaltung von Transaktionen. So können Zugriffe blockiert oder Konflikte erkannt werden. Dazu verwendet die Transaktionsschicht Optionen aus allen darunterliegenden Ebenen, z.B. aus dem Datenmodell, um zu erkennen, ob ein Element gleichzeitig bearbeitet wird. Dabei können u.a. durch die Lese- und Schreibfilter Konflikte erkannt werden, bevor sie auftreten.

Bei der Untersuchung von ECM war zu beobachten, dass die Länge der logischen Ausrücke nicht-linear in Anzahl der Optionen ansteigt, so dass sich sowohl der Bedarf an Speicherplatz als auch die Rechenzeit zur Auswertung immer weiter erhöht. Durch Wiederverwendung von Teilausdrücken und Zwischenspeichern ihrer Auswertung kann beides optimiert werden, so dass sowohl Lesen als auch Schreiben in linearer Zeit erfolgen. Auf Grund der Komplexität der Sichtbarkeitsregeln ist ein zusätzliches Werkzeug für ihre Verwaltung notwendig [WMC01].

Die Erfahrungen mit ECM zeigen, dass sich einzelne Bereiche in Schichten voneinander entkoppeln lassen und z.B. das Datenmodell und das Versionsmodell unabhängig voneinander definieren lassen. Insbesondere die Entkopplung des Datenmodells erhöht die Wiederverwendungsmöglichkeiten von ECM, da sich unterschiedliche Daten mit Hilfe des gleichen darunterliegenden Systems versionieren lassen. Die logischen Ausdrücke sind offenbar, auf Grund ihrer Länge, als Beschreibungssprache für SKM-Datenstrukturen nur bedingt geeignet, während die SKM-Verfahren immer noch implizit im Quellcode von ECM enthalten sind.

2.1.2. ICE

Das **I**ncremental **C**onfiguration **E**nvironment (ICE) von Andreas Zeller ist ein VKS, bei dem der Zugriff auf die verwalteten Dateien über ein virtuelles Dateisystem erfolgt. Grundlage bildet das Versionsmengen-Datenmodell, das die versionierten Dateien mittels Merkmalslogik (engl. feature logic) attributiert und so klassifiziert. Dies ermöglicht die Realisierung unterschiedlicher VKS-Datenmodelle auf Basis eines einheitlichen Modells, so dass das Versionsmengen-Modell ein vereinheitlichendes Datenmodell für VKS darstellt [ZS97].

Ein Term in Merkmalslogik besteht aus der Verbindung von „$Merkmal$: $Wert$“-Tupeln. Folgender Term T beschreibt einige linguistische Eigenschaften des Satzes „x

singt y“ mit Hilfe von Merkmalslogik:

$$T = \begin{bmatrix} zeit : präsens, \\ prädikat : [verb : singen, handelnder : x, was : y], \\ subjekt : [x, num : singular, person : dritte], \\ objekt : y. \end{bmatrix}$$

Den Merkmalen (engl. features) werden entweder Konstanten (z.B. *präsens*), Variablen (z.B. x) oder wiederum Terme (z.B. $[verb : singen, handelnder : x, was : y]$)zugewiesen. Zusätzlich existieren noch boolesche Mengenoperationen, wie Schnitt ($A \cap B = [A, B]$), Vereinigung ($A \cap B = \{A, B\}$) und Komplement ($\neg A = \sim A$).

Im ICE-Datenmodell wird jedes Element durch einen eindeutigen Merkmalsterm identifiziert. So lassen sich die Elemente anhand einzelner Merkmale klassifizieren, d.h. sie bilden Mengen bzgl. bestimmter „$Merkmals : Wert$“-Tupel bzw. bestimmter Merkmalsterme. So kann z.B. eine Komponente Drucker, die in zwei alternativen Versionen existiert, wie folgt beschrieben werden:

$$\begin{aligned} drucker_1 &= [objekt : drucker, druckersprache : postscript] \\ drucker_2 &= [objekt : drucker, druckersprache : ascii] \end{aligned}$$

Die Menge aller Drucker lässt sich damit wie folgt beschreiben:

$$\begin{aligned} drucker_{alle} &= drucker_1 \cup drucker_2 \\ &= [objekt : drucker, druckersprache : \{postscript, ascii\}] \end{aligned}$$

Ein Repositorium von ICE lässt sich als Vereinigung aller Terme T_{repo} beschreiben.

Entitäten aus VKS-Datenmodellen, wie z.B. Revisionen, Historien und Synchronisationsmechanismen können nun mit Hilfe von Merkmalen beschrieben werden, z.B. indem ein Merkmal „rev“ (für Revisionsnummern) eingeführt wird, um eine Versionskennung zu beschreiben, oder ein Merkmal „lockedBy“, um das Sperren durch einen Benutzer auszudrücken. Auf Grund der Länge der Merkmalsterme ist es jedoch unbedingt erforderlich, dass der Anwender durch eine Abstraktionsschicht von den Termen getrennt wird. So wird z.B. die 2. Revision von $drucker_1$ durch folgenden Term beschrieben:

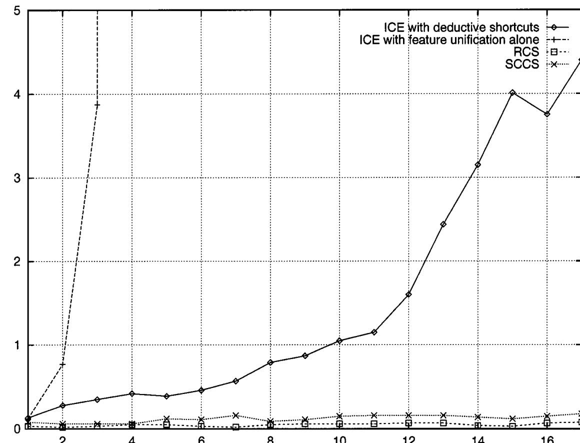
$$drucker_{1,2} = drucker_1 \cap [\neg rev : 193, \neg rev : 192, \ldots, \neg rev : 3, rev : 2, rev : 1]$$

Es sind in diesem Fall 193 „rev“-Merkmale nötig, um auszudrücken, dass in dieser Revision die Änderungen von Revision 1 und 2 aber nicht 3 bis 193 enthalten sind. Mit dem Merkmal „rev:2“ alleine wird die Menge alle Elemente identifiziert, welche die 2. Änderung enthalten. Im Falle einer linearen Entwicklung sind dies alle Revisionen von 2 bis 193 (denn auch Revision 3 enthält die Änderungen von Revision 2).

Um nun auf eine Menge E versionierte Elemente aus dem Repositorium T_{repo} zuzugreifen, gibt der Benutzer ebenfalls einen Merkmalsterm S ein. Die Auswahl erfolgt durch Schnitt des Anfrageterms S mit dem Repositoriums-Term T_{repo} des jeweiligen Elements, d.h. $E = T_{repo}$. Das virtuelle Dateisystem von ICE stellt eine typische Dateisystem-Kommandozeilenschnittstelle dar, die um die Eingabe von Merkmalstermen erweitert

```
commands.c[]
for (d = enter_file(".SUFFIXES")→deps; d ≢ 0; d = d→next)
  {
#if d370
    unsigned int slen = strlen(dep_name(d));
#else
    unsigned int len = strlen(file→name);
#endif
#if d374
    if (len > slen ∧ ¬strncmp(dep_name(d), name + (len − slen), slen))
#elif d370
    if (len > slen ∧ ¬strncmp(dep_name(d), name + len − slen, slen))
#else
    if (len > slen ∧ streq(dep_name(d), file→name + len − slen))
#endif
      {
#if d370
        file→stem = savestring(name, len − slen);
#else
        file→stem = savestring(file→name, len − slen);
#endif
        break;
      }
  }
if (d ≡ 0)
  file→stem = "";
```

(a) Multiversions-Datei



(b) Geschwindigkeit beim Commit

Abbildung 2.2.: Darstellung und Geschwindigkeit des ICE-VKS (nach [ZS97])

wurde. So kann durch Anhängen eines Merkmalsterms an einen Dateinamen auf eine bestimmte Datei zugegriffen werden (z.B. greift „drucker[druckersprache:postscript,rev:193, rev:192, rev:191,..., rev:2, rev:1]" auf $drucker_1$ zu). Wird durch den Anfrageterm eine Menge von Dateien ausgewählt, z.B. „commands.c[]" (keine weiteren Merkmale außer dem Namen), so werden diese als einzelne, mit C++-Präprozessoranweisungen ausgezeichnete, Datei dargestellt (vgl. Abbildung 2.2(a)).

Die Erfahrung mit ICE hat gezeigt, dass ein einheitliches Datenmodell für VKS spezifiziert werden kann. Die Definition der Versionsmodelle und die Implementierung des virtuellen Dateisystems zeigen, dass ein mehrschichtiges System möglich und – auf Grund der komplexen Merkmalsterme – auch zwingend notwendig ist. Untersuchungen des ICE-Systems haben jedoch gezeigt, dass die Algorithmen zur Verknüpfung der Merkmalsterme beim Lesezugriff NP-vollständig in Anzahl der Ausdrücke sind (ähnlich wie bei ECM, wo die Auswertung der Terme optimiert wurde, vgl. [WMC01], S. 1127ff.). Dadurch steigt die Laufzeit einiger VKS-Operationen immens. Abbildung 2.2(b) zeigt eine Vergleichsmessung der Zugriffszeit in Sekunden für 17 Commit-Vorgänge in Folge. Auf der x-Achse sind die Anzahl der Commits, auf der y-Achse die Zeit in Sekunden aufgetragen. Während die Zugriffszeit von RCS bzw. SCCS weitgehend konstant bleibt steigt sie bei ICE – insbesondere ohne Optimierung der Termauswertung – deutlich an [ZS97]. Die Verfahren zur Speicherung der versionierten Elemente und die Effizienz des SKMS sind offenbar eng miteinander gekoppelt.

Sowohl ECM als auch ICE verwenden einen sehr allgemeinen, logikbasierten Ansatz, um die Sichtbarkeit von Elementen in einer Konfiguration festzulegen. Über einen Lese- bzw. Schreibfilter werden die Elemente ausgewählt, die das Ziel der Operation sind. Diese Filter sind ebenfalls logische Ausdrücke. ECM verwendet dabei Aussagenlogik, während UVM Featurelogik einsetzt. Letztendlich bieten beide Ansätze den Vorteil der großen Allgemeinheit, bei dem sich alle Aspekte von Versionsmodellen mittels einziges Ansatzes

beschreiben lassen. Bei beiden Ansätzen wird aber auch deutlich, dass es Effizienzprobleme auf Grund der Komplexität der benötigten Algorithmen gibt, sowie eine aufwändige Anpassung der SKMS notwendig ist.

2.1.3. Molhado

Molhado von Tien N. Nguyen, Ethan V. Munson und John T. Boyland ist ein erweiterbares, objektorientiertes und wiederverwendbares Rahmenwerk für Produktversionierung. Molhado bietet ein Datenmodell, das auf attributierten Knoten und Kanten basiert, und das die Möglichkeit zu ihrer Versionierung besitzt. Auf Basis dieses Datenmodells wird ein Produkt-Datenmodell implementiert [NNMB05].

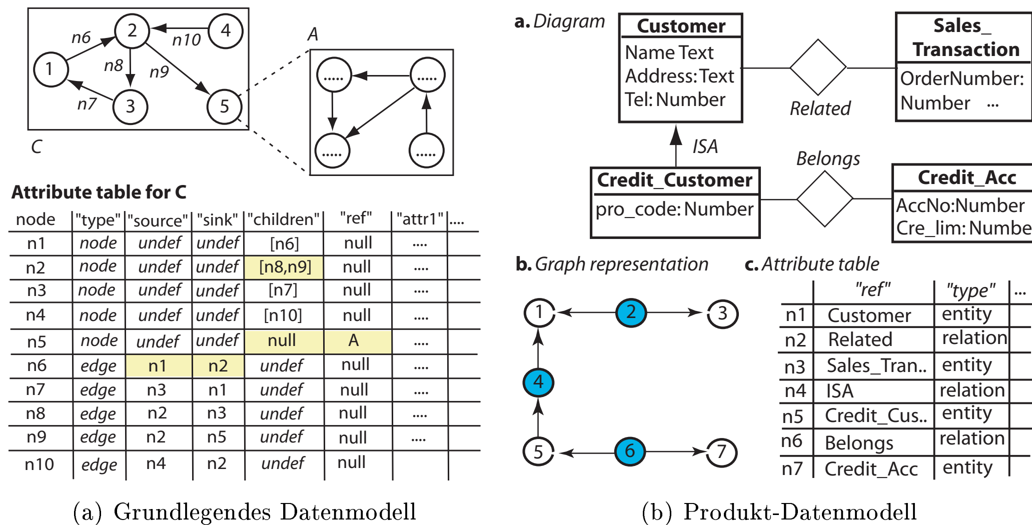
Abbildung 2.3(a) zeigt das grundlegende, graphbasierte Datenmodell von Molhado (vgl. [Ngu06]): In einer Attributetabelle werden Knoten und Kanten über eine eindeutige Kennung identifiziert, z.B. „n1“ oder „n7“. Beiden können Attribute (vgl. „Slots“ in [NNMB05]) zugewiesen werden, wobei jedes Attribut seinen eigenen Typ besitzt. In Abbildung 2.3(a) werden z.B. der Kante „n7“ die Attribute „source“ und „sink“ zugewiesen, da sie in der entsprechenden Zelle einen Attributwert besitzen. „children“ hingegen ist kein Attribut von „n7“ , da es undefiniert (vgl. Zelleneintrag „undef“) ist. Attribute können außer einzelnen Werten auch Sequenzen enthalten (vgl. „children“-Attribut von „n2“) oder Verweise auf andere Attributetabellen (vgl. „ref“-Attribut von „n5“). Somit lassen sich auch Hierarchien von graphbasierten Daten darstellen.

Molhado kann diese Attributstabellen versionieren, indem sie die Tabellen einem Projekt zuordnet. Zu diskreten Zeitpunkten lässt sich der vollständige Projekt-Zustand vom Anwender explizit sichern. Dabei werden jedoch nur veränderte Attribute gespeichert, unveränderte Attributwerte werden wiederverwendet, um Speicherplatz zu sparen. Molhado ist in der Lage, eine baumbasierte Projekthistorie zu verwalten, indem beim Speichern automatisch verzweigt wird. Ein Verschmelzen oder Vermischen von Projekt-Versionen ist nicht möglich [NNMB05].

Auf Basis dieses Datenmodells lassen sich komplexere Produkt-Datenmodelle implementieren. Abbildung 2.3(b) zeigt ein Beispiel für ein Entitäts-/Beziehungsmodell, das mit Hilfe von Molhado implementiert wurde. Dazu wurden zwei Klassen „Entität“ und „Relation“ implementiert, so dass sich deren Instanzen über die Schnittstelle des graphbasierten Datenmodells in Attributstabellen speichern lassen. Z.B. wird der Typ („Entität“ (engl. entity) oder „Relation“ (engl. relation)) als Attributwert eingetragen, und der Name einer Entität als Name der entsprechenden Attributetabelle verwendet. Diese Abbildung wird über den Quellcode der jeweiligen Klasse implementiert, indem z.B. dem „name“-Attribut der Klasse „Entität“ ein spezieller Datentyp aus dem Molhado-Rahmenwerk zugeordnet wird. Das Datenmodell wird, vermutlich über einen Generator, auf generisch implementierte Java-Klassen abgebildet, die mit Hilfe des Java-Reflection-Mechanismus als Attributetabelle serialisiert und wiederhergestellt werden können[2].

Die Erfahrungen mit Molhado zeigen, dass für graphbasierte Datenmodelle spezielle versionierbare Datenmodelle notwendig sind, die über das Versionieren von Dateien

[2]Die exakte Funktionsweise des Rahmenwerks wurde jedoch nie publiziert, und ist mir nur über ein unpubliziertes Papier bekannt



node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	node	undef	undef	[n6]	null	....	
n2	node	undef	undef	[n8,n9]	null	....	
n3	node	undef	undef	[n7]	null	....	
n4	node	undef	undef	[n10]	null	....	
n5	node	undef	undef	null	A	....	
n6	edge	n1	n2	undef	null	....	
n7	edge	n3	n1	undef	null	....	
n8	edge	n2	n3	undef	null	....	
n9	edge	n2	n5	undef	null	....	
n10	edge	n4	n2	undef	null		

	"ref"	"type"	...
n1	Customer	entity	
n2	Related	relation	
n3	Sales_Tran..	entity	
n4	ISA	relation	
n5	Credit_Cus..	entity	
n6	Belongs	relation	
n7	Credit_Acc	entity	

(a) Grundlegendes Datenmodell

(b) Produkt-Datenmodell

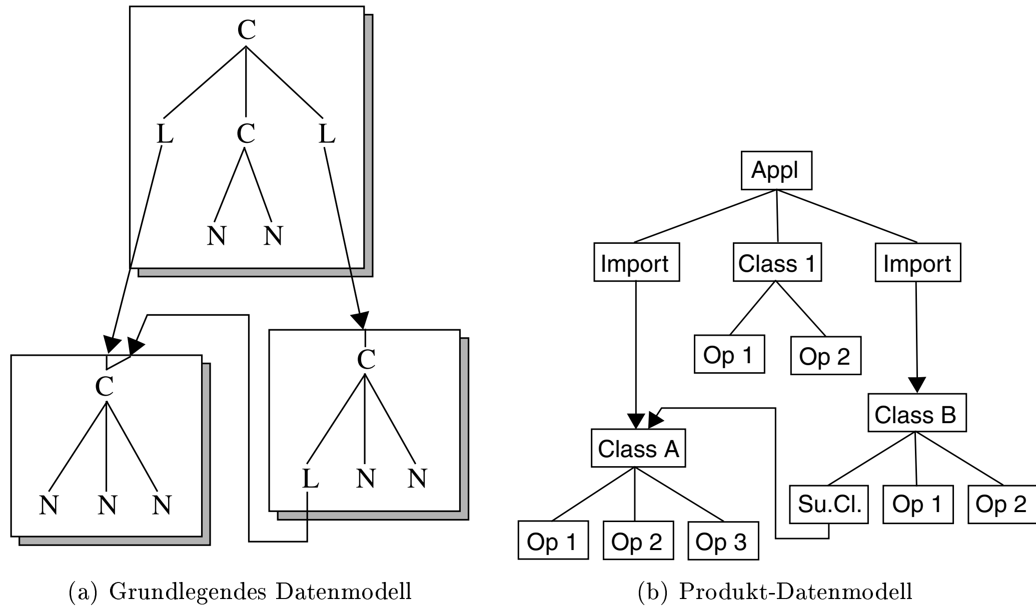
Abbildung 2.3.: Molhado Datenmodelle (nach [Ngu06])

hinausgehen. Es wird auch deutlich, dass eine enge Kopplung zwischen einem unversionierten Datenmodell und der internen Struktur eines versionierbaren Datenmodells bestehen: Molhado kann nur vollständige Projekte wiederherstellen. Eine partielle Wiederherstellung einer früheren Version, d.h. die Kombination verschiedener Versionen, ist deutlich aufwändiger zu realisieren. Zusätzlich ist es notwendig, ein versionierbares Datenmodell vollständig neu zu implementieren. Die Integration bestehender (unversionierter) Datenmodelle ist nicht möglich, d.h. es ist eine Neuimplementierung bzw. Anpassung aller verwendeten Werkzeuge nötig. Es besteht offenbar Bedarf an einer Entkopplung der unversionierten Datenmodelle und den versionierten Daten, um mit möglichst geringem Aufwand bestehende Datenmodelle und ihre Werkzeuge in ein SKMS zu integrieren. Außerdem ist – im Gegensatz zu den bisher vorgestellten Arbeiten – das Versionsmodell fest vorgegeben, d.h. es kann nur das Produktmodell neu definiert werden.

2.1.4. UEVM

Das **U**nified **E**xtensional **V**ersioning **M**odel (UEVM) von Ulf Asklund, Lars Bendix, Henrik B. Christensen und Boris Magnusson ist ein Datenmodell zur Versionierung atomarer und zusammengesetzter Elemente. Ziel ist die extensionale Versionierung strukturierter Daten, d.h. eine valide Kombination von Software-Elementen wird explizit zusammengebaut und gespeichert, anstatt durch Auswertung von Regeln selektiert zu werden. UEVM wurde in drei unterschiedlichen Werkzeugen verwendet und jedes Mal mit unterschiedlicher Funktionalität implementiert. Letztendlich besitzt keine Implementierung die vollständige UEVM-Funktionalität [ABCM99].

Abbildung 2.4(a) zeigt das UEVM Metamodell (vgl. [ABCM99]). Zentrales Element ist das Dokument-Element, das eine interne Datenstruktur in Form eines n-nären Baumes besitzt. In einer Abbildung wird ein Dokument als Kasten dargestellt, in dem seine inter-



(a) Grundlegendes Datenmodell

(b) Produkt-Datenmodell

Abbildung 2.4.: UEVM Datenmodelle (nach [ABCM99])

ne Baumstruktur definiert wird. Abbildung 2.4(a) enthält drei Dokumente. Die interne Baumstruktur besteht aus zusammengesetzten Knoten („C"-Knoten, engl. composite node), atomaren Knoten („N"-Knoten, engl. atomic node) und Verweisknoten („L"-Knoten, engl. link node). „C"-Knoten können Daten enthalten und bestehen aus weiteren „C"-, „N-", oder „L"-Knoten. „N"-Knoten enthalten nur noch Daten, d.h. sie bilden die Blätter der internen Baumstruktur der Dokumente. „L"-Knoten referenzieren andere Dokumente, so dass Hierarchien oder gerichtete Graphen von Dokumenten dargestellt werden können. Die Dokumente in Abbildung 2.4(a) bestehen alle aus mindestens einem „C"-Knoten und bilden bzgl. der „L"-Knoten einen gerichteten azyklischen Graphen: Das oberste Dokumente enthält referenziert die beiden unteren Dokumente, während das rechte auf das linke verweist.

Das Beispiel in Abbildung 2.4(b) zeigt, wie sich z.B. objektorientierter Quellcode auf das UEVM-Metamodell abbilden lässt (vgl. [ABCM99]). Dabei entspricht die interne Struktur des Beispiels dem Metamodell-Beispiel in Abbildung 2.4(a): Das oberste Dokument besteht aus dem Paket „Appl" und der Klasse „Class 1" – beide Elemente werden auf „C-Knoten" abgebildet. Die Operationen „Op1" und „Op2" werden dagegen auf „N"-Knoten abgebildet, während die beiden Import-Beziehungen „L"-Knoten entsprechen. Beide Importe referenzieren ihre zu importierende Klasse, die wiederum als Dokument zu sehen ist. Klasse „Class A" besteht aus drei Operationen, und Klasse „Class B" aus zwei Operationen und einer Subklassen-Beziehung zu „Class B" (die auf einen „L"-Knoten abgebildet wird).

Wird nun eine neue Version eines Knotens erstellt, führt diese dazu, dass alle Elternknoten bis zur Wurzel ebenfalls als neue Version angelegt werden. Wird z.B. die Operation

„Op1“ in der Klasse „Class B“ verändert, so wird auch eine neue Version von „Class B“, „Appl“ und ihrem „Import“ angelegt.

Die Erfahrungen mit UEVM zeigen, dass sich mit Hilfe eines kleinen, strukturierten Datenmodells unterschiedliche Produktmodelle versionieren lassen, z.B. strukturierte Dokumente und auch Quellcode. Für jedes Produktmodell wurde ein einzelnes Werkzeug entwickelt, und für jedes Werkzeug ein Teilaspekt des UEVM implementiert. Es gab keine Wiederverwendung eines einmalig implementierten UEVM-Modells, und auch andere SKM-Verfahren wurden anscheinend jedes Mal neu implementiert. Die jeweiligen Verfahren sind auch lediglich implizit über Quellcode beschrieben, so dass nicht klar ist, inwieweit sich die drei SKMS sonst noch unterscheiden. Die parallel implementierten Werkzeuge zeigen, dass offenbar Bedarf an der Modularisierung und Wiederverwendung von SKMS-Verfahren besteht. Dies würde auch einen systematischen Vergleich zwischen dem UEVM und intensionalen Verfahren erleichtern.

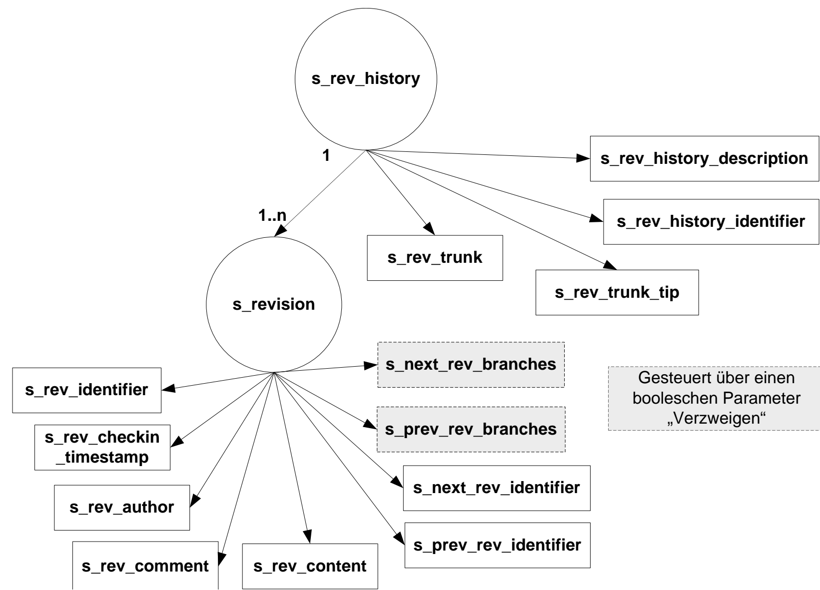
2.2. Quellcode-Kombinations-Rahmenwerke

Bei Quellcode-Kombinations-Rahmenwerken wird die Funktionalität eines SKMS manuell implementiert und anschließend der Quellcode in Fragmente zerlegt. Jedes Quellcode-Fragment wird nun ein oder mehreren SKMS-Implementierungen zugeordnet, in denen es verwendet wird. Durch Auswahl einer bestimmten Implementierung werden die benötigten Quellcode-Fragmente selektiert und zu einem SKMS zusammengesetzt. Wiederverwendung wird in dieser Methode durch Wiederverwendung von Quellcode-Fragmenten in mehreren Implementierungen erreicht. Bei den Quellcode-Fragmenten kann es sich entweder um vollständige Dateien oder auch einzelne Quellcode-Blöcke handeln. Diese werden dann in einer Vorlage an eine vordefinierte Position eingefügt. So kann invarianter Quellcode von Varianten getrennt werden. Doch obwohl Vorlagen verwendet werden, handelt es sich nicht um eine dynamische, vorlagenbasierte Quellcode-Generierung, da für jede Variante die Quellcode-Fragmente explizit vorliegen müssen [Guo08].

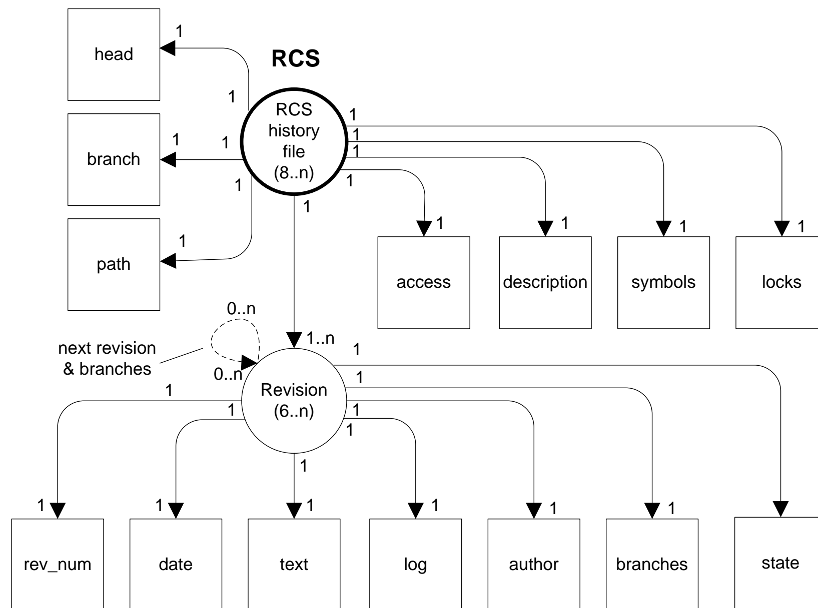
2.2.1. Bamboo

Bamboo von E. James Whitehead und GuoZheng Ge ist ein domänenspezifisches Quellcodekombinations-Rahmenwerk für Inhaltsverwaltungs-Systeme (engl. content management systems) [Guo08]. Mit Enthaltenseins-Modellen (ESM, engl. containment models), einer spezialisierten Fassung von Entitäts-/Beziehungsdiagrammen (engl. Entity/Relationship diagrams), wird das Datenmodell beschrieben. Abbildung 2.5(a) zeigt ein Beispiel für ein ESM (vgl. [WGP04]): Rechtecke stellen atomare Entitäten dar, die letztendlich genau ein Datenelement enthalten. Kreise stellen Container da, eine Entität, die sowohl andere Container als auch atomare Entitäten enthält. Die Pfeile zwischen den Entitäten repräsentieren eine „Enthalten-In“-Beziehung, wobei das Element an der Pfeilspitze im Element am Pfeilende enthalten ist. Bei Beziehungen zwischen Containern können – ähnlich wie in der UML – Kardinalitäten vergeben werden.

Das ESM in Abbildung 2.5(a) modelliert das Datenmodell einer geordneten Historie



(a) Geordnete Historie



(b) RCS Historie

Abbildung 2.5.: Bamboo Enthaltenseins-Diagramme (nach [Ge05])

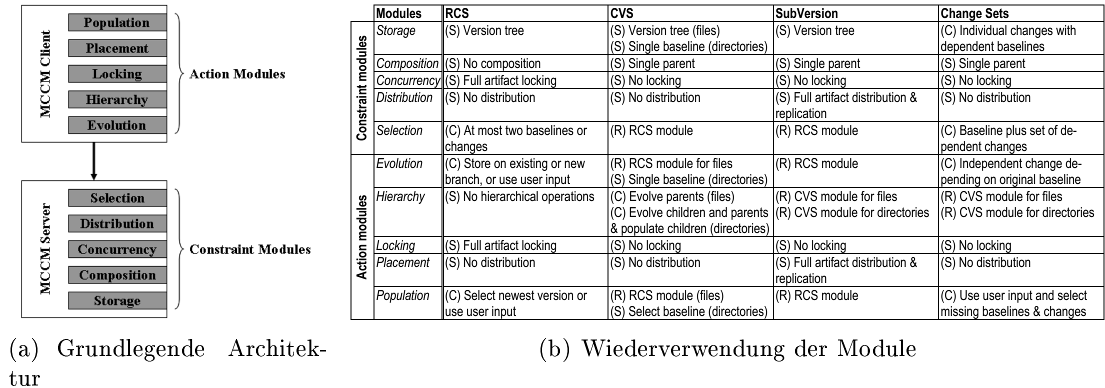
von Datei-Revision, wie sie z.B. in CVS verwendet wird[3]. Der Container „s_rev_history“ enthält eine Menge von „s_revision“-Containern und modelliert so eine Historie, die aus einer Menge von Revisionen besteht. Die atomaren Entitäten modellieren die Daten, die im entsprechenden Container gespeichert werden, z.B. die Kennung der neusten Version auf dem Hauptzweig in „s_rev_trunk_tip“, oder den Inhalt einer Datei-Revision in „s_rev_content“. Zusätzlich zum ESM wird eine Abbildung des ESM-Modells auf die Datentypen des verwendeten Repositoriums benötigt, im Falle von Bamboo auf eine SQL-Datenbank. So wird definiert, dass es sich z.B. bei „s_rev_content“ um Binärdaten und bei „s_rev_trunk_tip“ um einen String handelt.

Um eine konkrete Historie zu beschreiben, werden ebenfalls ESM verwendet. Abbildung 2.5(b) (vgl. [WGP04]) zeigt die Historie eines RCS-Systems: Eine Historie hält eine geordnete Menge an Revisionen. Über eine Abbildungs-Datei wird diese RCS-Historie auf die Elemente der geordneten Historie abgebildet, z.B. „revision“ auf „s_revision“ und „revision text“ auf „s_rev_content“. Dabei wird auch festgelegt dass z.B. kein Verzweigen unterstützt wird und der Verzweigen-Parameter daher den Wert „falsch“ besitzt.

Mit ESM und Datentyp-Abbildungen kann jedoch nur die Struktur des Repositoriums beschrieben und erzeugt werden. Ein Funktionsmodell existiert nicht, bzw. nur implizit als manuell implementierte Quellcode-Fragmente, z.B. durch ein Fragment in C, das eine neue Revision auf dem Hauptzweig anlegt und dabei den Wert in „s_rev_trunk_tip“ aktualisiert. Welches Quellcode-Fragment genutzt wird, hängt vom gewählten ESM und von den gewählten Parametern ab. So existieren z.B. Quellcode-Fragmente speziell für die geordnete Historie aus Abbildung 2.5(a) (vgl. [WGP04]) und zwar einmal mit Zweigen (engl. branches) und einmal ohne. Werden auf diese Art mehrere ESM kombiniert, so müssen, für alle möglichen Kombinationen aus ESM und ihren Parametrisierungen, Quellcode-Fragmente erstellt werden. Letztendlich wird der Quellcode für alle möglichen Produkte bereits im Vorfeld explizit implementiert. Bereits bei 10 booleschen Parametern führt dies zu 1024 verschiedenen Quellcode-Fragmenten für ein ESM. Dabei wird vollständig invarianter Quellcode von den Fragmenten getrennt und als Vorlage mit Platzhaltern für Quellcode-Fragmente abgelegt.

Die Erfahrungen mit Bamboo zeigen, dass sich Datenmodelle von VKS grafisch effektiv modellieren lassen, indem wenige verschiedene Datenmodelle durch Parameter konfiguriert werden. Es wird ebenfalls deutlich, dass die Kombination von Quellcode-Fragmenten nicht mit der Anzahl der Parameter skaliert, und die Autoren bezeichnen ihren Ansatz selbst als zu aufwändig: Bereits für 10 Parameter sei es einfacher das VKS neu zu implementieren, statt die Quellcode-Fragmente zu verwalten [Guo08]. Das Beschreiben der Funktionen durch – explizit vorhandene – Quellcode-Fragmente ist auf Grund der Variabilität offenbar nicht durchführbar. Es sind auf jeden Fall ausgereifte und mächtige Quellcode-Generatoren notwendig [Guo08].

[3] In den Bamboo-Datenmodellen werden die versionierten Daten nicht von den Metadaten getrennt, d.h. wenn ein anderes Datenmodell als Textdateien versioniert werden soll, muss ein vollständig neues Datenmodell erstellt werden.



	Modules	RCS	CVS	SubVersion	Change Sets
Constraint modules	*Storage*	(S) Version tree	(S) Version tree (files) (S) Single baseline (directories)	(S) Version tree	(C) Individual changes with dependent baselines
	Composition	(S) No composition	(S) Single parent	(S) Single parent	(S) Single parent
	Concurrency	(S) Full artifact locking	(S) No locking	(S) No locking	(S) No locking
	Distribution	(S) No distribution	(S) No distribution	(S) Full artifact distribution & replication	(S) No distribution
	Selection	(C) At most two baselines or changes	(R) RCS module	(R) RCS module	(C) Baseline plus set of de-pendent changes
Action modules	*Evolution*	(C) Store on existing or new branch, or use user input	(R) RCS module for files (S) Single baseline (directories)	(R) RCS module	(C) Independent change de-pending on original baseline
	Hierarchy	(S) No hierarchical operations	(C) Evolve parents (files) (C) Evolve children and parents & populate children (directories)	(R) CVS module for files (R) CVS module for directories	(R) CVS module for files (R) CVS module for directories
	Locking	(S) Full artifact locking	(S) No locking	(S) No locking	(S) No locking
	Placement	(S) No distribution	(S) No distribution	(S) Full artifact distribution & replication	(S) No distribution
	Population	(C) Select newest version or use user input	(R) RCS module (files) (S) Select baseline (directories)	(R) RCS module	(C) Use user input and select missing baselines & changes

(a) Grundlegende Architektur

(b) Wiederverwendung der Module

Abbildung 2.6.: MCCM-Rahmenwerk (nach [LH04])

2.2.2. MCCM

Das *MCCM*-Rahmenwerk von Ronald van der Lingen und André van der Hoek ist ein Rahmenwerk zur Komposition von SKMS-Strategien. Ziel von MCCM ist es, die Architektur von SKMS zu modularisieren und den Entwicklungsaufwand neuer Systeme durch Wiederverwendung bereits existierender Module zu reduzieren [LH04]. Dabei ist eine SKMS-Strategie ein Software-Modul, das auf Basis eines generisches Repositoriums ein bestimmtes SKMS-Verfahren implementiert. MCCM bietet dazu eine Modulbibliothek mit Standardmodulen an [Hoe00].

Abbildung 2.6(a) zeigt die grundlegende Architektur des MCCM-Rahmenwerks. Es wird zwischen den Komponenten „MCCM-Client" und „MCCM-Server" unterschieden. Der „MCCM-Server" besteht aus einem Repositorium mit vereinheitlichter Schnittstelle, die von möglichst viele SKMS-Strategien wiederverwendet werden kann. Mit Hilfe von Beschränkungsmodulen (engl. constraint modules) kann das Repositorium anschließend so verändert werden, dass es dem gewünschten Verfahren entspricht. Z.B. ist das generische Repositorium grundsätzlich in der Lage, Verzweigungen darzustellen, kann aber durch das Speicher-Beschränkungsmodul (engl. storage constraint module) so verändert werden, dass es keine Verzweigungen zulässt.

Im „MCCM-Client" existiert zu jedem Beschränkungsmodul ein zugehöriges Aktionsmodul (engl. action module). In Abbildung 2.6(a) sind die Modulpaare jeweils an der gleichen Position in ihrer Komponente aufgeführt, z.B. ist das Evolutions-Aktionsmodul der Partner des Speicher-Beschränkungsmoduls. Sie implementieren Aktionen, die – in den Grenzen des zugehörigen Beschränkungsmoduls – das Repositorium zielgerichtet manipulieren. Z.B. bietet das Evolutions-Aktionsmodul die Funktionalität, neue Versionen als Grundfassung (engl. baseline) oder als Delta zu speichern. Der Entwickler ist jedoch allein dafür verantwortlich, dass die Funktionalität des Modulpaars konsistent ist, z.B. muss er sicherstellen, dass das Aktionsmodul keine Verzweigungen anlegt, obwohl das Beschränkungsmodul sie verbietet.

Abbildung 2.6(b) zeigt die Zusammensetzung von vier Systemen, die mit Hilfe des MCCM-Rahmenwerks prototypisch nachimplementiert wurden. Module mit **(S)** sind

Standardmodule, die Teil der MCCM-Modulbibliothek sind. Module mit **(C)** sind speziell für das System implementiert worden. Einmal implementiert, wurden diese Module der MCCM-Modulbibliothek hinzugefügt, so dass sie anschließend wiederverwendet werden konnten. Solcherart wiederverwendete Module sind mit **(R)** gekennzeichnet. Z.B. wurde das Evolutions-Aktionsmodul für RCS neu implementiert und anschließend für CVS und Subversion wiederverwendet.

Die Erfahrungen mit MCCM zeigen, dass es möglich ist, Konzepte aus der SKM-Domäne zu modularisieren und in einer Modulbibliothek zu organisieren. Es wird ebenfalls deutlich, dass eine Unterstützung für die Konfiguration der Module notwendig ist, da mit steigender Modulzahl auch die Anzahl der Abhängigkeiten zwischen den Modulen steigt. Außerdem kommen die Autoren zu dem Schluss, dass bei modulübergreifenden Abhängigkeiten eine Anpassung des MCCM-Rahmenwerks notwendig wird, um diese aufzulösen [LH04]. Es ist offenbar notwendig, die Konfiguration von SKMS zu unterstützen und die Abhängigkeiten zwischen den einzelnen Modulen explizit zu beschreiben.

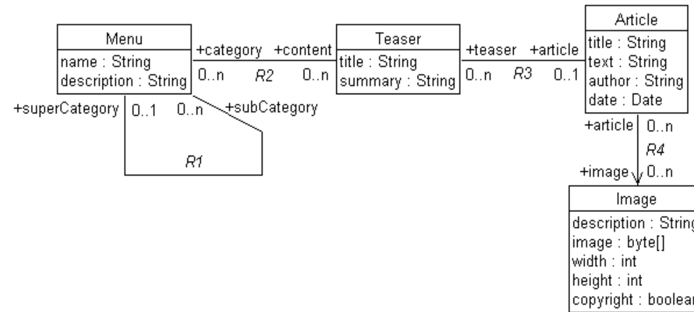
2.3. Quellcode-Generatoren

Quellcode-Generatoren erzeugen Quellcode aus grafischen Modellen, indem sie die Modellelemente auf Quellcode-Vorlagen abbilden. Durch Annotation der Modellelemente kann die Abbildung zusätzlich noch parameterisiert werden, um die Quellcode-Generation zu steuern. Meist lässt sich jedoch nur der Quellcode für die Struktur – und nicht für das Verhalten – generieren, so dass das Verhalten mit Hilfe der Quellcode-Vorlagen spezifiziert wird. Mit Hilfe der Annotationen können dann die benötigten Quellcode-Fragmente in der Vorlage gewählt werden, um so – ähnlich wie bei der bedingten Übersetzung – das gewünschte Verhalten zu erzeugen. Wiederverwendung wird in dieser Methode durch die Wiederverwendung der Quellcode-Vorlagen beim Generieren eines SKMS erreicht. Durch Annotation der Modellelemente mit den Merkmalen einer Produktlinie (s. Abschnitt 1.6) ist diese Methode ein Schritt in Richtung modellgetriebener Produktlinien.

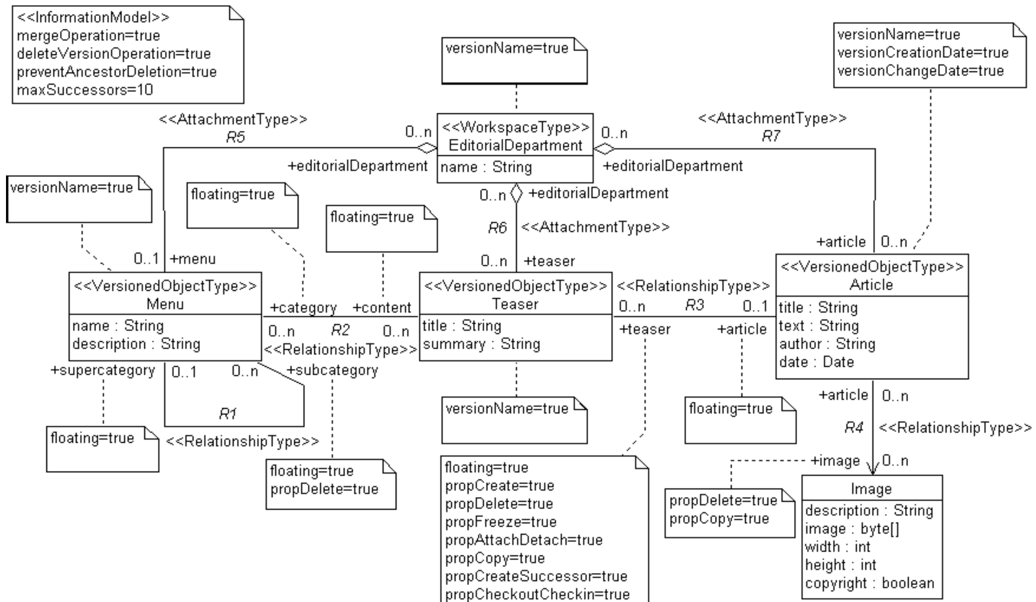
2.3.1. VS-Gen

Der ***V****ersioning* ***S****ystems* ***Gen****erator* (VS-Gen) von Jernej Kovše ist ein templatebasierter Quellcode-Generator für VKS. VS-Gen setzt UML-Stereotypen ein, um UML-Klassendiagramme mit zusätzlichen Informationen für ein VKS zu versehen, z.B. ob eine Klasse überhaupt Teil eines VKS-Datenmodells ist oder welche VKS-Operationen entlang der Assoziationen propagiert werden. Anschließend wird das mit VKS-Konzepten erweiterte Datenmodell als Eingabe für VS-Gen verwendet [Kov05].

Abbildung 2.7(a) zeigt ein Beispiel für ein UML-Klassendiagramm eines einfachen Inhaltsverwaltungssystems: Grundelement des Systems sind Artikel (Klasse "Article") die Bilder (Klasse „Image") enthalten können. Mit Hilfe von hierarchisch angelegten Kategorien (Klasse „Menu") werden Zusammenfassungen (Klasse „Teaser") verwaltet, die auf den eigentlichen Artikel verweisen. Ein Teaser kann mehreren Kategorien zugewiesen werden und zu einem Artikel können auch mehrere Zusammenfassungen existieren.



(a) Ohne Versionsinformationen



(b) Mit Versionsinformationen

Abbildung 2.7.: VS-Gen Datenmodell als UML-Klassendiagramm (nach [KG04])

Um mit VS-Gen für das Artikel-Datenmodell ein VKS zu generieren, muss das UML-Klassendiagramm mit zusätzlichen Informationen für den Generator angereichert werden. Dies geschieht mittels UML-Stereotypen [Kov05]. Abbildung 2.7(b) zeigt das entsprechend erweiterte UML-Diagramm für das Datenmodell aus Abbildung 2.7(a). Die **Klassen** „Menu", „Teaser" und „Article" wurden mit dem Stereotyp „«VersionObjectType»" gekennzeichnet, d.h. für jedes Objekt dieser Klassen existiert später eine eigene Historie im generierten VKS. Die **Assoziationen** zwischen diesen Klassen sind als „«RelationshipType»" markiert, d.h. sie werden ebenfalls versioniert.

Für Assoziationen existieren zwei Arten der Versionierung, die über die **Assoziationsenden** gesteuert werden: „floating" und „non-floating". **„Non-floating"** bedeutet, dass die Verbindung als fester Bestandteil eines versionierten Objektes betrachtet wird. Dies ist in Abbildung 2.7(b) bei „R4" (zwischen „Article" und „Image") der Fall: Jede

Version eines Artikels enthält auch eine Verbindung zu ihren Bildern (die selbst unversioniert sind; „Image“ ist kein VersionedObjectType). **„floating“** bedeutet, dass die Verbindung zu einer Menge von Versionen führt, der **Menge der Kandidatenversionen** (engl. **c**andidate **v**ersion **c**ollection, CVC). Dies gilt z.B. für „R3“ (zwischen „Article“ und „Teaser“). Jeder Artikel besitzt Beziehungen zu allen Teaser-Versionen, die ihm vom Benutzer zugeordnet werden.

Um ein unversioniertes Artikel-Objektmodell zu erhalten, ist es notwendig, die „floating“-Assoziationen zu binden, d.h. aus der Menge der CVC ein Objekt auszuwählen. Dabei bleibt die Verwaltung der CVC, d.h. zu welchen Versionen ein Objekt eine gültige Verbindung besitzt, dem Entwickler überlassen. Lediglich mit Hilfe des Stereotyps „«WorkspaceType»“ kann der Zugriff auf eine bestimmte Version des gesamten Datenmodells erleichtert werden. Diese Klasse besitzt eine Assoziation zu jeder Klasse mit Stereotyp „«VersionObjectType»“, die extra als „«AttachmentType»“ gekennzeichnet wird (Abbildung 2.7(b)). Eine solche Assoziation besagt, dass immer nur genau eine Version eines versionierten Objektes Teil des Arbeitsbereichs sein darf, was u.a. dazu führt, dass alle „floating“-Assoziationsenden gebunden werden. Das Erstellen einer CVC für eine Assoziation ist jedoch immer noch Aufgabe des Entwicklers. Über den „«InformationModel»“-Stereotyp wird schließlich noch das Paket, in dem das versionierte Datenmodell liegt, ausgezeichnet und so einige modellübergreifende Einstellungen für das VKS getroffen.

Die Erfahrung mit VS-Gen zeigen, dass sich Produktmodelle für VKS mit UML modellieren lassen. Außerdem wird gezeigt, dass sich das domänenspezifische Datenmodell der versionierten Elemente von vielen Komponenten des VKS entkoppeln lässt, da z.B. keine Informationen über Speichermechanismen oder die Historie benötigt werden. Es wird auch deutlich, dass ein explizites Modell der internen Konzepte von VKS benötigt wird, denn in VS-Gen sind diese Verfahren lediglich implizit in den Vorlagen (engl. templates) für den Generator vorhanden, und – auf Grund der Variabilität der Vorlagen – noch umständlicher zu erfassen als im Quellcode eines VKS. Es ist z.B. nicht mehr nachvollziehbar, welche Merkmale überhaupt in den Domänenvorlagen realisiert werden und wie die Konfiguration der Merkmale die Vorlagen beeinflusst. [Kov05]. Es besteht offenbar zum einen Bedarf an Werkzeugen für die Verknüpfung von Merkmalen und Domänenmodell und zum anderen an einer expliziten Modellierung der SKMS-Verfahren, um sie erforschbar zu machen.

2.4. Vergleich und Schlussfolgerungen

Die beschriebenen verwandten Arbeiten bieten unterschiedliche Lösungsansätze zu den vier beschriebenen Problemen: Problem 1: „Implizite Beschreibung“ (s. S. 22), Problem 2: „Dateisystem-fixiert“ (s. S. 22), Problem 3: „Monolithische Architektur“ (s. S. 24) und Problem 4: „Mangelnde Kombinationsmöglichkeiten“ (s. S. 24). Tabelle 2.1 zeigt, welche Arbeit Lösungsansätze zu welchem Problem liefert: Keine Arbeit bietet einen Lösungsansatz für die explizite Dokumentation von SKM-Verfahren. ECM/UVM kombiniert als einzige Arbeit ein generisches Versionsmodell mit beliebigen Produkt-Datenmodellen (wobei ICE auch diese Möglichkeit geboten hätte). Lediglich die Beschreibung der Verfahren er-

Verwandte Arbeit	**Lösung für Problem**			
	P1: Implizite Beschreibung	**P2: Dateisystem-fixiert**	**P3: Monolithische Architektur**	**P4: Mangelnde Kombinationsmöglichkeiten**
Bamboo			X	X
ECM/UVM		X	X	X
ICE			X	X
MCCM			X	X
Molhado		X		
UEVM		X		
VS-Gen		X		

Tabelle 2.1.: Lösungsansätze der verwandten Arbeiten

folgt implizit im Quellcode. Bamboo, ICE und MCCM konzentrieren sich auf generische Versionsmodelle (ICE) bzw. die Wiederverwendung bereits implementierter Komponenten (Bamboo, MCCM). Molhado und UEVM konzentrieren sich dagegen auf die Definition domänenspezifischer Produkt-Datenmodelle auf Basis eines fest implementierten VKS. VS-Gen soll die Möglichkeit zur Kombination verschiedener SKMS-Verfahren besitzen, jedoch sind diese implizit in Vorlagen zur Quellcode-Generierung enthalten und weder verfügbar noch dokumentiert [Kov05].

2.4.1. Generische Versionsmodelle

Lösungsansätze dieser Kategorie sind in Tabelle 2.1 farblich nicht hervorgehoben. Beim Vergleich der vier SKMS lassen sich zwei unterschiedliche Bereiche der Wiederverwendung erkennen: Wiederverwendung für unterschiedliche Produktmodelle und Wiederverwendung für unterschiedliche Versionsmodelle. *Molhado* und *UEVM* konzentrieren sich nur auf den ersten Bereich, *ICE* nur auf den zweiten und *ECM/UVM* unterstützt die Wiederverwendung in beiden Bereichen.

Die Lösungsansätze zeigen, dass die Unterstützung unterschiedlicher Produktmodelle ein Variationspunkt einer SKMS-Produktlinie sein sollte. Es wird auch deutlich, dass die Kopplungen zwischen dem Produktmodell und dem Versionsmodell nicht ausreichend untersucht sind, denn die SKM-Verfahren bei Molhado und UEVM sind nicht wiederverwendbar. Die Trennung von Produktmodell vom restlichen SKMS ist nicht ausreichend für eine effektive Wiederverwendung. Mit ICE und ECM/UVM lassen sich zwar unterschiedliche Versionsmodelle realisieren, doch die Komplexität der dabei entstehenden logischen Ausdrücke erhöht die Laufzeit der entstehenden SKMS drastisch. Durch die Abbildung der SKM-Verfahren auf logische Ausdrücke lassen sich die Verfahren nur mit großem Aufwand rekonstruieren und vergleichen.

2.4.2. Quellcode-Kombinations-Rahmenwerke

Lösungsansätze dieser Kategorie sind in Tabelle 2.1 hellgrau unterlegt. Bamboo und MCCM zeigen, dass die bei der Identifikation von SKMS-Komponenten und der Analyse ihrer Abhängigkeiten untereinander noch Forschungsbedarf besteht. Insbesondere die Verwaltung der dabei entstehenden Abhängigkeiten ist eine aufwändige Aufgabe: Bereits bei 10 Variationspunkten entstehen 1024 verschiedene SKMS. Das Fehlen von Vorgehensmodellen und Werkzeugen zur Identifikation, Dokumentation und Konfiguration der Variationspunkte macht die Verwaltung und Erstellung modularer SKMS nahezu unmöglich.

2.4.3. Quellcode-Generatoren

Der Lösungsansatz dieser Kategorie ist in Tabelle 2.1 dunkelgrau unterlegt. VS-Gen verwendet bereits Merkmale zur Beschreibung der Variationspunkte und UML-Klassendiagramme zur Beschreibung der Struktur des Produktmodells. Das Verhalten wird jedoch lediglich über Quellcode-Vorlagen in die Struktur eingefügt. Die Quellcode-Vorlagen werden dabei durch Annotation der Elemente des UML-Klassendiagramms konfiguriert. An VS-Gen wird deutlich, dass eine explizite Beschreibung und Modularisierung von SKM-Verfahren benötigt wird: Das Verhalten ist in VS-Gen in Quellcode-Vorlagen beschrieben, die noch schwieriger zu lesen und zu erweitern sind als implementierter Quellcode. Auch fehlen die expliziten Abhängigkeiten zwischen den Annotationen und den Quellcode-Vorlagen, so dass kaum nachvollziehbar ist, welche Merkmale realisiert werden. Und die Merkmalsanalyse konzentriert sich lediglich auf einige wenige ausgewählte Merkmale.

2.4.4. Schlussfolgerungen

Alle drei Kategorien von Lösungsansätzen zeigen, dass Bedarf an einer umfassenden Analyse der SKM-Domäne besteht, um geeignete SKMS-Komponenten mit minimalen Abhängigkeiten zu identifizieren. Dazu ist insbesondere eine Analyse der Architektur von SKMS zur Untersuchung der Kopplungen zwischen den einzelnen Modulen erforderlich. Die Beschreibung der SKM-Verfahren, die von einzelnen Komponenten implementiert werden, sollte nicht implizit im Quellcode erfolgen – oder sogar in Quellcode-Vorlagen – sondern mit Hilfe von leichter verständlichen, grafischen Modellen. Die Verwendung von Merkmalsmodellen und UML-Klassendiagrammen in VS-Gen geht einen Schritt in diese Richtung. Es wird jedoch deutlich, dass es einer expliziten Dokumentation der Kopplung zwischen Merkmalen und den annotierten Modellen bedarf. Insbesondere die Erweiterbarkeit und Korrektheit kann nur mit geeigneter Werkzeugunterstützung sichergestellt werden.

2.5. Zusammenfassung

Es existieren verschiedene Methoden, um die Wiederverwendung von SKM-Verfahren zu erhöhen. Generische Versionsmodelle setzen dazu ein einheitliches Datenmodell mit

Elementaroperationen ein, mit deren Hilfe die SKM-Verfahren implementiert werden. Quellcode-Kombinations-Rahmenwerke ordnen SKM-Funktionalitäten einzelnen Quellcode-Fragmenten zu, um ein Fragment in allen SKMS wiederzuverwenden, die die gleiche Funktionalität besitzen. Quellcode-Generatoren verwenden Vorlagen wieder, um aus grafischen Modellen Quellcode zu erzeugen. Die Auswahl der Vorlagen wird – ähnlich wie bei bedingter Übersetzung – durch Annotation der Modelle gesteuert. Alle drei Kategorien von Lösungsansätzen zeigen, dass eine umfassende Analyse der Abhängigkeiten zwischen möglichen SKM-Komponenten nötig ist, um die Architektur von SKMS zu modularisieren. Insbesondere besteht auch Bedarf an geeigneter Werkzeugunterstützung, um die Erweiterbarkeit und Korrektheit von modularen SKMS sicherzustellen.

3. Modellgetriebene Produktlinien

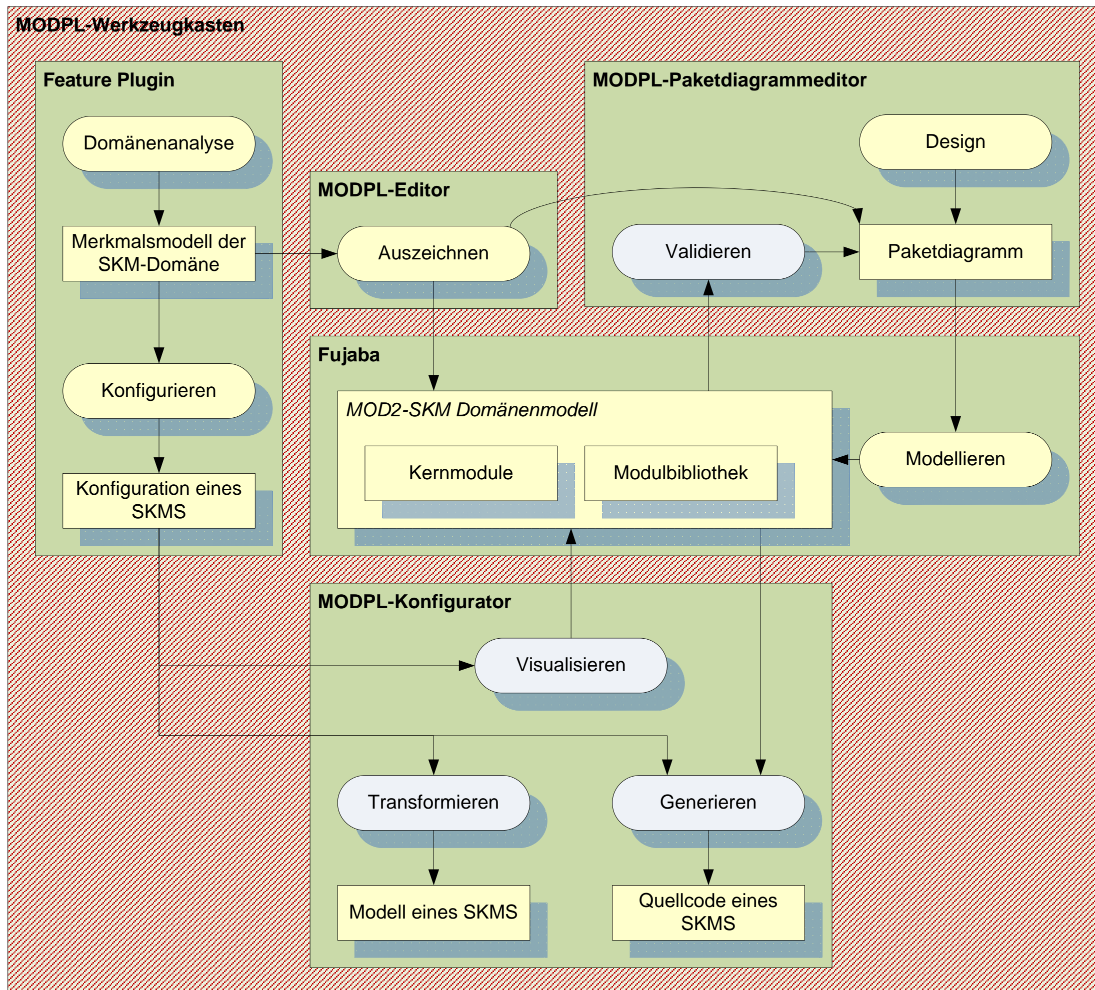


Abbildung 3.1.: Überblick über die MOD2-SKM Produktlinie

MOD2-SKM wird auf Grundlage des MODPL-Rahmenwerks entwickelt. Das folgende Kapitel gibt einen Überblick über seine Aktivitäten und Elemente, und betrifft daher den MODPL-Werkzeugkasten im Allgemeinen (vgl. schräg schraffierter Bereich in Abbildung 3.1).

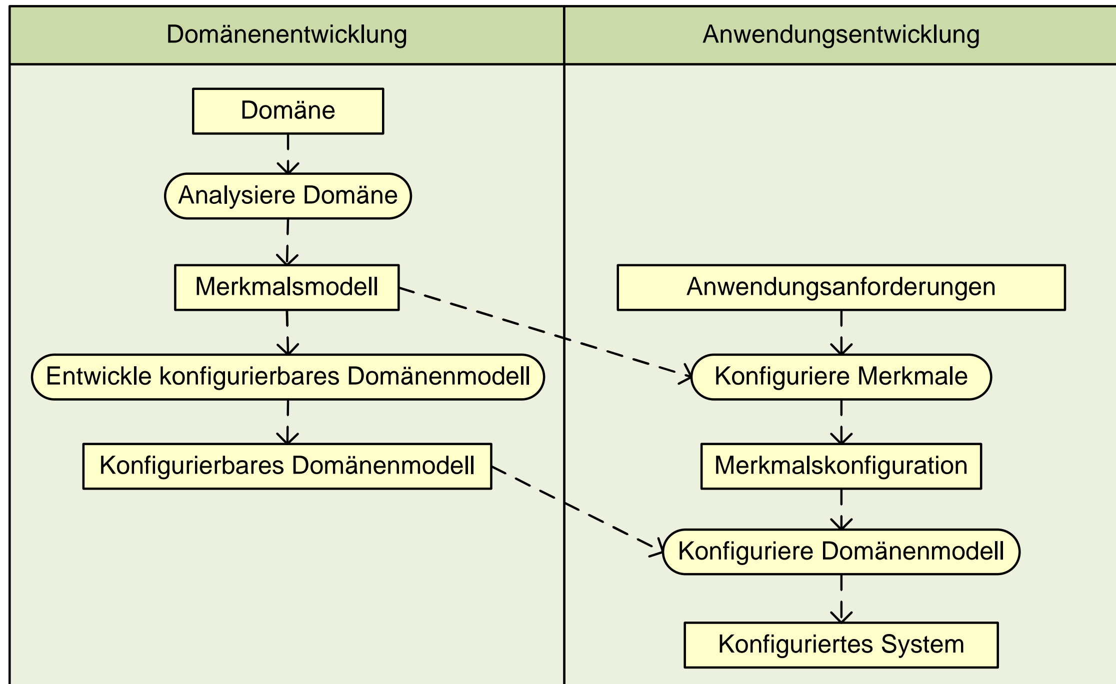


Abbildung 3.2.: Aktivitätsdiagramm der Aufgaben im MODPL-Rahmenwerk

3.1. Das MODPL-Rahmenwerk

Das MODPL-Rahmenwerk beschreibt die Aktivitäten und Software-Elemente, die für die modellgetriebene Entwicklung einer Produktlinie notwendig sind. Dabei wird zwischen der Domänenentwicklung und der Anwendungsentwicklung unterschieden [Buc10]. Auf der einen Seite betrachtet die Domänenentwicklung die Produktlinie als Ganzes. Sie beinhaltet Aktivitäten und Elemente, die für die gesamte Produktlinie relevant sind. Die Merkmale, anhand derer einzelne Systeme klassifiziert werden, oder das Domänenmodell sind Teil der Domänenentwicklung. Auf der anderen Seite betrachtet die Anwendungsentwicklung die Produktlinie als Menge einzelner Produkte, d.h. Software-Anwendungssysteme. Ein konkretes Software-System und seine spezifischen Merkmale sind Teil der Anwendungsentwicklung. Abbildung 3.2 zeigt ein Aktivitätsdiagramm der Aufgaben im MODPL-Rahmenwerk: Aktivitäten und Software-Elemente in der linken Spalte werden der Domänenentwicklung zugeordnet. Aktivitäten und Elemente in der rechten hingegen gehören in die Anwendungsentwicklung.

Am Beginn der Entwicklung einer neuen Produktlinie steht eine Domänenanalyse (vgl. Aufgabe „Analysiere Domäne“), so dass anschließend ein erstes Merkmalsmodell definiert und später verfeinert werden kann. Dieses Modell beschreibt die Domäne in Form von Merkmalen und ihren Abhängigkeiten untereinander. Auf dieser Grundlage lässt sich nun ein konfigurierbares Domänenmodell entwickeln (vgl. Aufgabe „Entwickle konfigurierbares Domänenmodell“). Es beschreibt die Produktlinie als Ganzes, d.h. es modelliert

Aufgabe MODPL-Prozess	**Aufgabe(n) MODPL-Werkzeugkasten**	**Verwendetes Werkzeug**
Analysiere Domäne	Domänenanalyse	Feature plugin
Entwickle konfigurierbares Domänenmodell	Auszeichnen, Design, Modellieren, Validieren	MODPL-Editor, MODPL-Paketdiagrammeditor, Fujaba
Konfiguriere Merkmale	Konfigurieren	Feature plugin
Konfiguriere Domänenmodell	Generieren, Transformieren, Visualisieren	MODPL-Konfigurator

Tabelle 3.1.: Werkzeuge und Einsatzbereiche des MODPL-Werkzeugkastens

alle vorhandenen Variationsmöglichkeiten und ordnet sie den entsprechenden Merkmalen aus dem Merkmalsmodell zu. Um nun ein Anwendungssystem aus der Produktlinie abzuleiten, wird eine Merkmalskonfiguration erstellt (vgl. Aufgabe „Konfiguriere Merkmale“), die konsistent mit den Bedingungen und Abhängigkeiten des Merkmalsmodells ist. Anschließend wird diese Konfiguration verwendet, um automatisiert aus dem konfigurierbaren Domänenmodell das gewünschte Anwendungssystem abzuleiten (vgl. Aufgabe „Konfiguriere Domänenmodell“) [Buc10].

Im MODPL-Rahmenwerk wird der MODPL-Werkzeugkasten eingesetzt, um sowohl Merkmalsmodelle und Konfigurationen zu bearbeiten, als auch das Domänenmodell zu editieren und konfigurierte Anwendungssysteme daraus zu generieren. Die entsprechenden Aufgaben aus Abbildung 3.2 lassen sich den Aufgaben und Werkzeugen des MODPL-Werkzeugkastens in Abbildung 3.1 zuordnen, wie in der Tabelle 3.1 gezeigt wird [Buc10].

Beispiel-Domäne: Mobiltelefon-Spiel

Spiele-Software, die für Mobiltelefone entwickelt wird, soll für eine möglichst große Zahl an Plattformen zur Verfügung stehen. Jedes Mobiltelefon stellt jedoch andere Anforderungen an ein Spiel, z.B. in Bezug auf Auflösung und Farbtiefe oder die Eingabemöglichkeiten. Die Spiele-Software muss auf die jeweiligen technischen Anforderungen des Mobiltelefons zugeschnitten sein, z.B. müssen die Grafiken in einer entsprechenden Auflösung vorliegen und die Eingaben des Spielers registriert und verarbeitet werden. Während also die Spielinhalte auf jedem Gerät gleich bleiben, verändert sich die Darstellung und Steuerung in Abhängigkeit von der Plattform. Eine MODPL für ein Mobiltelefon-Spiel hat zum Ziel, ein bestimmtes Spiel für möglichst viele Mobiltelefone verfügbar zu machen. Die Domäne ist in diesem Fall das Mobiltelefon-Spiel, während ein Produkt ein Spiel für ein bestimmtes Mobiltelefon ist.

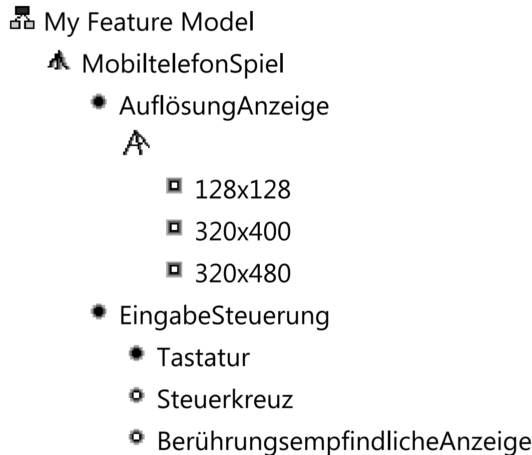


Abbildung 3.3.: Merkmalsmodell für ein Mobiltelefon-Spiel

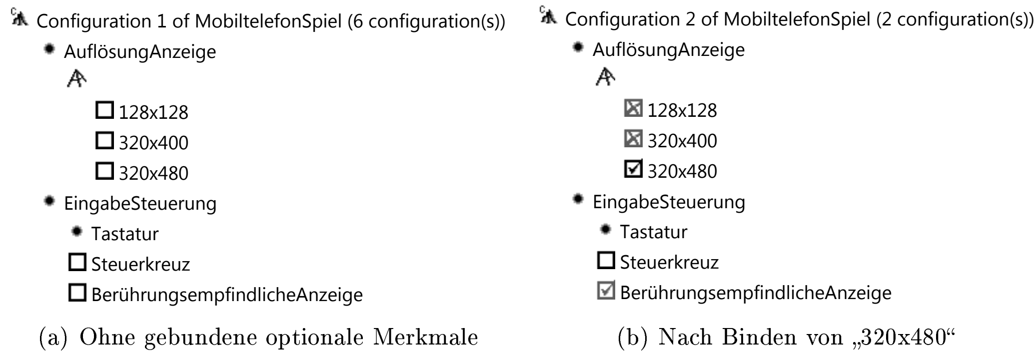
3.2. Das Merkmalsmodell

Ergebnis von: Analysiere Domäne
Verwendetes Werkzeug: Feature plugin

Das Merkmalsmodell ist ein Element der Domänenentwicklung [KKL+98]. Es definiert die Merkmale, die zur Klassifikation und Konfiguration der einzelnen Produkte verwendet werden können. Die Merkmale beschreiben, welche Eigenschaften alle Produkte gemeinsam haben und in welchen sie sich unterscheiden. Daher wird der Prozess zur Definition des Merkmalsmodells auch als *Gemeinsamkeiten-Unterschiede-Analyse* (engl. Commonality-Variablity-Analysis) bezeichnet [KCH+90] [KKL+98]. Abbildung 3.3 zeigt ein Merkmals-Modell für ein Mobiltelefon-Spiel (siehe auch Beispiel-Domäne: Mobiltelefon-Spiel). Jeder Knoten des Baumes repräsentiert ein Merkmal. Jedes von ihnen besitzt einen eindeutigen Namen, der als Knotenbezeichner verwendet wird, z.B. „AuflösungAnzeige" oder „Steuerkreuz" [AC04].

Außer den Merkmalen werden auch Abhängigkeiten zwischen ihnen definiert. Die häufigste Abhängigkeit ist die **Teil-Ganze-Beziehung** zwischen zwei Merkmalen [KCH+90]. Durch sie wird ausgedrückt, dass das untergeordnete Merkmal nicht ohne das übergeordnete Merkmal vorkommen kann. Um diese Beziehung zu modellieren, wird der Knoten des untergeordneten Merkmals (des „Teils") als Kindknoten des übergeordneten Merkmals (des „Ganzen") definiert. In Abbildung 3.3 ist z.B. „Steuerkreuz" ein Teil des Merkmals „EingabeSteuerung". Ein Merkmal kann aus mehreren Teilmerkmalen bestehen [KCH+90].

Die Teil-Ganze-Beziehung kann noch weiter verfeinert werden, indem **Einschränkungen bzgl. der Anzahl der Teilmerkmale** angegeben werden. So sind z.B. in Abbildung 3.3, die Untermerkmale von „AuflösungsAnzeige" nicht direkt als Kindknoten modelliert, sondern als sich gegenseitig ausschließende Merkmalsmenge (dargestellt durch das Ausschluss-Symbol). So wird ausgedrückt, dass die Teilmerkmale immer nur ein-



(a) Ohne gebundene optionale Merkmale (b) Nach Binden von „320x480“

Abbildung 3.4.: Partielle Konfigurationen

zeln und niemals gemeinsam vorkommen können (eine Anzeige besitzt immer genau eine Auflösung) [AC04].

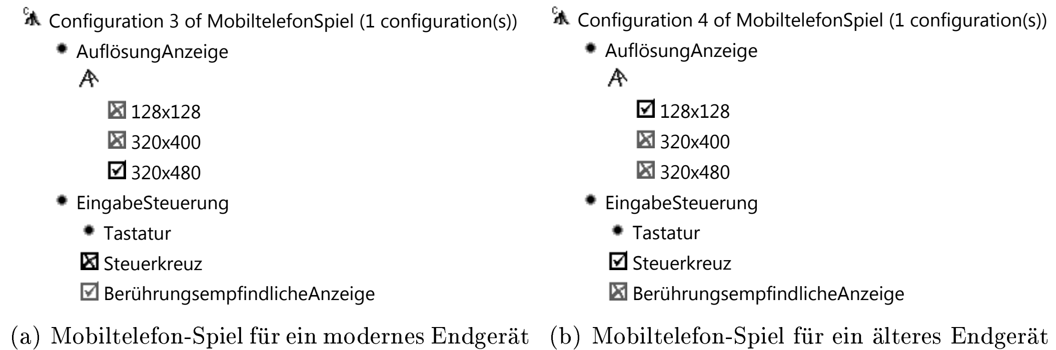
Zusätzlich können noch allgemeinere Abhängigkeiten in Form von **Wenn-Dann-Klauseln** definiert werden. Z.B. wird durch die Klauseln „Wenn *320x480* dann *BerührungsempfindlicheAnzeige*“ und „Wenn *BerührungsempfindlicheAnzeige* dann *320x480*“ modelliert, dass mit einer berührungsempfindlichen Anzeige auch eine bestimmte Auflösung einhergeht [AC04].

Um nun Gemeinsamkeiten und Unterschiede auszudrücken, werden die Merkmale als **verpflichtend** oder **optional** gekennzeichnet [KCH+90]. Verpflichtende Merkmale sind bei allen Produkten der Produktlinie vorhanden, d.h. es gibt kein Produkt ohne dieses Merkmal. Verpflichtende Merkmale werden durch einen schwarz ausgefüllten Kreis gekennzeichnet. Optionale Merkmale drücken die möglichen Unterschiede zwischen den einzelnen Produkten der Produktlinie aus, d.h. es gibt Produkte, die dieses Merkmal besitzen und welche, die es nicht besitzen. Optionale Merkmale werden durch einen nicht ausgefüllten Kreis gekennzeichnet. In Abbildung 3.3 sind die Merkmale „AuflösungAnzeige“ und „Tastatur“ verpflichtend, d.h. jede Implementierung des Mobiltelefon-Spiels wird auf einem Display angezeigt und kann durch Tasten gesteuert werden. Die Merkmale „Steuerkreuz“ und „128x128“ sind optional, d.h. es gibt Implementierungen, die ein Steuerkreuz und ein Display der Auflösung „128x128“ unterstützen, aber auch Implementierungen die nur eines der beiden Merkmale oder gar keines aufweisen [AC04].

3.3. Die Konfiguration

Ergebnis von: Konfiguriere Merkmale
Verwendetes Werkzeug: Feature plugin

Das Merkmalsmodell kann nun zum Erstellen von Konfigurationen verwendet werden. Sowohl die Aufgabe „Konfiguration erstellen“ als auch das Artefakt „Konfiguration“ sind bereits Teil der Produktentwicklung (vgl. MODPL-Prozess in Abbildung 3.2 auf S. 52), d.h. sie dienen dazu, aus der Produktlinie ein konkretes Produkt abzuleiten



(a) Mobiltelefon-Spiel für ein modernes Endgerät (b) Mobiltelefon-Spiel für ein älteres Endgerät

Abbildung 3.5.: Vollständige Konfigurationen

[Buc10]. Um eine Konfiguration zu erstellen, wird zunächst eine partielle Konfiguration aus dem Merkmalsmodell abgeleitet. In der partiellen Konfiguration sind alle Merkmale aus dem Merkmalsmodell vorhanden: Die verpflichtenden Merkmale sind bereits gebunden, da sie in jedem Produkt vorkommen. Die optionalen Merkmale sind ungebunden, d.h. der Entwickler entscheidet nun, welche (optionalen) Merkmale in der Konfiguration vorhanden sind und welche nicht. Abbildung 3.4(a) zeigt die partielle Konfiguration des Mobiltelefon-Spiel-Merkmalsmodells. Die verpflichtenden Merkmale sind automatisch gebunden, während die optionalen Merkmale noch ungebunden sind (dargestellt als leeres Kästchen). Die Anzahl der Konfigurationen zeigt an, wie viele verschiedene Konfigurationen (unter Einbeziehung aller Beschränkungen) aus dieser partiellen Konfiguration noch erstellt werden können [AC04].

Der Entwickler erstellt nun schrittweise die endgültige Konfiguration, indem er die Merkmale manuell bindet, d.h. er entscheidet, ob ein Merkmal Teil der Konfiguration ist oder nicht. Dies geschieht durch Markieren der ungebundenen Merkmale: Ein Haken markiert ein vorhandenes und ein Kreuz ein nicht vorhandenes Merkmal. Anschließend werden die Abhängigkeiten unter den Merkmalen ausgewertet und weitere Merkmale entsprechend gebunden. Diese propagierten Bindungen werden in grau angezeigt. Abbildung 3.4(b) zeigt die partielle Konfiguration nach Auswahl des Merkmals „320x480“. Durch die Beschränkung der Teil-Ganze-Beziehung zu „AuflösungAnzeige“ werden die alternativen Auflösungen abgewählt. Gleichzeitig wird durch die beispielhaft erwähnte Wenn-Dann-Klausel das Merkmal „BerührungsempfindlicheAnzeige“ ausgewählt. Die Anzahl der möglichen Konfigurationen ist aktualisiert worden: Es gibt nur noch zwei verschiedene Konfigurationen, eine mit und eine ohne „Steuerkreuz“ [AC04].

Mit Binden des letzten Merkmals ist die Konfiguration vollständig, da keine Alternativen mehr existieren [AC04]. Abbildung 3.5(a) zeigt eine Konfiguration des Mobiltelefon-Spiels für ein modernes Mobiltelefon. Durch unterschiedliche Bindungen von Merkmalen können so unterschiedliche Konfigurationen erstellt werden. Abbildung 3.5(b) zeigt eine weitere Konfiguration des Mobiltelefon-Spiels, diesmal für ein älteres Mobiltelefon mit niedrigerer Auflösung und einem Steuerkreuz.

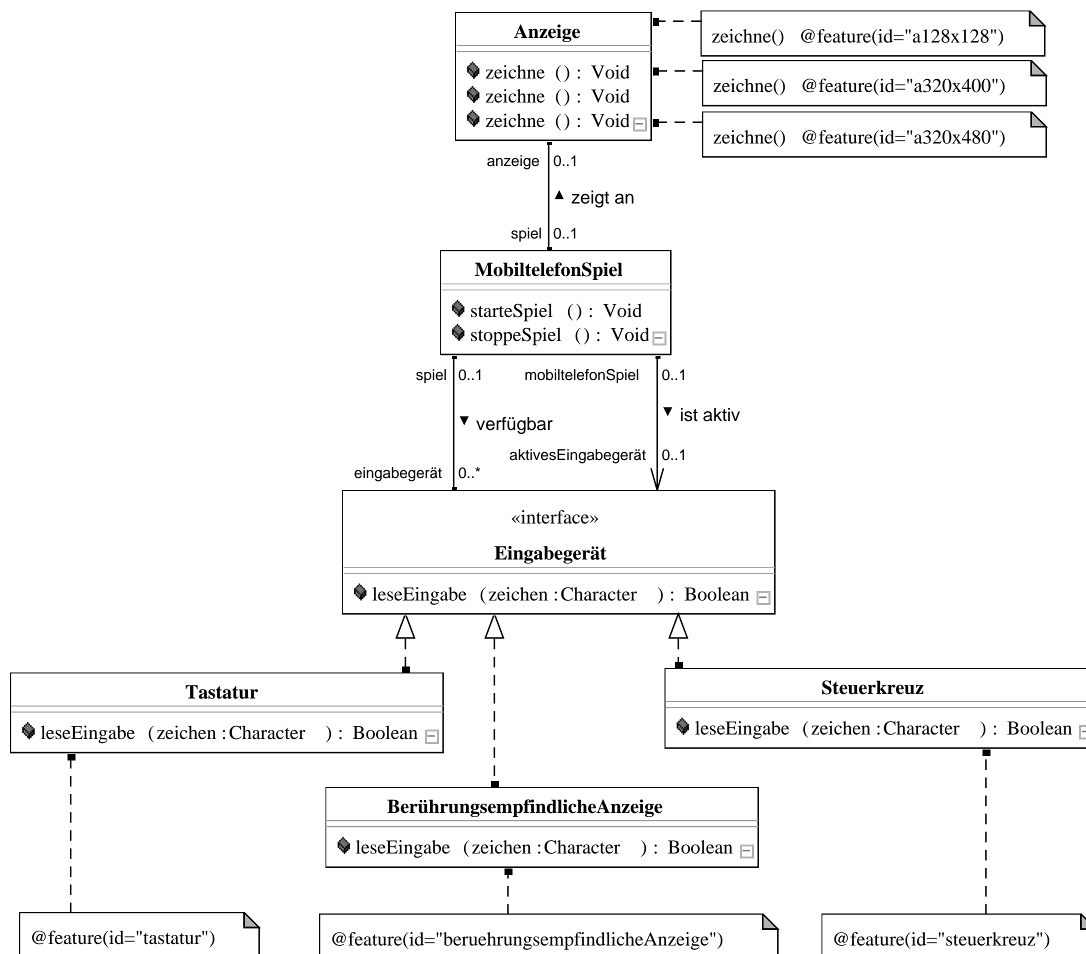


Abbildung 3.6.: Konfigurierbares Domänenmodel für das Mobiltelefon-Spiel

3.4. Das konfigurierbare Domänenmodell

Ergebnis von: Entwickle konfigurierbares Domänenmodell
Verwendetes Werkzeug: Fujaba mit MODPL-Editor

Das Merkmalsmodell beschreibt eine Produktlinie in Form eines Merkmals-Baums, und abstrahiert somit stark von der Software-Architektur und den operationalen Modellen einer Produktlinie. Dies ist die Aufgabe des Domänenmodells, das – wie das Merkmalsmodell – zum Bereich Domänenentwicklung gehört. Das Domänenmodell wird in Fujaba modelliert, wobei die Struktur der Produktlinie durch Klassendiagramme und die Operationen durch Storydiagramme beschrieben werden [Buc10]. Abbildung 3.6 zeigt das Klassendiagramm der Mobiltelefon-Spiel-Produktlinie[1]. Die Klasse „MobiltelefonSpiel“

[1]Das Beispielmodell ist ein stark vereinfachtes Modell, anhand dessen sich die Konzepte von MODPL erläutern lassen. Eine vollständig modellierte Produktlinie würde deutlich komplexer ausfallen.

ist das zentrale Element und enthält die Methoden, mit denen sich das Spiel starten und wieder anhalten lässt. Jedes Spiel besitzt eine „Anzeige“ (auf der die Spielgrafik „gezeichnet“ wird) und ein Menge von „Eingabegeräten“ (mit denen das Spiel gesteuert wird). Nur ein „Eingabegerät“ ist das aktive Gerät, dessen Eingaben gelesen werden. Jedes von ihnen besitzt die gleiche Schnittstelle (engl. interface) „leseEingabe“, implementiert diese aber unterschiedlich.

3.4.1. Verhaltensmodellierung

Das Verhalten der im Klassendiagramm definierten Methoden wird mit Hilfe von Story-Diagrammen modelliert [Buc10]. Diese Diagrammsorte verwendet Sprachelemente aus UML-Aktivitäts- und UML-Kommunikationsdiagrammen und lässt sich vollständig auf ausführbaren Quellcode abbilden. Aus diesem Grund werden Fujaba-Modelle auch als „ausführbar“ bezeichnet [Zün02]. Abbildung 3.7 zeigt ein Beispiel-Story-Diagramm für die Methode „starteSpiel“ der Klasse „MobiltelefonSpiel“. Wie ein UML-Aktivitätsdiagramm basiert ein Story-Diagramm auf einem Kontrollflussgraphen [Zün02]. Einstiegspunkt der Methode bildet der Startknoten (schwarz ausgefüllter Kreis). Ein Pfeil stellt den Kontrollflussverlauf zum nächsten Knoten dar, einer „Story-Aktivität“ (gelbes, abgerundetes Rechteck). Innerhalb der Aktivität wird – ähnlich wie bei einem Kommunikationsdiagramm – die Manipulation des Laufzeit-Objektgraphen angezeigt, d.h. es werden Objekte im Graph gesucht (schwarz), angelegt (grün) und gelöscht (rot, nicht abgebildet) sowie ihre Attribute und Beziehungen (engl. links) verändert [Zün02]. Die erste Story-Aktivität in Abbildung 3.7 erzeugt jeweils ein Objekt der drei „Eingabegerät“-Subklassen („BerührungsempfindlicheAnzeige“, „Steuerkreuz“ und „Tastatur“), sowie ein Objekt vom Typ „Anzeige“ (Objekte sind grün gefärbt). Anschließend wird das entsprechende „MobiltelefonSpiel“-Objekt sowohl mit den drei Objekten der „Eingabegerät“-Subklassen (Beziehung „verfügbar“) als auch mit dem „Anzeige“-Objekt (Beziehung „zeigt an“) in Beziehung gesetzt (grüne Beziehungen werden angelegt). Zum Schluss wird noch die Methode „zeichne“ am neu angelegten „Anzeige“-Objekt aufgerufen. Der Kontrollfluss verzweigt zur nächsten Story-Aktivität, wo zunächst ein beliebiges verfügbares Eingabegerät gesucht wird (schwarz gefärbt), das anschließend als „aktiv“ in Beziehung gesetzt wird (Assoziation ist grün gefärbt). Abhängig vom Erfolg der Aktivität verzweigt der Kontrollfluss. Kann die Aktivität erfolgreich ausgeführt werden („success“-Kante), erreicht die Methode einen Endzustands-Knoten. Wurde dagegen z.B. kein Eingabegerät gefunden, so schlägt die Aktivität fehl („failure“-Kante) und es wird in einer Statement-Aktivität eine Java-Ausnahme geworfen (weißes, abgerundetes Rechteck). Statement-Aktivitäten werden direkt in den Quellcode abgebildet, d.h. sie sind – im Gegensatz zu den Story-Aktivitäten – von der Zielsprache abhängig [Zün02].

3.4.2. Merkmalsmarkierungen

Während das Spiel in allen Produkten die gleichen Spielinhalte besitzt, unterscheiden sich die Eingabegeräte und die Auflösung der Anzeige (und damit die Implementierung von „zeichnen“). Eine Implementierung von „Eingabegerät“ ist nur nötig, wenn das ent-

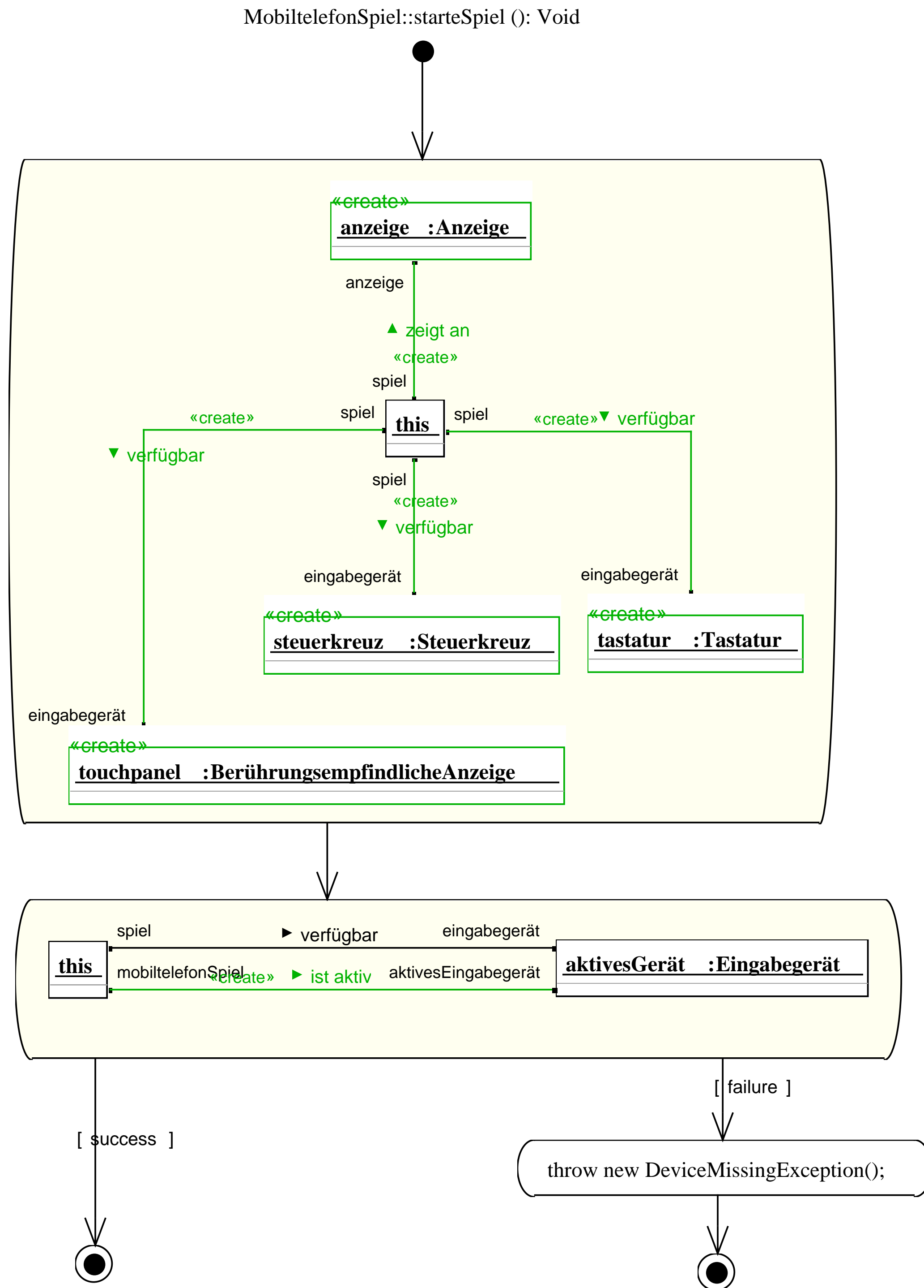


Abbildung 3.7.: Story-Diagramm der Methode „starteSpiel“

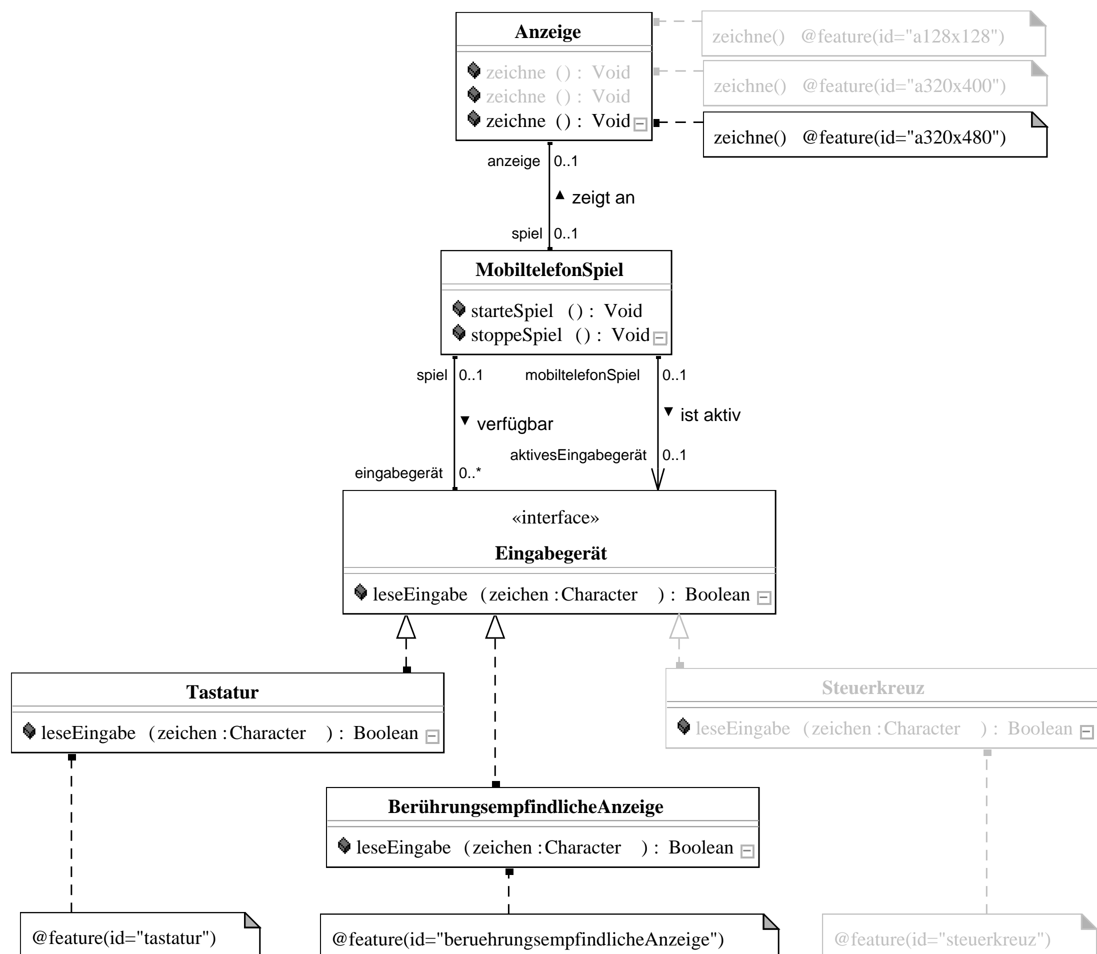


Abbildung 3.8.: Vorschau auf konfiguriertes Modell für ein modernes Endgerät

sprechende Gerät auch im Mobiltelefon vorhanden ist. Und die „Anzeige“ soll nur die Methode zum „zeichnen“ verwenden, die für die Auflösung des Endgerätes vorgesehen ist. Die entsprechenden Elemente im Domänenmodell sind damit direkt von Merkmalen aus dem Merkmalsmodell abhängig: Immer dann, wenn ein Merkmal Teil einer Konfiguration ist, sollen auch die entsprechenden Modellelemente im konfigurierten System auftauchen [BD09b].

Diese Abhängigkeit wird durch Merkmalsmarkierungen ausgedrückt. Alle Modellelemente, die einem Merkmal zugeordnet werden, erhalten eine solche Merkmalsmarkierung, die das entsprechende Merkmal über seine eindeutige Kennung referenzieren [BD09b]. In Abbildung 3.6 annotieren die Merkmalsmarkierungen die Elemente in der Form „@feature(id="*Kennung*")“. Durch Markieren der Modellelemente mit Merkmalen wird das Domänenmodell konfigurierbar: Mittels einer Konfiguration kann nun aus ihm ein Modell eines konfigurierten Anwendungssystems generiert werden, das aus allen Elementen des Domänenmodells besteht, deren Merkmalsmarkierungen Teil der Konfiguration sind.

Elemente, die mit nicht ausgewählten Merkmalen markiert sind, werden letztendlich entfernt [BD09a]. Abbildung 3.8 zeigt das markierte Domänenmodell unter Berücksichtigung der Konfiguration für ein modernes Endgerät mit berührungsempfindlicher Anzeige (vgl. Konfiguration in Abbildung 3.5(a), S. 56). Modellelemente, die auf Grund ihrer Markierung nicht Teil des konfigurierten Systems sind, werden in grau dargestellt, wie z.B. die Klasse „Steuerkreuz" oder die beiden „zeichnen"-Methoden für die niedrigen Auflösungen.

Der Entwickler wird dabei durch den MODPL-Editor unterstützt: Zunächst wird das Merkmalsmodell geladen, um mit dessen Merkmalen das Modell möglichst benutzerfreundlich markieren zu können. Wird ein Element, z.B. eine Klasse, ausgezeichnet, so wird die Markierung vor der Konfiguration des Modells an alle abhängigen Elemente propagiert, z.B. an ein- und ausgehende Assoziationen und Rollenenden. So wird sichergestellt, dass das konfigurierte Modell syntaktisch korrekt ist [BD09b]. Der MODPL-Editor bietet dazu die Möglichkeit, die Propagation der Markierungen vorab auszuführen und wieder rückgängig zu machen. Abbildung 3.9 zeigt das Story-Diagramm der Methode „starteSpiel" für ein modernes Endgerät. Elemente, die nicht Teil des Produktes sind, werden ebenfalls grau dargestellt, wie z.B. das „Steuerkreuz"-Objekt und die Beziehung zu ihm. Diese wurden durch automatische Propagation der Merkmalsmarkierung an der Klasse „Steuerkreuz" (vgl. Abbildung 3.6) entfernt [BD09a, Buc10].

3.5. Das konfigurierte System

Ergebnis von: Konfiguriere Domänenmodell
Verwendetes Werkzeug: MODPL-Konfigurator

Aus dem konfigurierbaren Domänenmodell kann nun – unter Verwendung einer Konfiguration – ein lauffähiges Produkt generiert werden. Dazu bietet der MODPL-Konfigurator drei Möglichkeiten [Buc10]:

1. Vorschau anzeigen
2. Konfiguriertes Modell generieren
3. Konfigurierten Quellcode generieren

Die Vorschau unterstützt den Entwickler bereits während der Bearbeitung des konfigurierbaren Domänenmodells, indem sie die Elemente einer Konfiguration hervorhebt: Lädt der Entwickler eine Konfiguration, so kann er alle Elemente, die nicht Teil der Konfiguration sind, ausblenden und wieder einblenden. So kann er direkt prüfen, ob das zugehörige Anwendungssystem der Konfiguration entspricht. Die Abbildungen 3.8 und 3.9 stellen solche Vorschauansichten dar. Zusätzlich werden auch die Abhängigkeiten zwischen einzelnen Merkmalen ausgewertet, um inkonsistente Auszeichnungen zu vermeiden. So wird z.B. geprüft, ob sich mehrere Merkmalsmarkierungen an einem Modellelement – auf Grund ihrer Abhängigkeiten im Merkmalsmodell – gegenseitig ausschließen [Buc10].

Aus einem konfigurierbaren Domänenmodell kann – mit Hilfe einer Konfiguration – ein Modell eines Anwendungssystems generiert werden. Dazu erhält der MODPL-Konfigurator das konfigurierbare Domänenmodell und eine Konfiguration als Eingabe und prüft

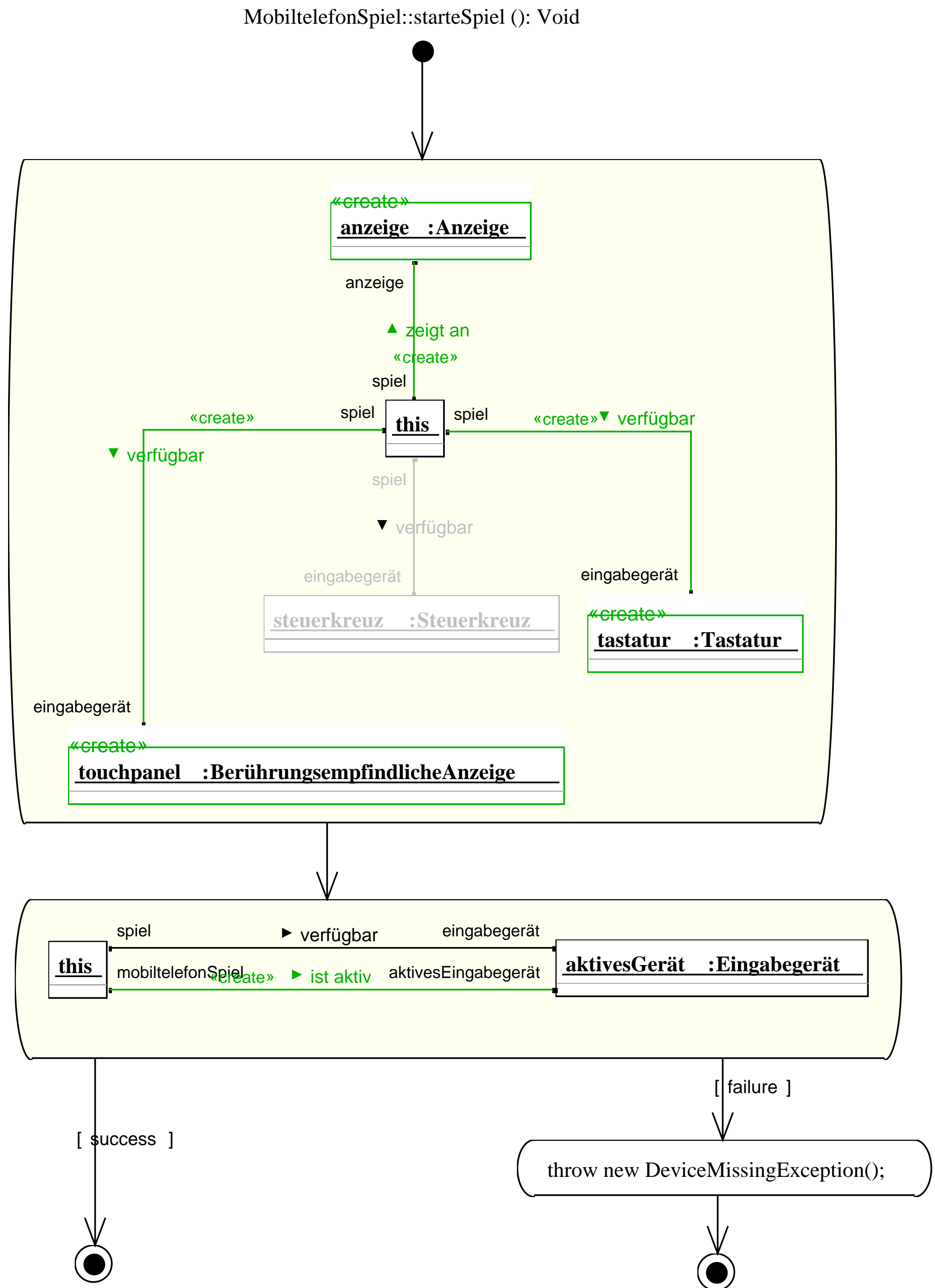
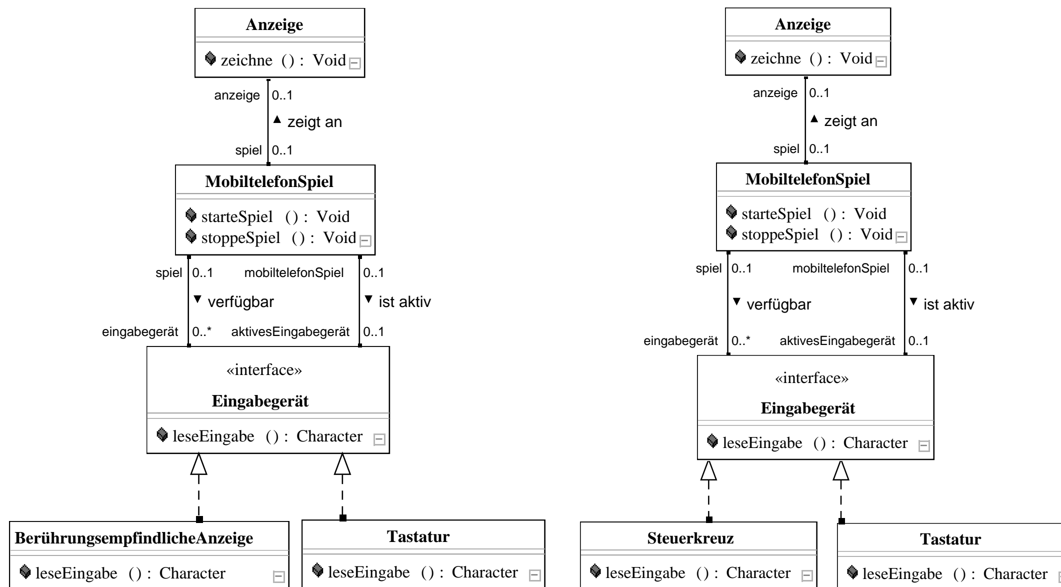


Abbildung 3.9.: Vorschau auf „starteSpiel“-Story-Diagramm für ein modernes Endgerät



(a) Konfiguriertes Mobiltelefon-Spiel für ein modernes Endgerät

(b) Konfiguriertes Mobiltelefon-Spiel für ein älteres Endgerät

Abbildung 3.10.: Vollständige Konfigurationen

nun die Merkmalsmarkierungen an den Elementen des Domänenmodells: Existieren alle Merkmalsmarkierungen an einem Domänenelement auch in der Konfiguration, dann ist das Domänenelement auch Teil des Anwendungssystems. Ist mindestens ein Merkmal aus den Merkmalsmarkierung in der Konfiguration abgewählt, so wird das Element nicht übernommen. Ein Sonderfall sind Domänenelemente ohne Markierungen (d.h. leere Konjunktion von Merkmalen): sie sind in jedem Fall Teil des konfigurierten Modells. Wie bei der Vorschau stellt der Konfigurator die syntaktische Korrektheit des Produktes sicher, indem er bestimmte Konsistenzbedingungen einhält. Z.B. ist eine Assoziation nur Teil des konfigurierten Modells, wenn auch Quelle und Ziel enthalten sind [BD09a]. Das konfigurierte Modell ist ein vollständiges Fujaba-Modell, d.h. es kann sowohl weiter bearbeitet als auch auf ausführbaren Quellcode abgebildet werden. Soll das konfigurierte Modell nicht mehr weiter bearbeitet werden, so kann mit Hilfe des MODPL-Konfigurators auch direkt der zugehörige Quellcode generiert werden [Buc10].

Die Abbildung 3.10(a) zeigt das Mobiltelefon-Spiel als Produkt für das moderne Mobiltelefon mit berührungsempfindlicher Anzeige (vgl. Konfiguration in Abbildung 3.5(a) auf S. 56). Die Anzeige besitzt nur noch die „zeichne“-Methode für die entsprechende Auflösung und die Implementierung für das „Steuerkreuz“ wurde entfernt.

Unter Verwendung der Konfiguration für das ältere Mobiltelefon (vgl. Konfiguration in Abbildung 3.5(b) auf S. 56) entsteht ein Produkt wie in Abbildung 3.10(b) gezeigt. Hier wurde die „zeichnen“-Methode für die niedrigere Auflösung ausgewählt und statt der Implementierung „BerührungsempfindlicheAnzeige“ ist „Steuerkreuz“ Teil des Produktes.

Beide Produkte lassen sich nun mittels Fujaba direkt auf ausführbaren Quellcode ab-

bilden und anschließend auf das entsprechende Mobiltelefon ausliefern [Zün02]. Somit beschränkt sich die Entwicklung neuer Produkte im Idealfall auf Erstellen einer Konfiguration und anschließendes Generieren von Produkt und Quellcode aus dem konfigurierbaren Domänenmodell [Buc10]. Gleichzeitig dient die Konfiguration als Klassifikation des entsprechenden Produkts. Mit ihrer Hilfe lassen sich gezielt die Unterschiede zwischen den beiden Versionen des Mobiltelefon-Spiels identifizieren: Andere Auflösung und unterschiedliches Eingabegerät.

3.6. Kernprobleme beim Entwurf konfigurierbarer Domänenmodelle

Die Konfiguration eines Modells für ein Anwendungssystem erfolgt durch Entfernen von Modellelementen, deren Merkmalsmarkierungen nicht Teil der jeweiligen Konfiguration sind [BD09a]. Jede Konfiguration teilt so die Modellelemente des Domänenmodells in zwei Klassen ein: Die Elemente des konfigurierten Modells und die entfernten Elemente. Während das Zuweisen und Entfernen einzelner Merkmalsmarkierungen ein einfacher Vorgang ist, müssen folgende Kernprobleme beim Entwurf und beim Markieren des Domänenmodells gelöst werden:

1. Die Merkmalsmarkierungen sollen so angebracht sein, dass jedes konfigurierte Modell ein vollständiges und syntaktisch korrektes Fujaba-Modell ist.

2. Jedes mit Merkmalsmarkierungen versehene Modellelement soll in mindestens einem konfigurierten Modell vorkommen.

3. Jedes Merkmal soll mindestens ein Modellelement markieren.

4. Die Architektur des Domänenmodells soll so beschaffen sein, dass für jede Konfiguration die entsprechenden Elemente entfernt werden können, ohne die syntaktische Korrektheit zu beeinträchtigen.

5. Das Domänenmodell soll so markiert sein, dass die Semantik des konfigurierten Modells konsistent mit der Semantik der Merkmale ist.

3.6.1. Kernproblem: Syntaktisch korrekte konfigurierte Modelle

Das erste Kernproblem ist eine Anforderung des MODPL-Rahmenwerks. Das Ergebnis der Aufgabe „Konfiguriere Domänenmodell“ (vgl. Abbildung 3.2, S. 52) ist ein lauffähiges Anwendungssystem, das nur aus einem vollständigen und syntaktisch korrekten Modell generiert werden kann [Buc10]. Diese Eigenschaft soll für jede valide Konfiguration gelten. Da beim Konfigurieren Modellelemente entfernt werden, kann – analog zum Löschen eines Elements – die syntaktische Korrektheit durch Entfernen weiterer Elemente wiederhergestellt werden. Wird z.B. eine Klasse entfernt, so verlieren zunächst alle ein- und ausgehenden Beziehungen (u.a. Assoziationen, Generalisierungen) einen Endpunkt und

sind damit invalide. Werden sie auch entfernt, so ist die syntaktische Korrektheit wiederhergestellt. Analog können die Merkmalsmarkierungen einer Klasse auch an ihren ein- und ausgehenden Beziehungen angebracht werden, um so sicherzustellen, dass die Klasse entweder samt ihrer Beziehungen in einem konfigurierten Modell enthalten ist oder vollständig entfernt wurde [BD09a, BD09b].

Der MODPL-Editor automatisiert die Propagation von Merkmalsmarkierungen an abhängige Elemente. Dadurch wird der Arbeitsaufwand bei der Erstellung des Domänenmodells deutlich reduziert und zusätzlich seine Lesbarkeit verbessert, da sonst das Domänenmodell mit Merkmalsmarkierungen „überflutet“ wird. Der MODPL-Editor verwendet für die Propagation eine Menge von Konsistenzregeln wie das o.g. Beispiel mit den Klassen und ihren Beziehungen. Die Konsistenzbedingungen werden in [BD09a] detailliert beschrieben.

3.6.2. Kernproblem: Verwaiste Modellelemente

Das zweite Kernproblem folgt aus den Beziehungen des Merkmalsmodells. Durch sie wird die Menge der gültigen Konfigurationen bestimmt, d.h. es wird u.a. festgelegt, welche Merkmale sich gegenseitig ausschließen und niemals gemeinsam in einer Konfiguration vorkommen können (vgl. Untermerkmale von „AuflösungAnzeige“ in Abbildung 3.3, S. 54) [KCH+90, AC04, CHE05]. Wird nun ein Modellelement mit zwei exklusiven Merkmalen markiert, ist es nur Teil aller konfigurierten Modelle, in deren Konfiguration beide Merkmale gleichzeitig vorkommen. Da diese Konfigurationen ungültig sind, ist dieses Element niemals Teil eines gültigen konfigurierten Modells.

Der MODPL-Editor analysiert die Mengen der Merkmalsmarkierungen an den Modellelementen, erkennt Mengen, die nur in ungültigen Konfigurationen auftauchen und markiert die Modellelemente. Dabei ist die Erkennung auf einfache Fälle beschränkt, z.B. auf sich gegenseitig ausschließende Teilmerkmale des selben Elternmerkmals. Komplexe Abhängigkeiten, die durch Wenn-Dann-Klauseln ausgedrückt werden, sind, auf Grund der komplexen Auswertung, nicht durch die Analyse abgedeckt [Buc10].

3.6.3. Kernproblem: Verwaiste Merkmale

Das dritte Kernproblem folgt aus der Abfolge der Aufgaben des MODPL-Rahmenwerks, die eine schrittweise Verfeinerung der Modelle vorsehen. Das Merkmalsmodell wird daher vor dem konfigurierbaren Domänenmodell erstellt und dessen Merkmale für die Merkmalsmarkierungen verwendet [Buc10]. Dadurch kann es vorkommen, dass ein oder mehrere Merkmale nicht für Markierungen verwendet werden. Dies ist ein Hinweis darauf, dass entweder das Merkmal noch nicht modelliert wurde oder das Merkmalsmodell zu umfangreich ist.

Der MODPL-Editor analysiert die bereits verwendeten Merkmale in den Merkmalsmarkierungen und hebt nicht verwendete Merkmale hervor. Die Beziehung zwischen den Merkmalen und ihren Markierungen wird zur Laufzeit hergestellt, so dass das zu Grunde liegende Merkmalsmodell jederzeit ausgetauscht werden kann. So können nicht verwendete Merkmale noch entfernt werden [Buc10].

3.6.4. Kernproblem: Modulare Architektur

Das vierte Kernproblem ist eine generelle Anforderung an konfigurierbare Domänenmodelle. Die Architektur des Domänenmodells muss so beschaffen sein, dass – nach Sicherstellen der syntaktischen Korrektheit – das konfigurierte Modell ein vollständiges und lauffähiges Anwendungssystem darstellt [PBL05]. Daher müssen die Abhängigkeiten zwischen den Modellelementen denen der Merkmale entsprechen: Sind zwei optionale Merkmale voneinander unabhängig, so gibt es z.B. Konfigurationen mit (1) beiden, (2) einem der beiden oder (3) keinem der Merkmale. In gleichem Maße müssen die gekennzeichneten Modellelemente voneinander unabhängig sein: Die Modelle der konfigurierten Systeme müssen korrekt sein, wenn die Elemente mit (1) beiden Merkmalsmarkierungen, (2) nur einer der beiden oder (3) keiner der beiden Merkmalsmarkierungen vorhanden sind. Z.B. müssen mehrere alternative Merkmale (vgl. Untermerkmale von „EingabeSteuerung“ in Abbildung 3.3, S. 54) so modelliert sein, dass sie im Domänenmodell austauschbar sind (vgl. Klassen „Steuerkreuz“ und „BerührungsempfindlicheAnzeige“ in Abbildung 3.6, S. 57).

Der MODPL-Editor bietet keine Unterstützung für den Entwurf der Architektur, so dass die Entwickler selbst Entwurfsmuster identifizieren oder festlegen müssen, die bestimmte Merkmalskonstellationen gut unterstützen. Ein Ergebnis dieser Arbeit ist die Identifikation von geeigneten Entwurfsmustern für bestimmte Merkmalskonstellationen anhand des MOD2-SKM Domänenmodells.

3.6.5. Kernproblem: Semantische Konsistenz mit Merkmalsmodell

Das fünfte Kernproblem ist ebenfalls eine generelle Anforderung an Produktlinien. Die Merkmale des Merkmalsmodells repräsentieren Eigenschaften von Anwendungssystemen. Die jeweils damit markierten Elemente des konfigurierbaren Domänenmodells sollten diese Eigenschaften modellieren – ansonsten entspricht das konfigurierte Anwendungssystem nicht der Konfiguration [KCH+90, KKL+98, AC04]. Ein einfaches Beispiel für dieses Kernproblem ist z.B. das versehentliche Vertauschen zweier Merkmale: Wäre z.B. in Abbildung 3.6 (s. S. 57) die Klasse „Tastatur“ mit dem Merkmal „Steuerkreuz“ und die Klasse „Steuerkreuz“ mit dem Merkmal „Tastatur“ markiert, würden sie in den falschen Anwendungssystemen auftauchen.

Die semantische Konsistenz kann vom MODPL-Editor nicht validiert werden, so dass die Entwickler selbst die Konsistenz prüfen müssen [Buc10]. Eine teilweise Automatisierung der Validierung kann z.B. durch Validieren aller konfigurierten Modelle erfolgen, bzw. durch Tests der konfigurierten Anwendungssysteme. Der Testaufwand ist jedoch, auf Grund der hohen Zahl an gültigen Konfigurationen, sehr hoch und sehr komplex. MODPL bietet zur Zeit keine Unterstützung für Produktlinientests, so dass die Entwickler eigene Testverfahren entwerfen und implementieren müssen. Daher wird im Rahmen dieser Arbeit die Verwendung von parametrisierbaren Tests für Produktlinien untersucht und analysiert.

3.7. Zusammenfassung

Das MODPL-Rahmenwerk bietet Werkzeuge, um eine modellgetriebene und modulare Produktlinie zu entwerfen und umzusetzen [Buc10]. Ergebnis der Arbeit sind ein Merkmalsmodell [KCH+90, KKL+98, AC04, CHE05] und ein konfigurierbares Domänenmodell, um die Produktlinie als Ganzes zu modellieren [Buc10]. Aus dem Merkmalsmodell werden, je nach Anforderungen, Konfigurationen abgeleitet, mit denen anschließend aus dem konfigurierbaren Domänenmodell ein konfiguriertes Modell eines Anwendungssystems generiert wird [BD09a]. Der Schwerpunkt der Produktlinienentwicklung liegt auf dem Entwurf und der Modellierung des konfigurierbaren Domänenmodells, aus dem sich vollständig lauffähige Anwendungssysteme generieren lassen. Kernproblem für die Entwickler ist es, die Vollständigkeit sowie die syntaktische und semantische Korrektheit dieser Systeme sicherzustellen [BD09a]. Der MODPL-Werkzeugkasten unterstützt diese Aufgabe, jedoch sind die Möglichkeiten und der Bedarf an Werkzeugunterstützung kaum erforscht. Ziel dieser Arbeit ist es daher auch, mögliche Ansatzpunkte und Erkenntnisse für die Weiterentwicklung der Werkzeugunterstützung zu liefern.

4. Software-Konfigurations-Management

Allgemein ausgedrückt ist Software-Konfigurations-Management (SKM) eine Methode, um die Entwicklung eines Software-Projektes zu steuern [BM05]. Dabei werden die Entwickler durch ein Software-Konfigurations-Management-System (SKMS) unterstützt, das die anfallenden Daten elektronisch speichert und die Aufgaben teilweise automatisiert. Es existiert jedoch keine allgemeingültige Definition des Begriffs SKM [BM05]. Daher ist es schwierig, allgemeingültige Kriterien zu finden, die SKMS von anderen Software-Anwendungssystemen abgrenzen, bzw. die Mindestanforderungen an den Funktionsumfang und die Aufgabengebiete eines SKMS zu formulieren. Bevor also SKM-Konzepte beschrieben werden können, ist es wichtig zu klären, welche Aufgaben und Ziele SKM besitzt und welche Anforderung an den Funktionsumfang von SKMS sich daraus ableiten lassen.

4.1. Definition von SKM und SKMS

Es existieren mehrere Ansätze, SKM zu definieren. Im Mittelpunkt stehen dabei sowohl die Aufgaben und Ziele von SKM, als auch die Funktionen und Eigenschaften, die speziell von SKMS realisiert werden und sie so von anderen Software-Anwendungssystemen abgrenzen. Im Folgenden werden mehrere Ansätze vorgestellt, um den unterschiedlichen Perspektiven auf die Methode SKM gerecht zu werden. Die Ansätze werden dabei in Reihenfolge ihrer Veröffentlichung beschrieben und zeigen so die Entwicklung der Definitionen.

4.1.1. Was ist SKM?

Edward Bersoff definiert SKM wie folgt:

> „SKM ist – wie KM – die Disziplin, die zu diskreten Zeitpunkten die Konfiguration eines [Software-Anwendungs]-Systems identifiziert, mit der Absicht, die Änderungen an dieser Konfiguration systematisch zu kontrollieren und ihre Integrität und Nachvollziehbarkeit während des [gesamten] Lebenszyklus des Systems zu erhalten.“ ([BHS78], S. 11)

Im Mittelpunkt steht dabei die, aus der Hardware-Entwicklung stammende, Disziplin **K**onfigurations**m**anagement (KM, *engl.* configuration management, CM) – angewendet auf Software. Aus ihr werden die Ziele, Anforderungen und Aufgaben in die Domäne der Software-Entwicklung transferiert und – wenn nötig – an die speziellen Bedürfnisse der Software-Entwicklung angepasst. Bersoff identifiziert als Hauptunterschiede (1) die

leichte Veränderbarkeit der Software und (2) fehlende physikalische Einschränkungen [BHS78].

> „Das Hauptziel von SKM ist die kosten-effiziente Verwaltung des Lebenszyklus eines Software-Anwendungssystems“ ([BHS78], S. 11).

Das zu entwickelnde Software-Anwendungssystem wird auch als Produkt (engl. product) bezeichnet, da es das Ergebnis eines Software-Entwicklungsprozesses ist. SKM wird während des gesamten Lebenszyklus des Produktes eingesetzt, dementsprechend kann sich der Funktionsumfang eines SKMS ebenfalls über den gesamten Lebenszyklus erstrecken. Dieser Funktionsumfang lässt sich auf vier Aufgabengebiete aufteilen:

1. **Identifikation:** Die Auswahl und Kennzeichnung von Software-Konfigurationselementen und deren Gruppierung in Grundfassungen (engl. baselines).
2. **Kontrolle:** Die Verwaltung von Änderungsvorschlägen.
3. **Statusbuchhaltung:** Die administrative Verfolgung und Übersicht, insbesondere der Historie des Produktes, über den gesamten Lebenszyklus hinweg.
4. **Prüfung:** Die Verifikation der Integrität über mehrere Änderungen hinweg, sowie die Validierung der Kundenanforderungen.

Die Definition macht deutlich, dass – auf Grund der komplexen Abhängigkeiten und der großen Datenmengen – die Verwaltung der Historie des Produktes, über den gesamten Lebenszyklus hinweg, nur mit Hilfe eines speziellen Software-Anwendungssystems – also eines SKMS – möglich ist. Trotzdem konzentriert Bersoff sich auf SKM als Methode und weniger auf das SKMS zu seiner Unterstützung. So lässt er offen, ob ein Anwendungssystem nur ein oder alle vier Aufgabengebiete unterstützen muss, um als SKMS zu gelten. Es werden auch weder die unterschiedlichen Rollen der Entwickler, noch Methoden für die Kollaboration betrachtet [Dar90].

SKM koordiniert mehrere Entwickler

Wayne Babich konzentriert sich in seiner SKM-Definition mehr auf die organisatorischen Aspekte:

> „Die Kunst, die Entwicklung von Software so zu koordinieren, dass die [bei Teamarbeit unausweichliche] Verwirrung minimiert wird, heißt Konfigurationsmanagement“ ([Bab86], S. 8).

Er führt drei Hauptprobleme an, die sich durch SKM kontrollieren lassen und deren Kontrolle durch SKMS automatisiert werden kann:

- **Das Doppel-Wartungs-Problem[1]:** Mehrere identische Kopien eines Software-Elements müssen gewartet werden. Änderungen in einem Element müssen in die Kopien übertragen werden, andernfalls divergieren die Elemente.

[1] engl. double maintenance problem

- **Das Gemeinsame-Daten-Problem[2]:** Um das Problem der doppelten Wartung zu verhindern, greifen mehrere Entwickler gleichzeitig auf die selben Daten zu und verändern sie auch. Dadurch beeinflussen sich die Änderungen der Entwickler sofort gegenseitig.
- **Das Gleichzeitige-Aktualisierungs-Problem[3]:** Um die Problematik der gemeinsam genutzten Daten zu umgehen, verwendet jeder Entwickler eine eigene Kopie der Daten, um sie nach Abschluss seiner Arbeit an alle freizugeben. Dadurch kann es zu konkurrierenden Änderungen an den gleichen Daten kommen, die sich gegenseitig überschreiben (vgl. Abbildung 1.2, S. 17).

Das Problem der konkurrierenden Änderungen lässt sich durch SKMS kontrollieren, die die gemeinsamen Daten mit den persönlichen Kopien der Entwickler abgleichen [Bab86]. Damit liegt der Fokus dieser Definition auf der Koordination mehrerer Entwickler, die in Teams zusammenarbeiten. Aufgabengebiet eines SKMS ist damit die teilweise Automatisierung des Datenabgleichs und die Unterstützung der Entwickler bei der Zusammenarbeit.

SKM unterstützt den Software-Entwicklungsprozess

Phillipe Kruchten definiert SKM als Unterstützungsprozess, der in ein größeres Prozessmodell integriert ist – in diesem Fall in den Rational Unified Process (RUP). Dieser Unterstützungsprozess wird von den Entwicklern mit Hilfe eines Software-Werkzeugs, eines SKMS, durchgeführt. Kruchten identifiziert drei Aspekte auf SKM: „Konfigurationsverwaltung“, „Änderungsanfragen-Verwaltung“ (engl. change request management), sowie „Status und Messungen“.

Der Aspekt der Konfigurationsverwaltung basiert auf dem Aufgabengebiet „Identifikation“ von Babich, erweitert um Konzepte für die Zusammenarbeit mehrerer Entwickler. Der Aspekt der Änderungsanfragen-Verwaltung entspricht dem Aufgabengebiet „Kontrolle“, erweitert um die Analyse von Auswirkungen und Aufwand der Änderungen. „Status und Messungen“ ist in Teilen sowohl im Aufgabengebiet „Statusbuchhaltung“ als auch in „Prüfung“ enthalten, wobei der Schwerpunkt auf Daten zum Projektmanagement liegt. Insbesondere die letzte Zuordnung zeigt, dass das letztendliche Ziel der Definition die Integration von SKM in den RUP ist [Kru03].

4.1.2. Was ist ein SKMS?

Susan Dart greift die Definition von Bersoff auf und erweitert sie – basierend auf der Definition von Babich – um zusätzliche Aufgabengebiete, wie z.B. Kollaboration und Prozessunterstützung. Dabei konzentriert sie sich weniger auf SKM als Methode, sondern vor allem auf die Anforderungen an SKMS und was sie von anderen Software-Anwendungssystemen abhebt [Dar90]:

[2] engl. shared data problem
[3] engl. simultaneous update problem

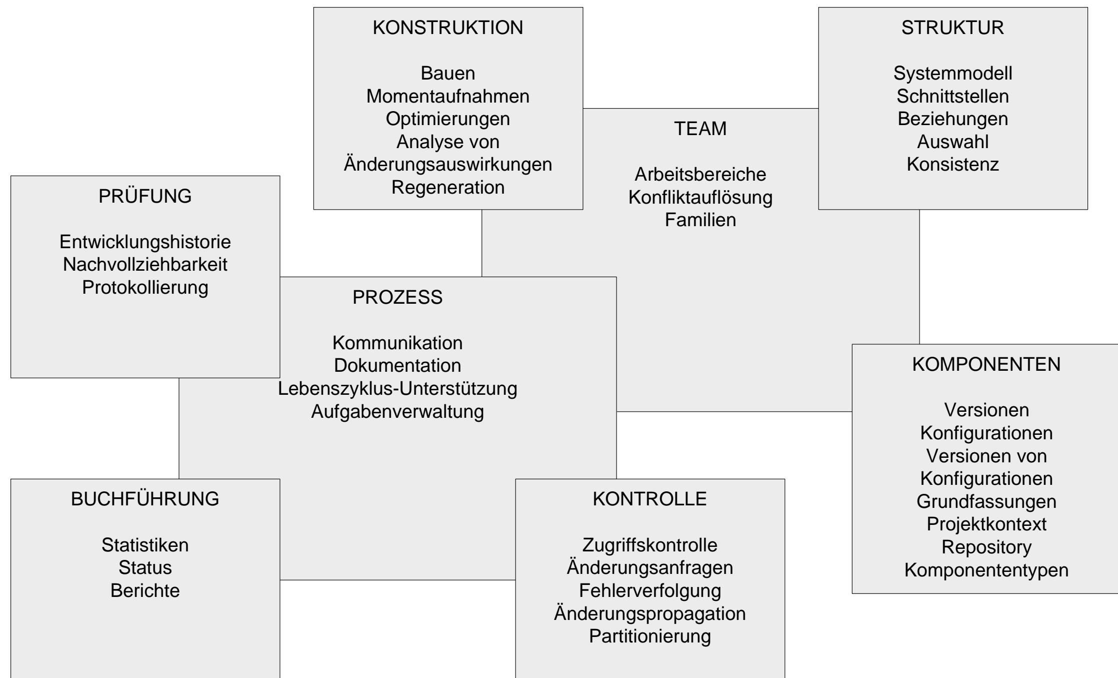


Abbildung 4.1.: Funktionale Anforderungen an SKMS (nach [Dar90])

> „Für das, was ein SKMS ausmacht, existiert keine allgemein anerkannte Definition. [...] Idealerweise deckt die Funktionalität ein SKMS alle Aufgabengebiete ab. In der Praxis wird jedoch jedes System, das eine Art Versionskontrolle, Konfigurationsidentifikation, Systemstrukturierung und Systemmodellierung bietet, von der Software-Engineering- (und Verkäufer-)Gemeinschaft als SKMS betrachtet.“ ([Dar90], S. 2)

Die Aufgabengebiete von SKMS werden erweitert, da ihr Funktionsumfang in der Praxis „über die oben genannte Definition [von Bersoff] hinausgeht“ ([Dar90], S. 1). Diese neuen Aufgabengebiete sind:

5. **Konstruktion:** Der Bau des Produktes, d.h. die Auswahl der Quell-Elemente, das Kompilieren oder Generieren, u.ä., erfolgt auf optimale Weise.

6. **Prozessmanagement:** Die Ausführung der Verfahren, Methoden und des Lebenszyklus-Modells der entwickelnden Organisation.

7. **Teamarbeit:** Die Kontrolle der Arbeit und Interaktion von mehreren Entwicklern an einem Produkt.

Abbildung 4.1 zeigt die sieben Aufgabengebiete – aufgeteilt auf acht Anforderungsbereiche – und nennt wichtige Unterkategorien in jedem Bereich. Dabei ist zu beachten, dass das Aufgabengebiet „Identifikation“ in die beiden Anforderungsbereiche „Struktur“ und „Komponenten“ zerlegt wird. Außerdem wird die Unterscheidung zwischen SKMS

und SKM-Werkzeug aufgegriffen: Die sechs Anforderungsbereiche am Rand sind Gebiete, die durch einzelne SKM-Werkzeuge abgedeckt werden (z.B. „Komponenten“ durch ein Werkzeug wie RCS), während die beiden zentral dargestellten Anforderungsbereiche „Prozessmanagement“ und „Teamarbeit“ die anderen sechs zu einem (vollständigen) SKMS integrieren.

4.1.3. Welche Aufgaben unterstützt ein SKMS?

Auch Bellagios und Milligans Definition basiert auf der von Bersoff und erweitert die Definition um eine Liste von bewährten Verfahren für SKM [BM05]:

- Artefakte identifizieren und in sicheren Repositorien speichern
- Änderungen an den Artefakten kontrollieren und prüfen
- Versionierte Artefakte als versionierte Komponenten gliedern
- Versionierte Komponenten und Subsysteme als versionierte Subsysteme organisieren
- Grundfassungen bei Projektmeilensteinen anlegen
- Änderungsanfragen aufnehmen und nachverfolgen
- Zusammengehörende Artefakte mit Hilfe von Aktivitäten organisieren und integrieren
- Stabile und konsistente Arbeitsbereiche pflegen
- Nebenläufige Änderungen an Artefakten und Komponenten unterstützen
- Früh und häufig integrieren
- Reproduzierbarkeit zusammengebauter Produkte sicherstellen

> „Bewährte Verfahren für SKM beziehen sich sowohl auf den SKM-Unterstützungsprozess als auch das SKMS.“ [BM05]

Bellagio bezieht sich hier ebenfalls auf die Definition von SKM als Unterstützungsprozess im RUP (vgl. [Kru03]), da beide Autoren Produkte der Firma *Rational* beschreiben.

4.1.4. Wer verwendet ein SKMS?

Conradi und Westfechtel greifen die Definitionen von Babich und Bersoff auf, und identifizieren sie als zwei unterschiedliche Perspektiven auf SKM:

> „SKM dient unterschiedlichen Bedürfnissen. [Zum einen] als Disziplin zur Unterstützung des Managements, wie durch Bersoff [BHS80] und den IEEE-Standard [IEE05] definiert [und zum anderen] als Disziplin zur Unterstützung der Entwickler, wie durch Babich [Bab86] definiert [...]“ ([CW98], S. 233).

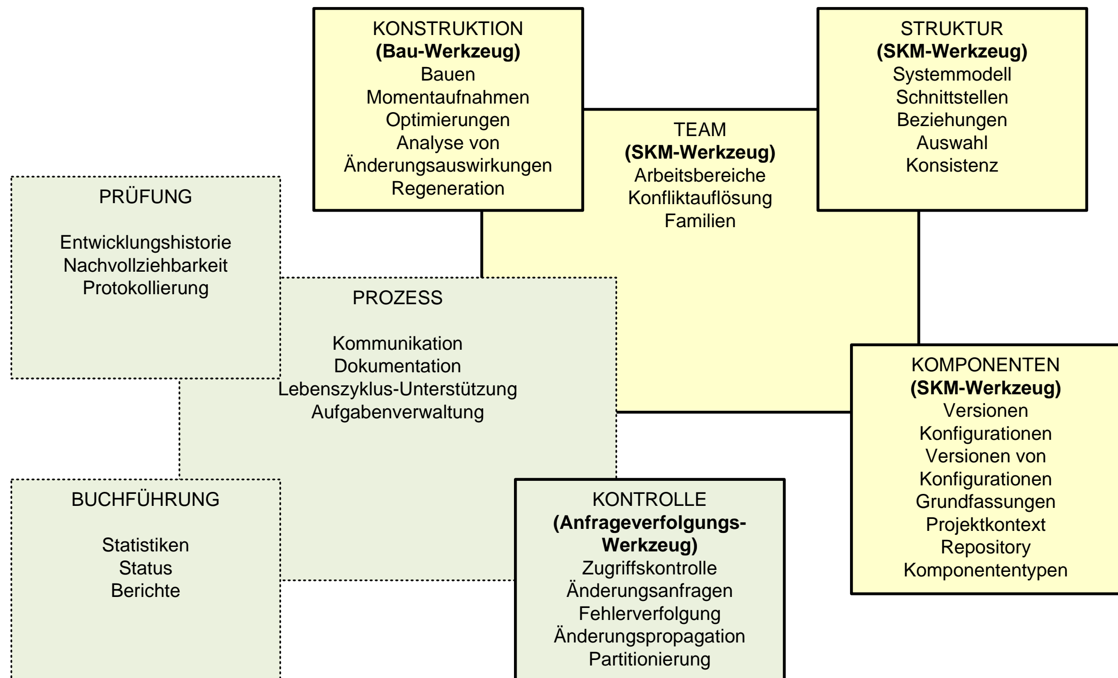


Abbildung 4.2.: Funktionale Anforderungen an SKMS aus Perspektive der Entwickler (*gelb*) und des Managements (*grün*)

Somit hängen die Anforderung an ein SKMS von der Perspektive der Nutzer ab [Dar90, CW98]. Dementsprechend lassen sich ihre Anforderungsbereiche für SKMS entsprechend der beiden Perspektiven gliedern. Abbildung 4.2 zeigt die Anforderungsbereiche, die der Management-Perspektive zugeordnet werden in grün und die Bereiche, die der Entwickler-Perspektive zugeordnet werden in gelb.

4.1.5. SKM-Umgebung oder SKM-Werkzeug?

Der Begriff SKMS beschreibt in der Praxis eine breit gefächerte Menge von Software-Anwendungssystemen, die in mindestens einem SKM-Aufgabengebiet einsetzbar sind. Dabei wird zwischen SKMS und SKM-Werkzeug unterschieden:

> „Ein SKMS ist Teil einer Umgebung, in der SKM-Unterstützung ein integraler Bestandteil ist. [...] Ein SKM-Werkzeug ist ein eigenständiges Werkzeug.“ ([Dar90], S. 2)

Als Beispiel für eine SKM-Umgebung nennt Dart die *Rational*-Umgebung und als Beispiel für ein SKM-Werkzeug das **R**evision **C**ontrol **S**ystem (*RCS*) (vgl. [Tic85]). Und auch Conradi und Westfechtel identifizieren eine Entwicklung von einzeln stehenden SKM-Werkzeugen hin zu integrierten SKM-Umgebungen [CW98]. Diese Unterscheidung zwischen SKM-Werkzeug und SKM-Umgebung ist bis heute aktuell. SKM-Umgebungen decken einen Großteil der Anforderungsbereiche ab, während sich SKM-Werkzeuge auf

einzelne Bereiche konzentrieren. In Abbildung 4.2 werden Anforderungsbereiche, die lediglich als Subsysteme einer vollwertigen SKM-Umgebung auftauchen, mit gestrichelten Linien begrenzt.

Gerade quelloffene Projekte setzen jedoch selten SKM-Umgebungen ein, sondern ebenfalls quelloffene SKM-Werkzeuge, wie z.B. *Subversion*, *Mercurial* oder *GIT* . Diese SKM-Werkzeuge unterstützen nur die Anforderungsbereiche „Team“, „Struktur“ und „Komponenten“ [CSFP08] [O'S09] [Ca09]. Zusätzlich werden für die anderen Anforderungsbereiche weitere Werkzeuge verwendet, die in der Lage sind, mit den SKM-Werkzeugen zusammenzuarbeiten: Bau-Werkzeuge (engl. build tools) übernehmen den Anforderungsbereich „Konstruktion“ (vgl. *make* [Fel79], *ant* [Hol05] oder *Maven* [Pop09]). Anfrageverfolgungssysteme (engl. issue tracking system) den Bereich „Prozess“ und „Controlling“, d.h. die Verwaltung der Aufgaben und der Änderungen (vgl. *Bugzilla* oder *Redmine* [Pop09]).

Mit diesen Werkzeugen lässt sich eine Werkzeugkette (engl. tool chain) aufbauen, die SKM-Werkzeuge mit anderen Systemen kombiniert und so eine leichtgewichtige SKM-Umgebung aufbaut [Pop09]. In Abbildung 4.2 sind die typischen Anforderungsbereiche einer Werkzeugkette mit durchgezogenen Linien begrenzt. Zusätzlich wird der Werkzeug-Typ unterhalb des Namens notiert. SKM-Werkzeuge werden auch oft als Revisionskontrollsysteme (RKS) oder Versionskontrollsysteme (VKS) bezeichnet.

4.1.6. MOD2-SKM oder MOD2-VKS?

Es existieren zwei Perspektiven auf SKM: Einmal aus Sicht der Manager und einmal aus Sicht der Entwickler. Dies spiegelt auch die Klassifizierung von SKM-Umgebung und SKM-Werkzeugen (insbesondere VKS) wider: Eine SKM-Umgebung versucht, die Anforderungen beider Perspektiven zu erfüllen, während VKS – als Teil einer Werkzeugkette – sich größtenteils auf die Anforderungen der Entwicklerperspektive konzentrieren. Der Umfang einer SKM-Umgebung ist zu groß, um ihn ihm Rahmen dieser Arbeit zu analysieren und in einer Produktlinie umzusetzen. Daher konzentriert sich die Implementierung von MOD2-SKM auf die Entwicklerperspektive und damit auf die Anforderungsbereiche eines VKS. Somit beschränken sich die im Folgenden beschriebenen SKM-Konzepte auf Verfahren und Methoden der Entwicklerperspektive.

Warum heißt die Produktlinie dann nicht gleich MOD2-VKS? MOD2-SKM wird als Kern einer SKM-Umgebung entwickelt. Während der Domänenanalyse wird die Produktlinie im Kontext einer SKM-Umgebung betrachtet, und explizit Anknüpfungspunkte für die Erweiterung um zusätzliche Anforderungsbereiche angeboten. Die Implementierung von MOD2-SKM beschränkt sich zwar auf die VKS-Anforderungsbereiche „Team“, „Struktur“ und „Komponenten“, doch die Architektur der Produktlinie ist ebenfalls mit Hinblick auf eine Erweiterung zu einer SKM-Umgebung entworfen worden.

4.2. Konzepte im Anforderungsbereich „Struktur“

Aufgabe ist die Modellierung der Struktur des Software-Anwendungssystems, des sogenannten **Produktraums**. Der Produktraum besteht aus den Software-Elementen des

Produktes und ihren Beziehungen untereinander, z.B. der Gliederung in eine Komponentenstruktur. Ergebnis der Modellierung ist das Produktmodell, das festlegt, aus welchen Element-Typen das zu versionierende System besteht und welche Beziehungs-Typen zwischen den Elementen existieren. Zentrale Bedeutung besitzt die Gruppierung einzelner Elemente zu zusammengesetzten Objekten – sog. Konfigurationen (engl. configurations). Mit ihrer Hilfe lassen sich die anderen Software-Elemente hierarchisch gliedern. Insbesondere lässt sich das gesamte Produkt so als Konfiguration erfassen [CW98]. Es kann zusätzlich auch Konsistenzbedingungen enthalten, um bei der Auswahl aus den versionierten Elementen sicherzustellen, dass eine korrekte und vollständige Instanz vorliegt.

4.2.1. Produktraum

Der Produktraum beschreibt die Struktur eines Software-Produkts. Er besteht aus Software-Elementen und ihren Beziehungen untereinander. Software-Elemente sind dabei nicht nur Quellcode, sondern jegliche Objekte, die während des Software-Lebenszyklus angelegt werden, z.B. Anforderungsspezifikationen, Entwürfe, Handbücher, Test- und auch Projektpläne. Um ein Software-Element im Produktraum zu identifizieren, wird jedem Element eine Objektkennung (engl. object identifier, OID) zugeordnet.

4.2.2. Die Objektkennung

Die OID hat die Aufgabe, ein Software-Element eindeutig im Produktraum zu identifizieren und die Identität der Elemente sicherzustellen [WJ95]. Es handelt sich bei ihr um einen symbolischen Namen, der dem Element zugeordnet wird. Die Zuordnung erfolgt über ein Namensschema, das einen symbolischen Namen einem Element zuordnet, und das folgende Eigenschaften besitzen sollte [WJ95]:

- **Singuläre Referenz:** In jeder endlichen Menge von Software-Elementen referenziert ein symbolischer Namen genau ein Element.
- **Singuläre Namen:** In jeder abzählbaren Menge von Software-Elementen trägt ein Element genau einen symbolischen Namen.
- **Monotone Benennung:** Für jedes Paar von aufeinanderfolgenden[4] Mengen von Software-Elementen gilt, dass die Namenszuweisungen der Vorgängermenge Teil der Zuweisungen der Nachfolgermenge sind, d.h. , eine OID darf weder erneut benutzt, noch ein Element umbenannt werden.

4.2.3. Beziehungen im Produktraum

Es existieren verschiedene Arten von Beziehungen zwischen den Software-Elementen des Produktraums, z.B. die Kompositions- oder die Ableitungsbeziehung. Mittels einer Kompositionsbeziehung lassen sich die Software-Elemente entsprechend ihrer Granularität

[4] Zwei Mengen von Software-Elementen stehen in einer Vorgänger-Nachfolger-Beziehung, wenn die zweite Menge durch Veränderung der ersten entsteht, z.B. durch Fortschreiten der Entwicklung.

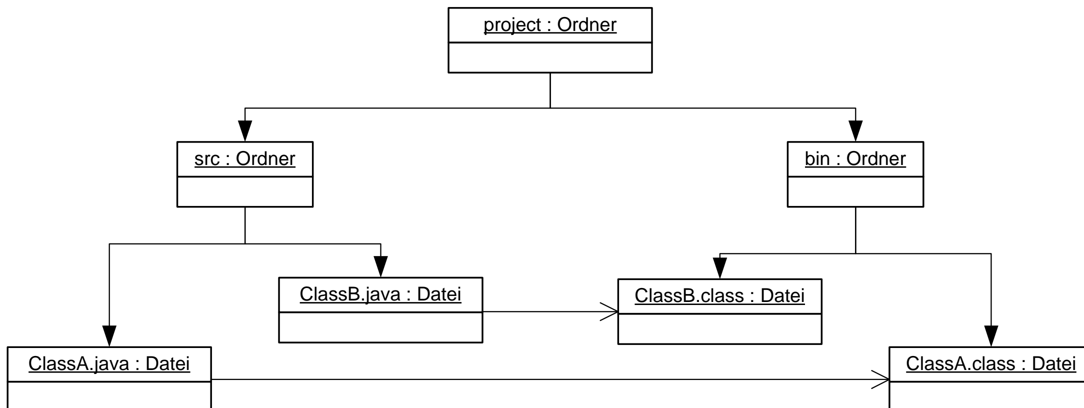


Abbildung 4.3.: Produktmodell für ein Dateisystem mit Kompositions- (volle Pfeilspitze) und Ableitungsbeziehungen (hohle Pfeilspitze)

strukturieren, z.B. indem mehrere Quellcode-Dateien zu einem Modul zusammengefasst werden oder mehrere Klassen zu einem Paket. So lässt sich eine Kompositionshierarchie aufbauen. Software-Elemente, die Beziehungen zu Unterelementen besitzen, werden auch als komplexe oder zusammengesetzte Elemente bezeichnet, bzw. auch als Konfiguration. Lässt sich ein Element nicht weiter zerlegen, so wird es atomar genannt [CW98].

Eine Ableitungsbeziehung drückt aus, dass sich Software-Elemente aus anderen automatisiert ableiten lassen. Sie besteht z.B. zwischen Quellcode-Dateien und ihrem Binärcode oder zwischen zwei Modellen, die sich automatisiert ineinander transformieren lassen. Wichtig ist, dass ein Software-Werkzeug existiert, um das Ziel-Element automatisiert aus dem Quell-Element abzuleiten. Das Werkzeug und der Ableitungsprozess sind jedoch nicht mehr Teil des Anforderungsbereichs „Komponenten“, sondern gehören zum Bereich „Konstruktion“. Die dort eingesetzten Bau-Werkzeuge nutzen dann die Ableitungsbeziehung, um den Bauprozess zu optimieren.

Abbildung 4.3 zeigt ein Beispiel für ein Produktmodell auf Basis von Ordnern und Dateien. Im Ordner „projekt“ sind die Ordner „src“ und „bin“ enthalten. Während in „src“ zwei java-Dateien liegen („ClassA.java“ und „ClassB.java“), befinden sich in „bin“ zwei kompilierte Klassen („ClassA.class“ und „ClassB.class“). Zwischen der jeweiligen Quellcode- und Binärdatei besteht eine Ableitungsbeziehung.

4.2.4. Produktmodelle

Der Produktraum wird in einem SKMS mit Hilfe eines Datenmodells dargestellt, das auch als Produktmodell bezeichnet wird. Es muss die Möglichkeit bieten, Software-Elemente und die Beziehungen zwischen ihnen zu beschreiben, sowie OIDs zu verwalten und den Elementen zuzuordnen. In der Praxis existieren viele unterschiedliche Produktmodelle (fast jedes SKMS hat sein eigenes Produktmodell), die eigene Konzepte für Software-Elemente, Beziehungen und OIDs besitzen.

CVS

Die Software-Elemente des CVS-Produktmodells sind lediglich Dateien. Kennung der Elemente ist der einfache Dateiname (ohne Pfadangabe), d.h. es handelt sich um keine „echte“ OID, da Umbenennungen erlaubt sind. Es gibt keine Möglichkeit, Beziehungen zwischen den Elementen auszudrücken, d.h. es gibt auch keine zusammengesetzten Elemente.

Subversion

Die Software-Elemente des SVN-Produktmodells sind sowohl Dateien als auch Verzeichnisse. Die Kennung der Elemente ist eine intern vergebene OID. Es existiert eine Kompositionsbeziehung zwischen einem Verzeichnis und den darin enthaltenen Software-Elementen, so dass Verzeichnisse zusammengesetzte und Dateien atomare Software-Elemente sind.

GIT

Die Software-Elemente des GIT-Produktmodells sind Dateiinhalte und Verzeichnisse. Die Kennung eines Elements ist der Wert der SHA-1-Hashfunktion, der die interne Repräsentation des Elements übergeben wurde. Es existiert eine Kompositionsbeziehung zwischen einem Verzeichnis und den darin enthaltenen Software-Elementen, so dass Verzeichnisse zusammengesetzte und Dateiinhalte atomare Software-Elemente sind.

Mercurial

Die Software-Elemente des Mercurial-Produktmodells sind Dateien und Verzeichnisse. Die Kennung eines Elements ist eine intern vergebene OID. Es existiert eine Kompositionsbeziehung zwischen einem Verzeichnis und den darin enthaltenen Software-Elementen, so dass Verzeichnisse zusammengesetzte und Dateien atomare Software-Elemente sind.

4.3. Konzepte im Anforderungsbereich „Komponenten“

Aufgabe ist die Identifikation, Klassifikation und Verwaltung der Versionen von Software-Elementen, des sogenannten **Versionsraums**. Das SKMS unterstützt die Entwickler, indem es die Entwicklung der Software-Elemente, über den gesamten Software-Lebenszyklus des Produktes hinweg, aufzeichnet und ihnen Zugriff auf die Elemente gewährt. Es wird die vollständige Entwicklungshistorie des Produktes gespeichert. In der Regel kommt dabei ein Repositorium (engl. repository) zum Einsatz, um die anfallenden Daten zu speichern und den Zugriff auf sie zu verwalten [Dar90].

Der Versionsraum besteht aus den einzelnen Versionen der Entwicklungshistorie der Komponenten des Produktmodells. Ein Kernproblem ist die Kombination einzelner Software-Elemente zu gültigen Instanzen des Produktmodells bzw. zu funktionsfähigen und

korrekten Konfigurationen aus dem Produktraum. Daher werden Konzepte für das **Zusammenspiel von Versions- und Produktraum** vorgestellt, welche die Software-Elemente des Produktraums mit ihrer Entwicklungshistorie in Bezug setzen [CW98]. Ein weiteres Problem ist die Speicherplatz- und Zugriffszeit-effiziente Speicherung der Historien, so dass abschließend noch einige Verfahren zur effizienten **Datenspeicherung** vorgestellt werden [HVT98, Ca09].

4.3.1. Versionsraum

Der Versionsraum beschreibt die Entwicklung eines Software-Produkts. Er besteht aus versionierten Software-Elementen, Versionen und ihren Beziehungen untereinander. Ein versioniertes Element ist ein Software-Element, dessen Entwicklung explizit verwaltet wird: Verändert ein Entwickler ein versioniertes Element, so wird es nicht überschrieben, sondern eine neue Version angelegt. Zu einem versionierten Element gehören also eine oder mehrere Versionen. Jede Version ist einem versionierten Element zugeordnet, d.h. sie beschreiben das gleiche Element. Zwischen den Versionen können Beziehungen bestehen, z.B. eine Vorgänger-Nachfolger-Beziehung, die ausdrückt, dass eine Version durch Veränderung einer anderen entstanden ist [CW98].

4.3.2. Die Versionskennung

Während das versionierte Element über seine Objektkennung eindeutig identifiziert werden kann, benötigt jede Version noch ihre eigene Versionskennung (engl. version identifier, VID) [CW98]. Auch diese sollte – wie die Objektkennung – eine OID sein (vgl. Objektkennung in Abschnitt 4.2.2). Eine Version wird so durch das Tupel (OID, VID) eindeutig identifiziert, d.h. die VID muss nicht im ganzen Versionsraum eindeutig sein, sondern nur in Bezug auf das versionierte Element. Unterschiede zwischen verschiedenen Versionierungsverfahren spiegeln sich häufig in unterschiedlichen Anforderungen an die Struktur und den Inhalt der VID wider [CW98].

Explizite und implizite Versionierung

Die Versionskennung kann explizit oder implizit verwendet werden, was auch als explizite bzw. implizite Versionierung bezeichnet wird [CW98]. Bei der expliziten Versionierung wird die VID explizit einer Version eines Software-Elements zugeordnet, um es später durch Angabe der VID wiederherstellen zu können. Bei der impliziten Versionierung werden einem Software-Element lediglich Eigenschaften zugeordnet. Als VID dient nun ein Ausdruck über der Menge der Eigenschaften, der die Menge aller Elemente auf das entsprechende Element einschränkt. Dabei muss die Version des Elements vorher nicht explizit vorgelegen haben, sie wird vielmehr implizit durch den Ausdruck erzeugt [CW98].

Zustandsbasierte und änderungsbasierte Versionierung

Eine Versionskennung kann entweder direkt den Zustand eines Software-Elements referenzieren, oder die Änderungen, die auf eine Grundfassung angewendet werden sollen, was

Abbildung 4.4.: Beispiel für eine Sequenz von Revisionen

auch als zustandsbasierte bzw. änderungsbasierte Versionierung bezeichnet wird [CW98]. Im Unterschied zur impliziten und expliziten Versionierung gibt es hier jedoch auch noch Unterschiede bei der Datenspeicherung (vgl. Abschnitt 4.3.6). Bei der zustandsbasierten Versionierung referenziert eine VID (implizit oder explizit) eine Version eines Software-Elements. Bei der änderungsbasierten Versionierung setzt sich die VID aus einer (explizit oder implizit definierten) Menge von Änderungskennungen (engl. change identifier, CID) zusammen, die auf eine Grundfassung des Software-Elements angewendet werden [CW98].

4.3.3. Beziehungen im Versionsraum

Es existieren verschiedenen Beziehungen zwischen den Versionen des Versionsraums. Die Vorgänger-Nachfolger-Beziehung drückt aus, dass die Nachfolger-Version durch Veränderung der Vorgänger-Version entstanden ist, z.B. wenn ein Entwickler neuen Quellcode hinzufügt oder bestehenden korrigiert. So lässt sich eine zeitliche Entwicklungshistorie aufbauen, die aus einer Sequenz von Versionen besteht, die durch Vorgänger-Nachfolger-Beziehungen verbunden sind. Diese Sequenz wird auch als Versionshistorie bezeichnet. Die aktuellste Version der Historie besitzt keinen Nachfolger und wird auch Kopf (engl. head) genannt. Versionen, die in einer Vorgänger-Nachfolger-Beziehung stehen, werden auch als Revisionen bezeichnet [CW98]. Bzgl. der Beziehung bilden die Revisionen eine Sequenz, wie in Abbildung 4.4 gezeigt wird: Drei Revisionen stehen in Vorgänger-Nachfolger-Beziehung, d.h. „Version 3“ bildet den Kopf der Sequenz.

Eine Alternativbeziehung wird dann verwendet, wenn zwei oder mehrere Versionen als Variante eines Software-Elements existieren. Im Gegensatz zu den Revisionen aus den Vorgänger-Nachfolger-Beziehungen sind Varianten Versionen, die parallel existieren und auch eigene Nachfolger besitzen können. Während eine Version nur eine Revision als Nachfolger hat, können mehrere Versionen in Alternativbeziehung zu ihr stehen. Dadurch bilden die Versionen bzgl. der beiden Beziehungen einen Baum, wie in Abbildung 4.5 gezeigt wird: Die Versionen „Version 1“ bis „Version 3“ stehen in Vorgänger-Nachfolger-Beziehung zueinander. „Version 2.1“ und „Version 3.1“ sind Alternativen zu „Version 2“ bzw. „Version 3“. „Version 2.1“ wird ebenfalls weiterentwickelt und besitzt bereits mit „Version 2.2“ einen Nachfolger. Alle Versionen, die miteinander in Vorgänger-Nachfolger-Beziehung stehen, bilden eine Revisions-Sequenz, die als Zweig des Baumes bezeichnet wird, an dessen Anfang immer eine Alternativbeziehung steht. Nur zur allerersten Version besteht keine Alternativbeziehung, so dass diese Revisions-Sequenz als Hauptzweig oder Stamm bezeichnet wird.

Mit einer Verschmelzungsbeziehung lässt sich das Zusammenführen von Versionen be-

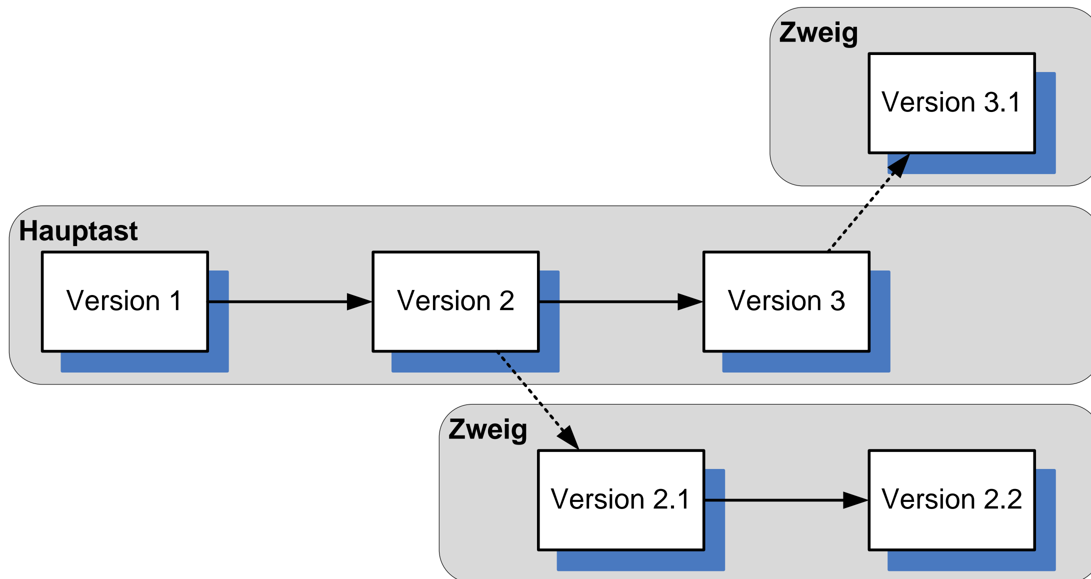


Abbildung 4.5.: Beispiel einer Historie als Baum

schreiben. Während eine Revision immer nur einen Vorgänger besitzt, kann sie Ziel von zwei (oder mehr) Verschmelzungsbeziehungen sein. In diesem Fall besteht die Ziel-Version aus der – automatischen oder manuellen – Verschmelzung der Quell-Versionen. Dadurch entsteht aus dem Baum mit seinem Hauptzweig und seinen Zweigen ein gerichteter azyklischer Graph (engl. directed acyclic graph, DAG), wie in Abbildung 4.6 gezeigt: „Version 4“ entsteht durch das Verschmelzen von „Version 3“ und „Version 2.2“.

Bäume und gerichtete azyklische Graphen aus Vorgänger-Nachfolger-Beziehungen

Anstatt Bäume mit Hilfe von Alternativbeziehungen, und gerichtete azyklische Graphen mit Verschmelzungsbeziehungen, zu erzeugen, können diese beiden Datenstrukturen auch schon mittels der Vorgänger-Nachfolger-Beziehungen entstehen, wenn Revisionen mehr als einen Nachfolger oder Vorgänger besitzen dürfen [CW98]. Ohne Beschränkung der Allgemeinheit lässt sich solch eine 1-Kantentyp-Datenstruktur mit der entsprechenden 2- oder 3-Kantentyp-Datenstruktur ausdrücken, indem einfach der Kantentyp ignoriert wird. Vielmehr ist es in der Praxis sogar so, dass in Systemen, die nur Vorgänger-Nachfolger-Beziehungen kennen, mit Hilfe der Versionskennung oder anderen Merkmalen implizit Alternativ- oder Verschmelzungskanten eingeführt werden (vgl. Magische Zweignummern in CVS [Ced05], S.44f.), so dass sich diese 1-Kantentyp-Datenstrukturen in eine entsprechende 2- oder 3-Kantentypen-Datenstruktur überführen lassen.

4.3.4. Integration von Produkt- und Versionsraum

Der Versionsraum versioniert – für sich allein betrachtet – einzelne Software-Elemente. Doch erst durch die Kombination des Versionsraumes mit dem Produktraum entsteht

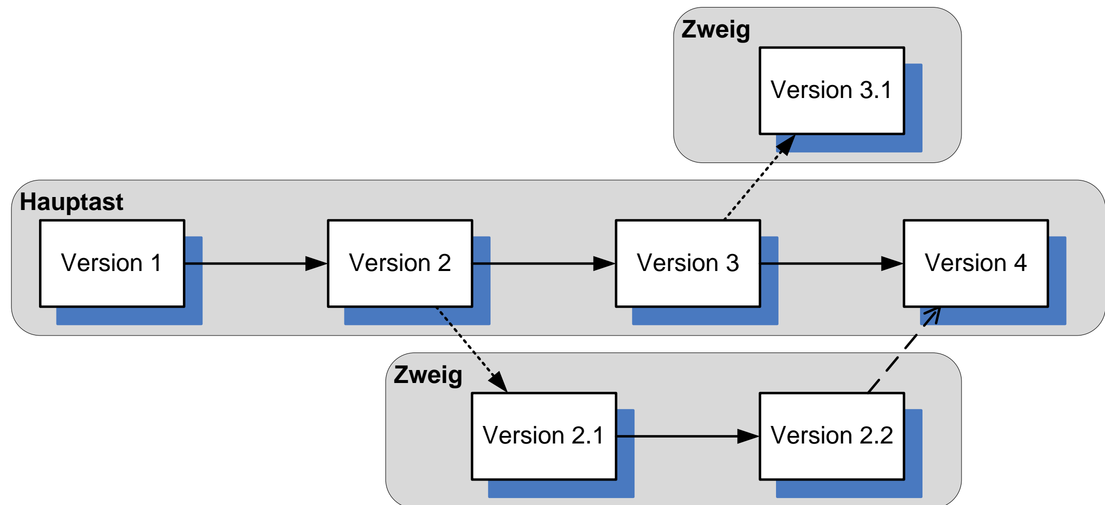


Abbildung 4.6.: Beispiel einer Historie als Gerichteter Azyklischer Graph

eine versionierte Objektbasis (engl. versioned object base, VOB) [CW98]. In ihr werden nicht nur Software-Elemente aus dem Produktmodell versioniert, sondern auch deren Beziehungen im Produktraum. Es existieren drei grundsätzliche Verfahren zur Integration von Produkt- und Versionsraum: der produktzentrierte (engl. product first), der versionszentrierte (engl. version first) und der verschränkte (engl. intertwined) Ansatz. Alle drei Ansätze lassen sich mit Hilfe von UND/ODER-Graphen veranschaulichen (vgl. Abbildung 4.8). Bei einem UND/ODER-Graphen existieren zwei Knotentypen: ODER-Knoten (Oval) und UND-Knoten (Rechteck), die durch ODER- (gestrichelt) bzw. UND-Kanten (durchgezogen) verbunden sind und jeweils vom Knoten des gleichen Typs ausgehen. Die Kompositions-Beziehungen im Produktraum lassen sich nun auf die UND-Kanten abbilden, und die Zuordnung der Versionen zu einem versionierten Element auf die ODER-Kanten. Jedes unversionierte Produkt kann somit durch einen reinen UND-Graphen beschrieben werden [CW98]. Um eine Version eines Produkts aus dem UND/ODER-Graphen abzuleiten, müssen die Knoten gebunden werden. Ein ODER-Knoten wird gebunden, indem ein Kindknoten ausgewählt wird. Ein UND-Knoten ist gebunden, wenn alle seine Kindknoten gebunden sind [CW98]. Gebundene Knoten sind in den Beispielen in Abbildung 4.7 und Abbildung 4.8 durch grauen Hintergrund und dickere Linien hervorgehoben.

Die drei Integrationsansätze lassen sich nun anhand des Beispiel-Produktmodells in Abbildung 4.3 beschreiben. Im Produktraum existieren zwei Quellcode-Dateien und zwei kompilierte Klassen – jeweils in separaten Unterverzeichnissen des Projektverzeichnisses. Im Versionsraum wurde jedes versionierte Element angelegt und – bis auf „ClassB.java“ – nicht mehr verändert, d.h. es existiert genau eine Revision dieser Elemente. „ClassB.java“ und die daraus abgeleitete „ClassB.class“ wurden einer Änderung unterzogen, d.h. es existieren jeweils zwei Revisionen dieser Elemente.

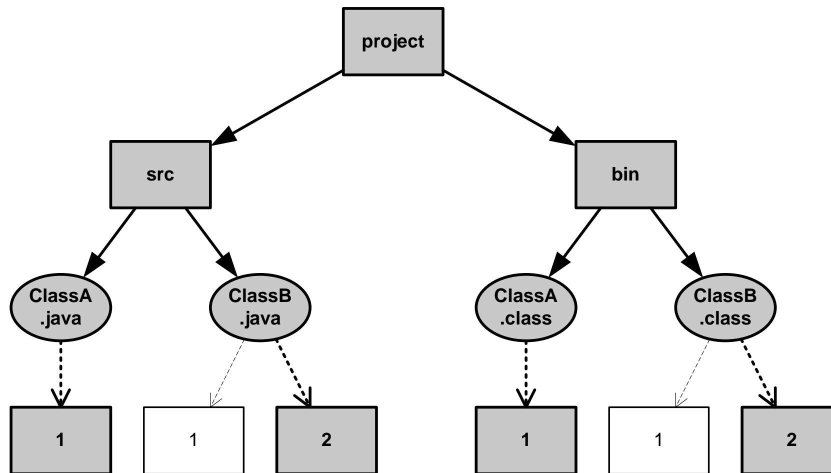


Abbildung 4.7.: Produktzentriertes Verfahren (Graue Elemente sind gebunden)

Produktzentrierte Integration

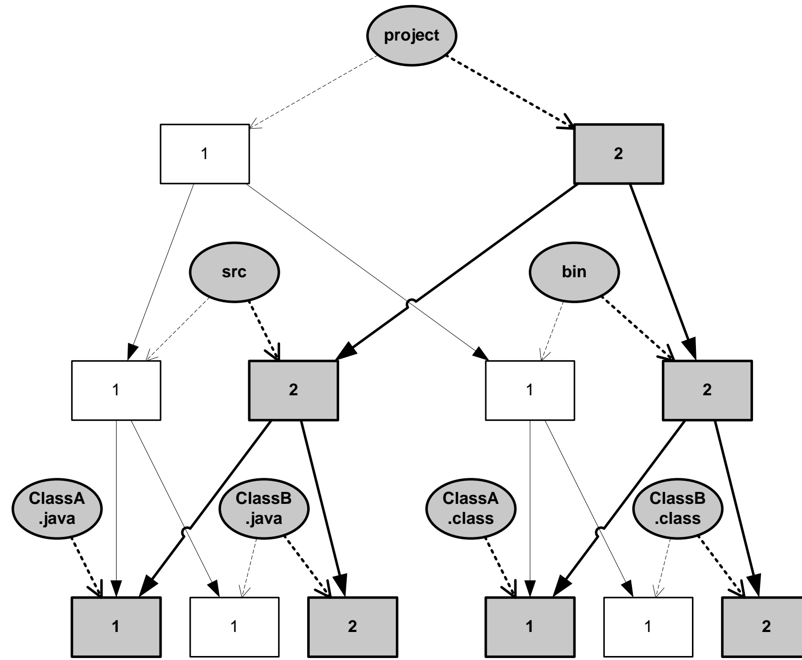
Im produktzentrierten Ansatz (Abbildung 4.7) wird zunächst das vollständige Produkt ausgewählt, wie die UND-Knoten zeigen. Erst wenn feststeht, welche Elemente Teil des Produktes sind, wird ihre Version festgelegt (vgl. ODER-Knoten). Mit diesem Ansatz ist es nicht möglich, die Beziehungen zwischen den Elementen zu versionieren [CW98], d.h. es ist möglich, inkompatible Versionen auszuwählen. Z.B. ist im UND/ODER-Graph in Abbildung 4.7 keine Beziehung zwischen „ClassB.java“ und „ClassB.class“ versioniert, so dass die ausgewählte java-Datei nicht unbedingt der kompilierten class-Datei entspricht.

Version-first

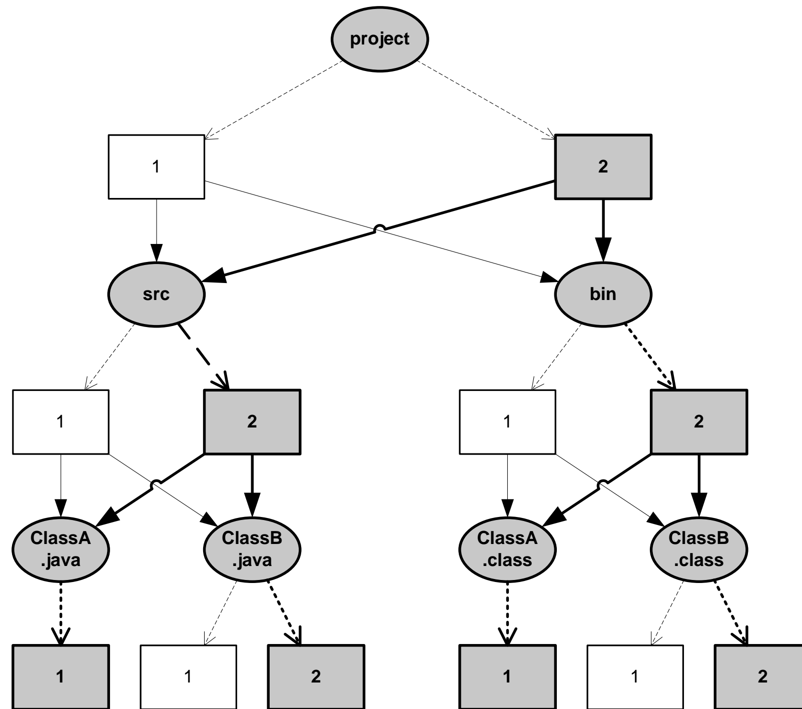
Im versionszentrierten Ansatz (Abbildung 4.8(a)) wird zunächst die Version des vollständigen Produkts ausgewählt, wie die UND-Knoten zeigen. Dadurch wird auch festgelegt, welche Elemente überhaupt Teil des Produktes sind; alle ODER-Knoten werden automatisch gebunden. Somit wird auch die Beziehung zwischen „ClassB.java“ und „ClassB.class“ versioniert: Quellcode-Datei und Binärdatei liegen immer in der gleichen Fassung vor.

Intertwined

Im verschränkten Ansatz (Abbildung 4.8(b)) wird zunächst nur die Version des Projektverzeichnisses ausgewählt. Dadurch wird festgelegt, welche Kindknoten in ihm enthalten sind. Nun wird erneut ausgewählt, in welcher Version die Kindknoten vorliegen. Dies setzt sich so lange fort, bis alle Knoten gebunden sind. Die Auswahl der Versionen der Kindknoten kann sowohl manuell durch den Entwickler als auch automatisch durch Auswahlkriterien erfolgen [CW98].



(a) Versionszentriertes Verfahren



(b) Verschränktes Verfahren

Abbildung 4.8.: Integration von Produkt- und Versionsraum (Graue Elemente sind gebunden)

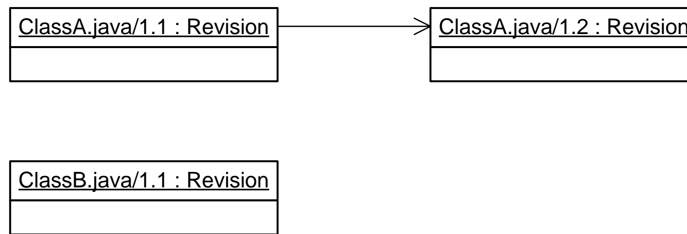


Abbildung 4.9.: Beispiel für Versionsmodell von CVS

4.3.5. Versionsmodelle

Der Versionsraum wird in einem SKMS mit Hilfe eines Datenmodells dargestellt, das auch als Versionsmodell bezeichnet wird. Es muss die Möglichkeit bieten, versionierte Software-Elemente, ihre Versionen und deren Beziehungen zu beschreiben, sowie VIDs zu verwalten und den Versionen zuzuordnen. In der Praxis existieren viele unterschiedliche Versionsmodelle (fast jedes SKMS hat sein eigenes Versionsmodell), die eigene Konzepte für Versionen, deren Beziehungen und VIDs besitzen. Die Abbildungen 4.9-4.12 zeigen jeweils ein Beispiel für das jeweilige Versionsmodell, d.h. es handelt sich in den Abbildungen nicht um die Versionsmodelle selbst, sondern um Instanzen der Versionsmodelle.

CVS

Versionierte Elemente sind Dateien. Jede versionierte Datei verwaltet ihre eigene VID der Form „1.x“ wobei „x“ der Anzahl der Versionen entspricht („1.42“ ist also die 42. Version). Versionen, deren letzte Stelle aufeinanderfolgt, während ihre anderen Stellen übereinstimmen (z.B. „1.41“ und „1.42“), stehen in Vorgänger-Nachfolger-Beziehungen. Dies wird zum Sparen von Speicherplatz ausgenutzt: Nur die Kopf-Revision einer Datei wird vollständig gespeichert. Statt den vollständigen Vorgängerversionen wird nur ein Patch vom Nachfolger auf den Vorgänger angelegt, um nur die Unterschiede zwischen Vorgänger und Nachfolger speichern zu müssen. Nachfolger mit einer ungeraden Anzahl von Stellen (z.B. „1.42.2“) stellen eine Verzweigungsbeziehung zur Version „1.42.2.1“ dar. Die Beziehungen werden also nicht explizit angelegt, sondern mit Hilfe des Namensschemas der VID verwaltet. Die Integration von Produkt- und Versionsraum erfolgt produktzentriert, da CVS die Dateien ohne ihre Beziehungen versioniert [Ced05].

Abbildung 4.9 zeigt ein Beispiel für das Versionsmodell von CVS. Als Grundlage dient das Beispiel-Produktmodell aus Abbildung 4.3, wobei nur die beiden „.java“-Dateien betrachtet werden. Initial liegen beide Dateien jeweils als Revision mit der VID „1.1“ vor. Verändert ein Entwickler nun die Datei „ClassA.java“, legt CVS eine neue Revision mit der VID „1.2“ an, und zwar als Nachfolger von Revision „1.1“. Die Revisionen von „ClassB.java“ werden vollständig unabhängig davon verwaltet.

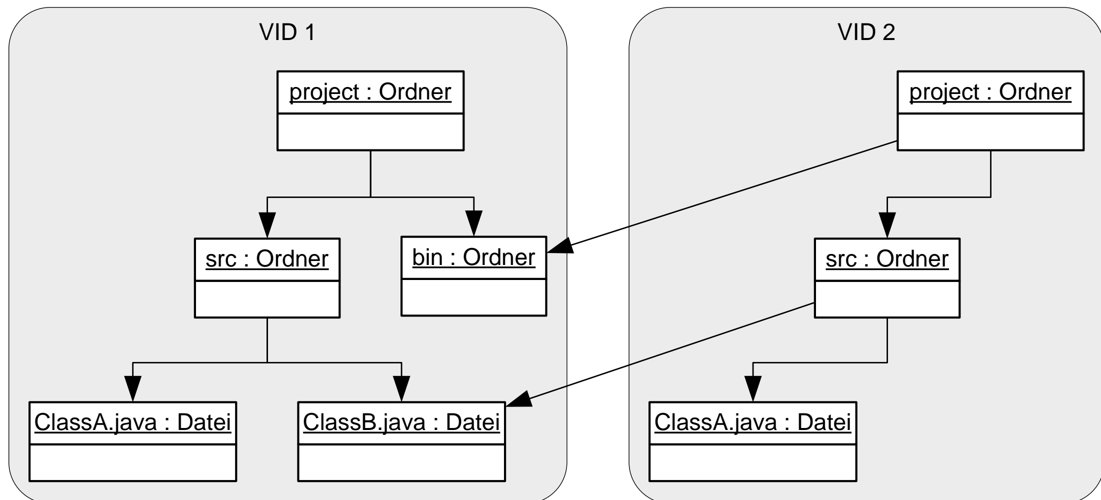


Abbildung 4.10.: Beispiel für Versionsmodell von Subversion

Subversion

Versionierte Elemente sind Dateien und Verzeichnisse. Sie bilden ein versioniertes Dateisystem. Für das Dateisystem wird eine VID in Form einer fortlaufenden Nummer verwaltet. Die Versionen einer Datei oder eines Verzeichnisses stehen in Vorgänger-Nachfolger-Beziehung, und sind der VID des Dateisystems zugeordnet, in der sie angelegt werden. Für ein Element wird also nur eine neue Version angelegt, wenn auch eine Änderung vorliegt. Die Richtung der Vorgänger-Nachfolger-Beziehungen folgt der VID des Dateisystems in aufsteigender Reihenfolge. Damit entspricht die Vorgänger-VID einer Version immer der nächstkleineren VID des Dateisystems, an dem eine Version des versionierten Elements angelegt wurde. Um Speicherplatz zu sparen wird der gleiche Mechanismus wie in CVS verwendet: Nur die Kopf-Revision wird vollständig gespeichert. Mit Hilfe von Kopien von Verzeichnissen werden auch Verzweigungsbeziehungen dargestellt: Um z.B. eine Alternative des ganzen versionierten Verzeichnisses zu erstellen, wird einfach das gesamte Verzeichnis kopiert. Später können Alternativversionen wieder verschmolzen werden, was mittels einer Verschmelzen-Beziehung ausgedrückt wird. Das Anlegen einer Verzweigung sowie das Verschmelzen, erhöhen somit auch die VID. Die Integration von Produkt- und Versionsraum erfolgt versionszentriert, da in Subversion die Version des Wurzelverzeichnisses die Versionen aller enthaltenen Elemente bestimmt [CSFP08].

Abbildung 4.10 zeigt ein Beispiel für ein Versionsmodell in Subversion. Grundlage bildet das Beispiel-Produktmodell aus Abbildung 4.3: Die Version mit VID 1 entspricht dem Beispielmodell mit 2 Verzeichnissen und 2 Dateien (die beiden „.class“-Dateien sind nicht enthalten). Wird nun die Datei „ClassA.java“ verändert, dann wird eine neue Version der Datei angelegt – unter der VID 2 – und auch von jedem übergeordneten Element. Diese referenzieren dann bereits existierende Elemente, so dass in der neuen Version „src“ auf die Datei „ClassB.java“ und „project“ auf das Verzeichnis „bin“ – jeweils in der Version mit VID 1 – verweisen.

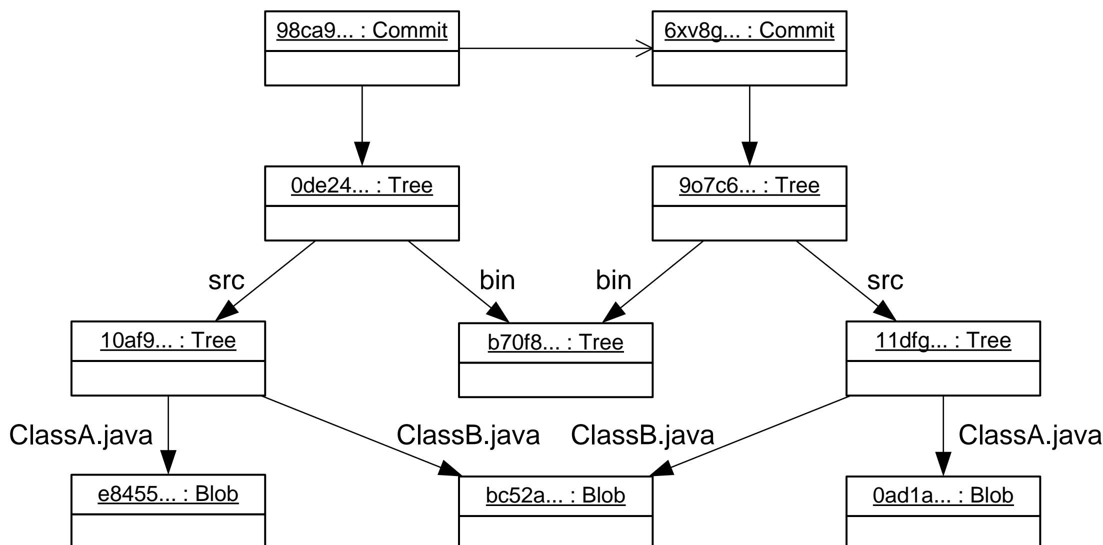


Abbildung 4.11.: Beispiel für Versionsmodell von GIT

GIT

Versionierte Elemente sind Dateiinhalte, Verzeichnisse und sog. „Commits“, die die gemeinsam übertragenen Änderung gruppieren. Ein „Commit“ umfasst immer alle Verzeichnisse und Dateiinhalte eines Projektes, wobei bereits existierende Dateiinhalte und Verzeichnisse referenziert werden, um Speicherplatz zu sparen. Zwischen den „Commits“ existieren Vorgänger-Nachfolger-, Alternativ- und Verschmelzungsbeziehungen. Für die „Commits“ wird auch eine VID in Form eines SHA-1-Hashwertes gespeichert. Die Integration von Produkt- und Versionsraum erfolgt versionszentriert, da in GIT die Version des Wurzelverzeichnisses die Versionen aller enthaltenen Elemente festlegt. Der Unterschied zu Subversion besteht in der Speicherung des Dateinamens: Während der Name in Subversion dem Dateisystem-Element selbst zugeordnet wird (um es zu identifizieren), ist er in GIT der Beziehung zwischen Verzeichnis und Datei zugeordnet (vgl. Abbildung 4.11) [Ca09].

Abbildung 4.11 zeigt ein Beispiel für ein Versionsmodell in GIT. Wie die vorherige Abbildung bildet das Beispiel-Produktmodell aus Abbildung 4.3 die Grundlage. Der erste „Commit“ referenziert das „Tree“-Element, das die Wurzel des Verzeichnisbaums ist. Darin enthalten sind zwei weitere Verzeichnisse die über den Namen „src“ und „bin“ referenziert werden. In „src“ befinden sich zwei „Blobs“, die mit dem Namen „ClassA.java“ und „ClassB.java“ referenziert werden. Jedes Element wird über seinen SHA-1-Hashwert identifiziert, was auf Grund der Länge des Wertes im Beispiel nur angedeutet wird. Verändert nun ein Entwickler den Inhalt der Datei „ClassA.java“, dann entsteht ein neuer Blob für den neuen Inhalt (rechter Blob) mit einem neuen Hashwert. Dadurch entsteht auch ein neues „Tree“-Objekt für das Verzeichnis „src“ und letztendlich das Wurzelelement, das

nun von einem neuen „Commit“ referenziert wird, der auch Nachfolger des 1. „Commit“ ist. Der „bin“-Tree und der „ClassB.java“-Blob haben sich jedoch nicht verändert, besitzen den gleichen Hashwert und werden somit von den neuen Elementen referenziert [Ca09].

Mercurial

Versionierte Elemente sind Dateien, Manifeste (engl. manifests) und Änderungsmengen (engl. change sets). Für jede Datei wird ein Dateilogbuch verwaltet, in dem die einzelnen Revisionen einer Datei unter einer VID eingetragen werden. Als VID dient in Mercurial – ähnlich wie in GIT – der Funktionswert einer auf den Eintrag angewendeten Hashfunktion. Verzeichnisse betrachtet Mercurial als Namenserweiterung des Dateinamens. Die Ordnerstruktur wird aus den erweiterten Dateinamen abgeleitet: Fehlende Order werden bei Bedarf erstellt bzw. leere Ordner gelöscht. Eine Änderung des erweiterten Dateinamens führt zu einem neuen versionierten Element, so dass Mercurial spezielle Kommandos zum Kopieren, Verschieben und Umbenennen besitzt. Ein Manifesteintrag hat ebenfalls eine VID und gruppiert die Revisionen zu einem vollständigen Zustand des Produktmodells zu einem festen Zeitpunkt. Solange eine Datei nicht verändert wird, referenziert jeder Manifesteintrag die gleiche Revision im entsprechenden Dateilogbuch. Letztendlich wird jeder Manifesteintrag von einer Änderungsmenge referenziert, die ebenfalls eine VID besitzt. Zwischen Revisionen, Manifesteinträgen und auch Änderungsmengen existieren jeweils Vorgänger-Nachfolger-Beziehungen. Alternativbeziehungen und Verschmelzungsbeziehungen werden aus den Vorgänger-Nachfolger-Beziehungen abgeleitet: Mehrere Nachfolger bedeutet Alternativen, zwei Vorgänger bedeutet Verschmelzen. Die Integration von Produkt- und Versionsraum erfolgt versionszentriert, da in Mercurial die Version der Änderungsmenge bzw. des Manifests die Versionen aller enthaltenen Dateien festlegt [O'S09].

Abbildung 4.12 zeigt ein Beispiel für ein Versionsmodell in Mercurial. Wie in den vorherigen beiden Abbildungen bildet das Beispiel-Produktmodell aus Abbildung 4.3 die Grundlage. Die linke Änderungsmenge im Änderungslogbuch referenziert den ersten Manifesteintrag, der jeweils auf eine Revision aus den Logbüchern von „ClassA.java“ und „ClassB.java“ verweist. Verändert ein Entwickler die Datei „ClassA.java“, so wird eine neue Revision in das entsprechende Dateilogbuch eingetragen, und zwar als Nachfolger der vorherigen Revision. Es wird auch ein neuer Manifesteintrag als Nachfolger des vorherigen angelegt, der auf die neue Revision von „ClassA.java“ und die unverändert gebliebene von „ClassB.java"verweist. Der Manifesteintrag wird von einer neuen Änderungsmenge referenziert, die ebenfalls als Nachfolger der bereits existierenden eingetragen wird.

4.3.6. Datenspeicherung

Während die Versionskennung eine Version eines Software-Elements identifiziert und die Versionshistorie die Elemente miteinander in Beziehung setzt, legt der Datenspeicher fest, in welcher Form die Software-Elemente gespeichert werden. Da die Anzahl der Versionen kontinuierlich ansteigt (die Historie wird niemals kleiner [CW98]), steigt auch der Spei-

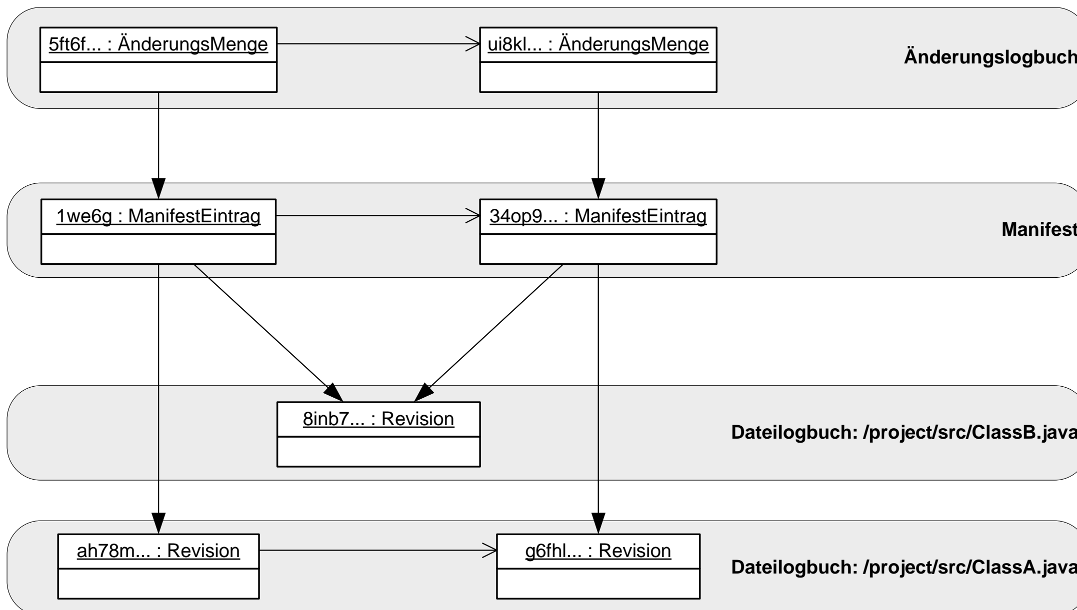


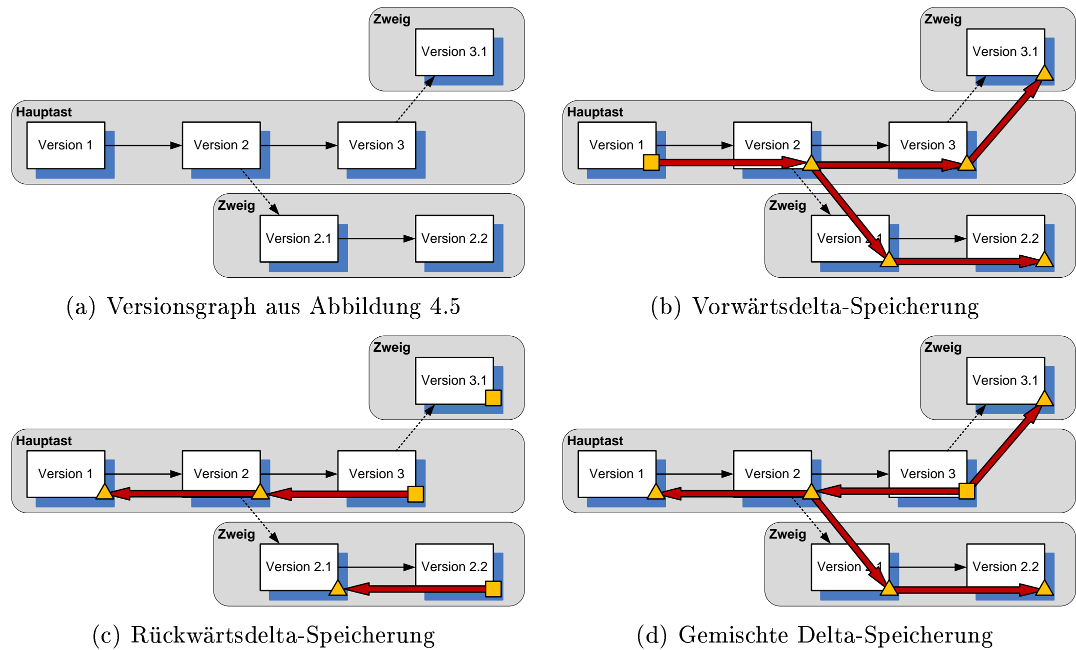
Abbildung 4.12.: Beispiel für Versionsmodell von Mercurial

cherplatzverbrauch monoton an. Daher existieren unterschiedliche Verfahren, um sein Wachstum zu begrenzen [Dar90, CW98, HVT98]. Von zentraler Bedeutung ist hier, ob der Zustand eines Software-Elements zu einem bestimmten Zeitpunkt (zustandsbasiert) oder die Änderungen ab einem bestimmten Zeitpunkt (änderungsbasiert) gespeichert werden [CW98].

Gerichtete Deltas

Bei der zustandsbasierten Versionierung wird der vollständige Zustand eines Software-Elements zu einem bestimmten Zeitpunkt gespeichert. Ohne weitere Mechanismen zur Reduktion des Speicherplatzbedarfs bedeutet dies, dass jede Version vollständig gespeichert wird. Mit Hilfe der Beziehungen zwischen den verschiedenen Versionen eines Software-Elements, versucht das Verfahren der Delta-Speicherung, den Platzbedarf zu reduzieren: Beim Vergleich zwischen zwei miteinander in Beziehung stehender Versionen v_1 und v_2 lässt sich häufig ein hoher Grad an Gemeinsamkeiten finden. Anstatt nun beide Versionen vollständig unabhängig voneinander zu speichern, wird nur v_1 vollständig gespeichert (d.h. als Grundfassung), während v_2 mit Hilfe des „gerichteten Deltas“ $\Delta_{v_1 \to v_2}$ (d.h. die Änderungen an v_1, um v_2 zu erhalten) abgelegt wird. Bei Bedarf wird v_2 durch „Anwenden“ des Deltas auf die Grundfassung v_1 wiederhergestellt (d.h. $v_2 = \Delta_{v_1 \to v_2}(v_1)$).

Delta-Speicherung führt zu einem erhöhten Rechenaufwand beim Speichern und Wiederherstellen von Versionen [Tic82, Tic85]. Wird die erste Version eines Software-Elements als Grundfassung abgespeichert und alle folgenden Versionen als Delta, wird dies auch als „Vorwärtsdelta-Speicherung“ bezeichnet, da die Anwendungsrichtung der Deltas auch der zeitlichen Entwicklung entspricht. Abbildung 4.13(b) zeigt diese anhand



(a) Versionsgraph aus Abbildung 4.5

(b) Vorwärtsdelta-Speicherung

(c) Rückwärtsdelta-Speicherung

(d) Gemischte Delta-Speicherung

Abbildung 4.13.: Verfahren zur Delta-Speicherung

der Beispielhistorie in Abbildung 4.13(a). Das orangenfarbene Viereck markiert eine als Grundfassung gespeicherte Version, ein orangenfarbenes Dreieck eine als Delta gespeicherte. Die roten Pfeile zeigen die Anwendungsrichtung der Deltas beim Wiederherstellen einer Version. Um im Beispiel „Version 3“ wiederherzustellen, wird auf die Grundfassung „Version 1“ zunächst das Delta auf „Version 2“ und darauf dann das Delta auf „Version 3“ angewendet. Das Beispiel zeigt bereits, dass dieses Verfahren zu einem monoton steigenden Rechenaufwand beim Zugriff auf die neuste Version führt.

Alternativ wird die neuste Version als Grundfassung und ältere Versionen als Deltas gespeichert. Da die Anwendungsrichtung der Deltas nun entgegengesetzt der zeitlichen Entwicklung läuft, wird dieses Verfahren auch als „Rückwärtsdelta-Speicherung“ bezeichnet (siehe Abbildung 4.13(c)) [Tic82]. Beim Einsatz von Verzweigungen führt dies dazu, dass jeder Zweig seine eigene Grundfassung besitzt (z.B. „Version 3.1“ und „Version 2.2“ in Abbildung 4.13(c)). Um genau eine Grundfassung zu speichern, wird das Verfahren der „Gemischten Delta-Speicherung“ eingesetzt (siehe Abbildung 4.13(d)). Die neuste Version des Hauptzweigs wird als (einzige) Grundfassung gespeichert. Evtl. auftretende Verzweigungen verwenden stattdessen Vorwärtsdeltas. Um nun eine Version auf einem Zweig wiederherzustellen, wird zunächst die entsprechende Version auf dem Hauptzweig mit Hilfe der Rückwärtsdeltas wiederhergestellt und anschließend die Vorwärtsdeltas auf dem Zweig angewendet [Tic82, Ced05].

Symmetrische Deltas

Gerichtete Deltas enthalten die Änderung in Bezug auf genau eine Grundfassung. Bei einem symmetrischen Delta werden die Inhalte aller Versionen überlappend gespeichert, indem der Inhalt – ähnlich der bedingten Kompilierung – markiert wird. Die Markierung legt fest, in welchen Versionen der entsprechende Teil des überlappenden Inhalts vorhanden ist. Da für MOD2-SKM kein Modul für symmetrische Deltas implementiert wurde, sei auf [CW98] für eine genauere Beschreibung verwiesen.

Änderungsbasierte Versionierung

In der Praxis der zustandsbasierten Versionierung sind gerichtete Deltas nur ein Mittel zum Sparen von Speicherplatz. Die Reihenfolge der Anwendungen ist über die Beziehungen der Versionshistorie fest vorgegeben [Tic82, Ced05]. Im Gegensatz dazu sind die Änderungen die primären Elemente in der änderungsbasierten Versionierung. Ihr Ziel ist es, die Änderungen – wenn möglich – voneinander unabhängig zu speichern. So kann nur ein Teil der Änderungen genutzt werden, um – ausgehend von einer Grundfassung – eine neue Version zu erzeugen. Eventuelle Abhängigkeiten unter den Änderungen werden, ähnlich wie in einer Versionshistorie, z.B. über Vorgänger-Nachfolger-Beziehungen verwaltet. Da für MOD2-SKM kein Modul für änderungsbasierte Versionierung implementiert wurde, sei auf [CW98] für eine genauere Beschreibung verwiesen.

4.4. Konzepte im Anforderungsbereich „Team“

Aufgabe ist die Koordination mehrerer Entwickler, um die gemeinsame Entwicklung eines Software-Anwendungssystems zu unterstützen [Dar90]. Das SKMS unterstützt die Entwickler, indem es den Zugriff und die Veränderung der Software-Elemente protokolliert und die Verbreitung geänderter Elemente über ein Repositorium steuert. Zentrales Konzept zu Vermeidung der hierbei auftretenden Probleme (vgl. Abschnitt 4.1.1, S. 70) ist hier der Arbeitsbereich (engl. workspace) eines Entwicklers: Um ein Element zu bearbeiten, muss jeder Entwickler eine Kopie des Elements aus dem Repositorium anfordern. Diese wird in seinem persönlichen Arbeitsbereich abgelegt und nur dort verändert. Nach Abschluss der Änderungen überspielt der Entwickler das veränderte Element wieder in das Repositorium. Die Überwachung und Ausführung dieser Kopiervorgänge erfolgt mittels eines SKMS. Hauptproblem ist vor allem das Gleichzeitige-Aktualisierungs-Problem [Bab86].

4.4.1. Repositorien

Das Konzept des Repositoriums ist essentiell für SKM. Es enthält die versionierte Objektbasis und bietet zentralisierten Zugriff auf deren versionierte Software-Elemente. Jedes Element, das Teil eines Repositoriums ist, gilt damit als durch SKM verwaltet. Die Elemente im Repositorium selbst sind unveränderlich, da sie einen Teil der Entwicklungsgeschichte dokumentieren. Das bedeutet, um ein Software-Element zu bearbeiten, muss

ein Entwickler eine Kopie aus dem Repositorium entnehmen (auschecken, engl. check out), verändern und als neue Version dem Repositorium hinzufügen (einchecken, engl. check in)[Dar90]. Solange nur ein Entwickler gleichzeitig die Elemente aus dem Repositorium bearbeitet, treten keine Nebenläufigkeiten auf. Sobald mehrere Entwickler mit den Elementen arbeiten, kann jeder Entwickler ein Element auschecken. Wenn nun mehrere Entwickler ein Element parallel bearbeiten, kommt es zum Gleichzeitige-Aktualisierungs-Problem (vgl. Abschnitt 4.1.1): Es existieren gleichzeitig zwei Versionen, die nun beide ins Repositorium eingecheckt werden sollen [CSFP08].

Pessimistische und Optimistische Sperren

Es existieren verschiedene Möglichkeiten, dieses Problem zu lösen. Im Falle eines unversionierten Elements überschreibt die zweite Änderung die erste. Befindet sich das Element in einem Repositorium unter Versionskontrolle, so ließen sich beide Elemente als Varianten auffassen [Tic82]. Beide Verfahren bieten jedoch keine zufriedenstellende Lösung für nebenläufige Änderungen, da kein Element entsteht, das beide Änderungen enthält. Zu diesem Zweck werden Sperren eingesetzt [Dar90, Pop09]. Pessimistische Sperren verbieten die Bearbeitung eines Elements, solange ein Entwickler ein Element aus dem Repositorium ausgecheckt hat. Alternativ lassen sich die Änderungen auf einen Seitenzweig einspielen und nach der Freigabe verschmelzen. Optimistische Sperren erlauben die Veränderung von bereits ausgecheckten Elementen. Sie erkennen jedoch beim einchecken eines geänderten Elements, ob im Repositorium das Element in unveränderter Form vorliegt oder in der Zwischenzeit verändert wurde [Dar90, Pop09]. Im ersten Fall wird die Änderung eingecheckt, im zweiten Fall wird der Vorgang abgebrochen und der Entwickler über einen Konflikt benachrichtigt. Der Entwickler muss nun die geänderte Fassung aus dem Repositorium auschecken und mit seinen Änderungen kombinieren. Dabei wird er, i.d.R. durch semi-automatisierte Verfahren zum Verschmelzen und entsprechende Software-Werkzeuge, unterstützt [CSFP08, Pop09]. In der Literatur wird der Einsatz von Sperren manchmal nach seinen Arbeitsschritten benannt: Sperren-Verändern-Entsperren für pessimistische und Kopieren-Verändern-Verschmelzen für optimistische Sperren [CSFP08, Pop09].

Replikation und Verteilte Repositorien

Bisher wurde angenommen, dass nur ein einziges Repositorium von allen Entwicklern verwendet wird. Es ist jedoch möglich, die Software-Elemente auf mehrere Repositorien zu verteilen. Dieses hat meist organisatorische Gründe, z.B. bei der Kooperation von Entwicklerteams unterschiedlicher Firmen, wobei jede Firma ihr eigenes Repositorium besitzt [Jäg03]. Weitere Gründe sind z.B. die Belastbarkeit oder Erreichbarkeit von Repositorien. Zu diesem Zweck werden Verfahren zur Repositorien-Replikation bzw. verteilte Repositorien eingesetzt.

Die Repositorien-Replikation hat das Ziel, mehrere Repositorien zu synchronisieren. Ziel ist es, Änderungen aus einem Repositorium in die anderen Repositorien zu übertragen, so dass alle Repositorien als exakte Kopie der anderen gelten [Hoe00]. So wird

die Erreichbarkeit und Belastbarkeit des Repositoriums erhöht. Es kann dabei jedoch zu nebenläufigen Änderungen in zwei oder mehr Repositorien kommen. In diesem Fall wird die Annahme getroffen, dass es sich hierbei nur um vollständig unabhängige Änderungen handelt und keine Konflikte auftreten, was z.B. durch organisatorische Maßnahmen sichergestellt werden muss [Hoe00]. Ein Spezialfall der Replikation ist der Einsatz eines Master-Repositoriums, dessen Änderungen an mehrere Nur-Lesen-Repositorien verteilt werden [CSFP08]. Da in MOD2-SKM bisher kein Replikations-Modul existiert, sei für eine detaillierte Beschreibung auf [Hoe00, CSFP08] verwiesen.

Bei verteilten Repositorien existieren zwei Varianten: Verborgen oder sichtbar verteilte Repositorien. Die verborgen verteilten Repositorien präsentieren sich dem Entwickler als ein (logisches) Repositorium, während intern mehrere (physikalische) Repositorien existieren. Damit müssen sie die Anforderung erfüllen, dass ein Protokoll für die notwendige Koordination und den Datenaustausch im Hintergrund existiert [Hoe00]. Bei sichtbar verteilten Repositorien erkennt der Entwickler die unterschiedlichen Repositorien und kann diese explizit miteinander synchronisieren. Konflikte bei nebenläufigen Änderungen werden analog zu den optimistischen Sperren behandelt: Sie werden semiautomatisch vom synchronisierenden Entwickler aufgelöst. Das Verschmelzen wird i.d.R. im Arbeitsbereich des Entwicklers durchgeführt, der anschließend die Änderungen in beide Repositorien überträgt. Analog zum Ein-/Auschecken wird dieses Verfahren auch als „Schiebe/Ziehe“-Verfahren (engl. push/pull) bezeichnet [Ca09]. Da in MOD2-SKM bisher kein Modul für verteilte Repositorien existiert, sei für eine detaillierte Beschreibung auf [Hoe00, Ca09, O'S09] verwiesen.

4.4.2. Arbeitsbereiche

Das Konzept des Arbeitsbereichs ist eng mit der Idee des Repositoriums (vgl. Abschnitt 4.4.1) verknüpft. Um ein Element im Repositorium zu verändern, muss ein Entwickler es zunächst auschecken, dann verändern und schließlich wieder einchecken. Diese Änderung nimmt er in seinem Arbeitsbereich vor, der mit dem Repositorium verbunden ist, so dass die Interaktion mit dem Repositorium (z.B. das ein- und auschecken) teilweise automatisiert werden kann. Zu diesem Zweck verwaltet der Arbeitsbereich auch Statusinformationen über die in ihm vorhandenen Software-Elemente [Dar90]. Mögliche Aktionen sind [Tic82, Ced05, CSFP08, Ca09, O'S09]:

- das Sperren eines Elements im Repositorium,
- das automatische Aufheben von Sperren beim Einchecken der Änderungen,
- das Verschmelzen bei Konflikten,
- das Rückgängigmachen von Änderungen,
- das Hinzufügen neuer Elemente in das Repositorium,
- das Ein-/Auschecken von Mengen bzw. Hierarchien von Elementen,
- der Vergleich von Elementen mit früheren Versionen,

- das Ersetzen von Elementen durch frühere Versionen.

Der Arbeitsbereich isoliert den Entwickler von den Änderungen im Repositorium und stellt gleichzeitig auch die Schnittstelle zu Softwareentwicklungs-Werkzeugen dar, die ohne SKM-Konzepte arbeiten [Dar90, CW98]. Der Arbeitsbereich verwaltet zwar Statusinformationen aus dem Repositorium, doch die in ihm enthaltenen Software-Elemente werden unversioniert präsentiert, d.h. sie können wie die unversionierten Elemente bearbeitet werden. Meist existiert jedoch ein Mechanismus, der die Elemente überwacht und ihre Veränderung registriert [CSFP08, Pop09].

4.5. Zusammenfassung

Das Software-Konfigurations-Management (SKM) transferiert die aus der Hardware-Entwicklung stammende Disziplin des Konfigurations-Management in die Softwareentwicklung [BHS78]. Im Zentrum stehen die Software-Elemente, die meist von mehreren Entwicklern bearbeitet werden und aus denen letztendlich ein vollständiges Software-Anwendungssystem konfiguriert wird. Zentrale Aufgabengebiete sind damit sowohl die Verwaltung der Software-Elemente als auch die Koordination der Entwickler und die Unterstützung des Entwicklungsprozesses. Dabei setzen die Entwickler ein SKM-System ein, das die Verwaltung der Elemente und typische Arbeitsabläufe teilweise automatisiert [Dar90]. In den Aufgabengebieten existieren viele unterschiedliche Verfahren und Konzepte, die meist in proprietären SKMS implementiert wurden [CW98]. Ein Großteil dieser Konzepte wurde in diesem Kapitel vorgestellt. Ziel dieser Arbeit ist es, eine Auswahl dieser Konzepte und Verfahren als Module zu implementieren und die Abhängigkeiten untereinander zu untersuchen. Die Verfahren dienen dabei auch als Grundlage für die technischen Merkmale in der folgende Merkmalsanalyse der SKM-Domäne.

5. MODPL in der SKM-Domäne

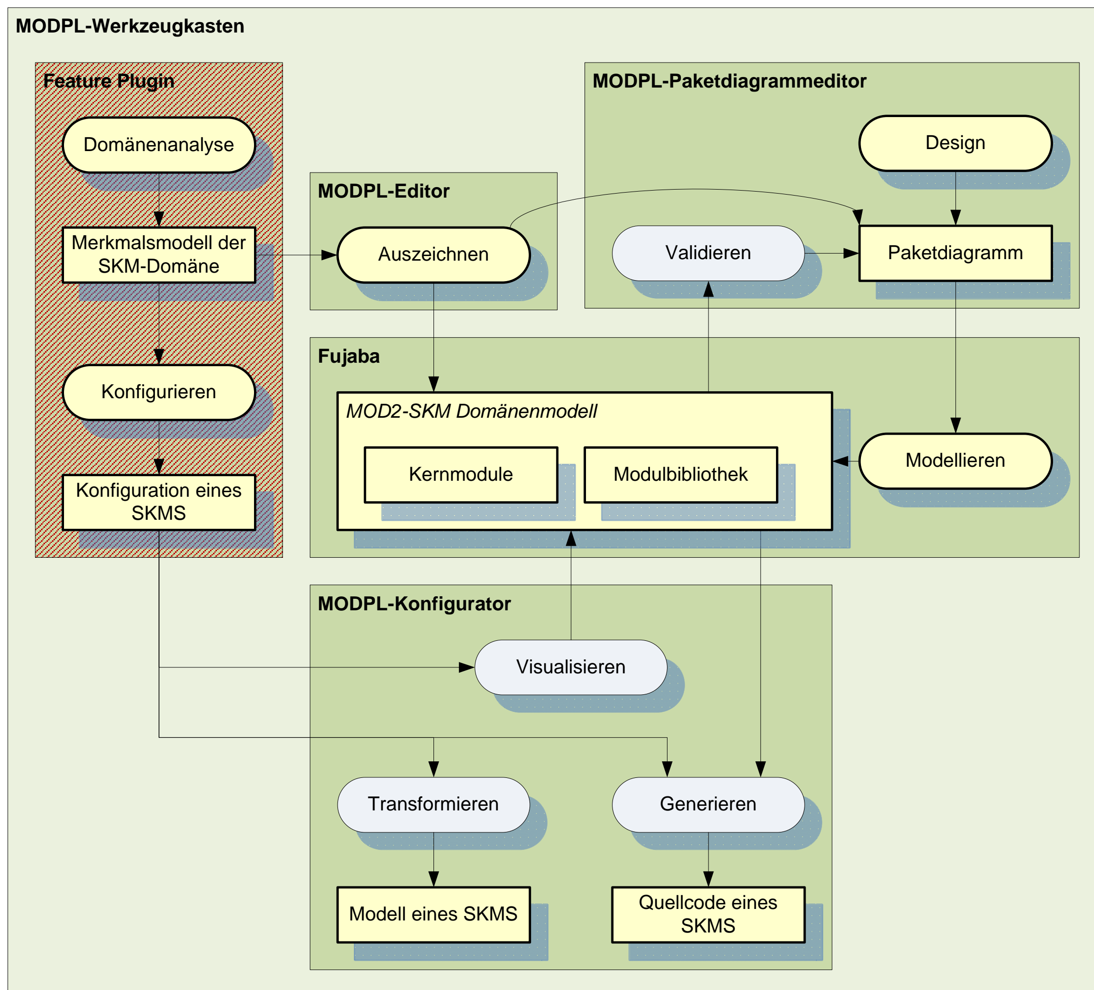


Abbildung 5.1.: Überblick über die MOD2-SKM Produktlinie

Entwurf und Realisierung von MOD2-SKM basieren auf meiner Domänenanalyse der SKM-Domäne. Das folgende Kapitel dokumentiert, präsentiert und diskutiert das Ergebnis der Analyse: das MOD2-SKM-Merkmalsmodell. Zu seiner Erstellung (und der seiner Konfigurationen) wird das Feature-Plugin [AC04] nach der FORM-Methode [KKL+98] eingesetzt. Die dazu verwendeten Werkzeuge und Elemente im MODPL-Werkzeugkasten sind in Abbildung 5.1 schraffiert. Anhang B enthält die vollständige Analyse (S. 359ff.).

5.1. Ablauf der MOD2-SKM Domänenanalyse

Der Ablauf der Merkmals-Analyse für MOD2-SKM ist wie folgt:

1. Eignungsfeststellung
2. Auswahl der Quellen
3. Identifikation von Merkmalen
4. Integration der Merkmale
5. Aufbau eines Merkmals-Modells
6. Validierung des Merkmals-Modells

5.1.1. Generelle Eignung von SKM für MODPL

Eine erfolgreiche Produktlinie benötigt sowohl Gemeinsamkeiten als auch Unterschiede. Die gemeinsamen Merkmale bilden die Grundlage, auf der die variablen Merkmale aufbauen [PBL05]. Ziel ist es, so viele Gemeinsamkeiten wie möglich auszunutzen und den Grad der Variabilität auf das notwendige Minimum zu beschränken [WL99]. Eine Domäne ist für die Entwicklung einer Produktlinie geeignet, wenn bereits einzeln entwickelte Produkte mit hohem Reifegrad existieren. Insbesondere beim Anlegen einer neuen Produktlinie sollten viele Gemeinsamkeiten ausgenutzt und Variationspunkte schrittweise hinzugefügt werden [KKL+98]. Eine vereinfachte Gemeinsamkeiten- Unterschiede-Analyse einer neuen Produktlinie sollte also (1) viele Gemeinsamkeiten und (2) wenige, leicht identifizierbare Variationspunkte aufweisen [KKL+98, PBL05].

Tabelle 5.1 zeigt eine einfache Gemeinsamkeiten–Unterschiede-Analyse nach [PBL05]. Verglichen wurden die vier SKMS „Concurrent Versioning System (CVS)“, „Subversion (SVN)“, „GIT“ und „Mercurial“, die bereits in Kapitel 4 beschrieben wurden. In den vier folgenden Spalten ist verzeichnet, ob das jeweilige SKMS diese Anforderung erfüllt („X“) oder nicht („-“). Weiße Zeilen enthalten von allen SKMS erfüllt Anforderungen, d.h. Gemeinsamkeiten. Farbige Zeilen enthalten Anforderungen, die nicht von allen vier SKMS erfüllt werden. Sie stellen Unterschiede zwischen den SKMS dar und lassen sich zu Variationspunkten gruppieren [PBL05]. Benachbarte Unterschiede in gleichem Farbton (gelb bzw. rot) bilden Varianten bzgl. eines Variationspunkts.

Diese stichprobenartige Analyse zeigt, dass die vier ausgewählten Systeme 8 von 18 Anforderungen gemeinsam haben. Die restlichen 10 Anforderungen verteilen sich auf 7 Variationspunkte. Das deutet auf ausreichend Gemeinsamkeiten und Unterschiede für den Aufbau einer Produktlinie hin. In [CW98, Ehr06, Fis09] wird eine größere Anzahl an SKMS systematisch verglichen, und auch hier lassen sich Gemeinsamkeiten und Variationspunkte identifizieren. Dies bestärkt die Hypothese, dass die SKM-Domäne für eine Produktlinie geeignet sei. Letztendlich ist es ja ein Ziel dieser Arbeit, diese Hypothese zu bestätigen oder zu widerlegen (vgl. Abschnitt 1.7, S. 26).

Anforderungen	CVS	SVN	GIT	Mercurial
Das SKMS ist ein SKM-Werkzeug.	X	X	X	X
Das Produktmodell verwendet eine echte OID.	-	X	X	X
Das Produktmodell ist Dateisystem-basiert.	X	X	X	X
Das Versionsmodell versioniert nur Dateien.	X	-	-	X
Das Versionsmodell versioniert Dateien und Verzeichnisse.	-	X	X	-
Das Versionsmodell ist zustandsbasiert.	X	X	X	X
Das Versionsmodell unterstützt Vorgänger-/Nachfolger-Beziehungen	X	X	X	X
Das Versionsmodell unterstützt Alternativ-Beziehungen	X	X	X	X
Das Versionsmodell unterstützt Verschmelzen-Beziehungen	-	-	X	X
Beide Modelle sind produktzentriert integriert.	X	-	-	-
Beide Modelle sind versionszentriert integriert.	-	X	X	X
Der Datenspeicher verwendet Rückwärts-Deltas, um Speicherplatz zu sparen.	X	X	-	X
Der Datenspeicher verwendet Referenzen auf gleiche Inhalte, um Speicherplatz zu sparen.	-	-	X	-
Das SKMS synchronisiert ein zentrales Repositorium und Arbeitsbereiche per „einchecken/auschecken“.	X	X	X	X
Das SKMS unterstützt die Replikation des Repositoriums.	-	X	X	X
Das SKMS unterstützt die Synchronisation verteilter Repositories per „schieben/ziehen“.	-	-	X	X
Das SKMS unterstützt pessimistische Sperren.	X	X	X	X
Das SKMS unterstützt optimistische Sperren.	X	X	X	X

Tabelle 5.1.: Gemeinsamkeit-Unterschiede-Analyse von vier SKM-Werkzeugen

5.1.2. Gliederung des Merkmals-Modells nach FORM

Auf der obersten Ebene ist das Merkmals-Modell nach der FORM-Methode gegliedert, d.h. es existieren die folgenden vier Ebenen [KKL+98]:

1. Befähigungs-Ebene (engl. capability layer)
2. Betriebsumgebungs-Ebene (engl. operating environment layer)
3. Domänentechnologie-Ebene (engl. domain technology layer)
4. Implementierungstechniken-Ebene (engl. implementation technique layer)

Die Merkmale der **Befähigungs-Ebene** beschreiben die Produktlinie mit Hilfe von Diensten, Operationen und Qualitäten, welche die Produktlinie anbieten soll. Sie ist von allen vier Ebenen am weitesten von der technischen Umsetzung entfernt und beschreibt die Produktlinie aus der Perspektive eines SKMS-Anwenders. Die Merkmale auf der **Betriebsumgebungs-Ebene** beschreiben die Produktlinie mit Hilfe von Eigenschaften der Hard- und Software-Umgebung, in denen die konkrete Anwendung eingesetzt wird. Typische Merkmale beziehen sich auf die Hardware, das Betriebssystem, Datenspeicher oder auch das Netzwerk. Diese Sicht lässt sich mit der eines System-Administrators vergleichen, der die Anwendung in Betrieb nehmen möchte. Die Merkmale auf der **Domänentechnologie-Ebene** beschreiben die Produktlinie mit Hilfe von Techniken, die speziell in der SKM-Domäne verwendet werden. Typische Merkmale behandeln die Realisierung von Elementen aus dem Produktraum und Versionsraum [KKL+98]. Diese Sicht lässt sich mit der eines Software-Entwicklers vergleichen, der ein SKM-Werkzeug für einen Entwicklungsprozess bzw. ein Software-Projekt entwerfen und implementieren soll. Die Merkmale auf der **Implementierungstechnik-Ebene** beschreiben die Produktlinie mit Hilfe der verwendeten grundlegenden Techniken. Diese sind allerdings nicht domänenspezifisch, sondern grundlegende Techniken der Datenübertragung und -verarbeitung. Diese Sicht lässt sich mit der eines Software-Entwicklers vergleichen, der die grundlegenden Datenstrukturen und Protokolle zur Datenübertragung festlegen soll.

5.1.3. Auswahl der Quellen

In FORM werden unterschiedliche Arten von Quellen für die Merkmals-Analyse genannt – je nachdem welcher Ebene die Merkmale zugeordnet werden. Für die Befähigungs-Ebene werden Handbücher empfohlen, für Betriebsumgebung und Domänentechnologie Dokumente aus der Anforderungs- und Design-Phase und für die Implementierungstechniken Design-Dokumente und Quellcode ([KKL+98], S. 10).

Für die Merkmals-Identifikation der Befähigungs-Ebene wurden die Handbücher der folgenden SKMS verwendet: CVS [Ced05], Subversion [CSFP08], GIT [Ca09] und Mercurial [O'S09]. Diese vier SKM-Systeme dienen auch in Schritt 5 zur abschließenden Validierung des Merkmals-Modells (siehe Abschnitt 5.4).

Für die Merkmals-Identifikation auf den Ebenen Betriebsumgebung und Domänentechnologie waren keine Dokumente aus den Anforderungs- bzw. Design-Phasen der vier SKM-Systeme verfügbar. Stattdessen wurden mehrere Forschungsarbeiten aus der SKM-Domäne ausgewählt. Kriterien für die Auswahl einer Quelle waren, dass

- die SKM-Konzepte möglichst **systemunabhängig** beschrieben werden: Dafür sind in der SKM-Domäne bekannte Übersichtsartikel bzw. Systemvergleiche gut geeignet.

- die SKM-Konzepte möglichst **fundamental** für die SKM-Domäne sind: Ein Indiz dafür ist die Anzahl der Zitate aus diesen Artikeln.

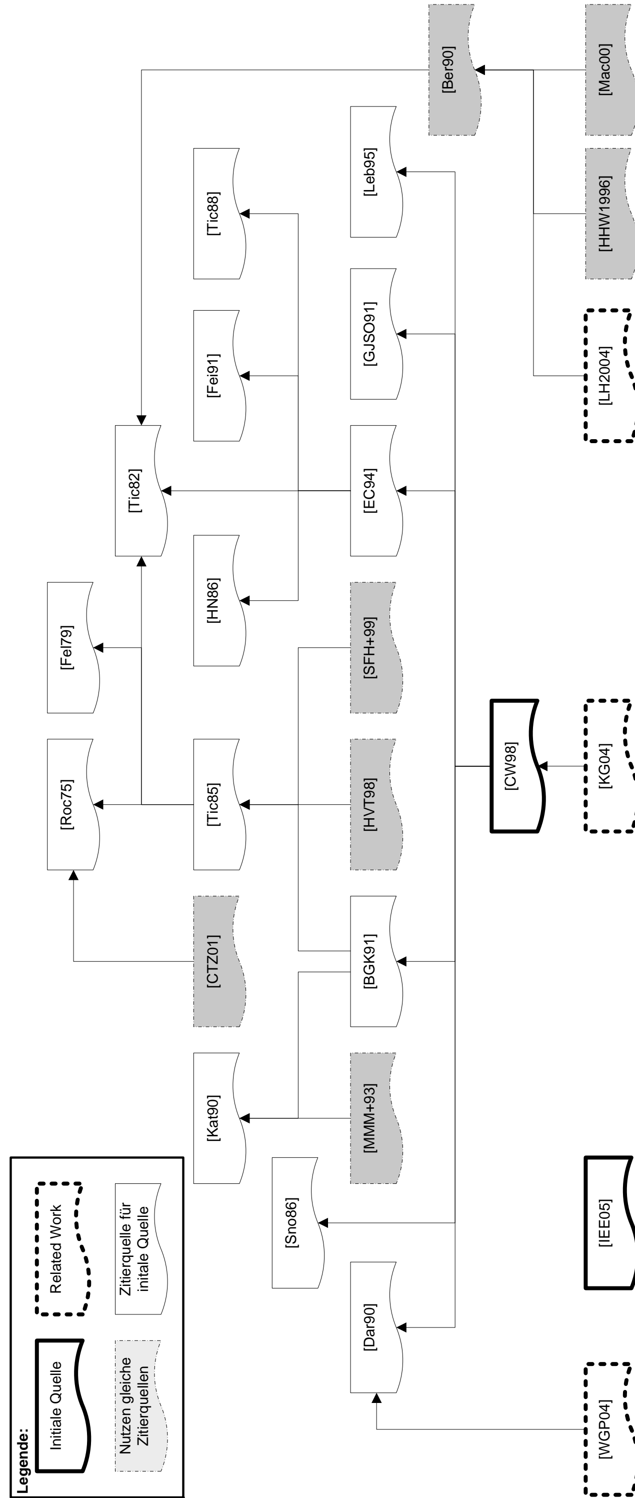


Abbildung 5.2.: Zitierbeziehung zwischen den ausgewählten Quellen

Abbildung 5.2 zeigt die ausgewählten Quellen und ihre Abhängigkeiten, basierend auf den Quellenangaben in ihren Literaturverzeichnissen: Als initiale Quelle wurde [CW98] ausgewählt. Ausgehend von diesem Übersichtsartikel wurden nun weitere Quellen einbezogen, die (a) von [CW98] zitiert werden, (b) von mehr als 50 anderen Quellen zitiert werden und (c) aus der SKM-Domäne stammen: [Roc75], [Fel79], [Tic82], [Tic85], [HN86], [Sno86], [Tic88], [Dar90], [Kat90], [BGK91], [Fei91], [GJSO91], [EC94] und [Leb95]. Um das Diagramm übersichtlich zu halten, sind die Abhängigkeiten in Abbildung 5.2 transitiv zu lesen.

Im nächsten Schritt wurden Quellen ausgewählt, die (a) mindestens einen der bisherigen Artikel zitierten und (b) von mehr als 50 anderen Quellen zitiert werden und (c) aus der SKM-Domäne stammen: [Ber90], [MMM+93], [HHW96], [HVT98], [SFH+99], [Mac00] und [CTZ01].

Zuletzt wurden bekannte *Themenverwandte Arbeiten* ([KG04], [LH04], [WGP04]), industrielle Vergleiche von SKM-Systemen ([Ehr06], [Fis09]) und Standards ([IEE05]) untersucht und (falls möglich) mit den bereits bestehenden Quellen in Verbindung gebracht. Dieser letzte Schritt war nötig, da keine der themenverwandten Veröffentlichungen bzw. der industriellen Vergleiche das Relevanzkriterium von 50 oder mehr Zitaten erreichte.

5.1.4. Identifikation und Integration von Merkmalen

Als Nächstes wurden die Merkmale in den ausgewählten Quellen identifiziert. Dazu wurde die Terminologie der Quellen analysiert und zentrale Begriffe ausgewählt, da "in einer ausgereiften und beständigen Domäne die Begriffe der Domänen-Fachsprache ein effizientes Mittel zur Identifikation der Merkmale bieten" [KKL+98].

In der SKM-Domäne existieren Bestrebungen, ein universelles Versionsmodell zu etablieren ([Zel95], [CW97]). Da jedoch ein Großteil der Quellen diesen Bestrebungen vorausgehen, bzw. die aktuellen Quellen diese Idee nicht aufgreifen, war es notwendig, unterschiedliche Begriffe zu integrieren. Ansonsten hätte es mehrere synonyme Merkmale gegeben. Unter der jeweiligen Beschreibung eines Merkmals werden ggf. die synonymen Begriffe aufgeführt.

5.2. Aufbau des MOD2-SKM Merkmals-Modell

Die vollständige Merkmals-Analyse im Anhang der Arbeit umfasst ca. 90 Seiten (vgl. Anhang B, S. 359ff.). Auf Grund des Umfangs sowie der Komplexität der Analyse erfolgt ihre Beschreibung in mehreren Schritten: Dieser Abschnitt beschreibt die Gliederung des Merkmals-Modells und die Notation der Merkmals-Analyse. Abschnitt 5.3 konzentriert sich anschließend auf die Beschreibung der Spitzenmerkmale (engl. top-level features) und hebt deren wichtigsten Untermerkmale hervor. Dies bietet einen Einstieg und Überblick in die Merkmals-Analyse. Ergebnis der Analyse ist ebenfalls das Merkmalsmodell in Anhang C, das anhand der vier SKMS – *CVS*, *GIT*, *Mercurial* und *Subversion* – in Abschnitt 5.4 validiert wird. Abschließend präsentiert Abschnitt 5.5 den Stand der Umsetzung des Merkmalsmodells, denn auf Grund ihres Umfangs ließen sich nicht alle Merkmale realisieren.

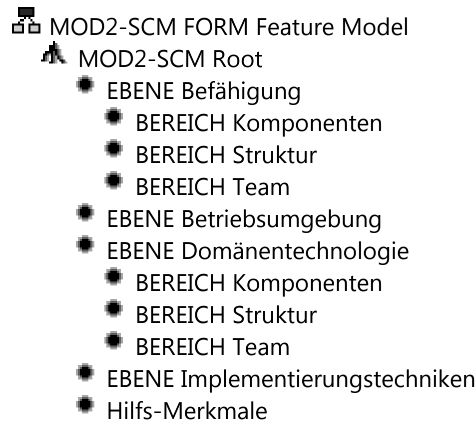


Abbildung 5.3.: Überblick über die FORM-Ebenen

5.2.1. Erstellung des MOD2-SKM Merkmalsmodells

Das Merkmals-Modell der SKM-Domäne ist mit Hilfe des Feature plugins aus dem MODPL-Rahmenwerk erstellt [AC04, Buc10]. Die Notation des Merkmalsmodells ist in Abschnitt 3.2 (s. S. 54) beschrieben. Es besteht aus insgesamt 147 Merkmalen und 36 Einschränkungen (als aussagenlogische Formeln). Das Merkmals-Modell ist zur Übersichtlichkeit nach den Ebenen der FORM-Methode [KKL+98] und den SKM-Anforderungsbereichen (vgl. Abschnitt 4.1.2 (s. S. 71) [Dar90]) gegliedert.

Auf Grund der Anzahl der Merkmale wurde von einer vollständigen Modellierung des gesamten Merkmalsraums aus folgenden Gründen Abstand genommen: (1) die Erfolgswahrscheinlichkeit einer erfolgreichen Modellierung sinkt bei großen Merkmalsräumen [KKL+98] und (2) liegt der Fokus dieser Arbeit auf dem gesamten MODPL-Prozess, so dass die Realisierung aller Merkmale den Rahmen dieser Arbeit sprengt. Daher sind in der Analyse einige Merkmale als „Erweiterungspunkt“ gekennzeichnet. Ein solches Merkmal kann entweder in zukünftigen Arbeiten analysiert und erweitert werden, oder als Wurzelknoten für ein existierendes Merkmalsmodell dienen, das diesen speziellen Bereich beschreibt.

Auf der obersten Ebene ist das Merkmals-Modell nach der FORM-Methode gegliedert. Das „Feature plugin“ ist jedoch nur auf Merkmals-Modelle der FODA-Methode ausgelegt [KCH+90, AC04], so dass der Editor die vier Ebenen nicht speziell unterstützt. Sie sind daher als vier verpflichtende Merkmale modelliert und durch den Bezeichner „EBENE“ kenntlich gemacht. Abbildung 5.3 zeigt die vier FORM-Ebenen als verpflichtende Merkmale. Auf Grund der fehlenden FORM-Unterstützung existieren ebenfalls keine Werkzeuge um die „Implementierung“-Beziehungen aus FORM zu modellieren. Mit ihrer Hilfe lässt sich ausdrücken, dass ein Merkmal einer niedrigeren (anforderungsorientierteren) Ebene durch ein Merkmal einer höheren (technischeren) Ebene implementiert wird (vgl. [KKL+98], S. 14). Mit Hilfe der aussagenlogischen Einschränkungen lässt sich dieser Beziehungstyp jedoch im „Feature plugin“ nachbilden.

Integration der Anforderungsbereiche

In Kapitel 4 sind die Konzepte und Methoden der SKM-Domäne entsprechend den drei Anforderungsbereichen aus Abbildung 4.2 (s. S. 74) gegliedert [Dar90]: Komponenten, Struktur und Team. Diese drei Bereiche bilden daher die zweite Gliederungsebene der Merkmals-Analyse. Als Hilfsmittel dienen ebenfalls verpflichtende Merkmale – analog zu den FORM-Ebenen – die mit dem Bezeichner „BEREICH“ hervorgehoben werden. Abbildung 5.3 zeigt ebenfalls die Anforderungsbereiche als verpflichtende Merkmale.

Allerdings sind lediglich die Merkmale der Befähigungs- und Domänentechnologie-Ebene in diese Bereiche gegliedert. In der Betriebsumgebungs- und Implementierungstechniken-Ebene ließen sich die Merkmale nicht eindeutig den einzelnen Anforderungsbereichen zuordnen. Der Grund ist in der Definition der FORM-Ebenen zu suchen: Die Befähigungs- und Domänentechnologie-Ebene erfassen die Funktionalität von SKMS bzw. die Konzepte der SKM-Domäne, d.h. es handelt sich vor allem um funktionale Anforderungen. Die Betriebsumgebungs- und Implementierungstechniken-Ebene erfassen betriebssystem-spezifische Anforderungen und domänenunabhängige Details der Implementierung, d.h. es handelt sich vor allem um technische Anforderungen. Da sich die Anforderungsbereiche auf funktionale Anforderungen beziehen, lassen sich auch nur die beiden Ebenen mit funktionalen Anforderungen entsprechend gliedern.

Beschreibung der aussagenlogischen Einschränkungen

Zusätzlich zu den strukturellen Bedingungen durch Merkmalsgruppierungen, existieren im Merkmalsmodell noch aussagenlogische Einschränkungen, die bei der Konfiguration ausgewertet werden. Zehn dieser Einschränkungen sind dabei lediglich für die korrekte Auswahl der Hilfsmerkmale nötig. Jede aussagenlogische Einschränkung ist als logische Implikation $A \rightarrow B$ formuliert. Wird während der Konfiguration das Merkmal A ausgewählt, führt dies zur Selektion der Merkmale unter B. Bei Abwahl eines Merkmals aus B wird dagegen auch das Merkmal A abgewählt. Somit ist die Implikation immer wahr. Merkmale unter B, die negiert werden (in der Beschreibung steht in diesem Fall ein „kein“ voran) verhalten sich bezüglich An-/Abwahl entgegengesetzt: Sie werden durch Auswahl von A selektiert und ihre Auswahl deselektiert A.

Hilfsmerkmale

Außer den Ebenen-Merkmalen in Abbildung 5.3 existiert noch die Gruppe der „Hilfsmerkmale“. Hierbei handelt es sich nicht um Merkmale der Produktlinie selbst, sondern um weitere Hilfskonstruktionen zur Negation existierender Merkmale: Da die Merkmalsmarkierungen in MODPL lediglich aus der Konjunktion selektierter Merkmale bestehen [Buc10], ist es nicht möglich, ein Modellelement an ein deselektiertes Merkmal zu binden. Stattdessen wird das Modellelement mit einem der Hilfsmerkmale gekennzeichnet, das genau dann selektiert wird, wenn das relevante Merkmale deselektiert ist. Die Verwendung der Hilfsmerkmale wird in Abschnitt 5.6.1 (s. S. 134) beschrieben.

Klassifikation vs. Beschreibung

Ziel der MOD2-SKM ist zum einen die Beschreibung der MOD2-SKM Produktlinie und zum anderen die Erstellung eines Schemas zur Klassifikation von SKMS im Allgemeinen. Diese beide Ziele lassen sich bei der Beschreibung der Merkmale in der Regel gut miteinander vereinbaren. Bei einigen Merkmalen hätte die explizite Auflistung aller Untermerkmale (zwecks Klassifikation) zu einer langen Liste von Merkmalen geführt, von denen nur ein einziges realisiert wurde (zwecks Beschreibung von MOD2-SKM), z.B. beim Spitzenmerkmal „Betriebssystem“ (s. S. 111): Während MOD2-SKM wegen des hohen Aufwands nur für ein einziges Betriebssystem modelliert wurde, benötigt eine Klassifikation eine umfangreiche Liste aller Betriebssysteme und -umgebungen. Aus Gründen der Lesbarkeit habe ich auf diese Auflistung verzichtet.

5.2.2. Beschreibung der Merkmale

Alle weiteren Merkmale sind gemäß FORM- bzw. FODA erfasst und anschließend in die bisher beschriebene Gliederung integriert. Zu jedem Merkmal existiert eine einheitliche und systematische Beschreibung ihrer Quellen und Beziehungen zu den anderen Merkmalen. Die vollständig Beschreibung aller Merkmale finden Sie in Anhang B. Die Beschreibung eines Merkmals ist wie folgt aufgebaut:

1. **Kennung:** Jedes Merkmal besitzt einen Namen und eine Kennung. Diese setzt sich aus dem Anfangsbuchstaben seiner Ebene und einer hierarchischen laufenden Nummer zusammen. Ist ein Merkmal ein Untermerkmal, so besteht seine Kennung aus der des Elternmerkmals, die um eine zusätzliche Hierarchieebene erweitert wurde.

2. **Beschreibung:** Beschreibt die Bedeutung und den Zweck des Merkmals genauer.

3. **Umsetzung:** Gibt an, inwieweit das Merkmal bereits in den Modulen der MOD2-SKM modelliert und implementiert wurde (vgl. Abschnitt 5.5).

4. **Ebene:** Nennt die Ebene und – falls vorhanden – den Anforderungsbereich, dem das Merkmal zugeordnet ist.

5. **Kinder:** Listet alle Untermerkmale des beschriebenen Merkmals auf.

6. **Setzt voraus:** Listet die Bedingungen für die Auswahl dieses Merkmals auf. Wird das beschriebene Merkmal während der Konfiguration selektiert, stellen die Beziehungen im Merkmalsmodell sicher, dass die aufgelisteten Merkmale ebenfalls selektiert werden. Die Beziehungen werden entweder durch Gruppierung der Merkmale oder aussagenlogische Einschränkungen beschrieben (vgl. Abschnitt 5.2.1, S. 102f.).

7. **Benötigt von:** Listet die Merkmale auf, die das Merkmal benötigen, d.h. wenn das beschriebene Merkmal deselektiert wird, werden auch alle aufgelisteten Merkmale deselektiert. Die Beziehungen werden entweder durch Gruppierung der Merkmale oder aussagenlogische Einschränkungen beschrieben (vgl. Abschnitt 5.2.1, S. 102f.).

8. **Quellen:** Führt die Quellen an, aus denen das Merkmal entnommen wurde.

Das unten stehende Beispiel zeigt das Merkmal „Verzweigen“ aus dem Anhang B (s. S. 395):

Beispiel: Merkmal „Verzweigen“

D.6 Verzweigen *engl.* Branching

Ebene	: Komponenten (Domänentechnologie)
Beschreibung:	Graph-basierte Historien unterstützen das Versionieren von Varianten durch Verzweigen [CW98]. Abbildung B.18 (siehe S. 414) gibt einen Überblick über die detaillierten Merkmale des Verzweigens.
Umsetzung	: Vollständig implementiert
Kinder	: „Private Zweige“ (D.6.1)
Setzt voraus	: ohne „Einfache Menge“ (D.1.2.1.2), ohne „Sequenz“ (D.1.2.1.4)
Benötigt von	: „Verschmelzen protokollieren“ (D.11.2.3.3)
Quellen	: [CW98]

5.3. Beschreibung der Spitzenmerkmale

Auf jeder Ebene befinden sich auf oberster Ebene nun die sogenannten Spitzenmerkmale (engl. top-level features). Abbildung 5.4 zeigt die 23 Spitzenmerkmale des MOD2-SKM-Merkmalsmodells in der oben beschriebenen Gliederung. 13 der Spitzenmerkmale sind verpflichtend, d.h. sie sind allen Konfigurationen gemeinsam. Sie besitzen jedoch optionale Untermerkmale, in denen sich einzelne SKMS der MOD2-SKM Produktlinie unterscheiden. Die restlichen 10 Spitzenmerkmale sind optional, d.h. es existieren SKMS, die ohne sie auskommen. Im Folgenden werden nun die Spitzenmerkmale entsprechend der Ebenen und der Reihenfolge im Merkmalsmodell vorgestellt. Dabei wird insbesondere Bezug auf ihre wichtigsten Untermerkmale genommen. Die vollständige Analyse ist in Anhang B zu finden (S. 359ff.).

5.3.1. Merkmale der Befähigungs-Ebene

Die Spitzenmerkmale dieser Ebene beschreiben den Funktionsumfang der SKMS („Versionskontrolle“), die Daten, die in ihm gespeichert werden („Produktmodell“) sowie die grundlegende Konzeption des Repositoriums und seiner Unterstützung der Teamarbeit

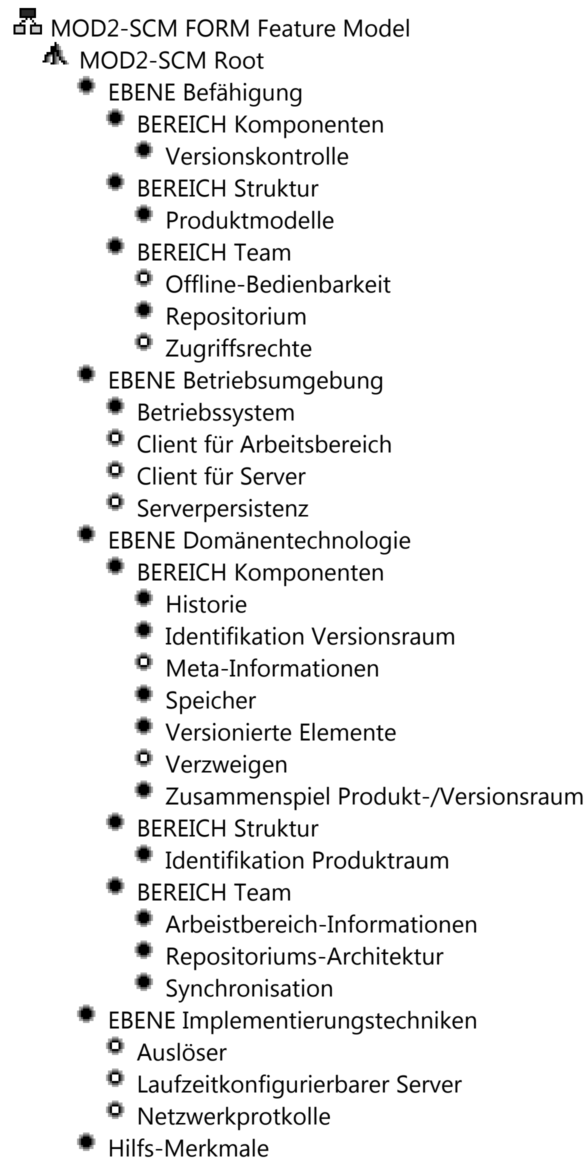


Abbildung 5.4.: Übersicht der Spitzenmerkmale

(„Offline-Bedienbarkeit“, „Repositorium“ und „Zugriffsrechte“). Die Merkmale stellen die Sicht eines Anwenders auf das SKMS dar.

Spitzenmerkmal: Versionskontrolle

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 364ff. detailliert beschrieben.

- Versionskontrolle
  - Commit/Update
    - Atomare Commits
    - Ausgewählte Elemente ignorieren
    - Commitauswahl
      - Einzelnes Element
      - Mehrere Elemente
      - Rekursion mit Tiefenbeschränkung
      - Vollständige Rekursion
    - Commits sammeln
    - Commits löschen
    - Kommentare
      - Kommentare erzwingen
    - Strikte Trennung von Lesen/Schreiben
    - Updateauswahl
      - Einzelnes Element
      - Mehrere Elemente
      - Rekursion mit Tiefenbeschränkung
      - Vollständige Rekursion
  - Versionskontroll-Operationen
    - Element kopieren
    - Element umbenennen
    - Elemente verschmelzen
    - Element verschieben

Abbildung 5.5.: Detaillierte Merkmale für die Versionskontrolle

Abbildung 5.5 zeigt die Untermerkmale dieses Spitzenmerkmals. Sie beschreiben sowohl den Funktionsumfang des SKMS als auch die Varianten der einzelnen Funktionen. Zentral ist insbesondere das Verhalten der „Commit/Update“-Operationen: Mit ihnen überträgt ein Entwickler Daten aus dem Arbeitsbereich in das Repositorium bzw. synchronisiert seinen Arbeitsbereich mit ihm. Die Operationen selbst sind ein verpflichtendes Merkmal und sie besitzen eine Vielzahl an Untermerkmalen, um ihr Verhalten detailliert zu beschreiben: „Atomare commits“ legt fest, ob die Datenübertragung in das Repositorium die Eigenschaften einer ACID-Transaktion aufweist. Dazu wird dann jedoch ein geeigneter Persistenzmechanismus benötigt (vgl. Spitzenmerkmal „Serverpersistenz“, S. 112) [Ehr06].

„Ausgewählte Elemente ignorieren“ ermöglicht dem Entwickler, Software-Elemente in seinem Arbeitsbereich zu ignorieren, so dass sie von den Commit-Operationen nicht be-

achtet werden [CSFP08, Ca09]. Die „Commitauswahl“ bietet mehrere Möglichkeiten, wie der Entwickler die Software-Elemente, die er bei einem Commit an das Repositorium übertragen möchte, auswählen kann. Es besteht immer die Möglichkeit, ein „einzelnes Element“ zu übertragen. Es können auch „mehrere Elemente“ gleichzeitig selektiert und in einem einzigen Commit übertragen werden. Bei größeren Elementmengen ist die manuelle Auswahl der Software-Elemente oft nicht praktikabel, so dass auch – ausgehend von einem Element – alle Unterelemente „vollständig rekursiv“ selektiert werden können. Falls nötig, kann die „Rekursion mit Tiefenbeschränkung“ ausgeführt werden, um z.B. nur die direkten Kindelemente für den Commit auszuwählen [CSFP08].

Das Merkmal „Commits sammeln“ bietet die Möglichkeit, noch komplexere Elementmengen auszuwählen, indem bei einem Commit die Element noch nicht direkt in das Repositorium, sondern einen Zwischenspeicher im Arbeitsbereich übertragen werden. So lassen sich die übertragenen Elemente gezielt zusammenstellen und auch wieder aus dem Zwischenspeicher entfernen. Dadurch ist es leichter sicherzustellen, dass eine logische Änderung (z.B. eine Änderungsanfrage) einem Commit entspricht [Ca09]. In der Regel arbeitet ein Repositorium strikt additiv, d.h. einmal übertragene Elemente existieren immer weiter im Repositorium (denn ihre Löschung bedeutet nicht, dass ihre Historie aus dem Repositorium entfernt wurde). In speziellen Fällen (z.B. aus Datenschutzgründen oder bei versehentlichem Übertragen vertraulicher Daten) ist es jedoch notwendig, eingespielte Daten zu löschen („Commits löschen“) [Ca09, O'S09]. Um den Grund eines Commits zu beschreiben, ist es möglich „Kommentare“ an ihn zu binden [CSFP08, Ca09, O'S09]. Da die Entwickler mit ihrer Hilfe die Historie deutlich leichter navigieren und verstehen, lassen sich „Kommentare erzwingen“ [KOL09].

Die „Strikte Trennung von Lesen/Schreiben“ regelt das Verhalten beim Einspielen zusammengesetzter Elemente [CSFP08]. Abbildung 5.6 illustriert dieses Untermerkmal anhand eines Beispielablaufs: Ein Elternelement e besitzt zwei Kindelemente k_1 und k_2. Entwickler A und B synchronisieren ihren Arbeitsbereich, so dass beide mit der gleichen Version arbeiten. Entwickler A verändert nun k_1 zu k_1' und überträgt die Änderungen ins Repositorium. e hat sich zwar inhaltlich nicht verändert, aber es existieren nun zwei Versionen e und e' bzgl. der Beziehung zu k_1 bzw.k_1' (vgl. „version proliferation“ [CW98]). Entwickler B verändert nun k_2 zu k_2' und überträgt die Änderung ebenfalls. Aus dem gleichen Grund wie bei A wird e' zu e''. Im Arbeitsbereich von A und B befindet sich aber immer noch e (und nicht e' bzw. e''). Beide müssen noch eine explizite Update-Operation ausführen. Den Grund demonstriert die Aktualisierung von B: Würde bereits die Commit-Operation e aktualisieren, müsste auch gleichzeitig k_1 aktualisiert werden, d.h. eine Commit-Operation würde nicht das Repositorium, sondern auch – wie eine Update-Operation – den Arbeitsbereich verändern [CSFP08].

Die „Updateauswahl“ regelt, analog zur „Commitauswahl“, das Auswahlverhalten für Elemente bei Update-Operationen [CSFP08]. Die nun folgenden Merkmale der „Versionskontroll-Operationen“ bestimmen, welche Operationen das SKMS im Arbeitsbereich anbietet – zusätzlich zu den Commit-/Update-Operationen. „Elemente kopieren“, „Elemente umbenennen“ und „Elemente verschieben“ ermöglicht, die drei entsprechenden Operationen so auszuführen, dass das SKMS explizit über die verwendete Operation informiert wird. Dies ist insbesondere notwendig, wenn das SKMS keine vollwertige OID besitzt

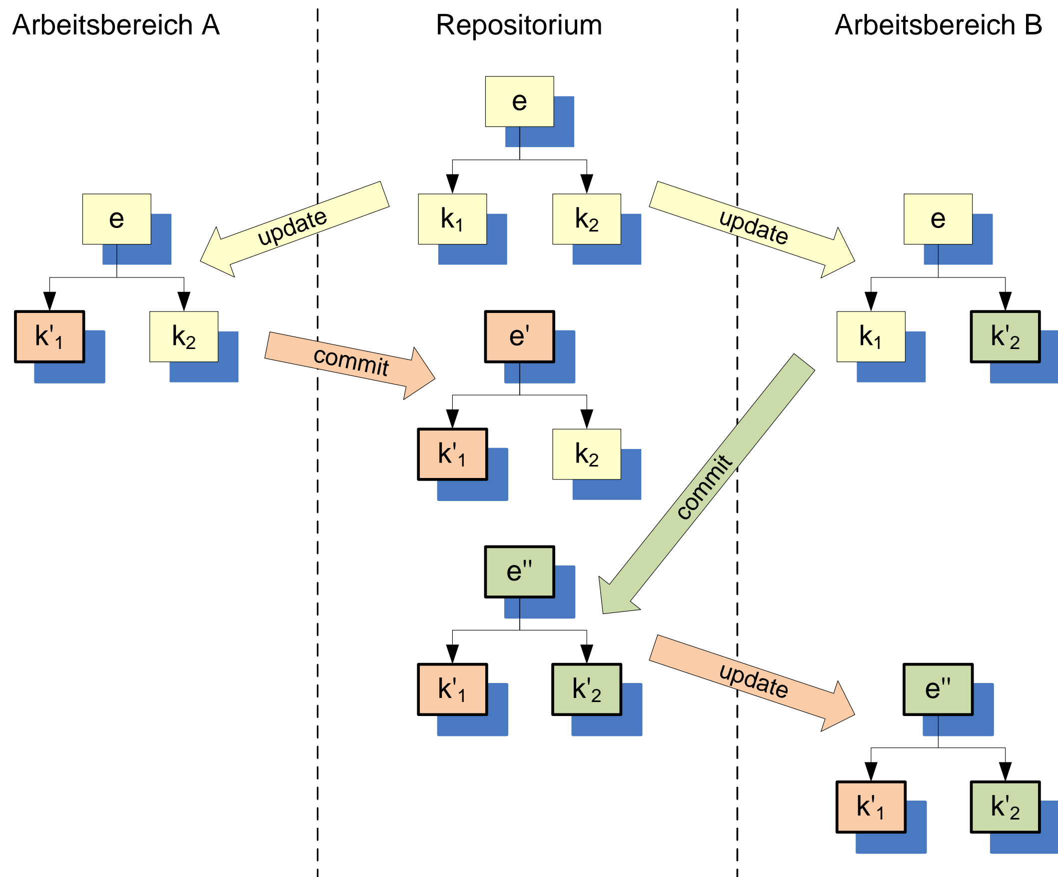


Abbildung 5.6.: Beispiel: Strikte Trennung von Lesen/Schreiben

(vgl. Abschnitt 4.2.2, S.76f. und Spitzenmerkmal „Identifikation Produktraum“, S.119): Die Änderung des Namens führt zu einer Veränderung der OID, und damit zum Beenden der Historie des alten Elements und zum Anlegen eines neuen Software-Elements. Um stattdessen die Historie fortzuführen, muss das SKMS diese Operationen explizit festhalten [CSFP08]. Das Merkmal „Elemente verschmelzen“ ermöglicht das Verschmelzen von zwei Elementen zwecks Zurückführen einer Verzweigung auf den Haupt- oder einen Seitenzweig [CW98].

Spitzenmerkmal: Produktmodell

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 374ff. detailliert beschrieben.

Abbildung 5.7 zeigt die Untermerkmale dieses Spitzenmerkmals. Das Produktmodell beschreibt die Software-Elemente und ihre Beziehungen untereinander (vgl. Abschnitt 4.2.3, S.76). MOD2-SKM unterstützt zur Zeit vier Produktmodelle: Das „Dateisystem“-Produktmodell ist das gebräuchlichste. Dateien und Verzeichnisse sind seine Software-

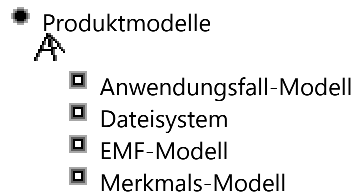


Abbildung 5.7.: Detaillierte Merkmale für die Produktmodelle

Elemente [Tic82, CW98, CSFP08, Ca09]. Das „Anwendungsfall-Modell“ dient vor allem zur Untersuchung der Versionierung von grafischen Modellen: Anwendungsfälle, Anwender und ihre Beziehungen sind hier die Software-Elemente. Das Produktmodell für „EMF-Modelle“ ist die Weiterentwicklung dieses Modells und soll die Versionierung von ecore-Instanzen unterstützen, d.h. die Elemente des ecore-Metamodells sind die Software-Element dieses Produktmodells [AKK+08, Pöh09]. Das „Merkmalsmodell“ ist ein Erweiterungspunkt. Die Elemente dieses Produktmodells wären Merkmale und ihre Beziehungen und würden die Versionierung der MODPL-Merkmalsmodelle erlauben. Dies wäre ein erster Schritt zur Versionierung von MODPL-Produktlinien. Die Aufzählung beschreibt letztendlich nur die bereits vorhandenen Produktmodelle. MOD2-SKM unterstützt die Erstellung und Verwendung eigener Produktmodelle, was sich jedoch nicht mit Hilfe einer expliziten Aufzählung beschreiben lässt.

Spitzenmerkmal: Offline-Bedienbarkeit

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 376ff. detailliert beschrieben.

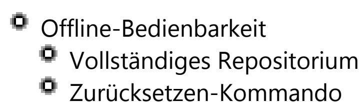


Abbildung 5.8.: Detaillierte Merkmale für die Offline-Bedienbarkeit

Abbildung 5.8 zeigt die Untermerkmale dieses Spitzenmerkmals. Ein Repositorium befindet sich, meist physikalisch von den Arbeitsbereichen getrennt, auf einem eigenen Computersystem. Der Datenaustausch zwischen Repositorium und Arbeitsbereich findet dabei über eine Netzwerkverbindung statt [CSFP08, Ca09]. Ist die Netzwerkverbindung nicht verfügbar, ist der Funktionsumfang des Arbeitsbereichs eingeschränkt. Durch Zwischenspeichern von Daten kann dieser jedoch vergrößert werden: Für die „Zurücksetzten-Operation“ wird die zuletzt synchronisierte Version eines Elements gespeichert, so dass sich alle Änderungen im Arbeitsbereich rückgängig machen lassen [CSFP08]. Beim „Vollständigen Repositorium“ ist jeder Arbeitsbereich an ein vollständiges, lokales Repositorium gekoppelt, dass sich mit anderen Repositorien synchronisieren lässt. Dies ist z.B. bei Peer-to-Peer-Repositorien der Fall, da hier die Synchronisation zweier Repositorien zur Basisfunktionalität gehört [Ca09] (vgl. Spitzenmerkmal „Repositorium“).

Spitzenmerkmal: Repositorium

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 377ff. detailliert beschrieben.

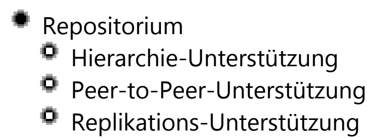


Abbildung 5.9.: Detaillierte Merkmale für das Repository

Abbildung 5.9 zeigt die Untermerkmale dieses Spitzenmerkmals. Bei diesen drei Untermerkmalen handelt es sich um Erweiterungspunkte, da die unterschiedlichen Verfahren zur Repositorien-Synchronisation aufwändig zu modellieren sind. Die „Hierarchie-Unterstützung“ würde die Bildung eines logischen Repositoriums aus mehreren verteilten Repositorien unterstützen [EC94, Hoe00, Ca09, Fis09], die „Peer-to-Peer-Unterstützung“ die Synchronisation verteilter (dezentralisierter) Repositorien [Ca09, O'S09] und die „Replikations-Unterstützung“ das Duplizieren bereits bestehender Repositorien [CSFP08, Fis09]. Letzteres ist auch immer ein Merkmal von Peer-to-Peer-SKMS, da hier – statt eines initialen Checkout – die Replikation eines bestehenden Repositoriums erfolgt [Ca09].

Spitzenmerkmal: Zugriffsrechte

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 379ff. detailliert beschrieben.

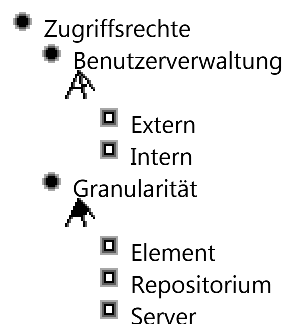


Abbildung 5.10.: Detaillierte Merkmale für die Zugriffsrechte

Abbildung 5.10 zeigt die Untermerkmale dieses Spitzenmerkmals. Die Untermerkmale beschreiben zum einen die „Benutzerverwaltung“, die entweder „Intern“ über das SKMS oder „Extern“ über ein eigenes Benutzerverwaltungssystem wie LDAP oder htaccess erfolgen kann. Zum anderen können einem Benutzer seine Zugriffsrechte auf unterschiedlichen Stufen der „Granularität“ gewährt werden. Entweder wird ihm Zugriff auf den vollständigen „Server“ gewährt oder nur auf einzelne „Repositorien“ oder sogar nur auf ausgewählte

„Elemente". Die Vergabe der Zugriffsrechte pro Element ist nur ein Erweiterungspunkt, da dieses Untermerkmal ein Regelsystem für die Deklaration der Zugriffsrechte benötigt [CSFP08].

5.3.2. Merkmale der Betriebsumgebungs-Ebene

Die Spitzenmerkmale dieser Ebene beschreiben die Hard- und Software der Systemumgebung, in der das SKMS eingesetzt wird, denn einige Merkmale sind z.B. nur auf einem bestimmten „Betriebssystem" verfügbar. Der „Client für den Arbeitsbereich" und der „Client für den Server" können auf verschiedene Arten in das Betriebssystem – oder sogar einzelne Software-Anwendungssysteme – integriert werden. Für die „Serverpersistenz" stehen ebenfalls unterschiedliche Persistenzmechanismen zur Auswahl, da einige Merkmale der Befähigungs-Ebene vom Funktionsumfang der jeweiligen Mechanismen abhängen.

Spitzenmerkmal: Betriebssystem

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 385ff. detailliert beschrieben.

- Betriebssystem
  - Java VM
  - MS Windows

Abbildung 5.11.: Detaillierte Merkmale für das Betriebssystem

Abbildung 5.11 zeigt die Untermerkmale dieses Spitzenmerkmals. Fujaba wird zum modellieren des Domänenmodells und zum generieren des konfigurierten SKMS eingesetzt. Fujaba generiert für das konfigurierte SKMS Java-Quellcode, so dass eine „Java Virtual Machine" benötigt wird, um es auszuführen [Buc10]. Das Dateisystem-Produktmodell (vgl. Merkmal „Produktmodell", S. 108) überwacht das lokale Dateisystem auf Änderungen, was jedoch nur im Betriebssystem „MS Windows" möglich ist.

Spitzenmerkmal: Client für Arbeitsbereich

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 386ff. detailliert beschrieben.

- Client für Arbeitsbereich
  - Browser-Schnittstelle
  - Grafischer Client
  - Integration in Eclipse
  - Kommandozeile

Abbildung 5.12.: Detaillierte Merkmale für den Arbeitsbereich-Client

Abbildung 5.12 zeigt die Untermerkmale dieses Spitzenmerkmals. Der Arbeitsbereich ermöglicht die Anbindung der versionierten Software-Elemente an Software-Anwendungs-

systeme, die lediglich unversionierte Elemente bearbeiten können (vgl. Abschnitt 4.4.2, S. 93). Diejenige Komponente des SKMS, die von den Entwicklern zur Verwaltung des Arbeitsbereichs sowie Synchronisation mit dem Repositorium eingesetzt wird, heißt Arbeitsbereich-Client. Der Client kann auf verschiedene Art und Weise in seine Software-Umgebung integriert sein: Als alleinstehendes Software-System, das entweder auf der „Kommandozeile“ nur textuelle Ein- und Ausgaben unterstützt oder als „Grafischer Client“, der Eingaben per Maus und komplexere grafische Ausgaben erlaubt [CSFP08, Ca09, O'S09]. Der Client kann auch als Dienst über eine „Browser-Schnittstelle“ arbeiten oder als „Integration in [die Entwicklungsumgebung] Eclipse“, die ein Rahmenwerk für die Integration von Arbeitsbereich-Clients anbietet [ecl10]. Auf Grund der umfangreichen Modellierung ist nur die Eclipse-Integration realisiert; alle anderen Merkmale sind Erweiterungspunkte.

Spitzenmerkmal: Client für Server

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 388ff. detailliert beschrieben.

- Client für Server
  - Browser-Schnittstelle
  - Grafischer Client
  - Kommandozeile

Abbildung 5.13.: Detaillierte Merkmale für den Server-Client

Abbildung 5.13 zeigt die Untermerkmale dieses Spitzenmerkmals. Analog zum Arbeitsbereich-Client existiert auch eine Komponente zur Verwaltung eines Repositoriums-Servers. Dieser „Server-Client“ kann auf verschiedene Art und Weise in seine Software-Umgebung integriert sein: Entweder mit textueller Ein- und Ausgabe auf der „Kommandozeile“ [CSFP08, Ca09, O'S09]. Oder als grafischer Client mit Mauseingabe und grafischer Ausgabe [KOL09]. Oder auch als Dienst einer „Browser-Schnittstelle“ über einen Webbrowser. Auf Grund der umfangreichen Modellierung ist nur die Verwaltung per Browser realisiert; alle anderen Merkmale sind Erweiterungspunkte. Der Browser-Client unterstützt sogar die Verwendung eines laufzeitkonfigurierbaren Servers, so dass ein Server beim Erstellen konfiguriert werden kann (vgl. Spitzenmerkmal „Laufzeitkonfigurierbarer Server“, S.122).

Spitzenmerkmal: Serverpersistenz

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 390ff. detailliert beschrieben.

Abbildung 5.14 zeigt die Untermerkmale dieses Spitzenmerkmals. Sie beschreiben verschiedene Verfahren, um die Daten der Repositorien zu persistenzieren und über den Halt des SKMS hinaus zu speichern. Persistenzmechanismen orientieren sich an den Anforderungen von Datenbanken und unterstützen Verfahren zum Sicherstellen der Datenkonsistenz, wie z.B. ACID- oder lange Transaktionen [CSFP08, Öhm10].

- EBENE Betriebsumgebung
  - Serverpersistenz
    - CoObRA Protokoll
    - Hibernate ORM
    - Java Objekt-Serialisierung

Abbildung 5.14.: Detaillierte Merkmale für die Serverpersistenz

Das „CoObRA-Protokoll“ ist ein änderungsbasierter Persistenzmechanismus, der die Änderungen an einem JavaBean-basierten Objektmodell protokolliert, und auch zum Speichern der Fujaba-Modelle (u.a. dem MOD2-SKM Domänenmodell) eingesetzt wird [Sch07]. Da als Grundfassung der Änderungen ein leeres Objektmodell dient, durchläuft ein Objektmodell beim Laden alle Zustände seit seiner Erstellung. Dadurch werden alle Deltaspeicherungs-Verfahren mit Rückwärtsdeltas (vgl. Abschnitt 4.3.6, S.89) wirkungslos, da CoObRA beim Laden jede Grundfassung anlegt und dann durch das entsprechende Delta ersetzt, d.h. , im CoObRA-Protokoll sind letztendlich alle Grundfassung und alle Deltas gespeichert, was mehr Platz benötigt als alle Grundfassungen alleine [Öhm10].

„Hibernate“ ist ein Persistenzrahmenwerk, das objektrelationale Daten in einer relationalen Datenbank speichert und wiederherstellt. Es bietet dadurch u.a. Unterstützung von ACID-Transaktionen [Öhm10] und ist damit Voraussetzung für „Atomare commits“ (vgl. Spitzenmerkmal „Versionskontrolle“, S. 106). Die „Java Objekt-Serialisierung“ verwendet den Persistenzmechanismus der Java-Laufzeitumgebung. Dazu wird der Zustand des Objektmodells lediglich als binärer Datenstrom oder XML-Repräsentation in eine Datei gesichert, ohne Unterstützung einer zusätzlichen Abstraktionsschicht oder Transaktionen [Ull09].

5.3.3. Merkmale der Domänentechnologie-Ebene

Die Spitzenmerkmale dieser Ebene beschreiben die domänenspezifischen Bestandteile einer SKMS. Im Bereich „Komponenten“ sind dies:

- Die „Historie“ zum Speichern der Versionen und ihrer Abhängigkeiten.
- Die „Identifikation [der Versionen im] Versionsraum“.
- Zusätzliche „Meta-Informationen“ zur Identifikation der Versionen.
- Der „Speicher“, der die Versionen – meist platzsparend – persistenziert.
- Die Komplexität der „Versionierten Elemente“.
- Die Möglichkeit, parallel Entwicklungen durch „Verzweigen“ zu unterstützen.
- Das „Zusammenspiel von Produkt-/Versionsraum“.

Im Bereich „Struktur“ ist dies die „Identifikation [der Elemente im] Produktraum“ und im Bereich „Team“, bestimmen die „Arbeitsbereich-Informationen“, welche Informationen die SKMS-Operationen im Arbeitsbereich benötigen. Und die „Repositoriums-

Architektur" bestimmt den Aufbau und Kommunikationswege des SKMS, während die „Synchronisation" Verfahren zur Koordination mehrerer Entwickler beschreibt.

Spitzenmerkmal: Historie

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 398ff. detailliert beschrieben.

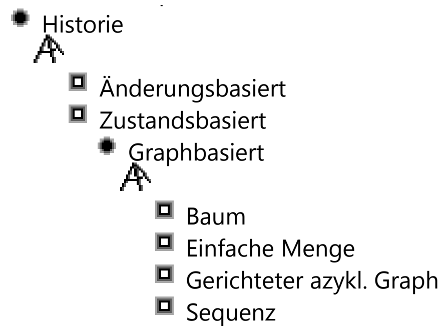


Abbildung 5.15.: Detaillierte Merkmale für die Historie

Abbildung 5.15 zeigt die Untermerkmale dieses Spitzenmerkmals. Die Elemente der Historie sind entweder Zustände eines Software-Elements („Zustandsbasiert") oder die Änderungen selbst („Änderungsbasiert"), wie in Abschnitt 4.3.2 beschrieben (s. S. 79). Während es sich bei der änderungsbasierten Historie um einen Erweiterungspunkt handelt, kann aus vier Arten der zustandsbasierten Historie gewählt werden. Diese unterscheiden sich in Art und Anzahl der verwendeten Beziehungen zwischen den Versionen (vgl. Abschnitt 4.3.3, S. 80). Die einfachste Historie bildet die „Einfache Menge", eine ungeordnete Menge von Versionen eines Software-Elements, ohne explizite Beziehungen untereinander. Beziehungen zwischen den Versionen lassen sich lediglich implizit, z.B. über die Versionskennung, ausdrücken. Die „Sequenz" verwendet Vorgänger-Nachfolger-Beziehungen, um die Revisionen eines Software-Elements zu ordnen. Beim „Baum" kommen zusätzlich noch Alternativbeziehungen hinzu, und der „Gerichtete azyklische Graph" erweitert dies noch um Verschmelzungsbeziehungen. Operationen zum Verzweigen (vgl. Spitzenmerkmal „Verzweigen", S. 117) und Verschmelzen (vgl. Spitzenmerkmal „Synchronisation", S. 120) setzen eine Historie mit entsprechenden Beziehungen voraus.

Spitzenmerkmal: Identifikation Versionsraum

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 402ff. detailliert beschrieben.

Abbildung 5.16 zeigt die Untermerkmale dieses Spitzenmerkmals. Dabei wird zwischen dem Typ der Versionskennung selbst („Versions-ID", vgl. Abschnitt 4.3.2, S. 79) sowie zusätzlichen Identifikationsverfahren, die indirekt auf Versionskennungen abgebildet werden („ID-Auflösung"), unterschieden. Die Versionskennung kann entweder der „Hashwert" einer Version sein, d.h. für das die Version repräsentierende Datenobjekt wird der

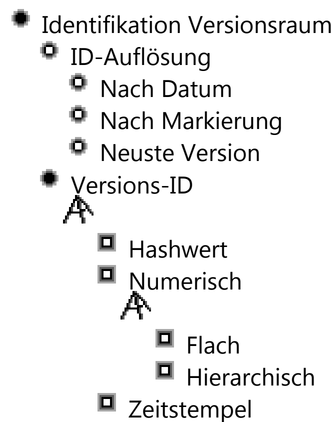


Abbildung 5.16.: Detaillierte Merkmale für die Identifikation im Versionsraum

Funktionswert einer Hashfunktion berechnet [Ca09]. Alternativ ist auch ein Zeitstempel verwendbar, der den genauen Zeitpunkt der Übertragung nutzt, wobei jedoch Ungenauigkeiten der internen Systemuhr bzw. der Systemzeit Probleme verursachen [FRH99]. Daher ist dieser Kennungstyp nur als Erweiterungspunkt vorgesehen. Am häufigsten wird eine „numerische" Repräsentation der Versionskennung verwendet: Entweder „flach", durch inkrementieren eines Zählers [Roc75], oder „hierarchisch" durch inkrementieren eines Zählers pro Verzweigung. Die Kennung entspricht dann der Konkatenation aller Zähler durch Punkte, z.B. „1.42.23" [Tic82].

Die Verfahren zur „ID-Auflösung" bieten die Möglichkeit – zusätzlich zur Versionskennung – Merkmale oder selbstdefinierte Kennungen zur Identifikation einzusetzen. „Nach Datum" ermöglicht Angabe eines Zeitpunktes, so dass ein Entwickler z.B. die „Version vom letzten Montag" referenzieren kann. Ebenso kann die „Neuste Version" referenziert werden, um z.B. den Arbeitsbereich auf den neusten Stand zu bringen [CSFP08]. Schließlich ist es auch möglich, selbstdefinierte Markierungen zu verwenden („Nach Markierung"), so dass Entwickler spezielle Versionen mit sprechenden Namen auszeichnen können, z.B. „1. Release" oder „Bugfix 9324". Mit ihrer Hilfe lassen sich z.B. auch Werkzeuge zur Änderungsverfolgung anbinden (vgl. Abschnitt 4.1.5, S. 74) [CSFP08, Pop09]. Alle drei Verfahren setzen zusätzliche Meta-Informationen über die Versionen voraus (vgl. Spitzenmerkmal „Meta-Informationen").

Spitzenmerkmal: Meta-Informationen

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 407ff. detailliert beschrieben.

Abbildung 5.17 zeigt die Untermerkmale dieses Spitzenmerkmals. Eine Version wird zwar über ihre Versionskennung eindeutig identifiziert, doch für einige Operationen werden zusätzliche Informationen über eine Version benötigt. Um eine Änderung den verantwortlichen Entwicklern zuzuordnen, lässt sich einer Version sowohl der „Autor" (der Urheber der Änderung) als auch der „Committer" (der Entwickler, der die Änderun-

- Meta-Informationen
  - Autor
  - Committer
  - Datum
  - Eigenschaften
  - Markierungen

Abbildung 5.17.: Detaillierte Merkmale für die Meta-Informationen

gen ins Repositorium eingespielt hat) zuordnen [CSFP08, Ca09, O'S09]. In viele Fällen handelt es sich um die gleiche Person, aber bei großen Projekten werden z.B. die Änderungen durch einen Entwickler überprüft, bevor sie ins Repositorium übertragen werden [Ca09]. Für die Zuordnung der Entwickler zu den entsprechenden Eigenschaften wird die Benutzerverwaltung verwendet (vgl. Spitzenmerkmal „Zugriffsrechte“, S. 110). Das „Datum“ speichert den Zeitpunkt, an dem eine Version in das Repositorium eingespielt wurde, so dass sich Versionen nicht nur über ihre Versionskennung, sondern auch über Zeitangaben identifizieren lassen. Zu einem ähnlichen Zweck können auch selbstdefinierte „Markierungen“ einer Version zugeordnet werden (vgl. Spitzenmerkmal „Identifikation Versionsraum“) [CSFP08, Ca09, O'S09]. Aus Gründen der Erweiterbarkeit lässt sich das Konzept der Meta-Informationen so erweitern, dass einer Version beliebige „Eigenschaften“ (engl. properties) – in Form von Schlüssel-Wertepaaren – zugeordnet werden können [CSFP08].

Spitzenmerkmal: Speicher

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 409ff. detailliert beschrieben.

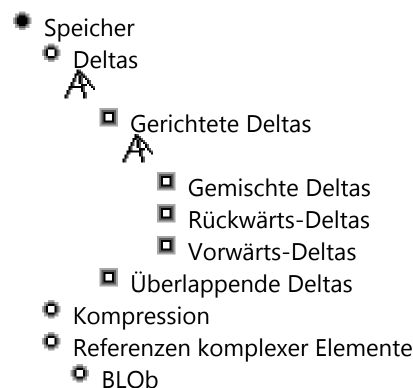


Abbildung 5.18.: Detaillierte Merkmale für den Speicher

Abbildung 5.18 zeigt die Untermerkmale dieses Spitzenmerkmals. Da ein SKMS die gesamte Historie aller Software-Elemente speichert, besitzen seine Repositorien einen hohen Speicherplatzbedarf (vgl. Abschnitt 4.3.6, S. 88), so dass verschiedene Verfahren zum Sparen von Speicherplatz existieren. Eine Möglichkeit bietet das Speichern von „Deltas“,

indem – statt der vollständigen Version – nur die Unterschiede zu einer Vorgängerversion gespeichert werden. Bei gerichteten Deltas unterscheiden sich die Verfahren, je nachdem ob die Vorgängerversion auch der zeitliche Vorgänger ist oder nicht, so dass zwischen „Vorwärts-", „Rückwärts-" oder „Gemischten Deltas" unterschieden wird (vgl. Abschnitt 4.3.6, S. 89) [Tic82, CW98, Mat09]. Überlappende Deltas bestehen dagegen aus den Inhalten aller Versionen und stellen durch Markierungen sicher (ähnlich der bedingten Kompilierung), dass nur der Inhalt der entsprechenden Version sichtbar ist (vgl. Abschnitt 4.3.6, S. 91) [Roc75, CW98]. Zusätzlich kann der Speicherplatzbedarf von Versionen bzw. Deltas per Byte-Kompression reduziert werden [Mat09]. Besitzt ein Produktmodell komplexe (zusammengesetzte) Software-Elemente (vgl. Abschnitt 4.2.3, S.76), kann durch den Einsatz von Referenzen auf unveränderte Kindelemente Speicherplatz gespart werden, indem nur veränderte Elemente im Repositorium gespeichert werden. Die unveränderten Kindelement werden entfernt und durch Referenzen auf die bereits existierenden Versionen gleichen Inhalts ersetzt (vgl. Abbildung 4.10, S. 86) [CSFP08]. Zusätzlich lässt sich der Inhalt noch von den Elementen trennen, so dass der gleiche Inhalt – über alle Elemente hinweg – nur einmalig gespeichert und von allen Versionen referenziert wird (vgl. Abbildung 4.11, S. 87) [Ca09].

Spitzenmerkmal: Versionierte Elemente

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 413ff. detailliert beschrieben.

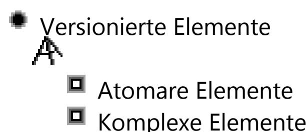


Abbildung 5.19.: Detaillierte Merkmale für die Versionierten Elemente

Abbildung 5.19 zeigt die Untermerkmale dieses Spitzenmerkmals. Sie legen fest, ob die Beziehungen zwischen den Elementen im Produktraum (vgl. Abschnitt 4.2.3, S. 76) ebenfalls im Repositorium gespeichert werden können. Ein Repositorium für „Atomare Elemente" speichert nur die Versionen der einzelnen Elemente unabhängig von ihren Beziehungen, so dass lediglich ein produktzentriertes Zusammenspiel von Produkt- und Versionsraum möglich ist (vgl. Spitzenmerkmal „Zusammenspiel Produkt-/Versionsraum", S. 118) [Tic82, CW98]. „Komplexe Elemente" versionieren dagegen die Beziehungen unter den Elementen im Produktraum, z.B. die Teil-Ganze-Beziehung. Dies ist Voraussetzung für ein versionszentriertes oder verwobenes Zusammenspiel der beiden Räume [CW98].

Spitzenmerkmal: Verzweigen

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 414ff. detailliert beschrieben.

Abbildung 5.20 zeigt die Untermerkmale dieses Spitzenmerkmals. Verzweigen ermöglicht das Anlegen von Alternativbeziehungen zwischen einzelnen Versionen, so dass z.B.

Verzweigen
Private Zweige

Abbildung 5.20.: Detaillierte Merkmale für das Verzweigen

mehrere Entwickler parallel Versionen einspielen können, so dass die Historie mindestens einen Baum bildet (vgl. Spitzenmerkmal „Historie“, S. 5.3.3). Soll die Parallelentwicklung später wieder auf den Hauptzweig zurückgeführt werden, wird als Historie sogar ein gerichteter azyklischer Graph und die Verschmelzen-Funktionalität benötigt (vgl. Spitzenmerkmal „Synchronisation“, S. 120). „Private Zweige“ erzwingen die Verwendung eines Zweiges für einen Entwickler. So kann niemand direkte Änderungen an der aktuellen Version des Produktes durchführen, sondern muss die Änderungen zunächst auf seinen privaten Zweig übertragen, bevor dann ein zuständiger Entwickler diese Änderung explizit in den Hauptzweig überträgt [Ehr06, CSFP08]. Dieses Untermerkmal ist ein Erweiterungspunkt.

Spitzenmerkmal: Zusammenspiel Produkt-/Versionsraum

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 415ff. detailliert beschrieben.

Zusammenspiel Produkt-/Versionsraum
Produktzentriert
Versionszentriert
Verwoben

Abbildung 5.21.: Detaillierte Merkmale für das Zusammenspiel Versions-/Produktraum

Abbildung 5.21 zeigt die Untermerkmale dieses Spitzenmerkmals. Das Zusammenspiel von Versions- und Produktraum legt fest, wie die Beziehungen im Produktraum vom SKMS gespeichert und ausgewertet werden (vgl. Abschnitt 4.3.4). „Produktzentriert“ bedeutet, dass keinerlei Beziehungen zwischen den Software-Elementen versioniert werden und sich zusammengehörige Elemente lediglich manuell vom Entwickler zusammenstellen lassen, bzw. indirekt über Meta-Informationen wie Markierungen (vgl. Spitzenmerkmal „Identifikation Versionsraum“, S. 5.3.3 und Abschnitt 4.3.4, S. 83) [CW98, Ced05]. In diesem Fall reicht es aus, wenn das Repositorium atomare Elemente unterstützt (S. 5.3.3).

„Versionszentriert“ bedeutet dagegen, dass, durch die Auswahl einer Version eines Elements, die Versionen aller Teilelemente festgelegt werden, die über den transitiven Abschluss der Teil-Ganze-Beziehungen erreichbar sind (vgl. Abschnitt 4.3.4, S. 83). „Verwoben“ ähnelt dem versionszentrierten Ansatz, erlaubt jedoch die (manuelle oder automatisierte) Auswahl der Versionen der Teilelemente (vgl. Abschnitt 4.3.4, S. 83) [CW98]. In beiden Fällen ist es nötig, dass das Repositorium komplexe (zusammengesetzte) Elemente unterstützt (vgl. Spitzenmerkmal „Versionierte Elemente“, S. 117).

Spitzenmerkmal: Identifikation Produktraum

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 417ff. detailliert beschrieben.

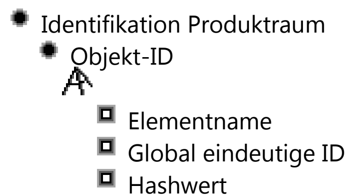


Abbildung 5.22.: Detaillierte Merkmale für die Identifikation im Produktraum

Abbildung 5.22 zeigt die Untermerkmale dieses Spitzenmerkmals. Um ein Software-Element im Produktraum zu identifizieren, existieren verschiedene Arten der Objektkennung. Mehrere Objekte mit der gleichen Objektkennung bilden unterschiedliche Versionen des gleichen Elements (vgl. Abschnitt 4.2.2, S. 76) [CW98]. Der „Elementname“ dient häufig als Kennung – obwohl er nicht alle Eigenschaften einer ordentlichen Objektkennung aufweist. Dies führt zu Zuordnungsproblemen von Versionen zu Elementen, sobald das Element umbenannt wird, die sich nur durch Einführung spezieller Versionskontrolloperationen zum Umbenennen und Verschieben von Elementen lösen lassen (vgl. Spitzenmerkmal „Versionskontrolle“, S. 106) [Ced05]. Daher ist die Vergabe einer ordentlichen „Objekt-ID“ – ähnlich einem Primärschlüssel in einer Datenbank – empfehlenswert [WJ95, CW98]. Eine interessante Alternative bietet auch die Verwendung eines Funktionswertes einer Hashfunktion, die auf das Element angewendet wird („Hashwert“): Damit wird für gleiche Elemente immer die gleiche Objektkennung vergeben. Zusätzlich dient sie auch zur Konsistenzprüfung und zur schnelle Prüfung auf Gleichheit [Ca09].

Spitzenmerkmal: Arbeitsbereich-Informationen

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 419ff. detailliert beschrieben.

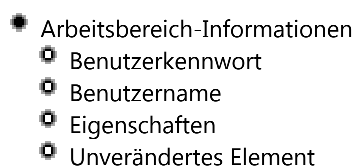


Abbildung 5.23.: Detaillierte Merkmale für die Arbeitsbereich-Informationen

Abbildung 5.23 zeigt die Untermerkmale dieses Spitzenmerkmals. Um die Arbeit der Entwickler im Arbeitsbereich zu erleichtern, können dort verschiedene Informationen gespeichert werden. Das Zwischenspeichern von „Benutzerkennwort“ und „Benutzername“ erleichtert den Zugriff auf ein Repositorium mit Benutzerverwaltung (vgl. Spitzenmerkmal „Zugriffsrechte“, S. 5.3.1) [KOL09]. Lassen sich Versionen mit selbstdefinierten Eigen-

schaften auszeichnen, so sind diese auch im Arbeitsbereich sichtbar und können hier ebenfalls ausgewertet werden (vgl. Spitzenmerkmal „Meta-Informationen“, S. 5.3.3) [CSFP08]. Und um die zuletzt im Arbeitsbereich ausgeführten Änderungen – ohne Repositoriums-Zugriff – rückgängig zu machen, kann die zuletzt aus dem Repositorium ausgelesene Version gespeichert werden (vgl. Spitzenmerkmal „Offline-Bedienbarkeit“, S. 5.3.1) [CSFP08].

Spitzenmerkmal: Repositoriums-Architektur

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 421ff. detailliert beschrieben.

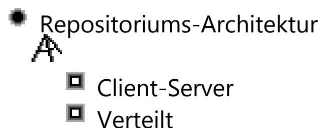


Abbildung 5.24.: Detaillierte Merkmale für das Repository

Abbildung 5.24 zeigt die Untermerkmale dieses Spitzenmerkmals. Die Architektur legt den grundlegenden Aufbau und die Kommunikationswege zwischen Arbeitsbereichen und Repositorien fest und somit die Möglichkeiten zur Synchronisation (vgl. Abschnitt 4.4.1, S. 91). Die „Client-Server“-Architektur verwendet ein einzige Repositorium (Server) mit dem sich alle Arbeitsbereiche (Client) synchronisieren lassen. Die Synchronisation zwischen Repositorien ist nicht zwingend erforderlich, was die Implementierung dieser Architektur erleichtert. Bei „verteilten“ Repositorien existiert nicht zwingend ein zentrales Repositorium. Stattdessen besitzt jeder Arbeitsbereich sein eigenes Repositorium, das sich mit allen anderen Repositorien synchronisieren lässt. Daher sind Funktionen zum replizieren und synchronisieren von Repositorien unbedingt erforderlich (vgl. Spitzenmerkmal „Repositorium“, S. 5.3.1). Auf Grund des hohen Modellierungsaufwands dieser Operationen sind die verteilten Repositorien ein Erweiterungspunkt.

Spitzenmerkmal: Synchronisation

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 422ff. detailliert beschrieben.

Abbildung 5.25 zeigt die Untermerkmale dieses Spitzenmerkmals. Sie koordinieren die Zusammenarbeit mehrerer Entwickler. Zum einen „Intern“, indem sie die parallelen Zugriffe der Entwickler auf das Repositorium organisieren; entweder darf nur ein Entwickler zur gleichen Zeit auf den Server zugreifen, oder es werden nur die Elemente gesperrt, die durch die Operation des Entwicklers betroffen sind [HR83, CSFP08]. Letzter Fall wird insbesondere dann eingesetzt, wenn die Beziehungen zwischen den Elementen nicht versioniert werden (vgl. Spitzenmerkmal „Zusammenspiel Produkt-/Versionsraum“, S. 118). In Persistenzmechanismen mit ACID-Transaktionen ist diese Funktionalität bereits integriert (vgl. Spitzenmerkmal „Serverpersistenz“, S. 112).

Für Probleme mit der „Nebenläufigkeit“ – wie das Gleichzeitige-Aktualisierungs-Problem (vgl. Abschnitt 4.1.1, S. 70) – existieren zwei Lösungsansätze: Optimistische und

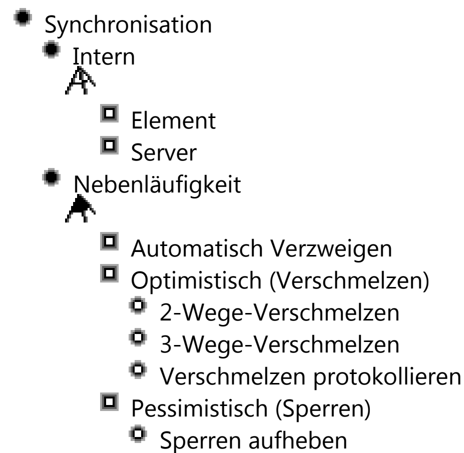


Abbildung 5.25.: Detaillierte Merkmale für die Synchronisation

pessimistische Sperren (vgl. Abschnitt 4.4.1, S. 92) [CW98]. Die pessimistischen Sperren („Sperren") blockieren den Zugriff auf ein Element, bis es der bearbeitende Entwickler wieder freigibt. Ein häufiges Problem ist, dass der Entwickler vergisst, das Element freizugeben, so dass ausgewählten Entwicklern (z.B. Administratoren) die Funktion angeboten werden kann, „Sperren aufzuheben" [CSFP08]. Optimistische Sperren verbieten das Einspielen einer Version, falls ihre Ursprungsversion bereits einen Nachfolger im Repositorium besitzt. In diesem Fall muss der Entwickler zunächst seine Änderungen mit den Nachfolgern "verschmelzen" – erst dann darf er das Ergebnis als neue Version in das Repositorium einspielen. Dazu stehen ihm verschiedene Arten der Verschmelzung zur Verfügung [CW98]. Verschmelzen wird auch bei SKMS eingesetzt, die das Verzweigen unterstützen (vgl. Spitzenmerkmal „Verzweigen", S. 117), um die Änderungen auf einem Zweig in den Hauptzweig zurückzuspielen und auch in der Historie durch eine Beziehung auszudrücken („Verschmelzen protokollieren") [CW98].

5.3.4. Merkmale der Implementierungstechniken-Ebene

Die Spitzenmerkmale dieser Ebene beschreiben unterschiedliche Techniken und Verfahren, die unabhängig von der SKM-Domäne sind. Über „Auslöser" (engl. trigger) können die Entwickler das Verhalten des SKMS erweitern, sobald bestimmte, vordefinierte Ereignisse eintreten. Der „Laufzeitkonfigurierbare Server" ermöglicht die Konfiguration ausgewählter Merkmale zur Laufzeit, was insbesondere das Testen der MOD2-SKM Produktlinie erleichtert. Der Datenaustausch zwischen Repositorien und Arbeitsbereichen erfolgt über eines der „Netzwerkprotokolle".

Spitzenmerkmal: Auslöser

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 429ff. detailliert beschrieben.

- Auslöser
  - Nach-Ereignis-Auslöser
    - E-Mail Benachrichtigung
  - Vor-Ereignis-Auslöser

Abbildung 5.26.: Detaillierte Merkmale für die Auslöser

Abbildung 5.26 zeigt die Untermerkmale dieses Spitzenmerkmals. Ein „Auslöser" bietet Entwicklern die Möglichkeit, das Verhalten des SKMS an vordefinierten Punkten zu erweitern. Dazu lösen ausgewählte Ereignisse eine Notifikation aus, woraufhin die Erweiterung ausgeführt wird. Die Notifikation kann sowohl vor („Vor-Ereignis-Auslöser") als auch nach („Nach-Ereignis-Auslöser") dem Ereignis erfolgen. Erstere können so vor einem Ereignis in die Verarbeitung eingreifen (z.B. Daten validieren und die Operation abbrechen), während letztere der Informationsverbreitung dienen, z.B. zum Versand einer „E-Mail Benachrichtigung". Alle Untermerkmale sind Erweiterungspunkte, da sie für ein funktionierendes SKMS nicht zwingend benötigt werden.

Spitzenmerkmal: Laufzeitkonfigurierbarer Server

Im MODPL-Prozess wird mit Hilfe einer Konfiguration aus dem Domänenmodell ein Modell eines konfigurierten SKMS (ein konfiguriertes Domänenmodell) erzeugt. Damit ist die Konfiguration abgeschlossen und zur Laufzeit des SKMS unveränderlich [Buc10]. Zum Testen des Zusammenspiels der Realisierungen der unterschiedlichen Merkmale existiert in MOD2-SKM ein spezieller laufzeitkonfigurierbarer Server. Er erlaubt es, bestimmte Merkmale bei Inbetriebnahme zu setzen, so dass die Konfiguration programmatisch zur Laufzeit erfolgen kann. Dies ermöglicht den Einsatz parametrisierbarer Tests, die das Kreuzprodukt aller Laufzeit-Merkmale überprüfen und so eine Teilmenge der gesamten Produktlinie testen.

Der Laufzeitkonfigurierbare Server stellt ein besonderes Merkmal dar, da er einen Teil der Konfiguration letztendlich ignoriert. Während ein vollständig konfiguriertes SKMS nur eine Art von Speichermechanismus oder Historie besitzt, soll der Laufzeitkonfigurierbare Server alle Kombinationen testen, d.h. er benötigt alle Arten, um die unterschiedlichen Konfigurationen zur Laufzeit zu erstellen. Letztendlich bedeutet dies, dass die Konfiguration eines Laufzeitkonfigurierbaren Servers lediglich partiell gebunden ist, da nicht alle Merkmale selektiert werden müssen.

Der Laufzeitkonfigurierbare Server der MOD2-SKM Produktlinie ignoriert die Untermerkmale folgender Merkmale: „Granularität" (B.5.2), „Graphbasiert" (D.1.2.1), „Speicher" (D.4), „Versionierte Elemente" (D.5) sowie das Merkmal „Verzweigen" (D.6).

Spitzenmerkmal: Netzwerkprotokolle

Die Untermerkmale dieses Spitzenmerkmals werden in Anhang B ab Seite 431ff. detailliert beschrieben.

Abbildung 5.27 zeigt die Untermerkmale dieses Spitzenmerkmals. Sie bestehen aus verschiedenen Protokollen, mit denen sich Daten über ein Netzwerk zwischen den Ar-

Abbildung 5.27.: Detaillierte Merkmale für die Netzwerkprotokolle

beitsbereichen und Repositorium austauschen lassen. „Java Remote Method Invocation (Java RMI)“ ist Teil der Java Standard Laufzeitumgebung, und macht die Aufrufe und Übertragung von Parametern, über Laufzeitumgebungen hinweg, transparent [Ull09]. Beim Einsatz eines „Webservers“ stehen Protokolle wie „HTTP“, „HTTPS“ oder auch „WebDAV“ zur Verfügung [CSFP08, Ca09, O'S09]. Auch eine Übertragung per E-Mail ist denkbar, z.B. um die Änderungen als eigenständige „Patches“ an einen zuständigen Entwickler zu senden [Ca09]. Da die Art des Protokolls die Funktionalität des SKMS kaum beeinflusst, wurde nur der Austausch per „Java RMI“ realisiert – alle anderen Merkmale sind Erweiterungspunkte.

5.3.5. Probleme und Grenzen

Die MOD2-SKM-Merkmalsanalyse stieß sowohl an einige konzeptionelle Grenzen des Merkmalsmodells als auch der Werkzeugunterstützung.

Konzeptionelle Grenzen

Zwei Merkmale ließen sich nicht mit Hilfe der Elemente des FORM-Modells adäquat beschreiben: Zum einen die Produktmodelle (vgl. „Produktmodelle“ (B.2), S. 108) und zum anderen der Laufzeitkonfigurierbare Server (vgl. „Laufzeitkonfigurierbarer Server“ (I.2), S. 122). MOD2-SKM bietet die Möglichkeit, eigene Produktmodelle zu definieren und anschließend zu versionieren, d.h. , die Menge der möglichen Produktmodelle lässt sich nur implizit beschreiben. FODA bzw. FORM erlauben jedoch nur explizite Aufzählungen, so dass dieser Sachverhalt nicht im Merkmalsmodell modelliert wird. Der laufzeitkonfigurierbare Server ermöglicht das automatisierte Testen eines Teils der Produktlinie: Anstatt alle SKMS einzeln zu konfigurieren und anschließend zu generieren, ermöglicht er die Konfiguration zur Laufzeit, so dass sich während der Tests eine neue Konfiguration erstellen lässt. So können die Testfälle automatisiert unterschiedliche Konfigurationen testen. Dafür ist es nötig, ein SKMS für eine partielle Konfiguration zu erstellen, da die Tests einen Teil der Konfiguration übernehmen. Dies lässt sich ebenfalls nicht modellieren, da eine partielle Konfiguration per Definition „unvollständig“ ist.

Werkzeugunterstützung

Auf Grund der wenigen Sprachelemente ließ sich die Erstellung eines initialen Merkmalsmodells nach FODA [KCH+90] schnell und einfach durchführen. Die 147 Merkmale des finalen Merkmalsmodells sind jedoch bereits zu unübersichtlich für ein unstrukturiertes Modell, so dass die FORM-Methode [KKL+98] eingesetzt wurde, um die Merkmale gemäß den vier Ebenen zu gruppieren. Auch diese Klassifizierung stellte sich als zu grob heraus, so dass zusätzlich die Anforderungsbereiche (vgl. Abschnitt 4.1.2) zur Gliederung dienen. Das für FODA implementierte Feature-Plugin [AC04] besitzt keinerlei Unterstützung für FORM-Ebenen oder andere Gliederungsebenen, so dass diese als verpflichtende Merkmale modelliert sind. Werkzeugunterstützung für „Modellieren im Großen“ ist hier notwendig.

Die 34 aussagenlogischen Einschränkungen beschränken sich in der Regel auf Implikationen mit zwei Merkmalen, wobei keine Klausel mehr als fünf Merkmale besitzt. Komplexere aussagenlogische Ausdrücke wären auch nur schwer zu pflegen, da sie im Feature-Plugin textuell als Wenn-Dann-Klauseln eingegeben werden, wobei eine interne Kennung des Merkmals verwendet wird [AC04]. Es existiert keinerlei Fehlerbehandlung, geschweige denn Validierung für die Klauseln, so dass bereits ein Tippfehler in einer Kennung keine Konfigurationsschritte mehr erlaubt. Die fehlerhafte Kennung ließ sich nur durch Setzen von Haltepunkten im Quellcode des Feature-Plugins selbst herausfinden. Ein mächtigeres Werkzeug zur Eingabe der Einschränkungen auf Basis der Merkmale selbst – und nicht ihren internen Kennungen – ist hier notwendig.

Auch die fehlende Visualisierung von Abhängigkeiten erschwert bereits bei 147 Merkmalen den Zugang zum Merkmalsmodell, da die 34 Einschränkungen global im Modell definiert werden, und nicht für ein Merkmal [AC04]. Es ist auch nicht möglich, die Einschränkungen zu filtern und z.B. nur diejenigen anzuzeigen, an denen ein ausgewähltes Merkmal beteiligt ist. Während der Konfiguration werden alle De-/Selektionen, die durch strukturelle oder aussagenlogische Einschränkungen vorgegeben sind, automatisch ausgeführt und farblich hervorgehoben [AC04]. Es ist jedoch nicht erkennbar welche Änderungen durch die letzte manuelle An- oder Abwahl eines Merkmals erfolgt sind, so dass die Auswirkungen eines Konfigurationsschritts mit steigender Modellgröße immer unklarer werden. Für die Konfiguration und Wartung von Merkmalsmodellen – und insbesondere den aussagenlogischen Einschränkungen – ist daher ein Werkzeug mit komplexen Visualisierungs- und Filtermöglichkeiten notwendig.

Eine Konfiguration ist immer von ihrem zugrunde liegenden Merkmalsmodell abhängig. Bei Änderungen am Merkmalsmodell bietet das Feature-Plugin die Möglichkeit das neue Modell mit den existierenden Konfigurationen zu synchronisieren [AC04]. Solange nur Merkmale hinzugefügt werden, funktioniert dieser Mechanismus, das Löschen oder sogar Verschieben von Merkmalen wird nicht erkannt, so dass eine manuelle Korrektur aller Konfigurationen notwendig ist. Insbesondere beim Verschieben von Merkmalen kommt es zu Problemen, da die Kennung eines Merkmals eindeutig sei muss. Aus diesem Grund wird bei den verschobenen Merkmalen eine laufende Nummer an die Kennung angehängt, so dass nach dem Löschen der alten Merkmale die Kennungen nicht mehr mit der vorherigen Konfiguration übereinstimmen. Taucht eine Kennung in einer aussagenlo-

gischen Einschränkung auf, lässt sich die Konfiguration nicht mehr verändern (s.o.). Hier benötigt der Anwender Werkzeugunterstützung, um eine schrittweise Synchronisation durchzuführen.

Die zugehörige Merkmalsanalyse musste manuell an das Merkmalsmodell angepasst werden. Jegliche Änderung an den Merkmalsnamen, der Hierarchie oder den Abhängigkeiten sind ohne Werkzeugunterstützung übertragen worden. Auch bei der Erstellung der Abbildungen der Modelle, sowohl in Anhang C als auch der Ausschnitte in diesem Kapitel, fielen langwierige Exportoperationen und aufwändige manuelle Bildbearbeitungsschritte an. Ähnlich wie bei Werkzeugen für die Anforderungs- bzw. Anwendungsfall-Analyse ist hier ein einheitliches Werkzeug notwendig. Dieses Werkzeug benötigt z.B. auch die Unterstützung zur Aufbereitung des Merkmalsmodells für die Druckausgabe, insbesondere für die mehrseitigen Merkmalsmodelle.

5.4. Validierung des MOD2-SKM Merkmals-Modells

Bevor das Merkmalsmodell verwendet wird, sollte es zunächst validiert werden. Die Validierung erfolgt durch Erstellen einer Konfiguration für jedes SKMS, das während der Merkmalsanalyse zur Identifikation von Merkmalen gedient hat. Dies stellt sicher, dass das Merkmalsmodell die Merkmale der analysierten SKMS korrekt repräsentiert [KKL+98]. Für die MOD2-SKM Merkmalsanalyse sind die SKMS CVS, GIT, Mercurial und Subversion verwendet worden. Die vier SKMS werden im Folgenden beschrieben. Zu ausgewählten Anforderungen wird das zugehörige Merkmal und dessen Spitzenmerkmal genannt, so dass seine Beschreibung sowohl in der Merkmalsanalyse in Anhang B (S. 359ff.) als auch in der Übersicht der Spitzenmerkmale in Abschnitt 5.3 (S. 104ff.) nachgeschlagen werden kann. Die vollständigen Konfigurationen des Merkmalsmodells sind in Anhang D (S. 439ff.) abgebildet.

5.4.1. Concurrent Versioning System

Mit Hilfe des MOD2-SKM Merkmalsmodells lässt sich eine Konfiguration für CVS definieren. Die Konfiguration ist in Anhang D auf S. 439f. abgebildet. Das **C**oncurrent **V**ersioning **S**ystem (CVS) wird seit 1986 entwickelt und ist als freie Software unter GNU Lizenz verfügbar. Ursprünglich eine Sammlung von Shell-Skripten, wurde es einige Jahre später sowohl in C als auch C++ implementiert [Ced05]. Beide Implementierungen besitzen die gleiche Kommandozeilen-Schnittstelle (*„Kommandozeile“ (O.2.4) aus „Client für Arbeitsbereich“, S. 111*) und sogar die gleiche Struktur: Die C++-Implementierung verwendet *keine* Klassen, um CVS zu beschreiben.

CVS verwaltet die Entwicklungshistorie von Dateien (*„Dateisystem“ (B.2.2) aus „Produktmodelle“, S. 108*), ignoriert jedoch die Historie von Verzeichnissen (*ohne „Komplexe Elemente“ (D.5.2) aus „Versionierte Elemente“, S. 117*) und verknüpft so Produkt- und Versionsraum produktzentriert (*„Produktzentriert“ (D.7.1) aus „Zusammenspiel Produkt-/Versionsraum“, S. 118*). Die Zusammenarbeit der Entwickler wird durch ein zentrales Repository unterstützt (*„Client-Server“ (D.10.1) aus „Repositoriums-Architektur“, S. 120*). Die Dateien werden anhand ihres Namens identifiziert (*„Elementname“ (D.8.1.1)*

aus „Identifikation Produktraum", S. 119). Umbenennen oder Verschieben von Dateien zerstört die Historie, da keine speziellen Operationen existieren (*ohne „Versionskontroll-Operationen" (B.1.2) aus „Versionskontrolle", S. 106*). Verzweigungen werden unterstützt (*„Verzweigen" (D.6), S. 117*) und mit Hilfe hierarchischer Versionskennungen ausgedrückt (*„Hierarchisch" (D.2.2.2.2) aus „Identifikation Versionsraum", S. 114*). Alternativ ist die Kennzeichnung mit Markierungen möglich (*„Nach Markierung" (D.2.1.2) aus „Identifikation Versionsraum", S. 114*). Der Speicherplatzbedarf wird mit Hilfe von vermischten Deltas reduziert (*„Gemischte Deltas" (D.4.1.1.1) aus „Speicher", S. 116*). Konflikte werden mit Hilfe von Sperren (*„Sperren" (D.11.2.2) aus „Synchronisation", S. 120*) oder Verschmelzen (*„Verschmelzen" (D.11.2.3) aus „Synchronisation", S. 120*) behoben.

CVS besitzt keine speziellen Erweiterungsmöglichkeiten , um es an neue Anforderungen anzupassen (*ohne „Auslöser" (I.1), S. 121*), was u.a. zu unterschiedlichen Funktionalitäten in der C- und C++-Implementierung führte. In der Praxis werden oft Skripte verwendet, um die CVS-Kommandos zu kapseln und anschließend die gewünschte Funktionalität hinzuzufügen. So kann z.B. ein Kommentar für einen Commit nur durch Einlesen und Prüfen des Kommentars im kapselnden Programm erzwungen werden. Eine Prüfung und Ablehnung eines Commits ohne Kommentar durch das Repository ist nicht möglich (*ohne „Kommentare erzwingen" (B.1.1.6.1) aus „Versionskontrolle", S. 106*).

5.4.2. GIT

Mit Hilfe des MOD2-SKM Merkmalsmodells lässt sich eine Konfiguration für GIT definieren. Die Konfiguration ist in Anhang D auf S. 441f. abgebildet. **GIT** wird seit 2005 entwickelt und ist als freie Software unter GNU Lizenz verfügbar. Es ist auf die Anforderungen der Entwicklungsprozesse für den Linux-Kernel abgestimmt. GIT ist in C implementiert und wird über die Kommandozeile (*„Kommandozeile" (O.2.4) aus „Client für Arbeitsbereich", S. 111*) mit Hilfe von Kommandos in Form von Shell-Skripten angesteuert.

GIT verwaltet die Entwicklungshistorie von Dateien und Verzeichnissen (*„Dateisystem" (B.2.2) aus „Produktmodelle", S. 108*, *„Komplexe Elemente" (D.5.2) aus „Versionierte Elemente", S. 117*) und speichert auch ihre Beziehungen (*„Referenzen komplexer Elemente" (D.4.3) aus „Speicher", S. 116*). Es verschränkt dabei den Produkt- und Versionsraum versionszentriert (*„Versionszentriert" (D.7.2) aus „Zusammenspiel Produkt-/Versionsraum", S. 118*). Die Historie unterstützt das Verzweigen (*„Verzweigen" (D.6), S. 117*) und das explizite Zusammenführen (*„„Verschmelzen protokollieren" (D.11.2.3.3) aus „Synchronisation", S. 120*) von Versionen. Jede Version wird dabei mit Hilfe ihres SHA-1-Hashfunktionswerts identifiziert (*„Hashwert" (D.2.2.1) aus „Identifikation Versionsraum", S. 114*). Um Platz zu sparen, werden die Versionen komprimiert (*„Kompression" (D.4.2) aus „Speicher", S. 116*) und der Inhalt unter seinem Hashwert gespeichert, d.h. er wird nur einmal abgelegt, selbst wenn er in verschiedenen Dateien des Dateisystems vorkommt (*„BLOb" (D.4.3.1) aus „Speicher", S. 116*). Zusätzlich lässt sich der Speicherplatzbedarf einer explizit angegebenen Versionsmenge durch Delta-Speicherung reduzieren (*„Gemischte Deltas" (D.4.1.1.1) aus „Speicher", S. 116*). Konflikte werden durch Verschmelzen behoben (*„Verschmelzen" (D.11.2.3) aus „Synchronisation", S. 120*).

Jeder Benutzer besitzt für seinen Arbeitsbereich ein lokales Repositorium (*„Vollständiges Repositorium“ (B.3.1) aus „Offline-Bedienbarkeit“, S. 109*), das vollständig repliziert (*„Replikations-Unterstützung“ (B.4.3) aus „Repositorium“, S. 110*) und mit anderen Repositories synchronisiert (*„Verteilt“ (D.10.2) aus „Repositoriums-Architektur“, S. 120*) werden kann (sowohl mit einem zentralen Repository als auch einem Repository eines anderen Entwicklers), d.h. GIT unterstützt Peer-to-Peer-Synchronisation (*„Peer-to-Peer-Unterstützung“ (B.4.2) aus „Repositorium“, S. 110*). Auch die Offline-Synchronisation durch per E-Mail versandte Patches wird unterstützt (*„E-Mail“ (I.3.1) aus „Netzwerkprotokolle“, S. 122*). GIT bietet keinen Auslöser zur Erweiterung (*ohne „Auslöser“ (I.1), S. 121*) oder eine spezielle Unterstützung für weitere Aktivitäten von Entwicklungsprozessen.

5.4.3. Mercurial

Mit Hilfe des MOD2-SKM Merkmalsmodells lässt sich eine Konfiguration für Mercurial definieren. Die Konfiguration ist in Anhang D auf S. 443f. abgebildet. **Mercurial** wird seit 2005 entwickelt und ist als freie Software unter GNU Lizenz verfügbar. Es ist in der Programmiersprache *Python* implementiert, so dass Mercurial mit unter 30.000 Codezeilen einen deutlich geringeren Umfang besitzt, als die drei anderen, in C implementierten, SKMS. Außer einer Kommandozeilen-Schnittstelle (*„Kommandozeile“ (O.2.4) aus „Client für Arbeitsbereich“, S. 111*) besitzt Mercurial auch einen Browser-Client (*„Browser-Schnittstelle“ (O.2.1) aus „Client für Arbeitsbereich“, S. 111*).

Die von Mercurial verwalteten Elemente sind nur Dateien (*„Dateisystem“ (B.2.2) aus „Produktmodelle“, S. 108*), während Verzeichnisse nicht explizit versioniert werden, sondern implizit über den Dateinamen (*„Atomare Elemente“ (D.5.1) aus „Versionierte Elemente“, S. 117*). Dieser wird um den relativen Pfad zum Wurzelverzeichnis des Arbeitsbereichs erweitert und fehlende Verzeichnisse so bei Bedarf angelegt oder gelöscht. Jeder Benutzer besitzt ein lokales Repositorium (*„Vollständiges Repositorium“ (B.3.1) aus „Offline-Bedienbarkeit“, S. 109*), das repliziert (*„Replikations-Unterstützung“ (B.4.3) aus „Repositorium“, S. 110*) und mit anderen Repositorien synchronisiert werden kann (*„Verteilt“ (D.10.2) aus „Repositoriums-Architektur“, S. 120*). Ein zentrales Repository ist nicht erforderlich. Die Historie verwaltet die Beziehungen unter den Dateien nur implizit über das versionszentrierte Zusammenspiel (*„ohne „Referenzen komplexer Elemente“ (D.4.3)“ aus „Speicher“, S. 116*) und verwendet – analog zu GIT – für die Identifikation einer Version den Hashfunktionswert einer Version (*„Hashwert“ (D.2.2.1) aus „Identifikation Versionsraum“, S. 114*). Es besteht die Möglichkeit, Zweige anzulegen (*„„Verzweigen“ (D.6)’, S. 117*) und explizit zusammenzuführen (*„Verschmelzen protokollieren“ (D.11.2.3.3) aus „Synchronisation“, S. 120*). Konflikte werden mittels Verschmelzen aufgelöst (*„Verschmelzen“ (D.11.2.3) aus „Synchronisation“, S. 120*). Die versionierten Dateien werden mit Hilfe von Vorwärtsdeltas platzsparend abgelegt (*„Vorwärts-Deltas“ (D.4.1.1.3) aus „Speicher“, S. 116*). Mercurial kann, wie Subversion, mit Hilfe von Auslösern erweitert werden (*„Auslöser“ (I.1), S. 121*), die Python-Code ausführen. Eine spezielle Unterstützung für Aktivitäten von Entwicklungsprozessen ist nicht vorgesehen.

5.4.4. Subversion

Mit Hilfe des MOD2-SKM Merkmalsmodells lässt sich eine Konfiguration für Subversion definieren. Die Konfiguration ist in Anhang D auf S. 445f. abgebildet. **S**ub**v**ersio**n** (SVN) wird seit 2000 entwickelt und ist als freie Software unter Apache Lizenz erhältlich. Es ist Ziel des Projektes „ein neues SKMS zu implementieren, das die grundlegenden Konzepte von CVS übernimmt, ohne seine Fehler zu kopieren.“ [CSFP08]. Subversion ist als Sammlung gemeinsam nutzbarer C-Bibliotheken strukturiert und besitzt eine explizit definierte Schnittstelle, um die Integration von SVN in andere Software-Systeme zu erleichtern. Daher existieren, außer dem Kommandozeilen-Client (*„Kommandozeile“ (O.2.4) aus „Client für Arbeitsbereich“, S. 111*), auch grafische Clients (*„Grafischer Client“ (O.2.2) aus „Client für Arbeitsbereich“, S. 111*) oder ein Client für die Entwicklungsumgebung *Eclipse* (*„Integration in Eclipse“ (O.2.3) aus „Client für Arbeitsbereich“, S. 111*).

SVN verwaltet die Historien von Dateien und Verzeichnissen (*„Dateisystem“ (B.2.2) aus „Produktmodelle“, S. 108*). Diese werden über ihren Namen identifiziert (*„Elementname“ (D.8.1.1) aus „Identifikation Produktraum“, S. 119*), so dass spezielle Operationen zum Umbenennen und Verschieben existieren, um die Historie nicht zu unterbrechen (*„Versionskontroll-Operationen“ (B.1.2) aus „Versionskontrolle“, S. 106*). Die Zusammenarbeit der Entwickler wird durch ein zentrales Repository unterstützt (*„Client-Server“ (D.10.1) aus „Repositoriums-Architektur“, S. 120*), das auch repliziert (aber nicht synchronisiert) werden kann (*„Replikations-Unterstützung“ (B.4.3) aus „Repositorium“, S. 110*). Konflikte lassen sich entweder pessimistisch über Sperren verhindern (*„Sperren“ (D.11.2.2) aus „Synchronisation“, S. 120*), oder optimistisch über Verschmelzen auflösen (*„Verschmelzen“ (D.11.2.3) aus „Synchronisation“, S. 120*). Das Zusammenspiel zwischen Produkt- und Versionsraum ist versionszentriert (*„Versionszentriert“ (D.7.2) aus „Zusammenspiel Produkt-/Versionsraum“, S. 118*), und die Versionen werden über die flache, numerische Versions-ID des Wurzelverzeichnisses identifiziert (*„Flach“ (D.2.2.2.1) aus „Identifikation Versionsraum“, S. 114*). Die Historie bietet die Möglichkeit zum Verzweigen (*„Verzweigen“ (D.6), S. 117*), während das Zurückführen von Zweigen nur unzureichend implementiert ist (*ohne „Verschmelzen protokollieren“ (D.11.2.3.3) aus „Synchronisation“, S. 120*). Verändern sich nicht alle Dateien in einem Verzeichnis, so werden – um Speicherplatz zu sparen – die vorhanden (unveränderten) Elemente referenziert (*„Referenzen komplexer Elemente“ (D.4.3) aus „Speicher“, S. 116*). Zusätzlich werden veränderte Dateien als gemischte Deltas abgelegt (*„Gemischte Deltas“ (D.4.1.1.1) aus „Speicher“, S. 116*) und so der Speicherplatzbedarf weiter reduziert.

Weitere Aktivitäten von Entwicklungsprozessen werden nicht speziell unterstützt, jedoch ist SVN als Bibliothek in Werkzeuge integrierbar, die andere Aktivitäten unterstützen. Zur Erweiterung von SVN werden „Auslöser“ verwendet (*„Auslöser“ (I.1), S. 121*): Vor oder nach fest definierten Aufgabenschritten im Synchronisierungsprotokoll[1] kann selbst implementierter Programmcode ausgeführt werden, z.B. ein Auslöser, der nach Abschluss einer Übertragung von Änderungen in das Repository ein selbst implementiertes Skript startet, um eine E-Mail-Benachrichtigung zu versenden.

[1] Es gibt insgesamt 10 Auslöser: Vor, nach und beim Auslösen eines Einspielvorgangs, vor und nach Änderung eines Meta-Attributs, sowie vor und nach dem Blockieren bzw. Freigeben eines Elements.

5.5. Stand der Umsetzung

Die MOD2-SKM Merkmals-Analyse besitzt einen größeren Umfang als das Domänenmodell der MOD2-SKM Produktlinie. Dies liegt insbesondere daran, dass der Aufwand, ein Merkmal im Domänenmodell zu modellieren, deutlich höher ist, als das entsprechende Merkmal zu identifizieren, zu beschreiben und in die Merkmals-Analyse aufzunehmen. Daher besitzt jedes Merkmal in seiner Beschreibung die Eigenschaft „Umsetzung“ (vgl. Abschnitt 5.2.2, S. 103). Diese Eigenschaft besitzt jeweils einen der drei folgenden Werte:

1. Vollständig implementiert,
2. Teilweise implementiert,
3. Erweiterungspunkt.

Ein Merkmal gilt als **vollständig implementiert**, wenn es im Domänenmodell realisiert ist. D.h. es ist mindestens einem Modellelement zugeordnet. Ein Elternmerkmal gilt dann als vollständig implementiert, wenn alle seine Untermerkmale vollständig implementiert sind. Tabelle E.1 im Anhang (s. S. 447) zeigt die Liste aller vollständig implementierten Merkmale. Ein Merkmal ist ein **Erweiterungspunkt**, wenn es nicht im Domänenmodell realisiert wird und weder es selbst, noch seine Untermerkmale einem Modellelement zugeordnet sind. Ein Elternmerkmal ist somit nur ein Erweiterungspunkt, wenn alle seine Kindmerkmale Erweiterungspunkte sind. Tabelle E.4 im Anhang (s. S. 449) zeigt die Liste aller Erweiterungspunkte. Ein Merkmal gilt als **teilweise implementiert**, wenn seine Untermerkmale sowohl vollständig implementierte als auch teilweise implementierte bzw. Erweiterungspunkte enthalten. Tabelle E.2 im Anhang (s. S. 448) zeigt die Liste aller teilweise implementierten Merkmale. Zusätzlich existieren noch **Hilfsmerkmale**, die lediglich dazu dienen, die Auszeichnung des Domänenmodells zu erleichtern. Sie fügen der Analyse keine neuen Konzepte hinzu, sondern erleichtern die Umsetzung der bereits existierenden Merkmale. Tabelle E.5 im Anhang (s. S. 449) zeigt die Liste aller Hilfsmerkmale.

Abbildung 5.28 zeigt die prozentuale Verteilung der Zustände über die 147 Merkmale. 30% (54 Merkmale) sind bereits vollständig implementiert. 11% (15 Merkmale) sind noch in Arbeit, d.h. sie werden im Rahmen von Abschlussarbeiten oder Praktika realisiert, die noch nicht beendet sind. Bei erfolgreichem Abschluss sind diese Merkmale dann ebenfalls vollständig implementiert. 26% (33 Merkmale) der Merkmale sind nur teilweise realisiert, d.h. sie sind Elternmerkmale der 45 Erweiterungspunkte, die 29% der Merkmale ausmachen. Lediglich 4% der Merkmale sind Hilfsmerkmale, die lediglich für die Auszeichnung des Domänenmodells benötigt werden.

Bei 30% vollständig implementierten Merkmalen liegt es nahe zu sagen, dass nur ein Drittel des Merkmalsmodells realisiert wurde. Es ist jedoch zu beachten, dass ein teilweise implementiertes Merkmal sowohl Erweiterungspunkte als auch – vollständig oder teilweise – implementierte Untermerkmale besitzt. Um als vollständig implementiert zu gelten, müssen erst alle Untermerkmale vollständig implementiert sein. Der Umsetzungsstand

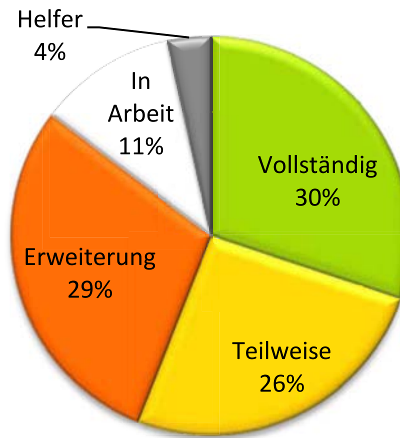


Abbildung 5.28.: Verteilung der Merkmale über die Umsetzungszustände

der teilweise implementierten Merkmale hängt also direkt von den Erweiterungspunkten ab: Sind alle Erweiterungspunkte realisiert, dann sind alle teilweise implementierten Merkmale vollständig implementiert. Da der Anteil der Erweiterungspunkte in etwa dem Anteil der vollständig implementierten Merkmale entspricht, bedeutet dies, dass das Merkmalsmodell zu etwa 50% realisiert wurde. Die Merkmale unterscheiden sich teilweise jedoch deutlich im Modellierungsaufwand untereinander, so dass sich hieraus keine Rückschlüsse auf den zu erwartenden Entwicklungsaufwand ziehen lassen.

5.5.1. Umsetzung der Spitzenmerkmale

Abbildung 5.29 zeigt den Stand der Umsetzung der Spitzenmerkmale der MOD2-SKM Produktlinie. Auf der x-Achse sind – nach Ebenen sortiert – die Spitzenmerkmale in Reihenfolge der Gliederung aus Abbildung 5.4 aufgetragen. Auf der y-Achse befindet sich die Anzahl der Untermerkmale. Die Gesamtlänge eines Balkens entspricht somit der vollständige Anzahl der Untermerkmale des jeweiligen Spitzenmerkmals. Der Balken ist zusätzlich noch farblich unterteilt und zeigt so die Anzahl der Merkmale mit einem bestimmten Umsetzungszustand an. Die Zuordnung der Farben zu den Umsetzungszuständen befindet sich in der Legende unterhalb des Diagramms. Die Segmentierung von links nacht rechts entspricht der Reihenfolge der Umsetzungszustände: „vollständig implementiert“, „teilweise implementiert“, „Erweiterungspunkt“ und „in Arbeit". An der Länge eines Segments ist die Anzahl der jeweiligen Untermerkmale im jeweiligen Umsetzungszustand ablesbar. Fehlt ein Segment vollständig, existiert kein Untermerkmal in diesem Zustand.

Umsetzung der Befähigungs-Ebene

Die Untermerkmale des umfangreichste Spitzenmerkmals „Versionskontrolle“ verteilen sich relativ gleichmäßig über alle Entwicklungszustände. Werden die laufenden Arbeiten abgeschlossen, ist etwa die Hälfte alle Versionskontroll-Operationen realisiert. Zwei

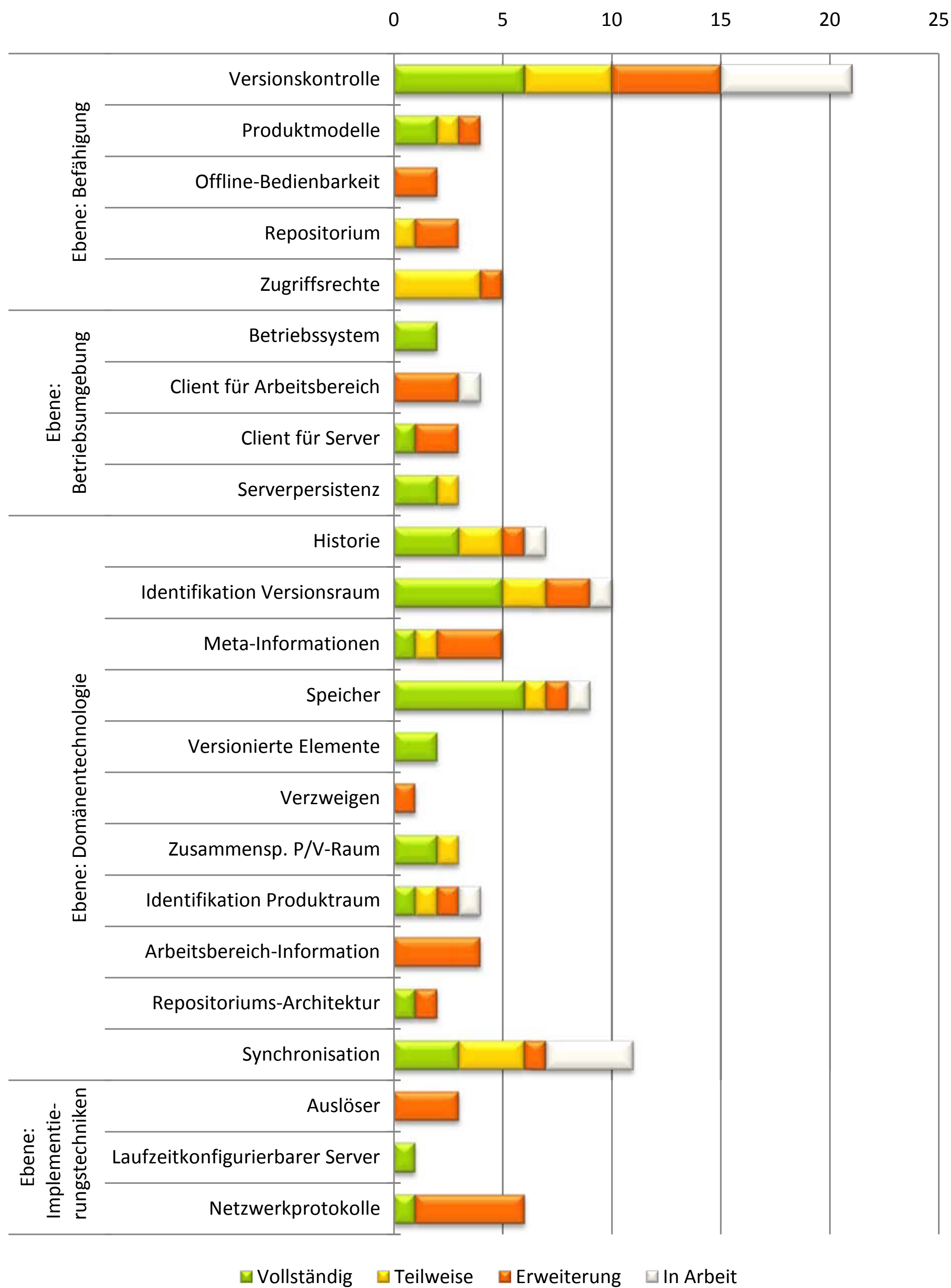


Abbildung 5.29.: Anzahl Untermerkmale und Umsetzung, je Spitzenmerkmal

„Produktmodelle" sind bereits realisiert, während die Spitzenmerkmale „Offline-Bedienbarkeit" und „Repositorium" größtenteils aus Erweiterungspunkten bestehen. Dabei handelt es sich vor allem um Merkmale für diverse Architekturen verteilter Repositorien, die einen hohen Modellierungsaufwand besitzen (vgl. Abschnitt 4.4.1, S. 92). Aus dem selben Grund sind die Zugriffsrechte nur teilweise realisiert, da nur eine rudimentäre Benutzerverwaltung realisiert wurde.

Umsetzung der Betriebsumgebungs-Ebene

Die Spitzenmerkmale „Betriebssystem" und „Serverpersistenz" sind fast vollständig realisiert. Das ist bei Ersterem leicht, da es lediglich die Informationen über das Betriebssystem enthält, die als Voraussetzung für andere Merkmale dienen. Letzteres besteht aus drei – bereits existierenden – Persistenzmechanismen für Java-Objekte, die an MOD2-SKM angepasst und nicht vollständig neu modelliert wurden. Die Modellierung eines Clients zur Verwaltung des Arbeitsbereichs bzw. des Server hingegen gestaltet sich sehr aufwändig, so dass jeweils nur ein Untermerkmal von „Client für Arbeitsbereiche" und „Client für Server" realisiert wurde.

Umsetzung der Domänentechnologie-Ebene

Die Merkmale im Bereich Domänentechnologie sind größtenteils vollständig realisiert. Es existieren mehrere graphbasierte „Historien": von einer einfachen Menge, über Sequenzen bis zu Graphen mit Verzweigungs- und Verschmelzungskanten. Mittels verschiedener Identifikationsverfahren aus „Identifikation Versionsraum" werden die Versionen referenziert, z.B. hierarchische Nummerierungen oder Hashwerten. Mehrere „Speicher"-Verfahren stehen zur Verfügung, um die Software-Elemente mit Hilfe von Delta-Verfahren platzsparend abzulegen bzw. um die Beziehungen zwischen den Elementen zu speichern. Dazu wurden auch alle drei Arten des „Zusammenspiels Produkt/Versionsraum" umgesetzt, d.h. eine Konfiguration lässt sich sowohl produktzentriert, versionszentriert als auch verwoben abrufen. Lediglich ein spezielles Verfahren zum „Verzweigen" wurde nicht umgesetzt, sowie auf den größten Teil der „Meta-Informationen" verzichtet, da diese von anderen Merkmalen, wie der Benutzerverwaltung, benötigt werden, die ebenfalls nur teilweise bzw. als Erweiterungspunkt realisiert sind. Die Modellierung dieser Informationen ist zwar leicht, hätte aber aus dem eben genannten Grund das Modell mit nicht verwendeten Elementen vergrößert.

Im Anforderungsbereich Struktur stehen mehrere Verfahren für die „Identifikation Produktraum" zur Auswahl, z.B. Elementnamen oder Hashwert. Im Bereich „Team" existieren mehrere Verfahren zu „Synchronisation" wie Sperren oder Verschmelzen. Das Spitzenmerkmal „Arbeitsbereich-Information" ist dagegen ein reiner Erweiterungspunkt: Er enthält diverse Daten, die im Arbeitsbereich zwischengespeichert werden, z.B. Benutzerdaten zum Anmelden an der Benutzerverwaltung. Da diese Daten jedoch nur von Erweiterungspunkten benötigt werden, wurde auf ihre Modellierung verzichtet, da sie – wie beim Spitzenmerkmal „Meta-Informationen" – nur den Umfang des Domänenmodells erhöht hätten, ohne verwendbare Funktionalität zu liefern.

Umsetzung der Implementierungstechniken-Ebene

Sowohl die „Auslöser“ als auch ein großer Teil der Untermerkmale der „Netzwerkprotokolle“ sind Erweiterungspunkte, da es sich in beiden Fällen – wie der Name der Ebene bereits ausdrückt – um Details der Implementierung handelt, deren Details auf die Modellierung der SKM-Konzepte aus Kapitel 4 nur einen geringen Einfluss haben. So ist z.B. für die Client-Server-Architektur ein Netzwerkprotokoll nötig, aber die Wahl des konkreten Protokolls von geringer Bedeutung. Stattdessen lag der Schwerpunkt der Modellierung auf dem „Laufzeitkonfigurierbaren Server“, der die Konfiguration ausgewählter Merkmale zur Laufzeit unterstützt, und so das automatisierte Testen eines Teils der Produktlinie ermöglicht.

5.6. Merkmalsmarkierungen im Domänenmodell

Das Merkmalsmodell erfüllt eine Doppelfunktion: Zum einen dient es zur Klassifikation bestehender SKMS, zum anderen werden die Merkmale zur Konfiguration eines SKMS verwendet (vgl. Abschnitt 3, S. 51ff.). Dazu werden die im Domänenmodell modellierten Variationspunkte mit Merkmalsmarkierungen ausgezeichnet. Das vollständige Merkmalsmodell besteht aus 147 Merkmalen und 34 aussagenlogischen Einschränkungen, doch nicht alle Merkmale werden als Merkmalsmarkierungen verwendet. 45 Merkmale sind Erweiterungspunkte, die lediglich zur Klassifikation dienen, und nicht implementiert wurden (vgl. Abschnitt 5.5, S. 129ff.). Von den verbliebenen 102 Merkmalen sind jedoch nur 45 Merkmale für Merkmalsmarkierungen verwendet worden. Dies hat mehrere Gründe: Erstens beeinflussen einige Merkmale über die aussagenlogischen Einschränkungen die Konfiguration. Zweitens dienen verpflichtende Merkmale zur Strukturierung der variablen Merkmale, wurden aber nicht zum Markieren verwendet.

Beeinflussung durch Einschränkungen

Die Auswertung der aussagenlogischen Einschränkungen eines Merkmalsmodells erfolgt während der Konfiguration. Wird ein Merkmal an- oder abgewählt, das Teil einer Einschränkungen ist, so werden die restlichen Merkmale ebenfalls konfiguriert, so dass der aussagenlogische Ausdruck den Wert „wahr“ ergibt (vgl. Abschnitt 3, S. 51ff.). Von den 34 aussagenlogischen Einschränkungen enthalten 30 Ausdrücke mindestens ein Merkmal, das als Markierung verwendet wird. Alle Merkmale in diesen Ausdrücken beeinflussen so das Domänenmodell indirekt, indem sie die Konfiguration zur Markierungen verwendeter Merkmale unterstützen.

Verpflichtende Merkmale

Verpflichtende Merkmale sind immer Teil einer Konfiguration und dienen insbesondere der Strukturierung des Merkmalsmodells (vgl. Abschnitt 3, S. 51ff.). Da sie in jeder Konfiguration enthalten sind, beeinflussen sie das konfigurierte SKMS nicht. Solche Merkmale können ebenfalls für Markierungen im Domänenmodell verwendet werden, besitzen dort

jedoch lediglich Kommentarcharakter, da mit ihnen markierte Elemente – wie unmarkierte – in jeder Konfiguration enthalten sind. Zwar wurde im MOD2-SKM Domänenmodell auf die Auszeichnung mit verpflichtenden Merkmalen verzichtet, doch zeigt sich nun, dass ihre Existenz das Verständnis für die Zusammenhänge zwischen Merkmals- und Domänenmodell erleichtert hätte.

5.6.1. Untersuchung der Merkmalsmarkierungen

Abbildung 5.30 und Abbildung 5.31 zeigen ein reduziertes Merkmalsmodell (vgl. Abschnitt 5, S. 95ff.), das nur aus den Merkmalen besteht, die zur Markierung eingesetzt wurden. Um die Struktur des Modells zu wahren, sind verpflichtende Merkmale übernommen worden, falls ihre Kindmerkmale für Markierungen benutzt werden. Auch die Einteilung in verpflichtend und optional blieb bestehen, auch wenn nur noch eine Auswahlmöglichkeit für ein optionales Merkmal existiert (z.B. bei „Benutzerverwaltung“). Letztendlich sind 75 Merkmale erhalten geblieben, von denen 45 optionale Merkmale für Merkmalsmarkierungen verwendet wurden. D.h. alle optionalen Merkmale im reduzierten Modell – außer „Zustandsbasiert“, „ID-Auflösung“, „Numerisch“ und „Deltas“ – dienen als Auszeichnung im Domänenmodell.

Die Merkmale stammen aus allen vier Ebenen der FORM-Methode, allein jedoch zwei Drittel aus der Ebene Domänentechnologie. Diese Beobachtung geht konform mit der Beschreibung dieser Ebene als Sicht eines SKMS-Entwicklers auf die Produktlinie: Merkmale, die einen SKMS-Entwickler interessieren, steuern die Konfiguration des Domänenmodells. Diese Merkmale betreffen vor allem die Konfiguration des Server-Teilsystems. Auf der Befähigungsebene (der Benutzersicht) steuern die Merkmale hingegen die Konfiguration des Arbeitsbereich-Teilsystems sowie der im SKMS gespeicherten Daten, d.h. Konzepte, mit denen ein Benutzer direkt in seinem Arbeitsbereich interagiert.

Hilfsmerkmale

Zusätzlich existiere noch 5 Hilfsmerkmale (s. Abb. 5.32), die ebenfalls zum Auszeichnen verwendet wurden. Sie dienen aber lediglich dazu, die Negation eines bereits existierenden Merkmals zu erfassen. Im MODPL-Werkzeugkasten bedeutet das Markieren eines Modellelements, dass dieses nur in einem konfigurierten SKMS existiert, wenn das Merkmal Teil der Konfiguration ist [Buc10]. Soll ein Modellelement existieren, wenn ein Merkmal nicht in einer Konfiguration ist – z.B. als Alternative zu einem Modellelement mit ausgewähltem Merkmal – lässt sich dies nicht modellieren, da keine Markierung mit Negation existiert. Um diese Einschränkung zu umgehen, führe ich Hilfsmerkmale ein, die über eine aussagenlogische Verknüpfung an das betroffene Merkmal gekoppelt sind und immer den gegenteiligen Wert annehmen.

Größe der Produktlinie

Für das reduzierte Merkmalsmodell existieren insgesamt **290.304.000** Konfigurationen, d.h. ebenso viele unterschiedliche SKMS lassen sich aus der MOD2-SKM Produktlinie erzeugen. Die Merkmale besitzen jedoch nicht alle den gleichen Umfang bzw. gleiche

Abbildung 5.30.: Merkmalsmodell, auf Merkmalsmarkierungen reduziert

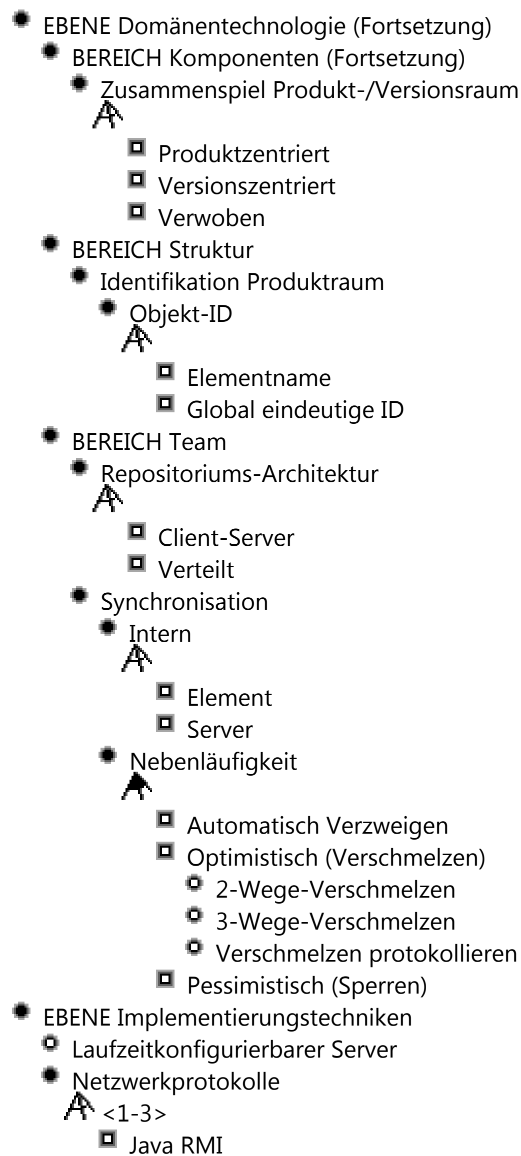


Abbildung 5.31.: Merkmalsmodell, auf Merkmalsmarkierungen reduziert (Fortsetzung)

- Hilfs-Merkmale
  - NOT Kompression
  - NOT Lauzeitkonfigurierbarer Server
  - NOT Komplexe Element-Referenzen
  - NOT JavaRMI
  - NOT Automatisch Verzweigen

Abbildung 5.32.: Hilfsmerkmale

Granularität. So steuert ein Merkmal das Erzwingen von Kommentaren („Kommentare erzwingen“, s. Abb. 5.30(a), S. 135), während ein anderes Merkmal die Integration von Produkt- und Versionsraum bestimmt („Verwoben“, s. Abb. 5.31, S. 136). Dennoch ist die Produktlinie damit so groß, dass sich nicht mehr alle Konfigurationen erzeugen und geschweige denn testen lassen. Lediglich mit Hilfe des laufzeitkonfigurierbaren Servers können ca. 1.000 Kombinationen von Servermodulen getestet werden (vgl. Abschnitt 6.4.20, S. 273ff.).

5.6.2. Untersuchung der aussagenlogischen Einschränkungen

Durch Merkmalsgruppierungen lassen sich bereits durch die Baumstruktur des Merkmalsmodells Abhängigkeiten ausdrücken und z.B. sich gegenseitig ausschließende Merkmale modellieren. Zusätzlich existiert noch die Möglichkeit, aussagenlogische Einschränkungen in Form von „Wenn-Dann-Klauseln“ anzugeben (vgl. Abschnitt 3.2, S. 54ff.). Im MOD2-SKM Merkmalsmodell existieren 34 dieser Einschränkungen. Dabei ist insbesondere von Interesse, inwieweit hier zusätzliche Abhängigkeiten erfasst werden.

In einem FODA-Merkmalsmodell repräsentiert jede Einschränkung eine Abhängigkeit [KCH+90]. Im MOD2-SKM Merkmalsmodell werden jedoch die aussagenlogischen Einschränkungen zusätzlich noch (1) zur Modellierung der „Implementierungs“-Beziehung zwischen den FORM-Ebenen und (2) zur Konfiguration der Hilfsmerkmale verwendet (vgl. Abschnitt 5.2, S. 100ff.). Die 34 Einschränkungen teilen sich wie folgt auf: 9 dienen entweder als Notlösung für „Implementierungs-Abhängigkeiten“ (Neun Klauseln) oder für Hilfsmerkmale (10 Klauseln). Damit verbleiben 15 Klauseln, die folgende – zur Baumstruktur orthogonale – Abhängigkeiten ausdrücken:

1. Wenn „Nach Datum“ (D.2.1.1), dann „Datum“ (D.3.3)
2. Wenn „Neuste Version“ (D.2.1.3), dann „Datum“ (D.3.3)
3. Wenn **ohne** „Nach Datum“ (D.2.1.1) und **ohne** „Neuste Version“ (D.2.1.3), dann **ohne** „Datum“ (D.3.3)
4. Wenn „Nach Markierung“ (D.2.1.2), dann „Markierungen“ (D.3.5)
5. Wenn **ohne** „Nach Markierung“ (D.2.1.2), dann **ohne** „Markierungen“ (D.3.5)
6. Wenn „Verzweigen“ (D.6), dann **ohne** „Einfache Menge“ (D.1.2.1.2) und **ohne** „Sequenz“ (D.1.2.1.4)
7. Wenn **ohne** „Verzweigen“ (D.6), dann **ohne** „Baum“ (D.1.2.1.1) und **ohne** „Verschmelzen protokollieren“ (D.11.2.3.3)
8. Wenn „Verschmelzen protokollieren“ (D.11.2.3.3), dann „Verzweigen“ (D.6) und „Gerichteter azyklischer Graph“ (D.1.2.1.3)
9. Wenn **ohne** „Verschmelzen protokollieren“ (D.11.2.3.3), dann **ohne** „Gerichteter azyklischer Graph“ (D.1.2.1.3)

10. Wenn „BLOb“ (D.4.3.1), dann „Hashwert“ (D.8.1.3)
11. Wenn „Referenzen komplexer Elemente“ (D.4.3), d. „Komplexe Elemente“ (D.5.2)
12. Wenn „Komplexe Elemente“ (D.5.2), d. „Referenzen komplexer Elemente“ (D.4.3)
13. Wenn „Produktzentriert“ (D.7.1), dann **ohne** „Komplexe Elemente“ (D.5.2)
14. Wenn „Verwoben“ (D.7.3), dann „Komplexe Elemente“ (D.5.2) und **ohne** „Element“ (D.11.1.1)
15. Wenn „Versionszentriert“ (D.7.2), dann „Element“ (D.11.1.1)

Die ersten fünf Klauseln stellen sicher, dass Verfahren zur Identifikation von Versionen auch die notwendigen Meta-Informationen vorfinden. Um die neuste Version bzw. eine Version anhand eines Datums auszuwählen, wird als Meta-Information das Datum gespeichert, an dem die Version in das Repositorium eingespielt wurde (*Klausel 1* und *2*). Für Markierungen (engl. tags) muss dagegen die Markierung selbst gespeichert werden (*Klausel 4*). Wird kein Verfahren verwendet, das eine entsprechende Meta-Information benötigt, so ist diese nicht notwendig (*Klausel 3* und *5*).

Die Klauseln sechs bis neun stellen sicher, dass zum Verzweigen und Verschmelzen die notwendigen expliziten Abhängigkeiten im Versionsraum existieren (vgl. Abschnitt 4.3.3, S. 80ff.). Beim Verzweigen lässt sich keine Historie ohne Alternativ-Beziehung verwenden (*Klausel 6*). Ohne Verzweigen ist aber auch diese Beziehung unnötig – ebenso wie eine explizite Verschmelzen-Beziehung beim Zurückführen von Zweigen (*Klausel 7*). Wird dies gewünscht, dann muss natürlich auch die Möglichkeit zum Verzweigen bestehen und ein azyklischer Graph als Historie dienen (*Klausel 8*). Diese Historie wird auch nur für die Rückführung von Zweigen benötigt (*Klausel 9*).

Klausel 10 bis 15 stellen sicher, dass bei atomaren bzw. zusammengesetzten Elementen nur Speicher- und interne Sperrverfahren verwendet werden, die auch mit diesen Elementen funktionieren. Das BLOb-Speicherverfahren setzt z.B. einen Hashwert als Objektkennung voraus (*Klausel 10*) [Ca09]. Zusammengesetzte (komplexe) Produktmodell-Elemente lassen sich nur effizient mit einem Speichermechanismus speichern, der Referenzen auf existierende Versionen unterstützt (*Klausel 11* und *12*). Komplexe Elemente sind eine Voraussetzung für eine versionszentrierte oder verwobene Integration von Produkt- und Versionsraum (vgl. Abschnitt 4.3.4, S. 81ff.) (*Klausel 14* und *15*), während bei produktzentrierter Integration nur atomare Elemente nötig sind (*Klausel 13*).

Klauseln im Merkmalsmodell oder Domänenmodell

Die Entwickler von Merkmals- und Domänenmodellen müssen entscheiden, ob sie eine Abhängigkeit zwischen zwei oder mehr Merkmalen entweder als Klausel im Merkmalsmodell oder über mehrere Merkmalsmarkierungen im Domänenmodell modellieren. Diese beiden Alternativen lassen sich z.B. anhand der Klauseln 1 und 2 illustrieren. Anstatt im Domänenmodell z.B. das Modellelement für die Datums-Meta-Information mit dem Merkmal „Datum“ (D.3.3) zu markieren, ließen sich auch die beiden Merkmale „Neuste

Version“ (D.2.1.3) und „Nach Datum“ (D.2.1.1) – über ein „oder“ verbunden – verwenden. Im Allgemeinen ist eine einzelne Merkmalsmarkierung mehreren vorzuziehen, da so die Abhängigkeiten ins Merkmalsmodell verlagert werden. Daraus lässt sich allerdings nicht ableiten, dass immer nur genau ein Merkmal zu Markierung genutzt wird. Kann ein Modul z.B. an mehreren Variationspunkten des konfigurierten Systems verwendet werden, dann lässt sich diese Mehrfachnutzung durch eine Veroderung der Merkmalsmarkierungen ausdrücken. Letztendlich müssen die Entwickler im Einzelfall entscheiden, ob sie mehrere Merkmalsmarkierungen verwenden, oder ob sie ein von ihnen abhängiges Merkmal einführen und zur Markierung nutzen. Das bedeutet, dass bei Merkmalsmarkierungen außer einem aussagenlogischen „nicht“ auch das „oder“ benötigt wird.

5.7. Zusammenfassung

Ob sich die SKM-Domäne überhaupt für den Einsatz einer Produktlinie eignet, wurde mit Hilfe einer Eignungsprüfung bestätigt [PBL05]. Danach erfolgt die Auswahl der geeigneten Quellen und Identifikation der Merkmale. Ergebnis der Analyse der SKM-Domäne nach dem MODPL-Prozess ist zum einen die Merkmals-Analyse in Anhang B (s. S. 435) und zum anderen ein Merkmalsmodell nach der FORM-Methode [KLD02], das in Anhang C (s. S. 435) gezeigt ist. Dabei wird deutlich, dass bereits für die Erstellung und Konfiguration des MOD2-SKM Merkmalsmodells deutlich komplexere Werkzeugunterstützung notwendig ist. Anhand der Konfiguration für die vier SKMS CVS, GIT, Mercurial und Subversion lässt sich das MOD2-SKM Merkmalsmodell validieren.

Auf Grund der aufwändigen und komplexen Modellierung einiger Merkmale, ist jedes Merkmal entweder vollständig oder teilweise implementiert – oder als Erweiterungspunkt gekennzeichnet, so dass festgestellt werden kann, dass ca. 50% der Merkmale im MOD2-SKM Domänenmodell umgesetzt sind, das im nun folgenden Kapitel beschrieben wird. Dies ist auch einer der Gründe, dass dort nur 45 der 147 Merkmale als Markierungen verwendet wurden. Trotzdem lassen sich aus diesem reduzierten Modell immer noch 290.304.000 unterschiedliche SKMS generieren.

6. Die MOD2-SKM Produktlinie

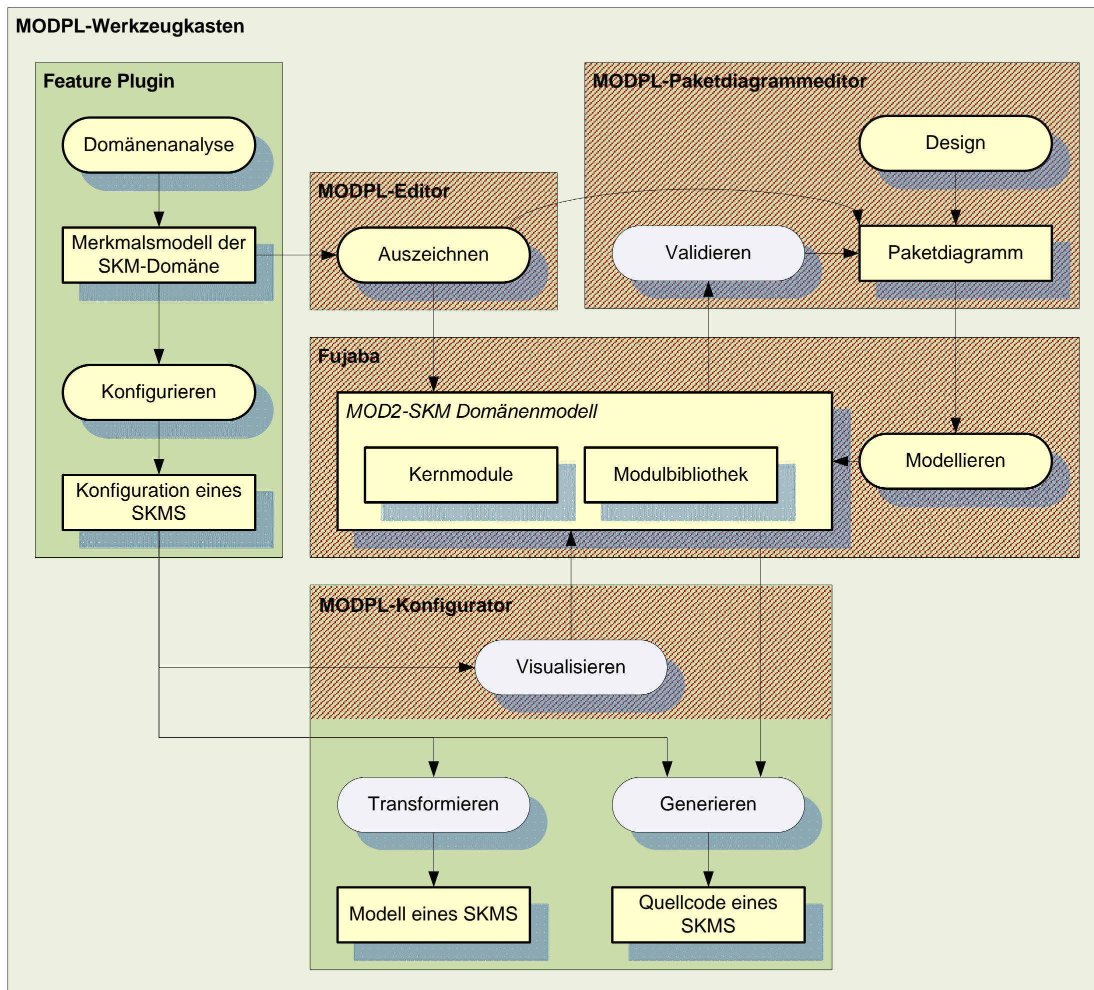


Abbildung 6.1.: Überblick über die MOD2-SKM Produktlinie

Das folgende Kapitel dokumentiert und präsentiert das Design und die Architektur des MOD2-SKM Domänenmodells. Das Domänenmodell besteht zum einen aus dem MOD2-SKM Paketdiagramm, das die Architektur der Produktlinie beschreibt und auch die Modularisierung der SKM-Konzepte anschaulich darstellt [Buc10]. Der zweite Teil des Domänenmodells ist das Fujaba-Modell, das die Struktur mittels Klassendiagrammen und das Verhalten mittels Storydiagrammen beschreibt [Zün02]. Beide Modelle model-

lieren sowohl gemeinsame als auch variable Teile der Produktlinie, so dass ein Teil der Elemente – wie in Kapitel 3 beschrieben – mit Merkmalskennungen aus dem MOD2-SKM Merkmalsmodell (vgl. Kapitel 5 und Anhang B) gekennzeichnet sind. Die dazu verwendeten Werkzeuge und Elemente im MODPL-Werkzeugkasten sind in Abbildung 6.1 durch eine Schraffur hervorgehoben.

6.1. Aufbau des MOD2-SKM Domänenmodells

Das MOD2-SKM Domänenmodell beschreibt die Klassenstruktur bzw. das Verhalten der einzelnen Methoden mit Hilfe von Klassen- bzw. Storydiagrammen in Fujaba. Es wird durch ein Paketmodell aus dem MODPL-Paketdiagrammeditor ergänzt, das die grundlegende Architektur der MOD2-SKM Produktlinie mit Hilfe von Paketdiagrammen beschreibt. Dabei besteht eine „1:1"-Verknüpfung zwischen den Paketen des MODPL-Paketmodells und des Fujaba-Modells (über den voll qualifizierten Namen). Zusätzlich ist auch jedes Klassendiagramm aus dem Fujaba-Modell genau einem Paket zugeordnet, dessen Inhalt es darstellt. Die Synchronisation der beiden Modelle erfolgt mittels des MODPL-Werkzeugkastens [Buc10].

Architektur: Paketdiagramme

Das MODPL-Paketdiagramm basiert auf einer vereinfachten Form des UML2-Paketdiagramms [OMG09b] [Buc10]. Jedes Paket stellt einen Namensraum dar, dessen Elemente einen eindeutigen **Elementnamen** besitzen müssen. In einem Paket können entweder weitere Pakete oder Klassen enthalten sein. Jedes Element ist über seinen **voll qualifizierten Namen** (die Konkatenation aller Paketnamen vom Wurzelpaket bis zum Element selbst) eindeutig referenzierbar. Weiterhin existieren Sichtbarkeiten (engl. visibilities), so dass sich ein Element nur mit seinem (kürzeren) Elementnamen referenzieren lässt – und nicht mit dem (längeren) voll qualifizierten Namen. Innerhalb eines Pakets können alle Elemente untereinander mittels des Elementnamens referenziert werden und ebenso alle Elemente der umgebenden Pakete [OMG09b].

Außerdem existieren zwei Arten von Import-Beziehungen zwischen Elementen, um ihre Sichtbarkeit auszudehnen. Ein **Element-Import** drückt aus, dass das Quellelement nur noch den Elementnamen des Zielelements verwenden muss und nicht mehr den voll qualifizierten Namen. Ein **Paket-Import** besagt, dass alle Element im Zielpaket vom Quellelement über den einfachen Namen referenziert werden können (was einem Element-Import auf alle Elemente des Pakets entspricht). Jede Import-Beziehung kann entweder als **privat** (engl. private) oder **öffentlich** (engl. public) deklariert werden. Öffentlich importierte Elemente sind auch für eingehende Import-Beziehungen sichtbar, d.h. sie können über mehrere Import-Beziehungen transitiv sichtbar sein. Private Import-Beziehungen sind dagegen nur im importierenden Element sichtbar und lassen sich nicht transitiv sichtbar machen [OMG09b]. Die ebenfalls in der UML2 vorhandene Paketverschmelzungs-Beziehung wird vom MODPL-Paketdiagrammeditor nicht unterstützt [Buc10].

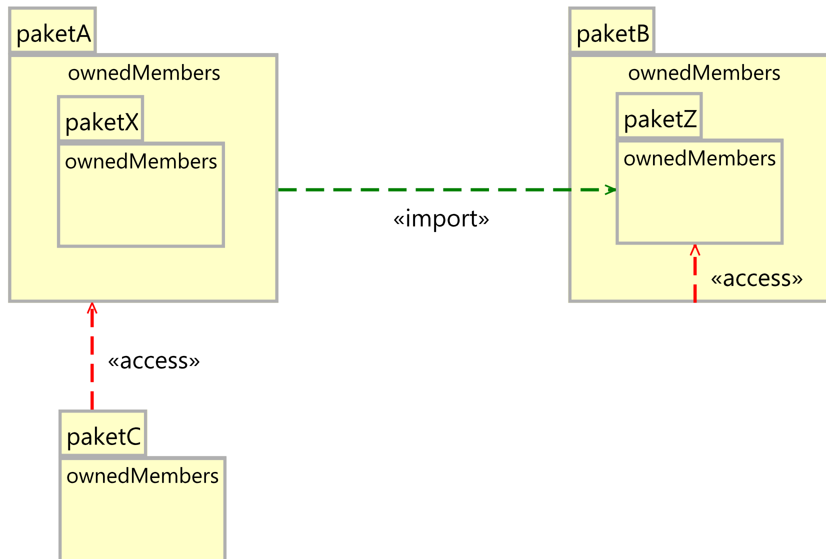


Abbildung 6.2.: Beispiel eines MODPL-Paketdiagramms

Abbildung 6.2 zeigt ein Beispiel für ein MODPL-Paketdiagramm. Auf der obersten Ebene sind die Pakete „paketA“, „paketB“ und „paketC“ deklariert. „paketX“ ist in „paketA“ und „paketZ“ in „paketB“. Beide „sehen“ damit auch die Elemente ihrer Elternpakete. Grüne Pfeile stellen öffentliche Import-Beziehungen dar und sind zusätzlich mit dem Stereotyp „import“ beschriftet. Rote Pfeile entsprechen privaten Import-Beziehungen und besitzen die Beschriftung „access“. „paketB“ „sieht“ somit auch die Elemente in „paketZ“, genauso wie „paketA“. „paketC“ kann sowohl auf die Elemente von „paketA“ als auch von „paketZ“ mit einfachem Namen zugreifen. Letzteres Paket ist auf Grund der öffentlichen Import-Beziehung sichtbar. Für eine detaillierte Beschreibung von Paketdiagrammen und Sichtbarkeitsregeln sei auf [HK99, OMG09b] verwiesen.

Besonderheiten von MODPL-Paketdiagrammen

Trotz Import-Beziehungen ist weiterhin die Verwendung von voll qualifizierten Namen möglich [OMG09b]. Dadurch sind dann jedoch die Abhängigkeiten zwischen den Elementen nicht mehr explizit (über die Import-Beziehungen) beschrieben [Buc10]. In MOD2-SKM wird auf die Verwendung voll qualifizierter Namen verzichtet, so dass über die Import-Beziehungen die Kopplung zwischen den Paketen bewertet werden kann (vgl. Import-Metrik in Abschnitt 6.5.2, S. 297).

Struktur: Klassendiagramme

Die Datenstrukturen von MOD2-SKM im Fujaba-Modell sind mit Hilfe von UML-Klassendiagrammen beschrieben [Zün02, OMG09b]. Jede Klasse ist dabei einem Paket aus dem oben beschriebenen MODPL-Paketmodell zugeordnet. Auch die Klassendiagramme

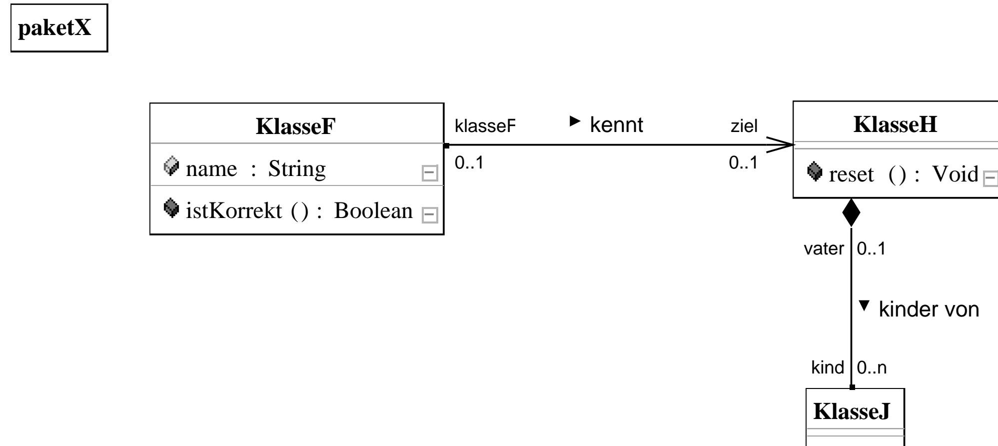


Abbildung 6.3.: Beispiel eines Fujaba-Klassendiagramms

lassen sich einem Paket zuordnen [Buc10], so dass in MOD2-SKM jedem Paket genau ein Klassendiagramm zugeordnet ist. Dies enthält alle Klassen des entsprechenden Pakets.

Die Elemente in den Fujaba-Klassendiagrammen orientieren sich an der UML-Spezifikation 1.4 [Zün02]. Jede Klasse besitzt einen eindeutigen Namen, Attribute und Methoden. Jedem Attribut wird ein Name, eine Sichtbarkeit (öffentlich, paketweit oder privat) und ein Typ zugeordnet. Methoden besitzen ebenfalls einen Namen, eine Sichtbarkeit, einen Rückgabetyp und eine abzählbare Menge an getypten Parametern. Zwischen den Klassen lassen sich Vererbungsbeziehungen und Assoziationen definieren. Eine Assoziation besitzt einen Namen, ist entweder gerichtet oder ungerichtet, und ist entweder generisch oder eine Aggregation oder sogar eine Komposition. Jede beteiligte Klasse nimmt eine Rolle in Bezug auf die Assoziation an. Jede Rolle besitzt ebenfalls einen Namen, eine Kardinalität und ist entweder unqualifiziert oder qualifiziert. Für eine ausführlichere Beschreibung von UML-Klassendiagrammen sei auf [HK99, Zün02, OMG09b] verwiesen.

Abbildung 6.3 zeigt ein Beispiel für ein Fujaba-Klassendiagramm. Die Beschriftung oben links zeigt, dass das Diagramm dem Paket „paketX“ aus Abbildung 6.2 zugeordnet ist. Die Klasse „KlasseF“ besitzt ein Attribut und eine Methode, „KlasseH“ nur eine Methode und „KlasseJ“ keines von beidem. „KlasseF“ und „KlasseH“ stehen in einer gerichteten „1:1“-Assoziation zueinander, wobei „KlasseH“ die Rolle „ziel“ annimmt. „KlasseH“ und „KlasseJ“ sind über eine „1:n“-Komposition miteinander verbunden, wobei „KlasseH“ die Rolle „vater“ und „KlasseH“ die Rolle „kind“ übernimmt.

Besonderheiten von Fujaba-Klassendiagrammen

Die in den Fujaba-Klassendiagrammen definierten Assoziationen werden später in den Storydiagrammen verwendet, um Muster im Laufzeit-Objektgraphen zu finden. Bei der letztendlichen Abbildung auf lauffähigen Quellcode werden die Assoziationen mit Hilfe von Referenzen und Methoden nachgebildet. So wird z.B. für ein „zu-1“-Rollenende eine Referenz des entsprechenden Typs einschließlich get/set-Methoden-Paar generiert. In

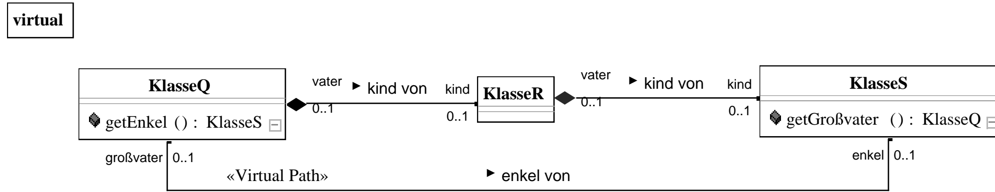


Abbildung 6.4.: Beispiel eines virtuellen Pfades

einigen Fällen ist es notwendig, diese Standard-Abbildung zu ersetzen, z.B. wenn keine eigene Referenz auf ein Laufzeit-Objekt gespeichert werden soll, sondern stattdessen bereits bestehende Assoziationen verwendet. In diesem Fall kann eine Assoziation als **„Virtueller Pfad“** deklariert werden [Zün02].

Abbildung 6.4 zeigt ein Klassendiagramm mit einem virtuellen Pfad. Die Klassen „KlasseQ“ und „KlasseR“ bzw. „KlasseR“ und „KlasseS“ stehen in Assoziation zueinander. Eine entsprechende Assoziation direkt zwischen „KlasseQ“ und „KlasseS“ führt zu eigenständigen Referenzen in den Quellcode-Dateien, die synchron mit den Referenzen der anderen beiden Assoziationen gepflegt werden müssen, d.h. es ist möglich, dass zur Laufzeit ein anderes Objekt als „enkel“ referenziert wird, als über die beiden „kind„-Objekte. Wird die Assoziation „enkel von“ jedoch als virtueller Pfad deklariert (wie in der Abbildung gezeigt), erzeugt Fujaba keinerlei Verhalten zur Navigation. Dies muss der Modellierer selbst definieren. So sind in der Abbildung die „get“-Methoden manuell modelliert, die auf die bereits bestehenden Assoziationen zugreifen. Ändert sich also das „kind“ an einem „KlasseR“-Objekt, so muss die korrespondierende „enkel“-Referenz nicht zusätzlich aktualisiert werden [Zün02].

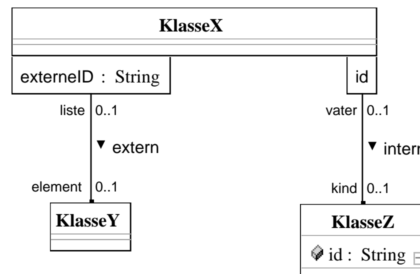


Abbildung 6.5.: Beispiel einer intern und einer extern qualifizierten Assoziation

Fujaba bietet die Möglichkeit, **qualifizierte Assoziationen** zu definieren. Statt einer ungeordneten Menge einer generischen „1:n“-Assoziation, dient ein Schlüssel (engl. key) dazu, jedes Element eindeutig zu identifizieren. Der Schlüssel kann zum einen **intern** vorliegen, d.h. als Attribut in der assoziierten Klasse. Zum anderen ist es auch möglich, **externe** Schlüssel anzugeben, die unabhängig vom assoziierten Objekt vergeben werden. Abbildung 6.5 zeigt beide Arten der qualifizierten Assoziation. Einmal eine intern quali-

fizierte zwischen „KlasseX“ und „KlasseZ“ und einmal eine extern qualifizierte zwischen „KlasseX“ und „KlasseY". Der Einsatz qualifizierter Assoziationen ist eine wichtige Voraussetzung für performanten Quellcode, da sie beim Zugriff auf ein Objekt eine Laufzeit von $O(1)$ bieten (statt $O(n)$ bei einer unqualifizierten Assoziation).

In Fujaba lassen sich einer Klasse ein oder mehrere **Stereotypen** (engl. stereotypes) zuordnen. Außer den gebräuchlichen UML-Stereotypen, wie z.B. „interface“, sind in Fujaba noch weitere Stereotypen vordefiniert: **„reference“** deklariert eine Klasse als Referenz auf eine außerhalb von Fujaba deklarierte Klasse, z.B. in einer Bibliothek. Für eine so ausgezeichnete Klasse wird kein Quellcode generiert. Stattdessen muss der Modellierer sicherstellen, dass sie beim Kompilieren und Ausführen des Quellcodes bereitsteht [Zün02]. Der Stereotyp **„JavaBean“** generiert zusätzliche Attribute und Methoden für eine Klasse, so dass sich ihre Instanzen – gemäß den JavaBean-Konventionen – auf Änderungen überwachen lassen und die Eigenschaften (engl. properties) der Klasse abgefragt werden können. Dies ist insbesondere beim Persistenzieren der Objekte über CoObRA notwendig [Sch07]. Die Verwendung des CoObRA-Persistenzmechanismus wird im Modul *persistency.coobra* (vgl. Abschnitt 6.4.11, S. 235ff.) beschrieben. Analog existiert auch der Stereotyp **„hibernate“**, um die Laufzeit-Objekte mit Hilfe des Persistenzrahmenwerks Hibernate zu sichern [BHRS07, Öhm10]. Die Verwendung von Hibernate wird im entsprechenden Modul *persistency.hibernate* (vgl. Abschnitt 6.4.12, S. 237ff.) näher beschrieben.

Verhalten: Storydiagramme

Die Storydiagramme in Fujaba modellieren das Verhalten einer Methode aus dem Fujaba-Klassendiagramm. Jedes Storydiagramm ist somit genau einer Methode aus dem Fujaba-Klassenmodell zugeordnet [Zün02]. Ein korrektes Story-Diagramm lässt sich auf lauffähigen Quellcode abbilden, so dass Fujaba-Modelle auch als „ausführbar“ bezeichnet werden [Zün02, Buc10]. Dazu fasst Fujaba die Objekte als Knoten und die Verknüpfungen zwischen ihnen als Kanten eines Graphen auf, der mittels Graphersetzung durchsucht und manipuliert wird. Die jeweiligen Knoten- und Kantentypen werden über die Klassen bzw. Assoziationen definiert [Zün02].

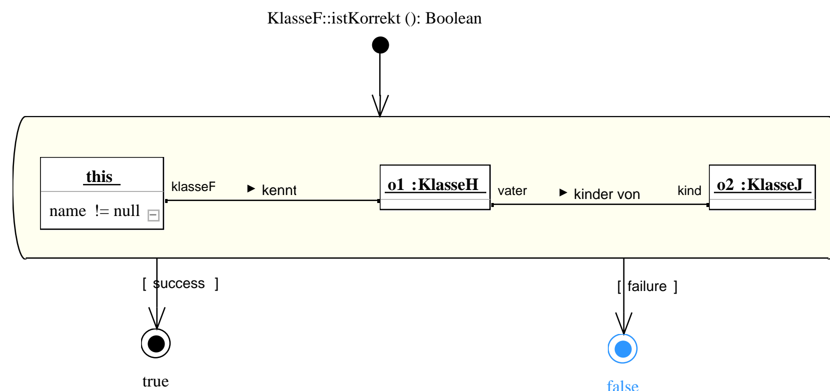


Abbildung 6.6.: Storydiagramm: Mustererkennung

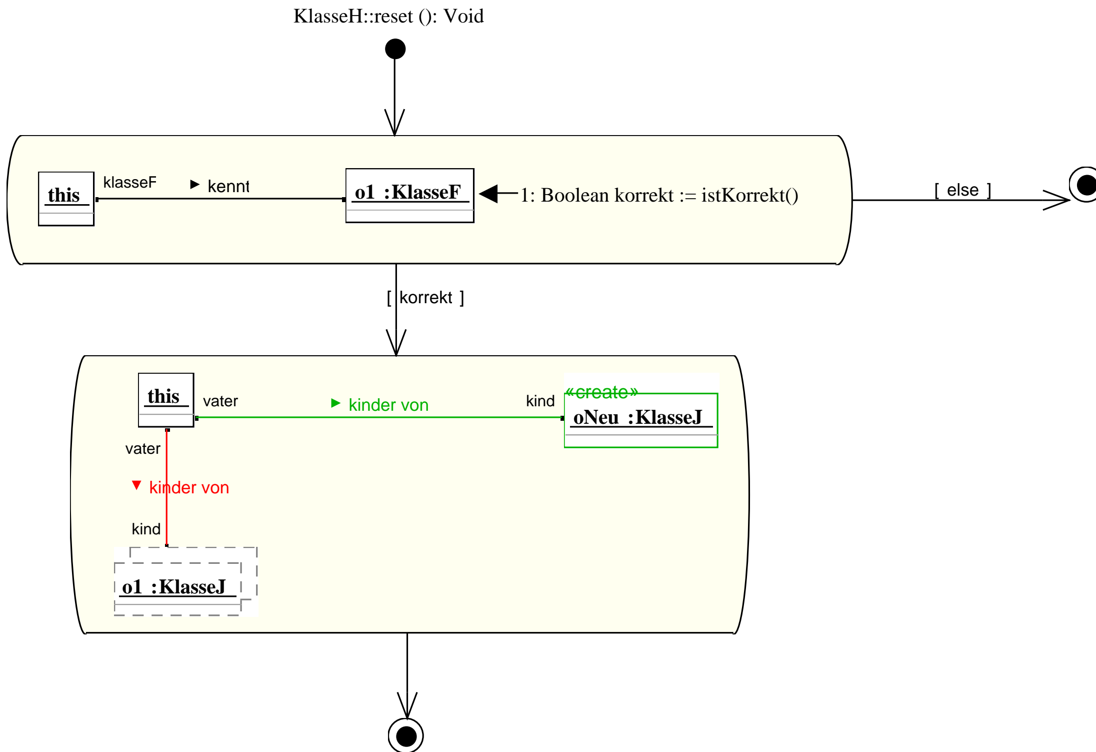
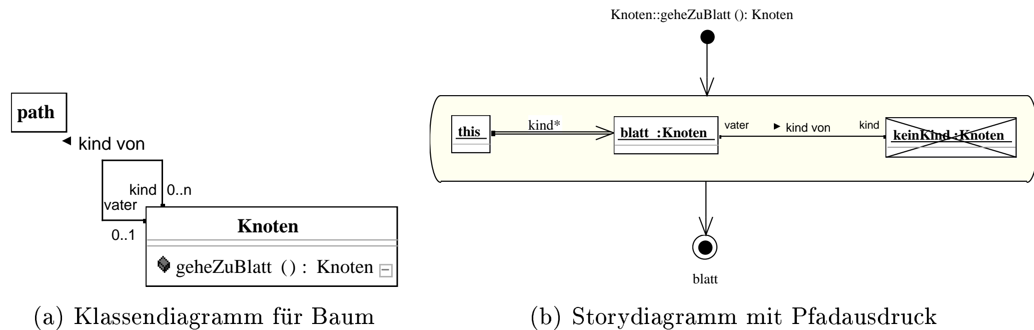


Abbildung 6.7.: Storydiagramm: Manipulation des Objektgraphen

Abbildung 6.6 zeigt ein Storydiagramm für die Methode „istKorrekt()“ aus dem Klassendiagramm in Abbildung 6.3. Jedes Storydiagramm beginnt bei einem Startknoten, der den Namen der zugehörigen Methode trägt. Von dort führt ein Kontrollflusspfeil zur ersten (und einzigen) Storyaktivität. Ausgehend vom „this“-Objekt, wird eine Verbindung zu einem Objekt des Typs „KlasseH“ gesucht, das eine Verbindung zu einem Objekt des Typs „KlasseJ“ besitzt. Außerdem wird der Wert des Attributs „name“ geprüft. Gelingt es, dieses Muster zu erfüllen, folgt der Kontrollfluss der „success“-Kante zum Endknoten „true“ (der Rückgabewert der Methode). Wird dieses Muster nicht gefunden, da (1) keine Verknüpfung zu einem Objekt vom Typ „KlasseH“ existiert, oder (2) kein verknüpftes „KlasseH“-Objekt mit einem „KlasseJ“-Objekt verbunden ist, oder (3) das Attribut „name“ nichts referenziert, dann folgt der Kontrollfluss der „failure“-Kante zum „false“-Endknoten.

Abbildung 6.7 zeigt ein Storydiagramm für die Methode „reset()“ aus dem Klassendiagramm in Abbildung 6.3 – diesmal mit Änderungen am Objektgraph. Die erste Storyaktivität sucht – ausgehend vom aktuellen Objekt – ein „KlasseF“-Objekt und ruft anschließend die Methode „istKorrekt()“ daran auf. Die Variable „korrekt“ speichert den Rückgabewert dieses Aufrufs. Anschließend verzweigt der Kontrollfluss in Abhängigkeit von einem Wahrheitswert, in diesem Fall der Variablen „korrekt“. Enthält sie den Wert „wahr“ geht es zur nächsten Storyaktivität, ansonsten wird das Methodenende erreicht. Die folgende Storyaktivität manipuliert den Objektgraphen. Schwarze Kanten und Ob-



(a) Klassendiagramm für Baum

(b) Storydiagramm mit Pfadausdruck

Abbildung 6.8.: Beispiel für Pfadausdrücke

jekte bleiben unverändert, rote werden gelöscht und grüne erzeugt. In diesem Fall werden zunächst alle „KlasseJ“-Kindobjekte gesucht und die Verknüpfung zu ihnen gelöscht. Da es sich bei „o1“ um einen Mengenknoten handelt, sind alle Verknüpfungen betroffen. Ohne Mengenknoten würde nur genau eine Verknüpfung zu einem beliebigen „KlasseJ“-Objekt gelöscht. Anschließend wird genau ein neues „KlasseJ“-Objekt erzeugt und mit dem aktuellen Objekt verknüpft. Dann ist ebenfalls das Methodenende erreicht. Für eine ausführlichere Beschreibung von Fujaba-Storydiagrammen sei auf [Zün02] verwiesen.

Besonderheiten von Fujaba-Storydiagrammen

Eine Verknüpfung in einer Storyaktivität bezieht sich immer auf direkt „benachbarte“ Objekte. Mit Hilfe von „Pfadausdrücken“ lassen sich in Fujaba beliebig viele Verknüpfungen navigieren, ohne die genau Anzahl an Verknüpfungen zu kennen. Mit ihrer Hilfe lässt sich somit der transitive Abschluss über die Verknüpfung bilden.

Abbildung 6.8(a) zeigt ein Klassendiagramm für einen einfachen n-ären Baum. Das Storydiagramm in Abbildung 6.8(b) verwendet den Pfadausdruck „kind*“ (dargestellt als Pfeil mit doppelter Linie) um, ausgehend vom aktuellen „Knoten“-Objekt, den transitiven Abschluss über alle Objekte in „kind“-Rolle zu bilden und ein Blatt auszuwählen (dargestellt durch die negative Bedingung).

In einigen Fällen ist es notwendig direkt Anweisungen in der Zielsprache der Quellcode-Generierung einzufügen – im Falle von MOD2-SKM ist dies Java. Grund sind z.B. Einschränkungen bei der Modellierung, die nur zwei ausgehende Kontrollflusskanten pro Storyaktivität erlauben. Da in Java öfters zusätzliche Ausnahmebehandlungen (engl. exception handling) nötig sind, müssen z.B. die „try-catch-Blöcke“ (vgl. [Ull09]) als Java-Anweisungen in den Quellcode geschrieben werden.

Abbildung 6.9 zeigt die Verwendung von zwei Anweisungsaktivitäten um einen try-catch-Block in Java einzufügen. Zunächst öffnet die „try“-Anweisungen den entsprechenden Block. Die nun folgende Storyaktivität wird nun innerhalb dieses Blocks ausgeführt, bevor die zweite Anweisungsaktivität ihn schließt und die Ausnahmen in den entsprechenden „catch“-Anweisungen behandelt [Ull09]. An dieser Stelle ist es nicht mehr möglich, das Verhalten mit Fujaba zu modellieren. Erschwerend kommt noch hinzu, dass

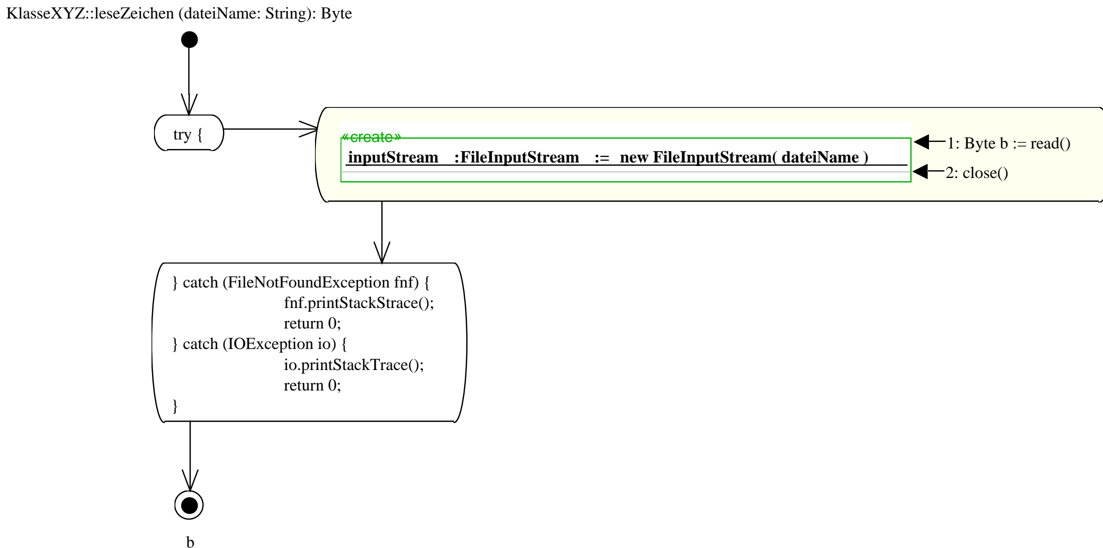


Abbildung 6.9.: Storydiagramm: Modellierung von try-catch-Blöcken in Java

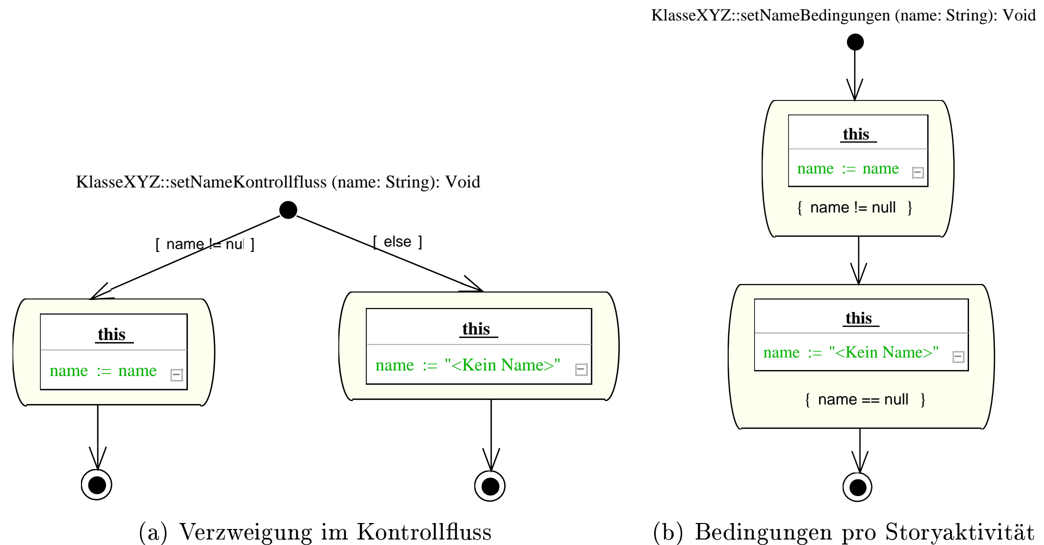
bei komplexeren Kontrollfluss-Verzweigungen die Platzierung der Anweisungsaktivitäten detaillierte Kenntnisse der Quellcode-Generierung von Fujaba voraussetzt.

Der Kontrollflussgraph eines Storydiagramms muss nicht einem Baum entsprechen, sondern darf auch Zyklen enthalten. Existiert mehr als ein Zyklus in einem Storydiagramm, kann es vorkommen, dass Fujaba die Verzweigungen und Zusammenführungen des Kontrollflusses nicht mehr auf Quellcode abbilden kann und die Codeerzeugung abbricht. Eine mögliche Lösung ist die Auslagerung eines Teil-Storydiagramms in eine eigene Methode und diese dann im ursprünglichen Storydiagramm aufzurufen. Ist dies nicht möglich, so kann alternativ der komplexe Kontrollfluss durch eine Sequenz von Storyaktivitäten ersetzt werden. Jeder der Aktivitäten wird nun eine Bedingung zugeordnet und über sie gesteuert, welche der Aktivitäten bei einem Methodenaufruf ausgeführt werden.

Abbildung 6.10 zeigt zwei Storydiagramme zum Setzen des „name“-Attributs. Ist der zugewiesene Wert leer, so wird stattdessen ein Standardwert verwendet. Das Storydiagramm in Abbildung 6.10(a) verwendet dazu zwei Kontrollflusskanten, um zur entsprechenden Storyaktivität zu verzweigen. In Abbildung 6.10(b) ist der Kontrollfluss dagegen linear, doch die Bedingungen der beiden Storyaktivitäten stellen sicher, dass nur die gewünschte Aktivität ausgeführt wird. Bereits dieses konstruierte Beispiel zeigt, dass die Lesbarkeit der Diagramme darunter leidet. Diese Art der Kontrollfluss-Modellierung wird daher lediglich als Hilfskonstruktion verwendet.

6.1.1. Modellierung von Variabilität

Der MODPL-Editor ermöglicht das Auszeichnen von Modellelementen mit Merkmalsmarkierungen, sowohl im Paketmodell als auch im Fujaba-Modell (vgl. Abbildung 6.1, S. 141 und Abschnitt 3.4.2, S. 58). Die Modelle selbst sind jedoch lediglich für die Mo-



(a) Verzweigung im Kontrollfluss (b) Bedingungen pro Storyaktivität

Abbildung 6.10.: Storydiagramm: Varianten der Kontrollfluss-Modellierung

dellierung einzelner Produkte vorgesehen – und nicht speziell zum Erfassen der variablen Komponenten einer Produktlinie. Z.B. sollen die Klassen in einem UML-Klassendiagramm einen eindeutigen Namen haben [OMG09b]. In einer Produktlinie können jedoch ohne weiteres zwei oder mehr Varianten eine Klasse existieren. Über die Merkmalsmarkierungen wird dann sichergestellt, dass im generierten Produkt immer nur eine der Varianten existiert (z.B. indem die Merkmale aus einer sich gegenseitig ausschließenden Merkmalsgruppe stammen, vgl. Abschnitt 3.4.2, S. 58ff.). Im Folgenden ist nun beschrieben, welche Möglichkeiten zur Modellierung der Variabilität die jeweiligen Modelle und ihre Editoren bieten.

Variabilität in Paketdiagrammen

Die MODPL-Paketdiagramme setzen die UML-Spezifikation für Paketdiagramme um (vgl. [OMG09b]). Daher müssen die voll qualifizierten Namen der Elemente eindeutig sein, und somit können in einem Paket nicht zwei Elemente mit dem gleichen Namen liegen. Dies ist jedoch nötig, um mehrere alternativen Fassungen eines Pakets zu modellieren, und dann eine Alternative per Merkmalsmarkierung auszuwählen. In diesem Fall muss ein Nicht-Produktlinien-Lösungsansatz verwendet werden: Die Varianten eines Pakets werden als benachbarte Unterpakete des entsprechenden Pakets modelliert. Die Merkmalsmarkierungen bestimmen dann lediglich, welches Unterpakete im konfigurierten SKMS enthalten ist.

Abbildung 6.11 zeigt das Paketdiagramm für die Historie des SKMS. Es existieren drei Pakete: „base“ für die Historie ohne explizite Beziehungen, „flat“ für die Historie mit Vorgänger-Nachfolger-Beziehung und „tree“ für die Historie mit Vorgänger-Nachfolger-Beziehung und Verzweigung (vgl. Abschnitt 4.3.3, S. 80ff.). Die drei Pakete sind je-

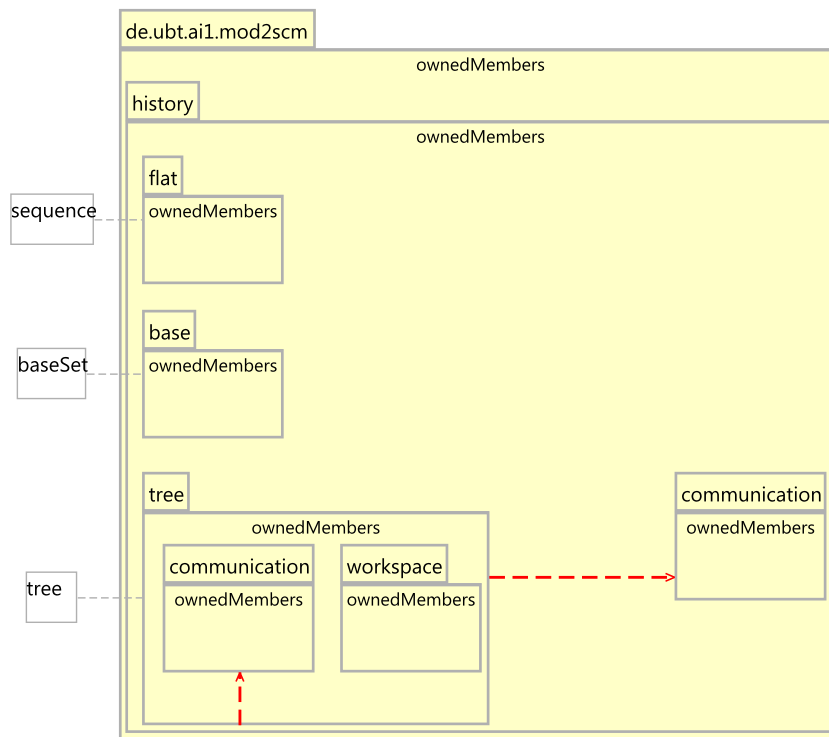


Abbildung 6.11.: Paketdiagramm: *history* mit Unterpaketen und Merkmalsmarkierungen

weils mit der entsprechenden internen Kennung der entsprechenden Merkmale markiert: „baseSet" (Merkmal „Einfache Menge" (D.1.2.1.2)), „sequence" (Merkmal „Sequenz" (D.1.2.1.4)) und „tree" (Merkmal „Baum" (D.1.2.1.1)). Alle drei Merkmale sind in der Merkmalsanalyse in Anhang B, S. 359ff. beschrieben. Hier ist auch schon zu sehen, dass sich die interne Kennung eines Merkmals (angezeigt als Wert von „id" in der Markierung) deutlich vom Merkmalsnamen unterscheidet. Dieses Problem wird in Abschnitt 6.1.2 besprochen.

Variabilität in Klassendiagrammen

Die Fujaba-Klassendiagramme orientieren sich an der UML-Spezifikation für Klassendiagramme [Zün02, OMG09b]. Wie bei den Paketdiagrammen soll der voll qualifizierte Name eindeutig sein, so dass Fujaba es verbietet, zwei Varianten der gleichen Klasse zu modellieren. Wie bei den Paketen ist es also nicht möglich, zwei Varianten einer Klasse zu modellieren. Doch während laut Spezifikation auch die Methoden und Attribute einer Klasse eindeutig benannt sein sollen, prüft Fujaba diese Bedingung nicht, sondern erzeugt den (unkompilierbaren) Quellcode. Diese „Lücke" erlaubt die Modellierung von Attributs- und Methoden-Varianten, indem mit Hilfe von Merkmalsmarkierungen sichergestellt wird, dass nur eine einzige Variante im konfigurierten SKMS enthalten ist.

Abbildung 6.12 zeigt eine Klasse, die zwei Varianten der Methode „handleCommitToPreviousVersion" enthält. Abhängig von den Merkmalen mit der Kennung „branchautomatically" (Merkmal „Automatisch Verzweigen" (D.11.2.1)) und „nOTBranchAutomatically" (das zugehörige Hilfsmerkmal, vgl. Abschnitt 5.2.1, S. 102ff.), wird entweder die erste oder die zweite Variante gewählt. Leider wird im Diagramm die Merkmalsmarkierung grafisch nicht direkt mit der Methode verbunden, sondern mit der Klasse. Das die Methode in der Markierung auftaucht, ist für die Modellierung der Varianten belanglos, da beide Methoden identische Namen und Signaturen besitzen. Hier muss der Modellierer entweder immer die Markierungen in der richtigen Reihenfolge halten oder mit Hilfe des MODPL-Konfigurators eine Konfiguration visualisieren (vgl. Abschnitt 3.5, S. 61). Alternativ kann auch für jede Methode eine eigene Unterklasse erstellt und markiert werden.

Variabilität in Storydiagrammen

Im Fujaba-Modell sind auch Merkmalsmarkierung in Storydiagrammen erlaubt [Buc10]. So lassen sich einzelne Storyaktivitäten oder sogar einzelne Elemente in den Aktivitäten auszeichnen. Die Beschränkungen bzgl. der Modellierung sind nur teilweise dokumentiert und lediglich implizit im Fujaba-Quellcode vorhanden. Daher sind nur die Einschränkungen bekannt, die Einfluss auf die Modellierung von MOD2-SKM hatten: Es ist z.B. nicht möglich, Quellcode für Storyaktivitäten mit mehr als zwei ausgehenden Kontrollflusskanten zu generieren. Es lassen sich daher keine alternativen Kontrollflüsse modellieren und anschließend über Merkmalsmarkierungen einschränken.

Abbildung 6.13 zeigt einen Ausschnitt aus einem Storydiagramm mit Merkmalsmarkierung. Nur wenn das Merkmal mit der Kennung „complexitems" in der Konfiguration

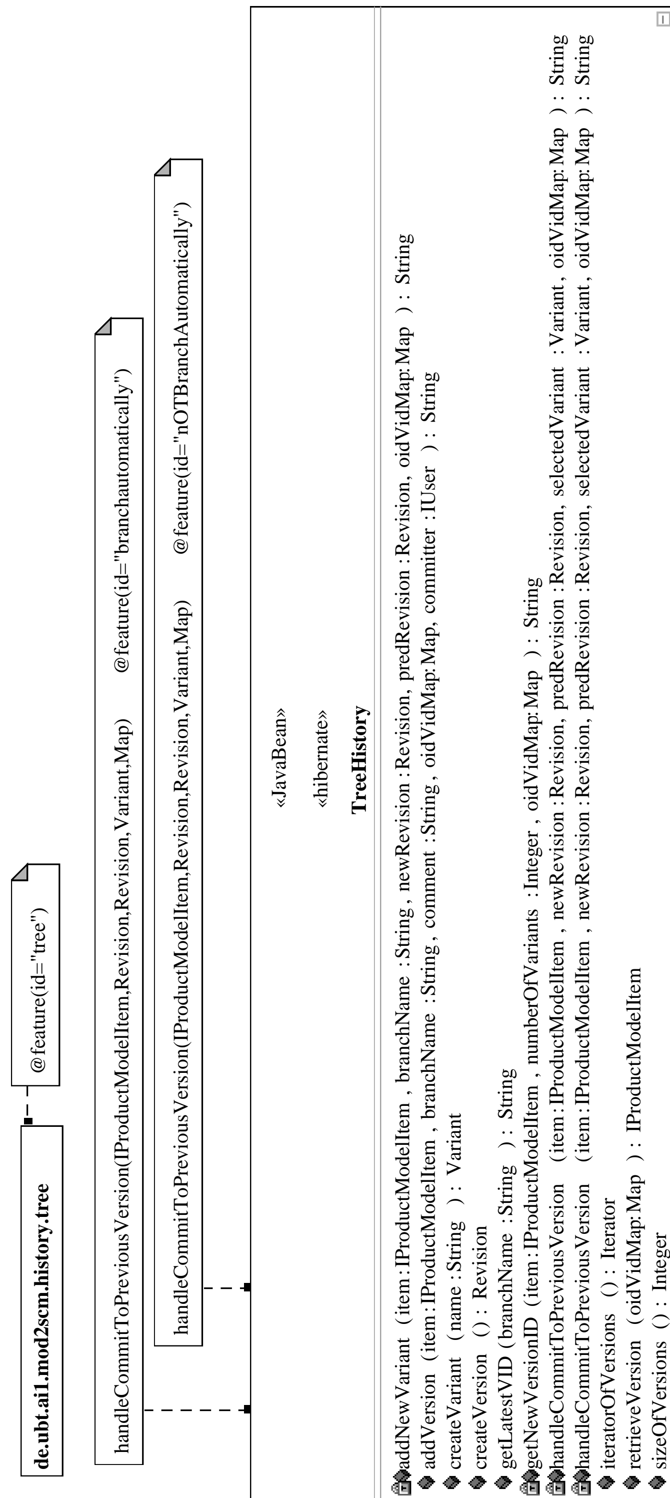


Abbildung 6.12.: Beispiel: Methoden-Varianten mit Merkmalsmarkierungen

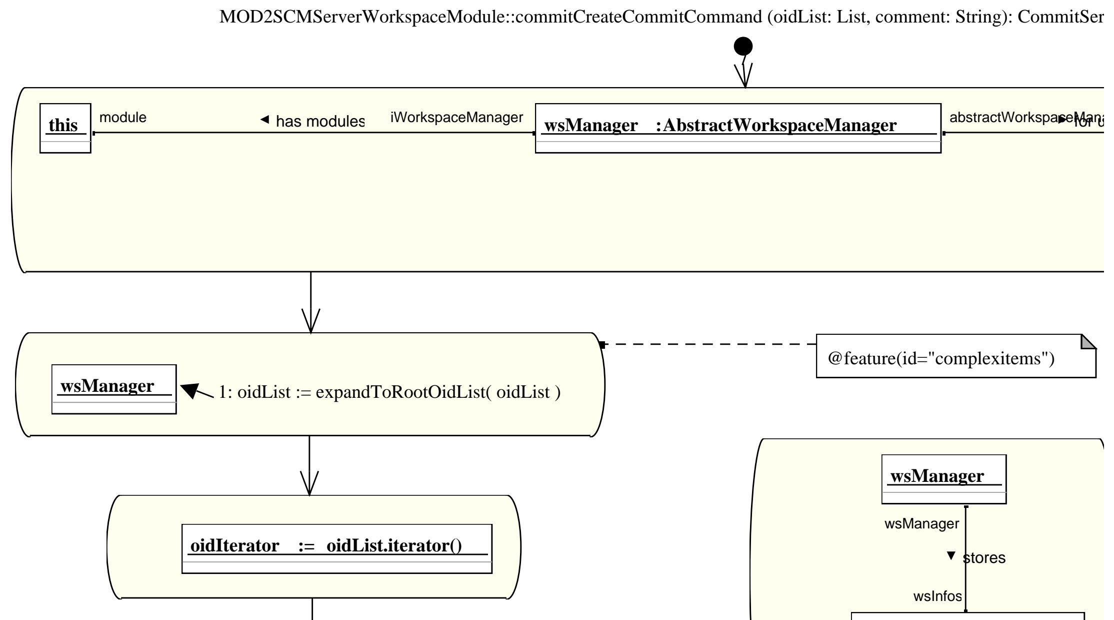


Abbildung 6.13.: Beispiel: Story-Diagramm Merkmalsmarkierungen

enthalten ist, wird auch die Anweisung in der Storyaktivität ausgeführt. Das Beispiel zeigt eine Merkmalsmarkierung, die an eine einzelne Aktivität innerhalb eines linearen Kontrollflusses gekoppelt ist. Die Änderung des Verhaltens bei Konfigurationen ohne das Merkmal sind so leicht zu erfassen. Bei komplexeren Kontrollflüssen oder Aktivitäten ist die Auswirkung einer Konfiguration mit fehlendem Merkmal schwer zu überblicken: Wie werden „success“- und „failure“-Kanten traversiert, wenn die Quellaktivität nicht zum konfigurierten System gehört? Diese Frage stellt sich ebenfalls bei „each“- und „end“-Kanten, ohne die zugehörige “foreach“-Aktivität (vgl. [Zün02]). Außerdem führt die Markierung einer Aktivität nur zum gewünschten Ergebnis, wenn der MODPL-Konfigurator direkt Quellcode generiert. Beim Erstellen eines konfigurierten Modells wird stattdessen der Kontrollfluss unterbrochen und das Story-Diagramm ist invalide.

6.1.2. Probleme und Grenzen

Die hier beschriebenen Techniken zur Modellierung der Variabilität stießen an die Grenzen der eingesetzten Modellierungswerkzeuge.

Zuordnung der Merkmalskennung

Die Merkmalsmarkierungen im MODPL-Werkzeugkasten verknüpfen die Elemente des Domänenmodells mit dem zugehörigen Merkmalsmodell. Diese Verknüpfung findet über die Kennung des Merkmals statt, die im Merkmalsmodell definiert wird [Buc10]. Diese Kennung ist im ganzen Merkmalsmodell eindeutig – im Gegensatz zum Merkmalsnamen – der mehrmals vorkommen kann (sogar als parallel liegende Untermerkmale). Die

Kennung wird aus dem beim Anlegen vergebenen Namen abgeleitet und wird auch bei Umbenennungen nicht mehr verändert. Sie wird jedoch nur intern verwendet, d.h. im Editor wird nur der Name angezeigt und auch die Merkmalsanalyse verwendet nur die Merkmalsnamen.

Ein Modellierer muss jedoch beim Auszeichnen die Merkmale anhand der Kennung identifizieren. Außerdem wird die Kennung auch in den Merkmalsmarkierungen an den Modellelementen angezeigt (vgl. Abbildungen 6.11, 6.12 und 6.13). Bei allen Merkmalsmarkierungen handelt es sich um Merkmale aus der Merkmalsanalyse in Anhang B. Eine Zuordnung der Markierung zum entsprechenden Merkmal ist kaum oder gar nicht möglich. Dies ist nur mit Hilfe einer zusätzliche Anmerkung im Text möglich: Die Kennung „tree“ entspricht dem Merkmal „Baum“ (D.1.2.1.1). Der MODPL-Werkzeugkasten sollte daher diese Kennung nur noch intern verwenden. Stattdessen sollte der Name des Merkmals oder sogar – analog zu den Paketen – der voll qualifizierten Merkmalsname angezeigt werden.

Einschränkung der Modellierung

Sowohl Fujaba als auch der MODPL-Paketdiagrammeditor verbieten – gemäß der UML-Spezifikationen – das Anlegen eines gleichnamigen Elements im selben Namensraum [OMG09b, Buc10]. Dies behindert jedoch die Modellierungen von Paket- bzw. Klassenvarianten: Es ist nicht möglich, zwei gleichnamige Elemente anzulegen und anschließend mit Merkmalsmarkierungen auszuzeichnen. Pakete und Klassen sind jedoch die größten Systembausteine, aus denen eine Produktlinie zusammengesetzt ist. Ein Modellierungswerkzeug für Produktlinien sollte diese Einschränkung unbedingt aufheben.

Um diese Einschränkung zu umgehen, wäre es möglich gewesen, eine Klasse anzulegen und dann jedes Attribut und jede Methode entsprechend auszuzeichnen. Die Merkmalsmarkierungen lassen sich jedoch einer Methode nur unter Schwierigkeiten zuordnen (vgl. Abbildung 6.12), daher scheidet auch diese temporäre Lösung aus. Somit ist die Variabilität im MOD2-SKM Domänenmodell größtenteils auf bereits existierende Architekturprinzipien und -konzepte zur Modularisierung angewiesen.

6.2. Architektur des MOD2-SKM Domänenmodells

Die Architektur des MOD2-SKM Domänenmodells lässt sich in zwei Teilbereiche gliedern. Zum einen der Architektur als Produktlinie, die gemeinsam und optional genutzte Elemente verbinden soll. Zum anderen der Architektur bzw. den Architekturen(!) der konfigurierten SKMS. In beiden Fällen kommen unterschiedliche Architekturprinzipien zum Einsatz.

6.2.1. Modularität: Architektur von Produktlinien

Der Schwerpunkt der Produktlinienarchitektur liegt auf dem Prinzip der **Modularität**, da sich die variablen Komponenten in mehreren Produkten wiederverwenden lassen sollen

[CN01]. Ziel einer modularen Architektur ist es, „wohl definierte Systembausteine[1] zu schaffen, [so] dass sie leicht austauschbar und in sich abgeschlossen sind.“ [VAC+09]. Das Modularitätsprinzip lässt sich dabei als Kombination fünf weiterer Architekturprinzipien auffassen [VAC+09]:

1. Seperation of Concerns
2. Abstraktion
3. Information Hiding
4. lose Kopplung
5. hohe Kohäsion

Separation of Concerns, d.h. die Trennung von Aufgabenbereichen, ist die Grundlage einer modularen Architektur. Sie bedeutet die Dekomposition des Gesamtsystems in Teilsysteme, von denen jedes einem eigenen Aufgabenbereich zugeordnet ist. Es wird dabei erwartet, dass die Aufteilung anhand der Aufgabenbereiche die Definition der Schnittstellen erleichtert, die Menge der sichtbaren Daten minimiert und die Kopplung und Kohäsion unterstützt [VAC+09].

Abstraktion im Bereich Software-Architektur bedeutet insbesondere Schnittstellenabstraktion, d.h. das Trennen der Schnittstelle von der eigentlichen Implementierung. Die Beziehungen werden dabei auf Basis der Schnittstellen definiert, so dass sich unterschiedliche Implementierungen mit gleicher Schnittstelle austauschen lassen [VAC+09].

Information Hiding bedeutet das Verbergen nicht benötigter Informationen. Dies betrifft insbesondere die Schnittstellen der Komponenten, die nur benötigte Informationen weitergeben sollten, um somit verständlicher zu bleiben [VAC+09].

Ziel der Anwendung dieser drei Architekturprinzipien ist letztendlich die Umsetzung zweier weiterer Prinzipien: der **losen Kopplung** und der **hohen Kohäsion**. Lose Kopplung bedeutet, dass von einer Komponente aus möglichst wenig Abhängigkeiten zu anderen Komponenten bestehen. Denn zum Verstehen bzw. Austauschen einer Komponente, müssen auch alle abhängigen Bausteine beachtet werden. Je weniger Abhängigkeiten existieren, desto leichter ist es, eine Komponente zu verstehen bzw. sie auszutauschen. Die Komponente selbst besteht jedoch meist aus mehreren Teilelementen, die Abhängigkeiten zu anderen Elementen besitzen. Die abhängigen Elemente stammen entweder ebenfalls aus der umgebenden Komponente, oder aber aus einer anderen. Je mehr Abhängigkeiten zwischen den Teilelementen innerhalb der Komponenten existieren, desto höher ist die Kohäsion – und desto weniger Abhängigkeiten bestehen zu anderen Komponenten [VAC+09].

[1] Ein Systembaustein ist ein klar abgegrenzter Teil eines Systems mit klar definierter Funktion und Schnittstelle, die von anderen Systembausteinen verwendet werden kann. Es kann sich im konkreten Fall sowohl um eine einzelne Klasse, eine Menge von Klassen oder auch eine vollständige Klassen- oder sogar Pakethierarchie handeln. [VAC+09]. Dies entspricht auch dem Komponentenbegriff [OMG09b], so dass die Begriffe „Systembaustein“ und „Komponente“ austauschbar verwendet werden.

6.2.2. Zentral vs. Dezentral: Architektur von SKMS

Der Schwerpunkt der SKMS-Architektur liegt auf ihrer Struktur und vor allem, wie zentralisiert bzw. dezentralisiert das SKMS arbeitet. An Hand existierender SKMS lassen sich grob drei verschiedene Architektur-Strukturen identifizieren [VAC+09]:

1. File Sharing
2. Client-/Server-Modell
3. Peer-to-Peer-System

Beim **File Sharing** sind die Daten (z.B. Dateien) in einem einzigen Speicher verfügbar, auf den von mehreren Rechnern aus zugegriffen werden kann. „SCCS“ [Roc75] und „RCS“ [Tic82] sind Beispiele für diese Architektur. Diese Struktur ermöglicht die Verwendung des SKMS-Konzepts „Repositorium“ (vgl. Abschnitt 4.4.1, S. 91ff.), aber nicht des „Arbeitsbereichs“ (vgl. Abschnitt 4.4.2, S. 93ff.). Da nur ein gemeinsam genutzter Speicher existiert, kann ein Entwickler die Daten nicht unabhängig von anderen Entwicklern bearbeiten.

Beim **Client-/Server-Modell** existiert ein zentraler Server, auf dessen Ressourcen ein oder mehrere Clients zugreifen. „CVS“ [Ced05] und „Subversion“ [CSFP08] sind Beispiele für diese Architektur. Diese Struktur ermöglicht die Trennung zwischen „Repositoriums“-Server und „Arbeitsbereichs“-Client. Dabei können die Entwickler ihren Arbeitsbereich mit dem zugehörigen Repositorium synchronisieren.

Beim **Peer-to-Peer-System** existiert kein zentrale Server mehr, sondern jeder Teilnehmer (engl. peer) kann Daten und Funktionen anbieten. Das bedeutet, dass die Teilnehmer u.a. erst herausfinden müssen, welche anderen Teilnehmer existieren und welche Daten oder Funktionen sie anbieten. „GIT“ [Ca09] und „Mercurial“ [O'S09] sind Beispiele für diese Architektur. Diese Struktur ermöglicht die Trennung zwischen „Repositorium“ und „Arbeitsbereich“, wobei jeder Arbeitsbereich sein eigenes Repositorium besitzt. Zusätzlich lässt sich das eigene Repositorium noch mit den Repositorien anderer Entwickler synchronisieren.

6.2.3. Umsetzung im MOD2-SKM Domänenmodell

Das MOD2-SKM Domänenmodell ist in Kernkomponenten und Module unterteilt. Die Kernkomponenten modellieren die gemeinsam genutzten Komponenten der Produktlinie und die Module die variablen Komponenten, die über die Konfiguration ausgewählt werden [PBL05, Buc10]. Die Architektur des SKMS ist ebenfalls variabel und wird über das Spitzenmerkmal „Repositoriums-Architektur“ festgelegt (vgl. Abschnitt 5.3.3, S. 120ff.). Da es sich jedoch bei den verteilten Repositorien um einen Erweiterungspunkt handelt (vgl. Abschnitt 5.5, S. 129ff.), ist als Architektur nur das Client-Server-Modell modelliert worden.

Abbildung 6.14.: Pakete des MOD2-SKM Domänenmodells

Kernkomponenten und Module

In einer Produktlinie existieren zum einen gemeinsam genutzte Komponenten, die in jedem konfigurierten SKMS vorhanden sind. Zum anderen gibt es variable Komponenten, die – abhängig von der Konfiguration – im konfigurierten System vorhanden sind oder nicht [Buc10]. Die Architektur des MOD2-SKM Domänenmodells spiegelt diese Unterscheidung wieder und ist eine erste Umsetzung des Prinzips „Separation of Concerns". Diese Trennung lässt sich gut mit Hilfe des Paketdiagramms in Abbildung 6.14 illustrieren: Das Paket „core", im oberen Teil der Abbildung, enthält die gemeinsam genutzten Komponenten, die in jedem SKMS vorhanden sind. Diese sind auf 14 weitere Pakete aufgeteilt – gemäß „Separation of Concerns" nach den SKM-Konzepten aus Kapitel 5. Ihre vollständige Beschreibung erfolgt in Abschnitt 6.3, S. 171ff. Anzumerken ist, dass kein Element einer Kernkomponente mit Merkmalsmarkierungen ausgezeichnet ist, da nur gemeinsam genutzte Komponenten enthalten sind.

Die 14 Pakete unterhalb des „core"-Paketes bilden die Modulbibliothek der MOD2-SKM Produktlinie, d.h. sie enthalten die variablen Komponenten, die nicht in jedem SKMS vorhanden sind. Jedes diese Pakete besitzt den gleichen Namen wie eines der Kernpakete und außerdem ein oder mehrere Unterpakete. Dadurch wird ausgedrückt, dass die jeweiligen Unterpakete alternative Implementierungen auf Basis der jeweiligen Kernkomponenten sind[2]. Z.B. sind die Unterpakete von „history" – „base", „flat" und „tree" – drei unterschiedliche Implementierungen einer Historie, basierend auf dem Kernpaket „core.history" (vgl. Abschnitt 4.3.3, S. 80ff.). Damit wird die in "core" eingeführte „Separation of Concerns" beibehalten.

Die Komponenten der Unterpakete müssen nun im besonderen Maße die Modularitätsprinzipien umsetzen, da sie austauschbar eingesetzt werden sollen – und zwar in allen möglichen Kombination mit allen anderen Modulen. Die einzelnen Module sind detailliert in Abschnitt 6.4 (S. 211ff.) beschrieben. Da es sich um den variablen Teil der Produktlinie handelt, sind ausgewählte Elemente von Modulen mit Merkmalsmarkierungen ausgezeichnet. Dies ist bereits deutlich in Abbildung 6.14 zu sehen. Hier sind eine Reihe von Paketen mit Merkmalsmarkierungen versehen, d.h. nur wenn das entsprechende Merkmal in einer Konfiguration enthalten ist, wird das Paket (mit all seinen enthaltenen Klassen und Unterpaketen) verwendet.

Tabelle 6.1 gibt sowohl einen Überblick über die Kernpakete als auch die Pakete der Modulbibliothek. Die Spalte „Kernpaket" listet die 14 Kernpakete in alphabetischer Reihenfolge auf. „Modulpakete" listet die Modulpakete auf, die das Kernpaket implementieren. Spalte drei verweist auf das modellierte SKM-Konzept aus Kapitel 4. Gibt es kein zugrunde liegendes Konzept, wird stattdessen kurz die Funktion beschrieben. Die letzte Spalte verweist schließlich auf die Abschnitte in diesem Kapitel, in denen das jeweilige Kernpaket (vgl. Abschnitt 6.3, S. 171ff.) bzw. seine Modulpakete (vgl. Abschnitt 6.4, S. 211ff.) detailliert beschrieben sind.

[2] Unterpakete mit den Namen „server", „communication" oder „workspace" sind keine alternativen Implementierungen, sondern verteilen ihre Komponenten auf die drei gleichnamigen Teilsysteme eines SKMS (vgl. Abschnitt 6.2.3, S. 161ff.).

Kern-paket	**Modulpakete**	**SKM-Konzept / Funktion**	**Details (Kern / Module)**
communication	–	Kommunikations-Middleware (vgl. Abschnitt 6.2.3, S. 161ff.).	Abschnitt 6.3.1, S. 173ff.
delta	storage.delta	Delta-Speicherung (vgl. Abschnitt 4.3.6, S. 89ff.).	Abs. 6.3.2, S. 175ff. Abs. 6.4.23, S. 283ff.
exceptions	–	Ausnahmen (engl. exceptions) zur Behandlung von Laufzeitfehlern.	Abschnitt 6.3.3, S. 177ff.
history	base, flat, tree	Versionshistorien (vgl. Abschnitt 4.3.1, S. 79ff.).	Abs. 6.3.4, S. 178ff. Abs. 6.4.1, S. 213ff.
id	hash, hierarchic, latest, primarykey, qualifiedname, tag	Verfahren zur Kennungsvergabe (vgl. Abschnitt 4.2.2, S. 76ff.).	Abs. 6.3.5, S. 182ff. Abs. 6.4.4, S. 222ff.
merge	eclipse, guiffy	Verfahren zum Verschmelzen zweier Elemente (vgl. Abschnitt 4.4.1, S. 92ff.).	Abs. 6.3.6, S. 184ff. Abs. 6.4.10, S. 233ff.
persistency	coobra, hibernate	Persistenzmechanismen für Repositorien und Arbeitsbereiche.	Abs. 6.3.7, S. 186ff. Abs. 6.4.11, S. 235ff.
product	ecore, filesystem, usecase	Produktmodelle (vgl. Abschnitt 4.2.1, S. 76ff.).	Abs. 6.3.8, S. 188ff. Abs. 6.4.13, S. 239ff.
repository	singleitem, complexitem	Repositorien (vgl. Abschnitt 4.4.1, S. 92ff.).	Abs. 6.3.9, S. 191ff. Abs. 6.4.16, S. 251ff.
server	base, rmi, runtimeconfig	Schnittstellen und Protokolle für SKMS-Server.	Abs. 6.3.10, S. 195ff. Abs. 6.4.18, S. 254ff.
storage	base, complex, delta	Speichermechanismen (vgl. Abschnitt 4.3.6, S. 88ff.)	Abs. 6.3.11, S. 200ff. Abs. 6.4.21, S. 279ff.
synchronisation	lockitem, lockserver	Verfahren zur pessimistischen Synchronisation mehrerer Entwickler (vgl. Abschnitt 4.4.1, S. 92ff.).	Abs. 6.3.12, S. 202ff. Abs. 6.4.24, S. 287ff.
user	flat	Benutzerverwaltungen (vgl. Abschnitt 4.4, S. 91ff.)	Abs. 6.3.13, S. 203ff. Abs. 6.4.26, S. 290ff.
workspace	base, runtimeconfig	Schnittstellen für Arbeitsbereiche (vgl. Abschnitt 4.4.2, S. 93ff.)	Abs. 6.3.14, S. 206ff. Abs. 6.4.27, S. 292ff.

Tabelle 6.1.: Überblick über Pakete der Kernkomponenten

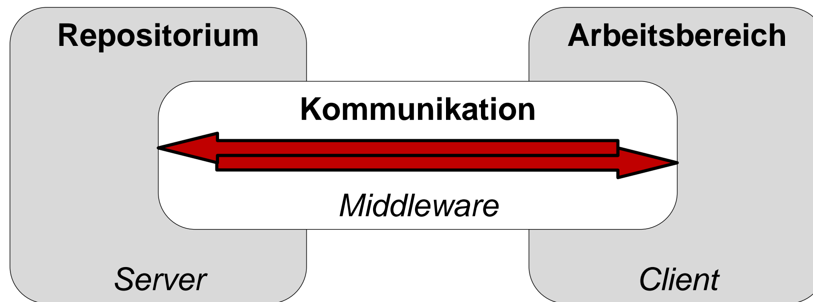


Abbildung 6.15.: Architektur-Struktur eines „MOD2-SKM"-SKMS

Client-/Server-Modell: Repositorium und Arbeitsbereich

Außer der Trennung zwischen gemeinsamen und variablen Komponenten, existiert noch die Aufteilung eines SKMS in Arbeitsbereich (vgl. Abschnitt 4.4.2, S. 93ff.) und Repositorium (vgl. Abschnitt 4.4.1, S. 91ff.). Diese Trennung ist in der Regel nicht nur konzeptionell, sondern auch physikalisch, indem Arbeitsbereich und Repositorien auf unterschiedlichen Computersystemen gespeichert sind. Grundlegende Architektur für MOD2-SKM ist damit auch die klassische Client-/Server-Architektur [BCK03, VAC+09]. Anstatt nun die Kommunikation zwischen beiden Anwendungen über ein Netzwerkprotokoll zu realisieren, wird zusätzlich noch eine Kommunikations-Middleware verwendet [3].

Abbildung 6.15 zeigt die drei Teilsysteme: Server (Repositorium), Client (Arbeitsbereich) und Middleware (Kommunikation). Komponenten des Repositoriums (linker Kasten) bzw. des Arbeitsbereichs (rechter Kasten) sind nur in diesem Teil des SKMS verfügbar und lassen sich nicht im jeweils anderen verwenden. Der „Kommunikationskanal" Middleware (mittlerer Kasten) tauscht Daten zwischen den beiden Teilsystemen aus, indem er eigens für die Kommunikation vorgesehene Komponenten überträgt (dargestellt durch die roten Pfeile). Alle Komponenten des Kommunikations-Bereichs müssen damit sowohl den Arbeitsbereich-Clients als auch den Repositoriums-Servern bekannt sein. Die Aufteilung betrifft auch Verteilung des SKMS zu Laufzeit: Zur Verwendung eines Arbeitsbereich-Clients reicht der kompilierte Quellcode für die Komponenten von Client und Kommunikation aus. Alle Server-Komponenten brauchen somit nicht an einen Client-Rechner ausgeliefert werden [BCK03].

Client/Server-Module

Sowohl die Kernkomponenten als auch die Module sind somit Teil eines SKMS nach Client-/Server-Modell. Dies ist auch im Paketdiagramm in Abbildung 6.14 (s. S. 158) zu sehen. Das „core-Paket" enthält die Pakete „server", „communication" und „workspace". Die Komponenten dieser drei Pakete legen das Client-/Server-Modell als Architektur-Struktur der konfigurierten SKMS fest.

[3] MOD2-SKM wird auf Java-Quellcode abgebildet, so dass es sich hier konkret um die „Java Remote Method Invocation (RMI)" handelt. Für weitere Informationen über Java RMI siehe [Ull09]

Die restlichen 11 Unterpakete des „core“-Pakets sind in diese Architektur integriert und lassen sich mehr oder weniger unabhängig von ihr verwenden. Besitzt ein Paket keine weiteren Unterpakete mehr, dann können alle Komponenten unabhängig von der Architektur verwendet werden. Dies trifft nur auf die drei Pakete „delta„, „persistency“ und „product“ zu. Alle anderen Pakete besitzen Komponenten, die sich nur in einem Teilsystem einsetzen lassen. Diese Komponenten liegen in einem Unterpaket, das den Namen des entsprechenden Teilsystems trägt: „server„, „communication“ oder „workspace“. Z.B. besitzt das Paket „user“ zum einen Komponenten, die sich nur auf dem „server“ einsetzen lassen, und zum anderen nur im „workspace“ verwendbar sind.

Die Pakete der Module übernehmen die Aufteilung ihrer Komponenten auf die drei Teilsysteme. So steuert z.B. die Baum-Historie (Paket „tree“) Komponenten zu allen drei Teilsystemen bei, was an den drei Unterpaketen „server„, „communication“ und „workspace“ zu sehen ist. Abschließend ist zu Abbildung 6.14 festzuhalten, dass aus Gründen der Lesbarkeit zwei Vereinfachungen vorgenommen wurden: Erstens wurde das „server“-Paket weggelassen, da ein Großteil der Module Komponenten für das Server-Teilsystem bereitstellt. Diese Komponenten befinden sich direkt im Modulpaket, und nur Komponenten für Kommunikation und Arbeitsbereich befinden sich in einem entsprechenden Unterpaket. D.h. das Paket „tree“ steuert auch Server-Komponenten bei. Ob ein Modul Server-Komponenten bereithält, ist immer am jeweiligen Kernpaket zu sehen: Enthält es ein „server“-Unterpaket, so handelt es sich beim Modul auch um ein Server-Modul. Zweitens sind in den Paketen „history“ und „product“ die Architektur-Unterpakete nur exemplarisch angegeben. Jede Historie steuert Komponenten zu allen drei Teilsystemen bei, und jedes Produktmodell zum Arbeitsbereich.

Architektur und Variabilität

Bei der Erstellung des MOD2-SKM Paketmodells spielte auch die Variabilität und insbesondere eine ökonomische Auszeichnung mittels Merkmalsmarkierungen eine Rolle. Daher wurden die Server-, Kommunikations- und Arbeitsbereich-Pakete ihren jeweiligen Modulen zugeordnet (anstatt z.B. drei Pakete für Server, Arbeitsbereich und Kommunikation zu wählen und die Komponenten darauf zu verteilen). Nur so ließ sich mit genau einer Merkmalsmarkierung am entsprechenden Unterpaket das Domänenmodell konfigurieren, z.B. im Falle der Historie mit genau einem Merkmal am entsprechenden ‘history“-Unterpaket (vgl. Abbildung 6.14, S. 158).

Ein offener Punkt ist die Realisierung einer variablen Architektur für die SKMS, damit auch „Peer-to-Peer“-Systeme (vgl. Abschnitt 6.2.2, S. 157ff.) durch MOD2-SKM unterstützt werden. Evtl. kann die existierende Architektur verwendet werden, da es sich bei den „Peer-to-Peer“-SKMS um hybride „Peer-to-Peer“-Systeme handelt: Jedes lokale Repositorium stellt einen Server dar, dem genau ein Arbeitsbereich zugeordnet ist (aufgebaut nach der jetzigen Architektur). Mit Hilfe von Operationen zur Synchronisation zweier Server ließe sich nun ein Abgleich zwischen zwei Repositorien realisieren, ohne das die Architektur-Struktur verändert werden muss. Alternativ könnte auch als Architektur-Struktur das „Peer-to-Peer“-System dienen, da sich mit ihm auch ein Client-Server-System nachbilden lässt. Ein Teilnehmer fungiert in diesem Fall als zentrales Re-

positorium, mit dem die anderen Teilnehmer synchronisieren [O'S09].

6.2.4. Module der MOD2-SKM-Produktlinie

Die folgenden Abschnitte beschreiben nun die einzelnen Module im Detail. Zum einen wird die Modellierung ihrer Struktur anhand ihrer Schnittstellen und Klassen beschrieben. Zu anderen wird für ausgewählte Operationen, wie Commit und Update, auch das Verhaltensmodell gezeigt und erläutert. Schwerpunkt der Beschreibung bilden dabei die Abhängigkeiten zwischen den Modulen (ausgedrückt durch Importbeziehungen) sowie die Variationspunkte der Module (modelliert durch Merkmalsmarkierungen).

Abbildung 6.16 zeigt alle Module der MOD2-SKM-Produktlinie, ihre wichtigsten Abhängigkeiten und ihre Merkmalsmarkierungen. Das Paket *core* enthält die 11 Kernmodule der Produktlinie. Sie modellieren die gemeinsam genutzten Komponenten von MOD2-SKM und sind daher nicht mit Merkmalen markiert. Unterhalb des *core*-Pakets befinden sich die Module der Modulbibliothek. Jedes Modul implementiert (mindestens) eine Schnittstelle eines Kernmoduls, was durch die entsprechenden Importbeziehungen gezeigt wird. Die Module modellieren die variablen Bestandteile von MOD2-SKM, so dass nahezu jedes Bibliotheksmodul eine Merkmalsmarkierung besitzt, d.h. es ist nur Teil eines SKMS, dessen Konfiguration das entsprechende Merkmal enthält. Die einzelnen Module sind fast vollständig entkoppelt und lassen sich daher gegen beliebige andere Module austauschen, welche die gleiche Kernmodul-Schnittstelle implementieren. Es existieren nur wenig Abhängigkeiten, welche die Kombination der Module einschränken. Diese sind an den roten privaten Importbeziehungen erkennbar.

Nun folgt ein Überblick über die Aufgaben der einzelnen Module und ihre wichtigsten Abhängigkeiten. Die Abschnitte 6.3 (S. 171ff.) und 6.4 (S. 211ff.) beschreiben die einzelnen Module detaillierter.

Überblick Kernmodule

Die folgenden Module sind im Paket *core*, im oberen Teil von Abbildung 6.16, enthalten. Sie modellieren die gemeinsam genutzten Schnittstellen und Komponenten von MOD2-SKM. Sie besitzen daher keine Abhängigkeiten zu konkreten Modulen der Modulbibliothek, stattdessen werden ihre Schnittstellen von konkreten Modulen implementiert, was durch die entsprechenden Importbeziehungen in Abbildung 6.16 dargestellt wird.

core.communication (vgl. Abschnitt 6.3.1, S. 173ff.) modelliert die Schnittstelle des Kommunikations-Teilsystems. Sie wird vom Arbeitsbereichs-Teilsystem genutzt, um Kommandos einschließlich ihrer zugehörigen Daten an das Server-Teilsystem zu übertragen. MOD2-SKM unterstützt die Kommunikation auf dem lokalen Computersystem (*server.base*) und über ein Netzwerk (*server.rmi*).

core.delta (vgl. Abschnitt 6.3.2, S. 175ff.) modelliert eine einheitliche Schnittstelle für deltifizierbare Elemente. Die Schnittstelle wird von allen Produktmodell-Elementen implementiert, die als Deltas gespeichert und geladen werden können (*product.filesystem*). Damit lassen sich diese Produktmodell-Elemente in einem Speicher ablegen, der diese

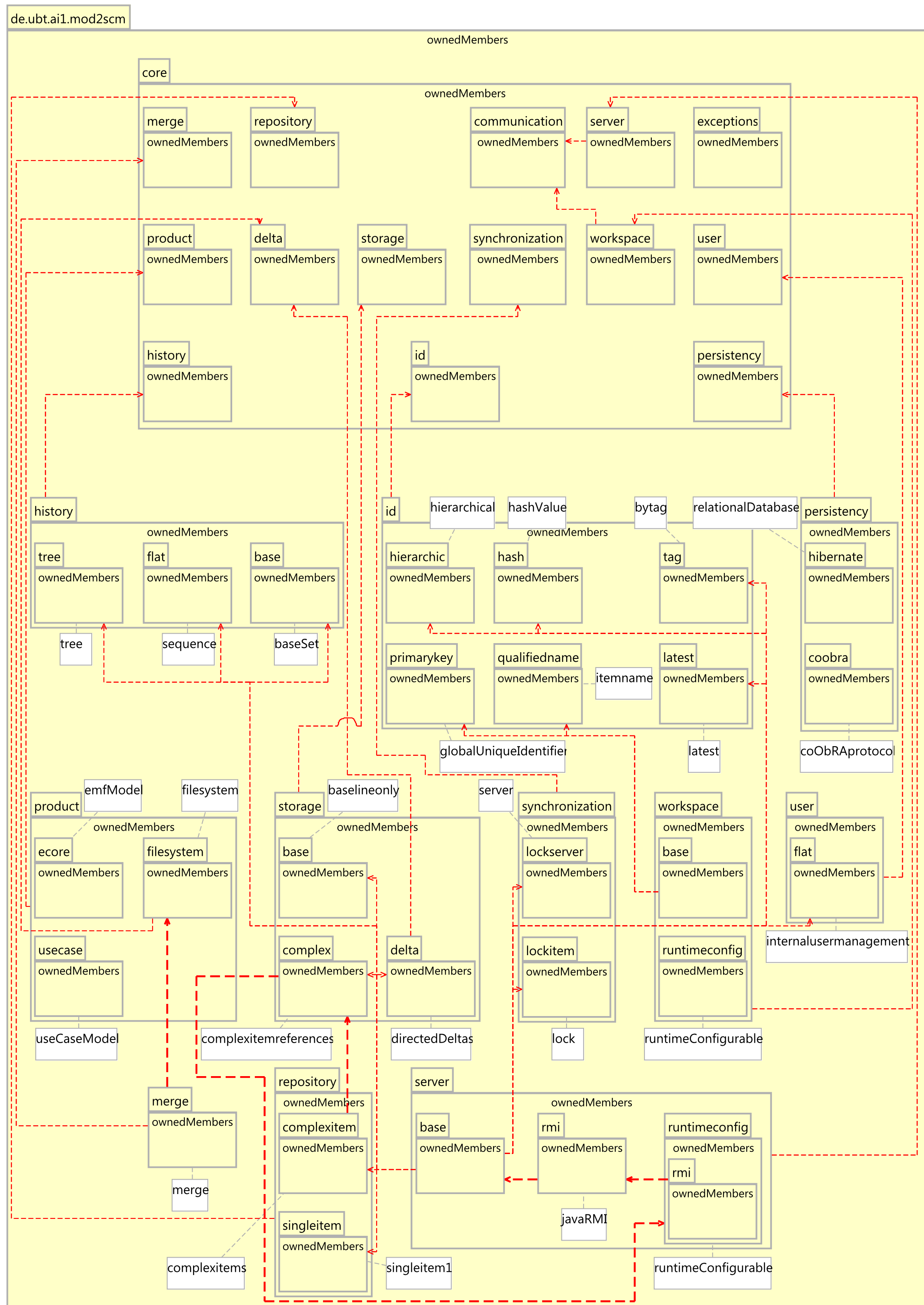


Abbildung 6.16.: Module des MOD2-SKM Domänenmodells

Schnittstelle verwendet (*storage.delta*). So kann der Delta-Speicher unabhängig von Algorithmus und Element-Typ verwendet werden.

core.exceptions (vgl. Abschnitt 6.3.3, S. 177ff.) modelliert generische Ausnahmen, die zur Laufzeit in einem SKMS auftreten können, und wird in vielen Modulen verwendet. Ausnahmen werden bis zur Benutzerschnittstelle propagiert, die mit ihrer Hilfe Fehlermeldungen anzeigt.

core.history (vgl. Abschnitt 6.3.4, S. 178ff.) modelliert die Schnittstellen für Historien. Sie ermöglicht den Zugriff auf die Versionen einer Historie, nur durch Angabe von Objekt- und Versionskennung (d.h. (oid, vid)-Tupel). Alle drei Historien (*history.base*, *history.flat* und *history.tree*) implementieren diese Schnittstelle und lassen sich so gegeneinander austauschen.

core.id (vgl. Abschnitt 6.3.5, S. 182ff.) modelliert die Schnittstelle für Kennungsgeneratoren und Kennungserweiterung. Erstere bieten eine einheitliche Schnittstelle für Module, die Objekt- oder Versionskennungen erzeugen (*id.hash*, *id.hierarchic*, *id.primarykey*, *id.qualifiedname*). Letztere kapseln zusätzliche Identifikationsverfahren, wie Markierungen (*id.tag*) oder Regeln (*id.latest*), anhand derer sich Objekt- und Versionskennungen ableiten lassen.

core.merge (vgl. Abschnitt 6.3.6, S. 184ff.) modelliert eine Schnittstelle für Verschmelzungsverfahren, um Konflikte bei optimistischer Synchronisation zu beheben. Das Zurückführen von Zweigen ist bisher nicht modelliert. Die Schnittstelle ist im Rahmen der Arbeit von Christopher Bär definiert und durch ein Modul (*merge*) realisiert [Bär10].

core.persistency (vgl. Abschnitt 6.3.7, S. 186ff.) modelliert eine Schnittstelle, um Persistenzmechanismen zu kapseln, d.h. um unabhängig vom konkreten Mechanismus (*persistency.coobra*, *persistency.hibernate*) den Zustand eines Teilsystems – gesichert durch Transaktionen – speichern zu können.

core.product (vgl. Abschnitt 6.3.8, S. 188ff.) modelliert eine Schnittstelle für atomare und zusammengesetzte Produktmodell-Elemente. Sie kapselt die Struktur und die Inhalte unterschiedlicher Produktmodelle (*product.ecore*, *product.filesystem*, *product.usecase*) durch eine generische Baumstruktur, die Operationen zum Navigieren, Einfügen bzw. Ersetzen von Elementen und Teilbäumen bietet.

core.repository (vgl. Abschnitt 6.3.9, S. 191ff.) modelliert eine Schnittstelle für Repositorien, welche die Operationen zum Einspielen und Auslesen von Versionen kapselt. So kann die Synchronisation zwischen Arbeitsbereich und Repositorium unabhängig vom konkreten Repositorium (*repository.complexitem*, *repository.singleitem*) erfolgen.

core.server (vgl. Abschnitt 6.3.10, S. 195ff.) modelliert eine Schnittstelle für das Server-Teilsystem und bietet ein einheitliches Rahmenwerk, um Server-Module an das Kommunikations-Teilsystem anzubinden. Der Server ist vom Arbeitsbereich entkoppelt, daher existiert nur eine Beziehung zu *core.communication* – und nicht zu *core.workspace*.

core.storage (vgl. Abschnitt 6.3.11, S. 200ff.) modelliert eine Schnittstelle für Speicherverfahren, die versionierte Produktmodell-Elemente speichern und laden können. Analog zur *core.history*, werden Versionen nur anhand ihres (oid, vid)-Tupels eindeutig referenziert, so dass sich unterschiedliche Speicherverfahren (*storage.base*, *storage.complex*, *storage.delta*) kapseln lassen.

core.synchronisation (vgl. Abschnitt 6.3.12, S. 202ff.) modelliert eine Schnittstelle,

um beliebigen Objekten den Status „gesperrt“ zuweisen zu können. Sie kapselt die Aufrufe zum Anlegen, Entfernen sowie Abfrage des Status für unterschiedliche Verfahren (*synchronisation.lockitem*, *synchronisation.lockserver*).

core.user (vgl. Abschnitt 6.3.13, S. 203ff.) modelliert eine Schnittstelle, um Benutzer in einem SKMS verwalten und in ihrem Namen Elemente sperren zu können. So lange ein Element gesperrt ist, kann es nur von diesem Benutzer verändert werden (vgl. Abschnitt 4.4.1, S. 92ff.). Zurzeit existiert nur ein Modul für eine rudimentäre Benutzerverwaltung (*user.flat*).

core.workspace (vgl. Abschnitt 6.3.14, S. 206ff.) modelliert eine Schnittstelle für das Arbeitsbereichs-Teilsystem und bietet ein einheitliches Rahmenwerk, um Arbeitsbereichs-Module an das Kommunikations-Teilsystem anzubinden. Der Arbeitsbereich ist vom Server entkoppelt, daher existiert nur eine Beziehung zu *core.communication* – und nicht zu *core.server*.

Überblick Modulbibliothek

Die folgenden Module sind im Paket *mod2scm*, im unteren Teil von Abbildung 6.16, enthalten. Sie modellieren die variablen Komponenten von MOD2-SKM. Sie implementieren i.d.R. die Schnittstelle eines Kernmoduls, was durch die entsprechenden Importbeziehungen in Abbildung 6.16 dargestellt wird. Damit sind sie gegen andere Module mit der gleichen Schnittstelle austauschbar und mit diesen auch im gleichen Paket gruppiert (z.B. enthält das Paket *mod2scm.history* alle austauschbaren Historienmodule). Die Abhängigkeiten zwischen den Modulen der Modulbibliothek wurde gezielt reduziert, um eine lose Kopplung zu erreichen. Die verbliebenen Abhängigkeiten sind als rote private Importbeziehungen in Abbildung 6.16 enthalten. Die Kopplungen der Module werden in Abschnitt 6.5.3 ausführlich untersucht.

Historien

history.base (vgl. Abschnitt 6.4.1, S. 213ff.) modelliert die Historie eines Produktmodell-Elements als ungeordnete Menge von Versionen, zwischen denen keine expliziten Beziehungen existieren (diese müssten implizit über die Kennung abgeleitet werden). Die Kennungen für neue Versionen werden über die Schnittstelle *core.id* erzeugt. Das Modul ist von der Konfiguration des Merkmals „Einfache Menge“ (D.1.2.1.2) (Markierung: *baseSet*) abhängig und ist nur partiell implementiert, d.h. es bietet bisher keine Kommandos zum Abfragen der Versionsinformationen in der Historie an (s. *history.tree* für ein vollständiges Modul).

history.flat (vgl. Abschnitt 6.4.2, S. 214ff.) modelliert die Historie eines Produktmodell-Elements als eine einzige Sequenz von Versionen, die in Vorgänger-Nachfolger-Beziehung stehen. Die Kennungen für neue Versionen werden über die Schnittstelle *core.id* erzeugt. Das Modul ist von der Konfiguration des Merkmals „Sequenz“ (D.1.2.1.4) (Markierung: *sequence*) abhängig und ist nur partiell implementiert, d.h. es bietet bisher keine Kommandos zum Abfragen der Versionsinformationen in der Historie an (s. *history.tree* für ein vollständiges Modul).

history.tree (vgl. Abschnitt 6.4.3, S. 216ff.) modelliert eine Historie eines Produktmodell-Elements als Baum aus Revisionen und Varianten. Die Revisionen stehen, wie bei *history.flat*, in Vorgänger-Nachfolger-Beziehungen. Zusätzlich existieren jedoch noch Verzweigungsbeziehungen, um Varianten einer Revision speichern zu können. Die Kennungen für neue Versionen werden über die Schnittstelle *core.id* erzeugt. Das Modul ist von der Konfiguration des Merkmals „Baum“ (D.1.2.1.1) (Markierung: *tree*) abhängig und bietet Kommandos zum Abfragen der Versionsinformationen in der Historie sowie zum Anlegen von Verzweigungen. Letzteres wird in den o.g. Historien nicht benötigt, d.h. die Historienmodule besitzen offensichtlich keine einheitliche Kommunikations-Schnittstelle.

Kennungen

id.hash (vgl. Abschnitt 6.4.4, S. 222ff.) modelliert einen Kennungsgenerator, der den SHA-1-Hashwert eines Produktmodell-Elements berechnet, um ihn als Objekt- bzw. Versionskennung zu verwenden. Das Modul *id.hash* ist von der Konfiguration des Merkmals „Hashwert“ (D.8.1.3) (Markierung: *hashValue*) abhängig.

id.hierarchic (vgl. Abschnitt 6.4.5, S. 223ff.) modelliert einen Kennungsgenerator, der eine numerische, hierarchische Kennung für ein Produktmodell-Element generiert, um sie als Objekt- bzw. Versionskennung zu verwenden. Das Modul ist von der Konfiguration des Merkmals „Hierarchisch“ (D.2.2.2.2) (Markierung: *hierarchical*) abhängig.

id.latest (vgl. Abschnitt 6.4.6, S. 224ff.) modelliert ein ergänzendes Kennungsmodul, mit dem sich eine Liste von (oid, vid)-Tupeln erzeugen lässt, welche die jeweils aktuellste Versionskennung jedes Elements enthält. Diese Liste kann dann zum Auslesen der Elemente an das Repositorium übergeben werden. Das Modul ist von der Konfiguration des Merkmals „Neuste Version“ (D.2.1.3) (Markierung: *latest*) abhängig.

id.primarykey (vgl. Abschnitt 6.4.7, S. 225ff.) modelliert einen Kennungsgenerator, der eine laufende Nummer (d.h. Primärschlüssel) als Kennung für ein Produktmodell-Element generiert, um sie als Objektkennung zu verwenden. Das Modul ist von der Konfiguration des Merkmals „Global eindeutige ID“ (D.8.1.2) (Markierung: *globalUniqueIdentifier*) abhängig.

id.qualifiedname (vgl. Abschnitt 6.4.8, S. 226ff.) modelliert einen Kennungsgenerator, der den qualifizierten Namen eines Produktmodell-Elements (z.B. den Dateinamen) ausliest, um ihn als Objektkennung zu verwenden. Das Modul ist von der Konfiguration des Merkmals „Elementname“ (D.8.1.1) (Markierung: *itemname*) abhängig.

id.tag (vgl. Abschnitt 6.4.9, S. 227ff.) modelliert ein ergänzendes Kennungsmodul, mit dem sich eine Liste von (oid, vid)-Tupeln (z.B. der aktuelle Stand eines Arbeitsbereichs) unter einem Namen speichern und später wiederherstellen lässt. Diese Liste kann dann zum Auslesen der Elemente an das Repositorium (*core.repository*) übergeben werden. Das Modul ist von der Konfiguration des Merkmals „Nach Markierung“ (D.2.1.2) (Markierung: *bytag*) abhängig.

Verschmelzen

merge (vgl. Abschnitt 6.4.10, S. 229ff.) wurde von Christopher Bär im Rahmen von [Bär10] entwickelt. Es modelliert mehrere Module zum Verschmelzen von Dateien und kann daher nur zusammen mit dem Dateisystem-Produktmodell (*product.filesystem*) verwendet werden. Das Verschmelzen erfolgt bei Konflikten während der Aktualisierung des Arbeitsbereichs – sowohl interaktiv als auch automatisiert. Die Verfahren integrieren – als Untermodule – zwei unterschiedlichen Verschmelzungsrahmenwerke [Bär10]. Das Modul ist von der Konfiguration des Merkmals „Verschmelzen" (D.11.2.3) (Markierung: *merge*) abhängig.

Persistenz

persistency.coobra (vgl. Abschnitt 6.4.11, S. 235ff.) modelliert ein Modul zum persistenzieren eines Teilsystems. Dazu delegiert es die Aufrufe des Persistenz-Kernmoduls (*core.persistency*) an das CoObRA2-Rahmenwerk von Christian Schneider [Sch07]. Das Modul delegiert zwar alle Aufrufe, bietet jedoch, auf Grund des Entwicklungsstands von CoObRA2, keine Transaktionen. Das Modul ist von der Konfiguration des Merkmals „CoObRA Protokoll" (O.4.1) (Markierung: *coObRAprotocol*) abhängig.

persistency.hibernate (vgl. Abschnitt 6.4.12, S. 237ff.) basiert auf der Integration des Hibernate-Rahmenwerks in Fujaba von Stefan Oehme [Öhm10]. Dazu delegiert es die Aufrufe des Persistenz-Kernmoduls (*core.persistency*) an das Hibernate-Rahmenwerk (die Unterstützung für Transaktionen ist noch nicht modelliert). Das Modul ist von der Konfiguration des Merkmals „Hibernate ORM" (O.4.2) (Markierung: *relationalDatabase*) abhängig.

Produktmodelle

product.ecore (vgl. Abschnitt 6.4.13, S. 239ff.) basiert auf dem Entwurf eines Ecore-Produktmodells von Udo Pöhner [Pöh09]. Die Arbeit beschreibt lediglich unterschiedliche Modellierungsansätze, d.h. es existiert noch kein verwendbares Produktmodell. Ein lauffähiges Modul wäre von der Konfiguration des Merkmals „EMF-Modell" (B.2.3) (Markierung: *emfModel*) abhängig.

product.filesystem (vgl. Abschnitt 6.4.14, S. 240ff.) modelliert ein Dateisystem-Produktmodell, das mit einem Teilbereich des Betriebssystems synchron gehalten wird. Ein Dateisystem-Arbeitsbereich eines Entwicklers ist eine Instanz dieses Produktmodells. Die Synchronisation erfolgt bidirektional: Einerseits werden Änderungen des Betriebssystem-Dateisystems in die Instanz übernommen, z.B. nach Bearbeitung einer Datei. Andererseits werden Änderungen an der Instanz in das Betriebssystem übertragen, z.B. nach einer Aktualisierung des Arbeitsbereichs. Dieses Modul unterstützt – dank Stefan Matthaei [Mat09] – die Delta-Speicherung von Dateien, ist vollständig implementiert und ausführlich getestet, und wurde bereits genutzt, um z.B. den Quellcode eines generierten SKMS zu versionieren. Das Modul ist von der Konfiguration des Merkmals „Dateisystem" (B.2.2) (Markierung: *filesystem*) abhängig.

product.usecase (vgl. Abschnitt 6.4.15, S. 247ff.) modelliert ein Produktmodell für Anwendungsfall-Diagramme. Instanzen dieses Modells können zwar versioniert, aber – im Gegensatz zu *product.filesystem* – nicht produktiv verwendet werden, da keine Anbindung an einen Diagrammeditor besteht. Zweck dieses Produktmodells ist vielmehr eine Untersuchung der Grenzen der Schnittstellen von *core.product*. Die notwendigen Hilfskonstruktionen zeigen, dass für graphbasierte Modelle (wie *product.ecore*) vermutlich erweiterte Schnittstellen notwendig sind. Das Modul ist von der Konfiguration des Merkmals „Anwendungsfall-Modell" (B.2.1) (Markierung: *useCaseModel*) abhängig.

Repositorien

repository.complexitem (vgl. Abschnitt 6.4.16, S. 251ff.) modelliert ein Repositorium für zusammengesetzte Elemente. Das Modul delegiert Anfragen zum Einspielen und Auslesen von Versionen an Historie und Speicher. In Bezug auf zusammengesetzte Elemente stellt es insbesondere sicher, dass aus den impliziten Parametern der Anfrage eine explizite Liste der betroffenen Elemente erstellt wird. Z.B. wird eine Anfrage zum Auslesen des Repositoriums-Wurzelelements zu einer Liste aller in ihm enthaltenen Elemente erweitert. Diese Auswahl kann entweder versionszentriert oder verwoben erfolgen (vgl. Abschnitt 4.3.4, S. 81ff.), und ist auch nur in Kombination mit dem Speicher für zusammengesetzte Elemente möglich, so dass eine Abhängigkeit zum Modul *storage.complex* besteht. Das Modul ist von der Konfiguration des Merkmals „Komplexe Elemente" (D.5.2) (Markierung: *complexitems*) abhängig.

repository.singleitem (vgl. Abschnitt 6.4.17, S. 253ff.) modelliert ein Repositorium für atomare Elemente, d.h., es unterstützt keine Wiederherstellung überlappend gespeicherter zusammengesetzter Elemente. Zusammengesetzte Elementen können zwar eingespielt werden, müssten aber immer vollständig (mit allen Kindelementen) gespeichert werden, was zu einem extrem hohen Speicherplatzbedarf führt. Die Abhängigkeiten im Merkmalsmodell verbieten daher Konfigurationen, die dieses Modul mit zusammengesetzten Elementen kombinieren. Das Modul ist von der Konfiguration des Merkmals „Atomare Elemente" (D.5.1) (Markierung: *singleitem1*) abhängig.

Server

server.base (vgl. Abschnitt 6.4.18, S. 254ff.) modelliert das zentrale Modul des Server-Teilsystems, das die Kommandos des Kommunikations-Teilsystems an ihre entsprechenden Server-Module delegiert. Das Modul legt fest, aus welchen Kernmodulen ein Server zusammengesetzt ist (genau ein Historien-Modul, genau ein Speichermodul, usw.). Außerdem ist es auch für die Instanzierung der konkreten Modul-Fabriken verantwortlich, d.h., es initialisiert diejenigen Module aus der Modulbibliothek, die anhand der Konfiguration ausgewählt wurden. Aus diesem Grund bestehen Abhängigkeiten zu nahezu jedem Modul (s. Abb. 6.16). Weiterhin modelliert der Server das Verhalten beim Einspielen und Auslesen von Versionen über die Kernmodul-Schnittstellen der verschiedenen Module, so dass diese unabhängig vom konkreten Bibliotheksmodul ablaufen. Es existiert bisher keine alternatives Servermodul, so dass *server.base* Teil jedes SKMS ist.

server.rmi (vgl. Abschnitt 6.4.19, S. 271ff.) verwendet das Modul *server.base* wieder und passt lediglich dessen Schnittstelle an die Vorgaben des „Java RMI"-Protokolls an. Damit lassen sich die Kommandos des Kommunikations-Teilsystems auch über eine Netzwerkverbindung an das Server-Teilsystem übertragen. Das Modul ist von der Konfiguration des Merkmals „Java RMI" (I.3.2) (Markierung: *javaRMI*) abhängig.

server.runtimeconfig (vgl. Abschnitt 6.4.20, S. 273ff.) verwendet das Modul *server.rmi* (und damit auch *server.base*) wieder und verändert den Instanzierungsprozess der Modul-Fabriken. Die statisch konfigurierte Methode aus *server.base* wird durch eine Serverfabrik ersetzt, die, anhand einer Merkmalsmenge, die entsprechenden Modul-Fabriken instanziert und dem Server zuweist. Diese Konfiguration erfolgt dynamisch zur Laufzeit, z.B. während der parametrisierbaren Tests. Diese Art der Konfiguration ist nur mit den in Abb. 6.16 gezeigten Merkmalen möglich, da jedes Merkmal eine andere Modulfabrik besitzt. Das Modul ist von der Konfiguration des Merkmals „Laufzeitkonfigurierbarer Server" (I.2) (Markierung: *runtimeConfigurable*) abhängig.

Speicher

storage.base (vgl. Abschnitt 6.4.21, S. 279ff.) modelliert einen einfachen Speicher, der jede Version eines Produktmodell-Elements als vollständige Grundfassung abspeichert. Zusammengesetzte Elemente werden vollständig mit allen Kindelementen – und nicht überlappend – gespeichert. Das Modul ist von der Konfiguration der Merkmale „Deltas" (D.4.1) und „Referenzen komplexer Elemente" (D.4.3) abhängig.

storage.complex (vgl. Abschnitt 6.4.22, S. 280ff.) modelliert einen Speicher, um zusammengesetzte Produktmodell-Elemente überlappend zu speichern. Atomare Elemente werden in einem eigenen Speicher abgelegt, so dass diese entweder als Grundfassungen (*storage.base*) oder als Deltas (*storage.delta*) gespeichert werden können. Dieses Modul muss bei Verwendung von *repository.complex* ebenfalls Teil eines SKMS sein. Das Modul ist von der Konfiguration des Merkmals „Referenzen komplexer Elemente" (D.4.3) (Markierung: *complexitemreferences*) abhängig.

storage.delta (vgl. Abschnitt 6.4.23, S. 283ff.) modelliert einen Speicher, um Produktmodell-Elemente, die auf der Schnittstelle *core.delta* basieren, platzsparend abzulegen. Das Modul ist von der Konfiguration des Merkmals „Deltas" (D.4.1) (Markierung: *directedDeltas*) abhängig. Anhand seiner Untermerkmale lässt sich noch die genau Art der Delta-Speicherung (vorwärts, rückwärts, gemischt) konfigurieren.

Sperren

synchronisation.lockitem (vgl. Abschnitt 6.4.24, S. 287ff.) modelliert einen Mechanismus, um Produktmodell-Elemente zu sperren. Dazu verwaltet das Modul eine Liste von gesperrten Kennungen und bietet Operationen an, um den Status zu ändern und abzufragen. Die Sperren kommen im Benutzermodul zum Einsatz (*user.flat*), könnten aber auch zum Synchronisieren parallel eintreffender Kommandos genutzt werden. Das Modul ist von der Konfiguration des Merkmals „Sperren" (D.11.2.2) (Markierung: *lock*) abhängig.

synchronisation.lockserver (vgl. Abschnitt 6.4.25, S. 289ff.) modelliert einen Mechanismus, der, im Gegensatz zum eben beschriebenen *synchronisation.lockitem*, den Status für einen ganzen Repositoriums-Server verwaltet. Die Sperre wird zum Synchronisieren parallel eintreffender Kommandos genutzt, ließe sich aber auch, über das Benutzermodul, zum Sperren des ganzen Servers einsetzen. Das Modul ist von der Konfiguration des Merkmals „Server" (D.11.1.2) (Markierung: *server*) abhängig.

Benutzerverwaltung

user.flat (vgl. Abschnitt 6.4.26, S. 290ff.) modelliert eine rudimentäre Benutzerverwaltung ohne Hierarchie oder Zugriffsrechte. Mit diesem Modul lassen sich nur Benutzer anlegen, damit sie dann den von ihnen eingespielten Versionen zugeordnet werden können. Zusätzlich kann ein Benutzer – über *synchronisation.lockitem* – Elemente in seinem Namen sperren, so dass nur noch er Änderungen an den gesperrten Elementen durchführen darf. Das Modul ist von der Konfiguration des Merkmals „Intern" (B.5.1.2) (Markierung: *internalusermanagement*) abhängig.

Arbeitsbereiche

workspace.base (vgl. Abschnitt 6.4.27, S. 292ff.) modelliert das zentrale Modul des Arbeitsbereichs-Teilsystem. Analog zu *server.base* legt es fest, welche Kernmodule im Arbeitsbereich vorkommen müssen (genau ein Produktmodell-Modul, usw.). Außerdem initialisiert es auch die Fabriken der konkreten Module. Da zurzeit keine alternativen Arbeitsbereichs-Module existieren, ist es Teil jedes konfigurierten SKMS.

workspace.runtimeconfig (vgl. Abschnitt 6.4.28, S. 295ff.) modelliert einen Arbeitsbereich, der – abhängig von der aktuellen Konfiguration des laufzeitkonfigurierbaren Servers – Elemente im Arbeitsbereich auswählen kann. Dieses Modul ist lediglich für die Verwendung in den parametrisierbaren Tests vorgesehen. Das Modul ist von der Konfiguration des Merkmals „Laufzeitkonfigurierbarer Server" (I.2) (Markierung: *runtimeConfigurable*) abhängig.

6.3. Die MOD2-SKM Kernkomponenten

Die Kernkomponenten modellieren die gemeinsam genutzten Komponenten des MOD2-SKM Domänenmodells. Ihr Aufgabe ist es, von den konkreten Modulen der Modulbibliothek (Abschnitt 6.4, S. 211) zu abstrahieren und somit eine Grundlage für die Modularisierung der SKMS zu schaffen. Wie bereits in Abschnitt 6.2.3 (s. S. 157) beschrieben, sind die Kernmodule gemäß „Separation of Concerns" aufgeteilt. Die Kernmodule selbst basieren auf Schnittstellen bzw. abstrakten Klassen, um durch Schnittstellen-Abstraktion (vgl. Abschnitt 6.2.1, S. 155ff.) eine einheitliche Schnittstelle für die jeweiligen Module zu definieren. Es wird verlangt, dass alle konkreten Module der Modulbibliothek die entsprechenden Schnittstellen implementieren. Eine Abhängigkeit von einer Schnittstelle der Kernmodule bedeutet nun, dass sich alle konkreten Module, die sie implementieren, austauschbar verwendet lassen [VAC+09].

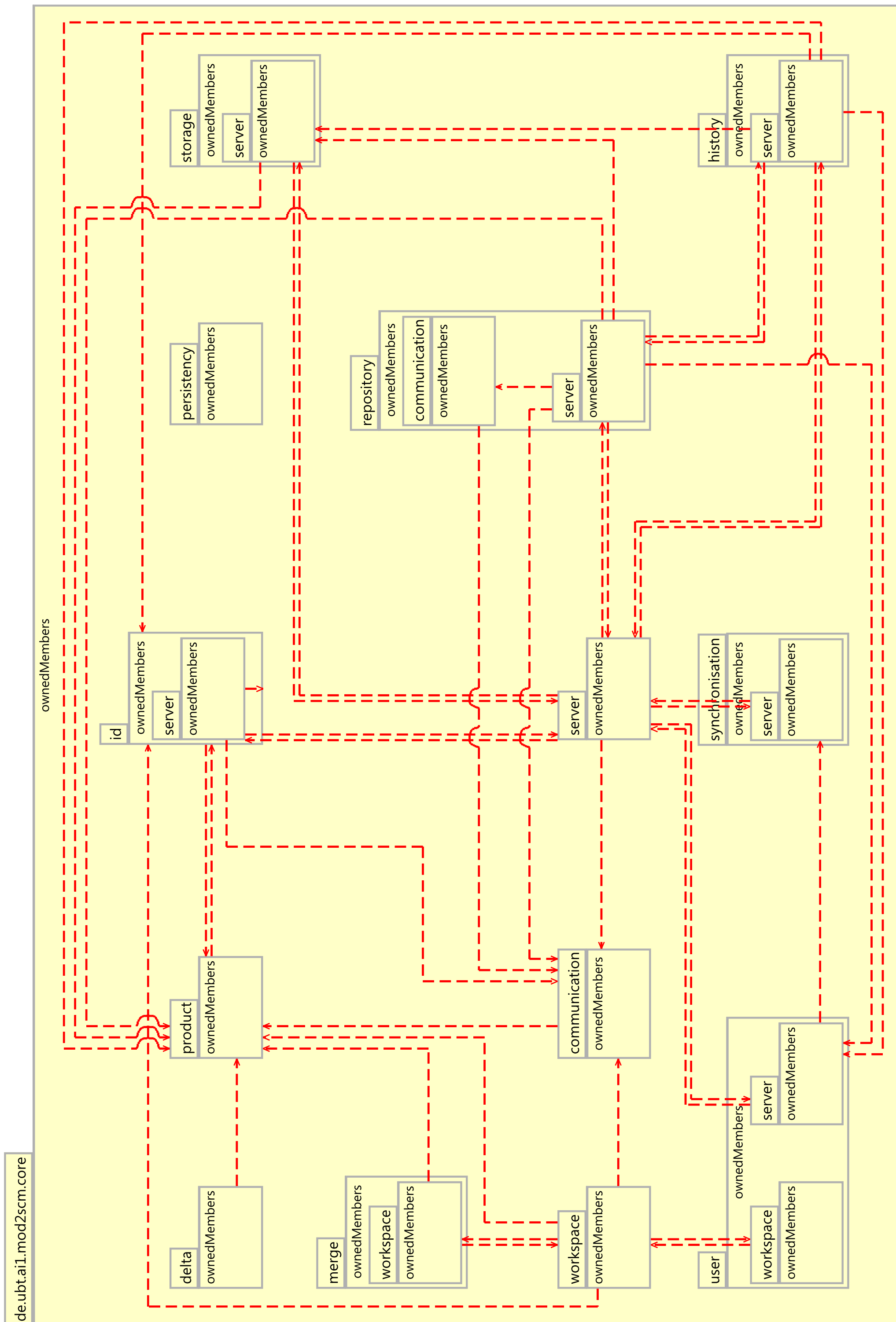


Abbildung 6.17.: Paketdiagramm der Kernkomponenten

Abbildung 6.17 zeigt die Abhängigkeiten der Kernmodule anhand ihrer privaten Paketimporte (vgl. Abschnitt 6.2, S. 155ff.). Diese werden in den nun folgenden Abschnitten für jedes Kernmodul detailliert beschrieben. Das Paketdiagramm zeigt alle ausgehenden Import-Beziehungen der Kernpakete, d.h. es existieren keine weiteren Import-Beziehungen zu den Paketen der Modulbibliothek. Damit enthalten die Kernmodule keine Abhängigkeiten von konkreten Implementierungen.

Die drei Pakete *core.communication*, *core.server* und *core.workspace* bilden die Client-Server-Struktur nach (vgl. Abschnitt 6.2.3, S. 161ff.), so dass auch keine Import-Beziehungen zwischen *core.server* und *core.workspace* existieren. Stattdessen importieren beide das Paket *core.communication*. Alle Kernpakete, die an eines der drei Teilsysteme gebunden sind (z.B die Historie an den Server), besitzen entsprechende Unterpakete, die auch immer das entsprechende Kernpaket importieren (im Falle der Historie importiert *core.history.server* auch *core.server*). Kernpakete ohne Teilsystem-Unterpaket (z.B. *core.product*) lassen sich unabhängig vom Teilsystem einsetzen.

Hinweise zur Modulbeschreibung

Im Folgenden werden nun die einzelnen Kernmodule und ihre Abhängigkeiten beschrieben. Um die Paketnamen im Text kurz zu halten, wird das Präfix *de.ubt.ai1* bzw. *de.ubt.ai1.mod2scm* ignoriert. Aus *de.ubt.ai1.mod2scm.core.history* wird so z.B. *core.history*. Da im ganzen Paketmodell nur genau ein einziges Paket mit Namen *core* existiert, sind die gekürzten Namen trotzdem eindeutig zuzuordnen. Jedes Klassendiagramm ist einem Paket zugeordnet, dessen Namen im Diagramm angezeigt wird. Um paketübergreifende Assoziationen darzustellen, muss auch die paketfremde Klasse angezeigt werden. Da Fujaba diese Klassen nicht kennzeichnet, werden sie in den Diagrammen mit zugeklappten Attributs- und Methodenfeldern präsentiert. In der zugehörigen Modulbeschreibung ist dann angegeben, aus welchem Paket die Klasse stammt.

6.3.1. Kernmodul: core.communication

Überblick

core.communication ist das Kernmodul für die Middleware-Kommunikation (vgl. Abschnitt 6.2.3, S. 161ff.). Es definiert die abstrakte Schnittstelle zwischen *core.server* (vgl. Abschnitt 6.3.10, S. 195ff.) und *core.workspace* (vgl. Abschnitt 6.3.14, S. 206ff.). In MOD2-SKM sollte jedes Modul für ein Kommunikationsverfahren auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.18 zeigt die Abhängigkeiten von *core.communication*. Die Kommunikation lässt sich über *util.logging* in ein Logbuch schreiben. Während der Kommunikation können Ausnahmen aus *core.exceptions* auftreten (vgl. Abschnitt 6.3.3, S. 177ff.). Und es existieren spezielle Datenstrukturen, um Elemente eines Produktmodells (Abhängigkeit zu *core.product*, vgl. Abschnitt 6.3.8, S. 188ff.) zu übertragen. Dabei kann es

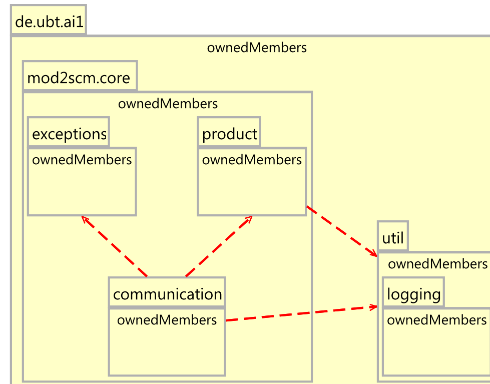


Abbildung 6.18.: Paketdiagramm: core.communication

sich jedoch um beliebige Produktmodell-Module handeln, da nur eine Abhängigkeit zum Produktmodell-Kernmodul besteht.

Detailbeschreibung

Das Kernmodul *core.communication* definiert eine abstrakte Schnittstelle zur Datenübertragung zwischen einem – ebenfalls abstrakten – Arbeitsbereich-Client und Repositoriums-Server (vgl. Abschnitt 6.2.1, S. 155ff.). Ihr Aufbau ist im Klassendiagramm in Abbildung 6.19 dokumentiert: Jeder Repositoriums-Server bietet eine Schnittstelle (*IMODSCMServerAccess*), um ein Kommando auszuführen (*executeCommand()*). Jedes Modul implementiert später seine eigenen Kommandos. , so dass das Kernmodul ebenfalls nur eine Schnittstelle vorgibt (*IMOD2SCMServerCommand*). Jedes Kommando lässt sich duplizieren (*duplicate()*), kennt seinen Ausführungserfolg (*successful*) und sein Ausführungsergebnis (*getResult()*). Jedes konkrete Kommando legt selbst fest, welche Daten an den Server übertragen werden. Es muss daher eine Methode zur Vollständigkeitsprüfung implementieren (*hasRequiredData()*). So lässt sich in einem konkreten Arbeitsbereich die Übertragung unvollständiger Kommandos verhindern.

Der Datenaustausch zwischen Arbeitsbereich-Client und Repositoriums-Server erfordert häufig die Übertragung von Elementen des Produktmodells (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.). Zu diesem Zweck existiert eine spezielle Datenstruktur (*PMIList*). Sie lässt sich u.a. leicht mit dem Wurzelelement eines ganzen Teilbaums von Produktmodell-Elementen erzeugen (*PMIList(item: IComplexProductModelItem)*) und bietet Methoden zum schnellen Zugriff auf ihre Elemente (*contains()*, *select()*) bzw. die Objekt-Kennung (*getOIDList()*). *PMIList* setzt dabei nur abstrakte Datentypen für Produktmodell-Elemente voraus, so dass sich die Liste mit beliebigen Produktmodell-Modulen verwenden lässt.

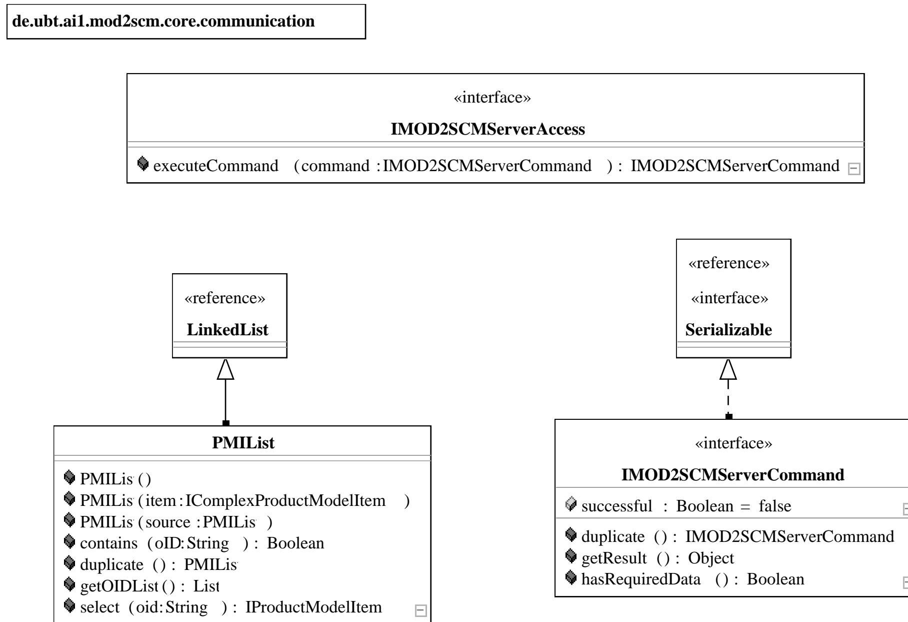


Abbildung 6.19.: Klassendiagramm: core.communication

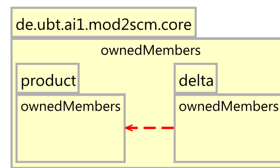


Abbildung 6.20.: Paketdiagramm: core.delta

6.3.2. Kernmodul: core.delta

Überblick

core:delta ist das Kernmodul für Verfahren zur Delta-Berechnung und -Anwendung. Es definiert eine abstrakte Schnittstelle, die zur platzsparenden Speicherung der Elemente verwendet werden kann (vgl. Abschnitt 4.3.6, S. 89ff.). In MOD2-SKM sollte jedes Produktmodell, das Delta-Speicherung unterstützt, auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.20 zeigt die Abhängigkeiten von *core.delta*. Es besteht nur eine Abhängigkeit zu *core.product* (vgl. Abschnitt 6.3.8, S. 188ff.), da es sich bei einem deltifizierbaren Produktmodell-Element um einen speziellen Subtypen eines Produktmodell-Elements handelt.

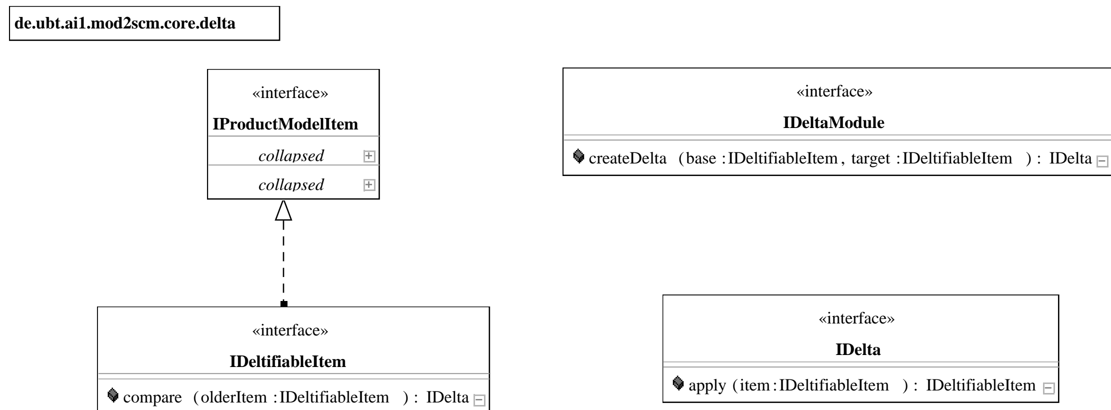


Abbildung 6.21.: Klassendiagramm: core.delta

Detailbeschreibung

Das Kernmodule *core.delta* definiert eine abstrakte Schnittstelle zur Implementierung von Verfahren zur Delta-Berechnung und -Anwendung. Ein Produktmodell-Element kann mit Hilfe der sogenannten „Delta-Speicherung“ speicherplatzfreundlich im Repository abgelegt werden (vgl. Abschnitt 4.3.6, S. 89ff.). Zum Einsatz eines solchen Speicherverfahrens müssen die Produktmodell-Elemente zwei Voraussetzungen erfüllen: Zum einen ist ein Verfahren zur Berechnung eines Deltas aus zwei Elementen notwendig. Zum anderen muss es auch möglich sein, ein Delta auf ein vollständiges Element anzuwenden, um ein entsprechend geändertes Element zu erhalten.

Dies wird durch die beiden Schnittstellen *IDeltifiableItem* und *IDelta* erreicht, die in Abbildung 6.21 gezeigt sind. Ein Produktmodell-Element, das als Delta gespeichert werden soll, muss nicht nur die Schnittstelle *IProductModelItem* (vgl. Abschnitt 6.3.8, S. 188ff.) sondern die Sub-Schnittstelle *IDeltifiableItem* implementieren. Diese verlangt vom Produktmodell-Designer die Implementierung einer konkreten Methode *createDelta(x:IDeltaItem)*, die ein gerichtetes Delta von x nach *this* liefert.

Der konkrete Typ des Deltas wird ebenfalls vom Produktmodell-Designer definiert, da er vom konkreten Produktmodell (z.B. Dateisystem, ecore-Modell) abhängt, bzw. sogar vom Typ des Produktmodell-Elements (z.B. Textdatei, Binärdatei, Videodatei, etc.). Einzige Vorgabe an die Deltas ist die Implementierung der Schnittstelle *IDelta*. Sie legt fest, dass für das konkrete Delta die Methode *applyDelta(x:IDeltaItem)* definiert werden muss. Sie wendet das Delta *this* auf das vollständige Element x an und liefert ein entsprechend geänderte Produktmodell-Element zurück.

Die Schnittstellen abstrahieren (vgl. Abschnitt 6.2.1, S. 155ff.) die Delta-Berechnung und -Anwendung von ihrer konkreten Implementierung. Somit lassen sich Delta-Speicher, die auf diesem Kernmodul basieren, für beliebige Produktmodelle nutzen. Der abstrahierte Zusammenhang zwischen zwei deltifizierbaren Items $x, \acute{x}$ und ihrem gerichteten Delta $\vec{\delta}(x, \acute{x})$ lässt sich wie folgt beschreiben: Sei *createDelta(x:IProductModelItem, y:IProductModelItem)* die Operation, die ein gerichtetes Delta $\vec{\delta}(x, \acute{x})$ erzeugt und *applyDelta(*

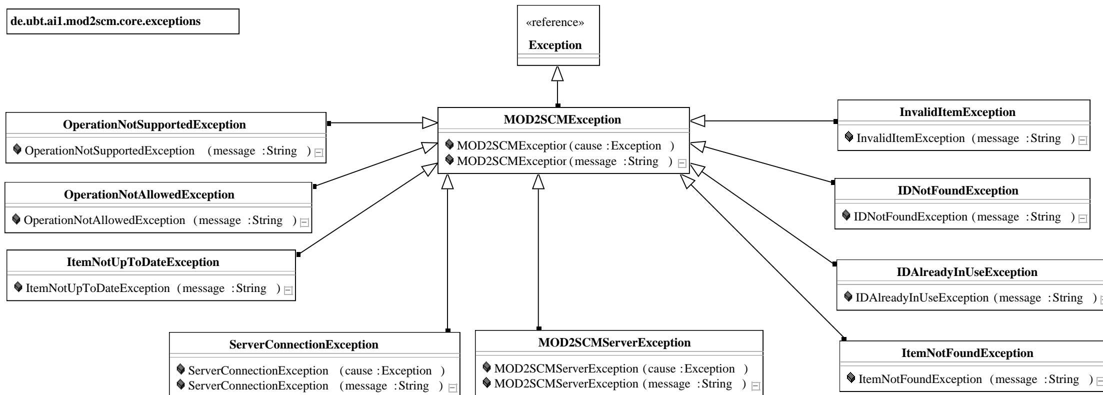


Abbildung 6.22.: Klassendiagramm: core.exceptions

x:IProductModelItem, a:Delta) die ein gerichtetes Delta a auf ein Produktmodell-Item x anwendet. Dann muss gelten:

$$\acute{x} = applyDelta(x, createDelta(x, \acute{x}))$$

Letztendlich lassen sich unterschiedliche Delta-Verfahren als Variationspunkt auffassen, so dass auch ein *IDeltaModule* existiert. Für jedes Verfahren wird ein eigenes Modul implementiert, an das die Aufrufe von *IDeltifiableItem.compare()* delegiert werden – wie z.B. im Dateisystem-Produktmodell (vgl. Abschnitt 6.4.14, S. 240ff.).

6.3.3. Kernmodul: core.exceptions

Überblick

core.exceptions ist das Kernmodul für die Ausnahmen (engl. exceptions) die zur Laufzeit auftreten können. Anhand des Ausnahmen-Typs sowie der Fehlerbeschreibung, können SKMS-Benutzern geeignete Fehlermeldungen angezeigt werden.

Abhängigkeiten

core.exceptions besitzt keine Abhängigkeiten zu anderen Modulen.

Detailbeschreibung

Abbildung 6.22 zeigt die in *core.exceptions* definierten Ausnahmen. Da Fujaba das Domänenmodell in Java-Quellcode übersetzt [Zün02], sind diese als Subklasse der Java-Basisklasse für Ausnahmen (*java.lang.Exception*, [Ull09]) modelliert. Es existieren folgende Ausnahme-Typen, die alle Subklassen von *MOD2SCMException* sind:

- *IDAlreadyInUseException:* Eine Kennung ist bereits belegt, z.B. beim Versuch, ein Repositorium anzulegen, das bereits existiert.

- *IDNotFoundException:* Eine Kennung kann nicht gefunden werden, z.B. bei der Angabe einer Markierung (engl. tag), die gar nicht existiert.
- *InvalidItemException:* Der Typ eines Elements ist ungeeignet, z.B. beim Versuch ein nicht-deltifizierbares Element in einem Delta-Speicher abzulegen.
- *ItemNotFoundException:* Ein Objekt kann nicht gefunden werden, z.B. bei der Angabe einer Versionskennung, die nicht existiert.
- *ItemNotUpToDateException:* Ein Element ist nicht auf dem neusten Stand, z.B. beim Versuch ein veraltetes Element in ein Repositorium einzuspielen.
- *MOD2SCMServerException:* Ein Behälter für alle Ausnahmen, die vom Repositoriums-Server an einen Arbeitsbereich-Client weitergeleitet werden. Damit wird die konkrete Ausnahme versteckt (vgl. Abschnitt 6.2.1, S. 155ff.), so dass für neue Ausnahme-Typen die Schnittstelle nicht verändert werden muss.
- *OperationNotSupportedException:* Eine Operation existiert nicht. Diese Ausnahme dient dazu, während der Modellierung neuer Module noch nicht modellierte Methoden zu kennzeichnen.
- *OperationNotAllowedException:* Ein Operation darf nicht ausgeführt werden, z.B. beim Versuch, eine geänderte Fassung eines gesperrten Elements in ein Repositorium einzuspielen.
- *ServerConnectionException:* Ein Behälter für alle Ausnahmen, die die Kommunikations-Middleware zwischen Arbeitsbereich-Client und Repositoriums-Server betreffen. So werden die Ausnahmen der konkreten Middleware versteckt (vgl. Abschnitt 6.2.1, S. 155ff.), so dass für Ausnahme-Typen einer neuen Middleware die Schnittstelle nicht verändert werden muss.

6.3.4. Kernmodul: core.history

Überblick

core.history ist das Kernmodul für die Historie eines Produktmodell-Elements. Es ist ein abstraktes Modell für Versionsräume (vgl. Abschnitt 4.3.1, S. 79ff.), das Schnittstellen für Historien und Versionen bietet, aber keine Annahmen über konkrete Beziehungen trifft. In MOD2-SKM sollte jedes Modul für eine Historie auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.23 zeigt die Abhängigkeiten von *core.history*. Es besteht eine – bidirektionale – Bindung an den abstrakten Server (*core.server*) und das abstrakte Repositorium (*core.repository*), da eine Historie einem Repositorium auf einem Server zugeordnet ist. Ebenso benötigt ein Server ein auf *core.history* basierendes Modul, das von seinen

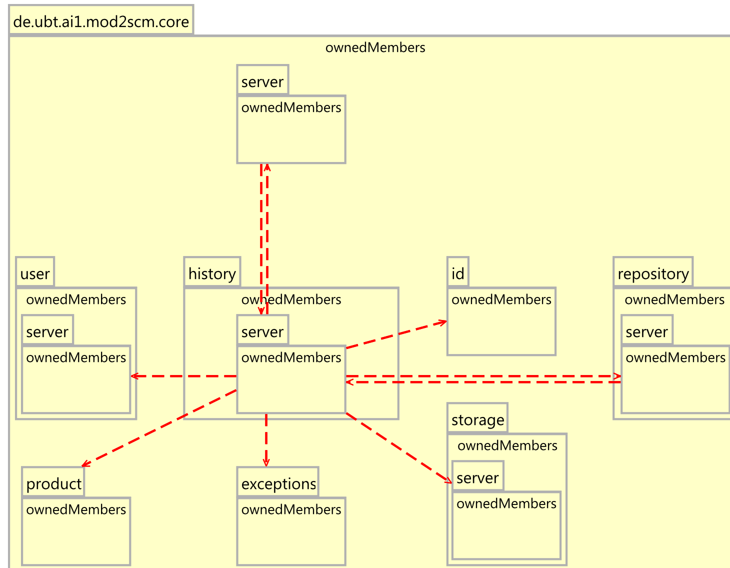


Abbildung 6.23.: Paketdiagramm: core.history

Repositorien genutzt wird. Weiterhin bestehen uni-direktionale Abhängigkeiten zum abstrakten Produktmodell (*core.product*), da eine Version den Zustand eines bestimmten Produktmodell-Elements bezeichnet. Diese wird in einem beliebigen Speicher-Modul abgelegt, das auf *core.storage* basiert. Beim Anlegen neuer Versionen wird eine neue Versionskennung benötigt, die durch ein auf *core.id* basierendes Modul vergeben wird. Außerdem gibt es auch immer einen Benutzer, der für das Anlegen einer neuen Version verantwortlich ist (*core.user*).

Bei all diesen Modulen handelt es sich um Kernmodule, d.h. es existieren keine Abhängigkeiten zu konkreten Implementierungen. Eine Historie kann damit unterschiedliche Module zur Speicherung, VID-Vergabe und Benutzerverwaltung verwenden und ist auch unabhängig vom konkreten Produktmodell. Außerdem lässt sich eine Historie mit unterschiedlichen Repositorien- und Server-Modulen einsetzen.

Detailbeschreibung

Das Kernmodul *core.history* definierte Schnittstellen für ein Modell eines Versionsraums, die in Abbildung 6.24 gezeigt sind. Alle Historien-Module (*IHistoryModule*) laufen auf einem Repositoriums-Server und sind daher Subtypen eines *IMOD2SCMServerModule*. Jedes Modul verwaltet (über die 1:n-Komposition „maintains“) seine Historien (*IHistory*), die über die Methode *createVersionSet* angelegt werden. In Anlehnung an das Fabrik-Entwurfsmuster [FF04] ist der Aufruf zum Erstellen neuer Historien-Objekte damit unabhängig vom konkreten Typ (vgl. Abschnitt 6.2.1, S. 155ff.).

Jede Historie (*IHistory*) ist über die Objektkennung (*oid*, vgl. Abschnitt 4.2.2, S. 76ff.) an ein Produktmodell-Element gekoppelt, deren Versionen (*AbstractVersion*) sie enthält. Eine Änderung der Objektkennung eines Elements bedeutet somit auch die Zuordnung zu

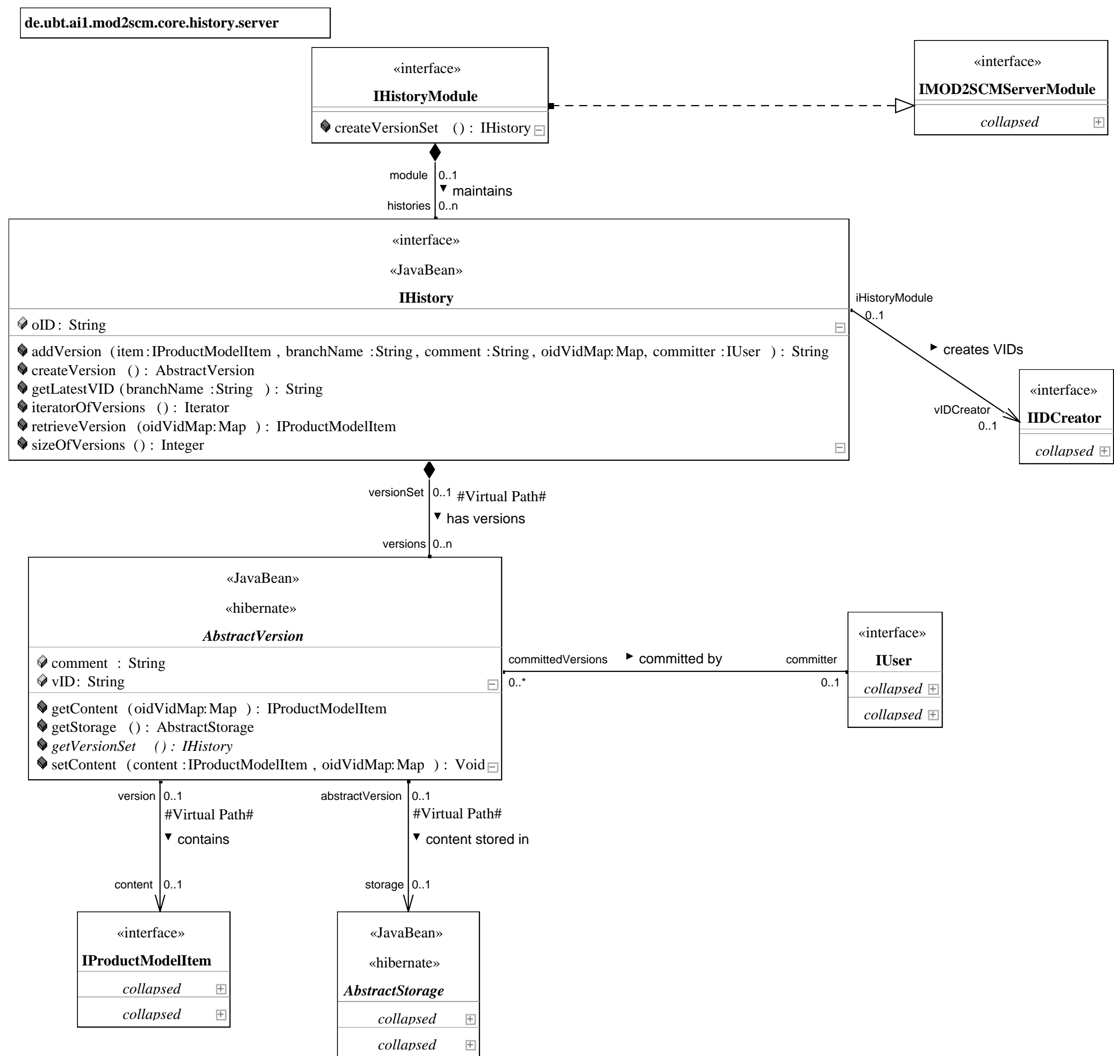


Abbildung 6.24.: Klassendiagramm: core.history

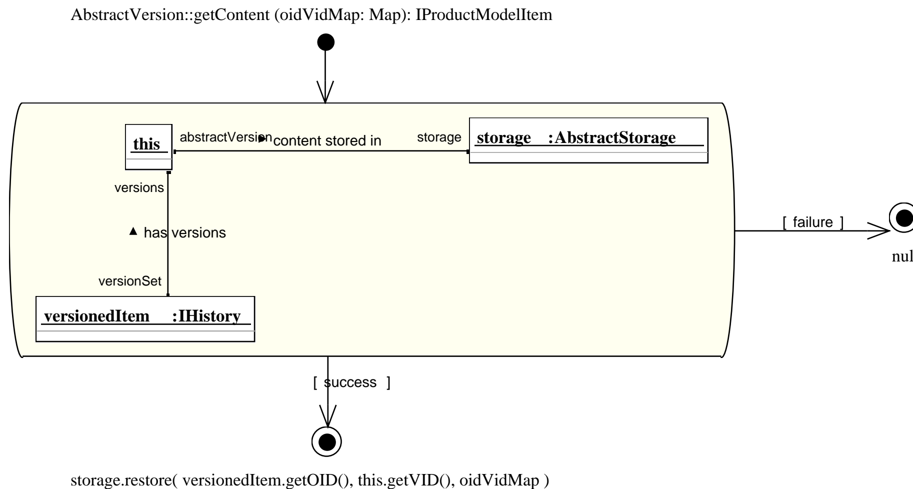


Abbildung 6.25.: Storydiagramm: AbstractVersion.getContent()

einer neuen Historie. Jede Version besitzt – außer einem Kommentar (*comment*) und einem Autoren (*IUser* aus *core.user*, vgl. Abschnitt 6.3.13, S. 203ff.) – eine Versionskennung (*vid*, vgl. Abschnitt 4.3.2, S. 79ff.), so dass über das Tupel (oid, vid) eine Version eindeutig identifiziert werden kann. Beim Hinzufügen einer neuen Version (*IHistory.addVersion()*) wird ihr eine neue Versionskennung zugewiesen und an den Aufrufer zurückgegeben (d.h. eine Versionskennung ist immer eine Zeichenkette). Die Versionskennung wird dabei von einem beliebiges ID-Modul erstellt, das die Schnittstelle *IIDCreator* (aus *core.id*, vgl. Abschnitt 6.3.5, S. 182ff.) implementiert.

Das abstrakte Produktmodell-Element (*IProductModelItem* aus *core.product*, vgl. Abschnitt 6.3.8, S. 188ff.), das jede Version (*AbstractVersion*) eindeutig referenziert, wird mit Hilfe eines Speichermechanismus (*AbstractStorage*) gespeichert. Dies wird jedoch mit Hilfe des virtuellen Pfads „contains“ und den Methoden *AbstractVersion.getContent()* und *AbstractVersion.setContent()* verborgen (vgl. Abschnitt 6.2.1, S. 155ff.). Abbildung 6.25 zeigt das Storydiagramm der Methode *getContent()*: Die Historie dient lediglich zur Bestimmung des Tupels (oid, vid). Dann wird die Anfrage an den Speichermechanismus weitergeleitet, der das vollständige Produktmodell-Element zurückliefert. So wird z.B. verborgen, dass ein Element nur als Delta in Bezug auf eine Grundfassung vorliegt. Um welchen Speichermechanismus es sich konkret handelt, ist damit unabhängig von der Historie.

core.history modelliert lediglich eine unstrukturierte Versionsmenge, denn es sind keine Beziehungen zwischen Versionen definiert – dies erfolgt erst in den konkreten Modulen (vgl. Abschnitt 6.4.1, S. 213ff.). Selbst die Assoziation zwischen *IHistory* und *AbstractVersion* ist als virtueller Pfad definiert (vgl. Abschnitt 6.1, S. 144ff.), d.h. sogar die interne Struktur der Historie wird nicht vorgegeben. Die Schnittstelle modelliert nur das Ablegen eines Produktmodell-Elements unter einem eindeutigen Tupel (oid, vid) (*IHistory.addVersion()*) und dem Zugriff auf bereits gespeicherte Versionen (*IHistory.retrieveVersion()*). Dabei ist jedoch anzumerken, dass die Signatur von *addVer-*

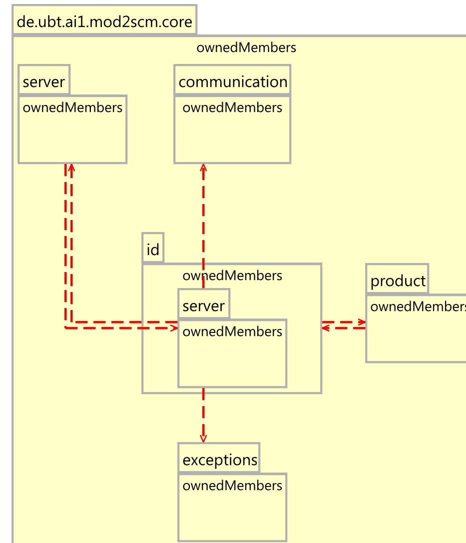


Abbildung 6.26.: Paketdiagramm: core.id

sion() Informationen enthält, die nicht jedes Historien-Modul benötigt, z.B. den Parameter *branchName* zur Identifikation des Seitenzweigs (siehe *history.tree*, vgl. Abschnitt 6.4.3, S. 216ff.). Hier setzt die Kopplung zwischen einer konkreten Historie und ihren Kennungen der Abstraktion Grenzen. Es bestünde die Möglichkeit, aus zwei alternativen Methoden – mit Hilfe einer Merkmalsmarkierung – auszuwählen, doch das widerspräche der Anforderung an die Kernmodule, dass sie nur gemeinsam genutzte Komponenten enthalten. Eine „sauberere“ Alternative bietet evtl. das Verbergen der konkreten Kennung mit Hilfe eines abstrakten ID-Objektes (vgl. Abschnitt 6.2.1, S. 155ff.). Oder aber das Implementieren einer weiteren *addVersion()*-Methode in der konkreten Historie, mit dem Parameter *branchName* in der Signatur. Diese Lösungsansätze wurden jedoch nicht mehr in dieser Arbeit verfolgt.

6.3.5. Kernmodul: core.id

Überblick

core.id ist das Kernmodul für die Erstellung und Übersetzung von Kennungen. Es bietet sowohl eine Schnittstelle für Module die Kennungen erzeugen (vgl. Abschnitt 4.2.2, S. 76ff. und vgl. Abschnitt 4.3.2, S. 79ff.), als auch eine Schnittstelle zur Übersetzung, um z.B. Markierungen (engl. tags) in die zugehörigen (oid, vid)-Tupel zu übersetzen. In MOD2-SKM sollte jedes Modul für Kennungen auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.26 zeigt die Abhängigkeiten von *core.id*. Das Paket selbst enthält die Kernkomponenten für das Anlegen von Kennungen. Diese sind (bidirektional) mit *core.product*

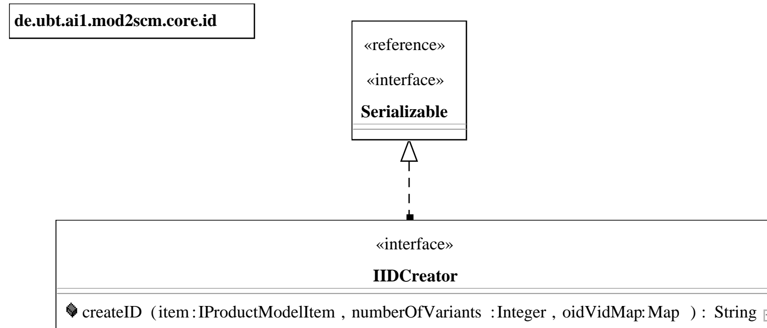


Abbildung 6.27.: Klassendiagramm: core.id

verbunden, da zum einen die Kennung für ein Produktmodell-Element erzeugt wird, aber zum anderen auch jedes Produktmodell ein Modul zum Erzeugen der Objektkennung benötigt (vgl. Abschnitt 4.2.2, S. 76ff.). Das Paket *core.id.server* enthält Komponenten zum Anlegen – und insbesondere Übersetzen – von Kennungen auf einem Repositoriums-Server (Abhängigkeit von *core.server*). Damit diese Module vom Arbeitsbereich aus angesprochen werden können, besteht ebenfalls eine Abhängigkeit von *core.communication*. Alle Abhängigkeiten bestehen zu Kernmodulen, so dass sich Kennungen unabhängig vom konkreten Produktmodell oder Repositoriums-Server vergeben lassen.

Detailbeschreibung

Das Kernmodul *core.id* definiert die Schnittstelle für Module zur Kennungserzeugung, z.B. für Objekt- oder Versionskennungen (vgl. Abschnitt 4.2.2, S. 76ff. und vgl. Abschnitt 4.3.2, S. 79ff.). Dazu wird die Annahme getroffen, dass jedes versionierte Element über eine Objekt- und eine Versionskennung identifiziert wird. Sein Unterpaket *core.id.server* bietet eine Schnittstelle für zusätzliche Identifikationsmethoden, z.B. Markierungen (engl. tags) oder Regeln (neuste Version). Beide Schnittstellen sind in die Standardabläufe zum Einspielen (engl. commit) und Aktualisieren (engl. update) eingebunden (vgl. Abschnitt 6.4.18, S. 254ff.).

Abbildung 6.27 zeigt das Klassendiagramm von *core.id* und die Schnittstelle für Module zur Kennungsgenerierung (*IIDCreator*). Sie besitzt genau eine Methode zum Erzeugen einer Kennung (eine Zeichenkette (engl. string)), die als Parameter ein Produktmodell-Element, die Anzahl der Varianten und eine Liste von Tupeln (oid, vid) erhält. Wie beim Anlegen einer neuen Version (vgl. Abschnitt 6.3.4, S. 178ff.), benötigt nicht jedes Modul alle Parameter (vgl. Abschnitt 6.4.4, S. 222ff.). Doch eine Anpassung der Schnittstelle, in Abhängigkeit von der Konfiguration, würde mit der Anforderung an die Kernmodule brechen, dass sie nur gemeinsam genutzte Elemente enthält.

Abbildung 6.28 zeigt das Klassendiagramm von *core.id.server* und enthält zwei Schnittstellen. Zum einen ein abstraktes Server-Modul (*AbstractIDCreatorModule*), um Kennungsgeneratoren auf dem Repositoriums-Server einzusetzen. Dies verlangt eine Fabrik-Methode, die – unabhängig vom konkreten Typ – einen Generator anlegt (*createID-Creator()*). Zum anderen eine Schnittstelle (*IIDTranslator*) zur Übersetzung zusätzlicher

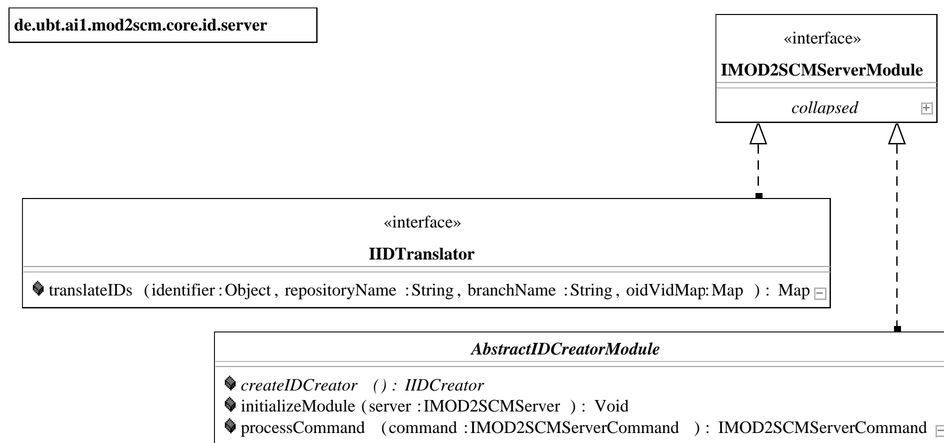


Abbildung 6.28.: Klassendiagramm: core.id.server

Identifikationsmerkmale in (oid, vid)-Tupel. Diese müssen eine Methode implementieren, die auf Basis einer zusätzlichen Kennung (Parameter: *identifier*) – und weiteren Identifikationsmerkmalen – eine Liste von (oid, vid)-Tupeln zurückgibt. Diese sind dann mit den Schnittstellen in den Kernmodulen der Historie (vgl. Abschnitt 6.3.4, S. 178ff.) und des Speichers (vgl. Abschnitt 6.3.11, S. 200ff.) kompatibel, die einzelne Elemente nur anhand des (oid, vid)-Tupels identifizieren.

In einem MOD2-SKM-Repositorium wird ein Elemente über das Tupel aus Objekt- und Versionskennung, (oid, vid), eindeutig referenziert. Das Kernmodul *core.id* besitzt nur Abhängigkeiten zu anderen Kernmodulen, so dass die Berechnung der Kennung unabhängig vom konkreten Produkt- oder Versionsmodell erfolgen kann. Zusätzliche Identifikationsverfahren lassen sich – ebenfalls unabhängig von konkreten Modulen – realisieren, solange sie sich als Abbildung auf (oid, vid)-Tupel modellieren lassen.

6.3.6. Kernmodul: core.merge

Überblick

Das Modul *core.merge* ist das Kernmodul für Verschmelzungsverfahren (vgl. Abschnitt 4.4.1, S. 92ff.). Es bietet sowohl eine Schnittstelle für die Verfahren selbst, als auch Datenstrukturen, um den Zustand der beteiligten Produktmodell-Elemente zu verfolgen.

Abhängigkeiten

Abbildung 6.29 zeigt die Abhängigkeiten von *core.merge*. Das es lediglich im Arbeitsbereich-Teilsystem verwendet wird, existiert nur das Unterpaket *workspace*, das entsprechenden von *core.workspace* (bidirektional) abhängig ist. Verschmelzungsmodule greifen auf die zu verschmelzenden Produktmodell-Elemente (Abhängigkeit zu *core.product*) im Arbeitsbereich zu, während der Arbeitsbereich, bei der Synchronisation mit dem Repositorium, das Modul über Konflikte benachrichtigt (vgl. Abschnitt 4.4.1, S. 92ff.). Bei

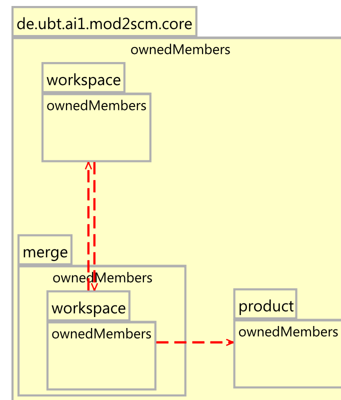


Abbildung 6.29.: Paketdiagramm: core.merge

beiden Modulen handelt es sich um Kernmodule, so dass die Verschmelzungsmodule unabhängig vom konkreten Arbeitsbereich oder Produktmodell modelliert werden können. Da ein Verschmelzungsverfahren i.d.R. für einzelne Produktmodelle implementiert wird, existiert dann vom konkreten Verschmelzungsmodul eine Abhängigkeit zu einem konkreten Produktmodell (vgl. Abschnitt 6.4.10, S. 233ff.).

Detailbeschreibung

Das Kernmodul *core.merge* definiert die Schnittstellen und Datenstrukturen für Verschmelzungsmodule, die zwei Produktmodell-Elemente im Arbeitsbereich miteinander verschmelzen (vgl. Abschnitt 4.4.1, S. 92ff.). Die Schnittstelle ist in den Ablauf zur Aktualisierung des Arbeitsbereichs eingebunden, so dass eine Benachrichtigung bei konkurrierenden Änderungen erfolgt (siehe auch *core.workspace*, vgl. Abschnitt 6.3.14, S. 206ff.).

Abbildung 6.30 zeigt das Klassendiagramm von *core.merge*. Da die Verschmelzungsmodule nur im Arbeitsbereich eingesetzt werden, implementiert *IMergeModule* die Schnittstelle *IWorkspaceModule* aus *core.workspace* (vgl. Abschnitt 6.3.14, S. 206ff.). Die drei Methoden der Schnittstelle dienen dazu, zwei Produktmodell-Elemente zu verschmelzen (*merge()*) und das Modul über einen Konflikt (*notifyMergeNecessary()*) bzw. den Abschluss eines Verschmelzungsvorgangs zu informieren (*markAsMerged()*). Alle drei Methoden nutzen Objekte vom Typ *WSInfo* aus *core.workspace*, d.h. es handelt sich um Produktmodell-Elemente einschließlich ihrer Versionsinformationen (vgl. Abschnitt 6.3.14, S. 206ff.).

Die *MergeInfo*-Objekte speichern Informationen über den Zustand eines Konflikts (*mergeSuccessful*) bzw. die beteiligten Elemente (Assoziationen „info about“ und „is partner“). Da es sich um uni-direktionale Assoziationen handelt, wird die Existenz eines Verschmelzungsmoduls bzw. die Beteiligung an einem Konflikt vor dem Produktmodell verborgen (vgl. Abschnitt 6.2.1, S. 155ff.).

Das Kernmodul abstrahiert von konkreten Verschmelzungsverfahren wie z.B. Zwei- oder Drei-Wege-Verschmelzen, indem es das Modul auf eine Schnittstelle zur Benachrichtigung, einschließlich Statusverwaltung, reduziert. Genauso wird der Grad der Auto-

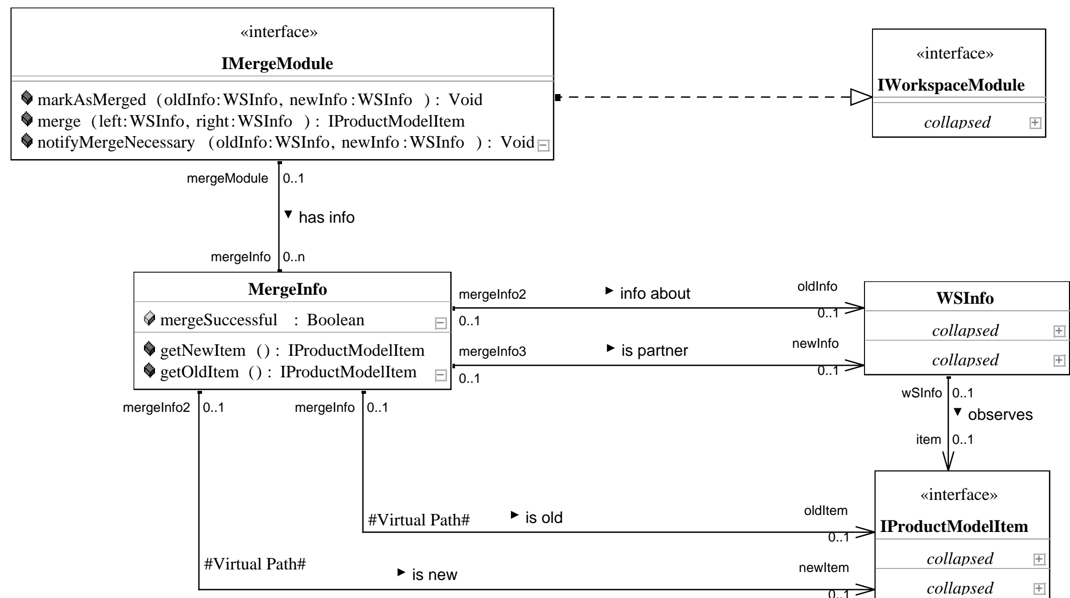


Abbildung 6.30.: Klassendiagramm: core.merge

matisierung beim Verschmelzen (automatisiert oder mit Benutzereingabe) gekapselt. Alle Abhängigkeiten von *core.merge* sind ebenfalls Kernmodule, so dass sich die Verschmelzen-Module unabhängig vom konkreten Arbeitsbereich oder Produktmodell modellieren lassen. Da ein Algorithmus zum Verschmelzen jedoch selten mit Produktmodell-Elementen von mehr als einem konkreten Produktmodell umgehen kann, existiert eine entsprechende Abhängigkeit unter den konkreten Modulen (vgl. Abschnitt 6.4.10, S. 233ff.).

6.3.7. Kernmodul: core.persistency

Überblick

core.persistency ist das Kernmodul für die Anbindung von Persistenzmechanismen. Es bietet eine Schnittstelle für einen Repositoriums-Server (vgl. Abschnitt 6.3.10, S. 195ff.), um seinen Zustand – transaktionsgestützt – zu persistenzieren. In MOD2-SKM sollte jeder Adapter [FF04] für einen Persistenzmechanismus auf diesem Kernmodul basieren.

Abhängigkeiten

Es existieren keine Abhängigkeiten zu anderen Modulen. Die konkreten Persistenzmodule besitzen jedoch oft Abhängigkeiten zu externen Bibliotheken eines bereits existierenden, generischen Persistenzmechanismus (vgl. Abschnitt 6.4.11, S. 235ff.).

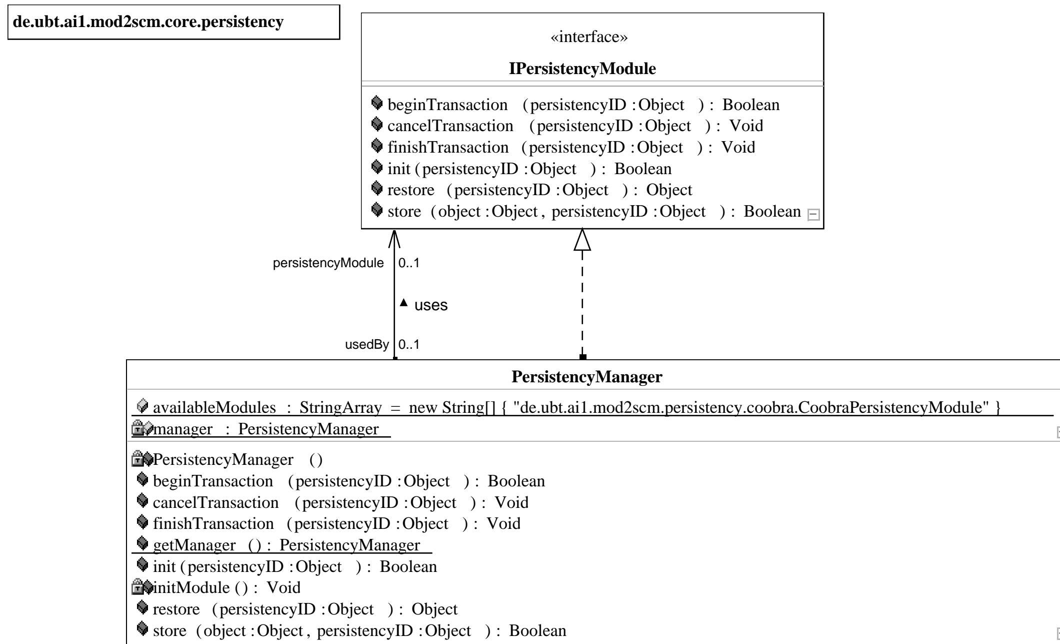


Abbildung 6.31.: Klassendiagramm: core.persistency

Detailbeschreibung

Das Kernmodul *core.persistency* definiert eine Schnittstelle, um bereits bestehende generische Persistenzmechanismen anzubinden. Theoretisch ist es sogar möglich, eigene Mechanismen zu modellieren, was auf Grund der benötigten ACID-Transaktionen [Ehr06] sehr aufwändig erscheint.

Abbildung 6.31 zeigt das Klassendiagramm von *core.persistency*. *IPersistencyModule* definiert eine einheitliche Schnittstelle für die Persistenzschicht, so dass die konkreten Persistenzmodule die proprietäre Schnittstelle des existierenden Persistenzmechanismus verstecken (vgl. Abschnitt 6.2.1, S. 155ff.). Dies entspricht dem Adapter-Entwurfsmuster [FF04]. Die fünf verlangten Methoden dienen zum einen zum Starten (*beginTransaction()*) und Beenden (*finishTransaction()*) bzw. Abbrechen (*cancelTransaction()*) von Transaktionen. Zum anderen dienen sie zum Speichern (*store()*) und Wiederherstellen (*restore()*) eines Objekts.

Dazu muss zunächst ein persistenter Speicher initialisiert werden (*init()*). Diesem Speicher wird eine Kennung (*persistencyID*) zugewiesen, die nun bei allen Aufrufen der o.g. Methoden verwendet wird, um den Aufruf diesem Speicher zuzuordnen. Dadurch abstrahiert die Schnittstelle von der Granularität der gespeicherten Daten (vgl. Abschnitt 6.2.1, S. 155ff.), und es obliegt dem Modellierer eines konkreten Servers (vgl. Abschnitt 6.4.18, S. 254ff.), welche Komponenten in separaten Speichern abgelegt werden. So kann z.B. der vollständige Server (mit mehreren Repositorien) unter dem Servernamen abgelegt werden. Oder aber jedes Repositorium erhält (unter seinem Namen) seinen eigenen Speicher.

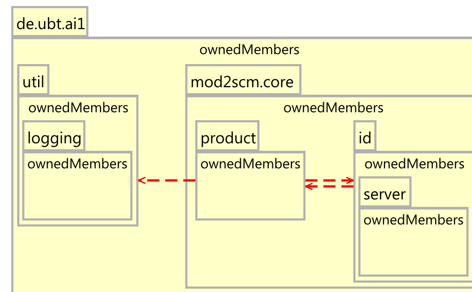


Abbildung 6.32.: Paketdiagramm: core.product

Prinzipiell ist es möglich, mehrere Persistenzmodule parallel zu verwenden, um z.B. verschiedene Repositorien mit unterschiedlichen Persistenzverfahren zu speichern. Für diesen Fall ist die Klasse PersistencyManager vorgesehen. Aus pragmatischen Überlegungen wird im vorliegenden Modell nur ein Persistenzmodul unterstützt (vgl. „uses"-Assoziation). Die Klasse bietet jedoch einen idealen Erweiterungspunkt, um eine Abbildung zwischen *persistencyID* und konkretem *IPersistencyModule*-Subtyp zu realisieren. Da *PersistencyManager* ebenfalls *IPersistencyModule* implementiert, würde es die unterschiedlichen Persistenzmechanismen elegant verbergen (vgl. Abschnitt 6.2.1, S. 155ff.).

6.3.8. Kernmodul: core.product

Überblick

core.product ist das Kernmodul für die Modellierung eines (unversionierten) Produktmodells (vgl. Abschnitt 4.2.1, S. 76ff.). Es bietet Schnittstellen, um Typen von Produktmodell-Elementen und ihre Beziehungen untereinander zu modellieren, um diese anschließend – mit Hilfe eines MOD2-SKM Repositoriums – zu versionieren. In MOD2-SKM sollte jedes Produktmodell auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.32 zeigt die Abhängigkeiten von *core.product*. Es besteht eine Abhängigkeit zum Kernmodul für Kennungen (*core.id*), da jedem Produktmodell-Element eine Objektkennung zugewiesen werden soll (vgl. Abschnitt 4.2.2, S. 76ff.). Diese Abhängigkeit ist bidirektional, da die Elemente an das Modul zur Kennungsgenerierung übergeben werden (vgl. Abschnitt 6.3.5, S. 182ff.). Zusätzlich besteht eine Abhängigkeit zum Logbuch-Mechanismus (*util.logging*), um Änderungen am Produktmodell protokollieren zu können. Es bestehen keine Abhängigkeiten zu Kernmodulen bzgl. Historie und Repositorium, so dass die konkreten Produktmodelle keinerlei Versionsinformationen enthalten. Diese werden erst über das Arbeitsbereich-Kernmodul (*core.workspace*) hinzugefügt (vgl. Abschnitt 6.3.14, S. 206ff.).

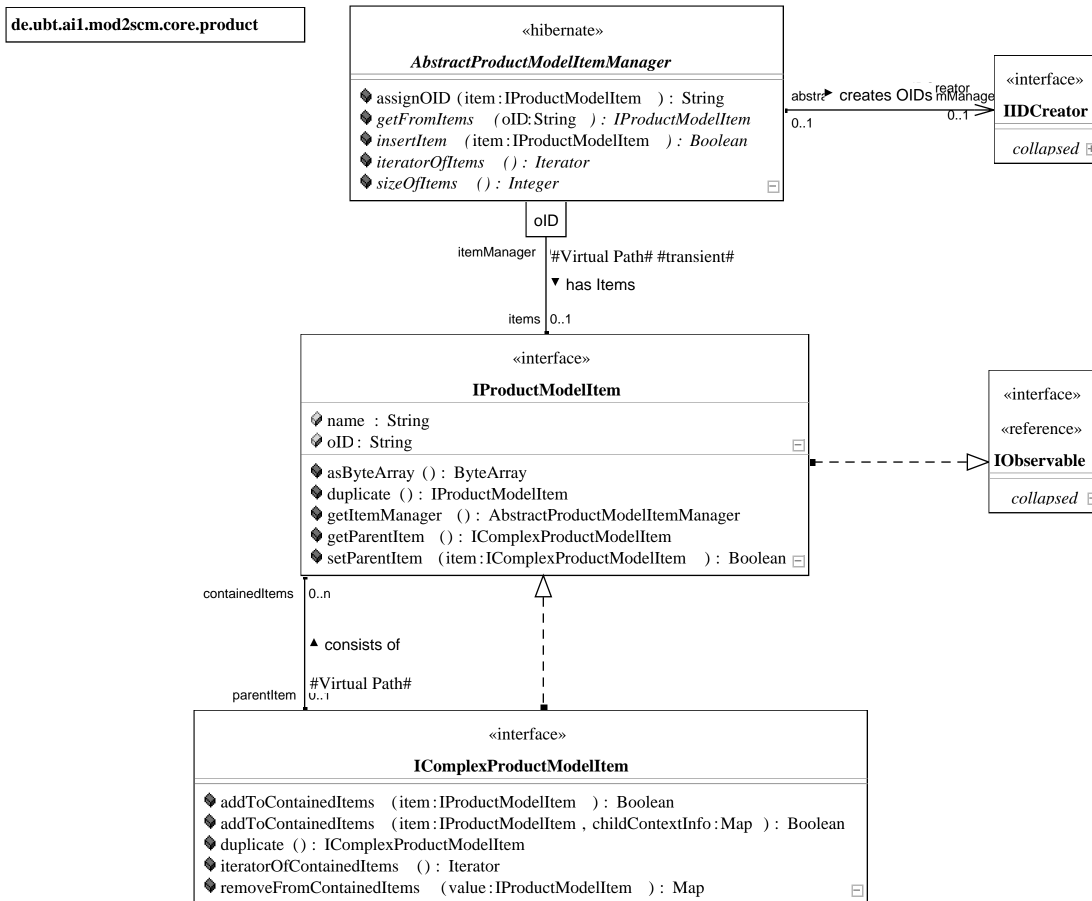


Abbildung 6.33.: Klassendiagramm: core.product

Detailbeschreibung

Das Kernmodul *core.product* definiert die Schnittstelle, die Produktmodell-Elemente und -Beziehungen implementieren müssen, um in MOD2-SKM verwendet zu werden. Die korrekte Implementierung der Schnittstellen stellt sicher, dass sich ein Produktmodell in einem Arbeitsbereich verwalten und mit einem Repositorium versionieren lässt. Alle Kernmodule, die Produktmodell-Elemente verarbeiten, basieren auf diesen Schnittstellen. So sind sie unabhängig vom konkreten Produktmodell, was die Modellierung eines SKMS für ein neues Produktmodell erheblich erleichtert: Bei monolithischen SKMS ist es nötig, die 14 Kernmodule zu implementieren. In MOD2-SKM genügt ein einziges Modul, das Produktmodell!

Abbildung 6.33 zeigt das Klassendiagramm von *core.product*. Jedes Produktmodell besitzt einen sog. Produktmodell-Manager (*AbstractProductModelItemManager*), der eine Referenz auf einen Kennungsgenerator besitzt („creates OIDs„-Assoziation auf *IIDCreator*). Die Assoziation besteht zur Schnittstelle des Kernmoduls *core.id* (vgl. Abschnitt 6.3.5, S. 182ff.), so dass sich verschiedene Kennungsgeneratoren nutzen lassen (vgl. Ab-

schnitt 6.4.4, S. 222ff.). Mit Hilfe dieses Generators kann nun neuen Produktmodell-Elementen ihre Objektkennung (vgl. Abschnitt 4.2.2, S. 76ff.) zugewiesen werden (*assignOID()*).

Weiterhin bietet der Produktmodell-Manager Zugriff auf jedes Produktmodell-Element anhand seiner Objektkennung (qualifizierte Assoziation „has Items“). Diese Beziehung ist als virtueller Pfad modelliert (vgl. Abschnitt 6.1, S. 144ff.), d.h. die interne Struktur des Produktmodells wird zwar modelliert, aber hinter dieser Assoziation versteckt (vgl. Abschnitt 4.2.1, S. 76ff.). So lässt sich jedes Produktmodell als Menge seiner Elemente betrachten, die über ihre Objektkennung effizient referenziert werden können.

Jedes versionierbare Produktmodell-Element muss die Schnittstelle *IProductModelItem* implementieren. Diese verlangt, dass das Element einen Namen (*name*) und eine Objektkennung (*oID*) besitzt. Sobald ein Produktmodell-Element in ein Repositorium übertragen wird, muss sichergestellt sein, dass es sich um eine Kopie (und nicht um eine Referenz) handelt, d.h. es muss möglich sein, ein Element zu kopieren (*duplicate()*). Im Hinblick auf eine Übertragung zwischen zwei Computersystemen ist auch eine Konvertierung in eine Bytefolge vorgesehen (*asByteArray()*). Weiterhin ist es notwendig, dass sich jedes Produktmodell-Element überwachen lässt, um Änderungen zu registrieren (vgl. Abschnitt 6.3.14, S. 206ff.). Dies wird mit Hilfe eines Überwachungs-Entwurfsmusters [FF04] modelliert (*IObservable*).

Produktmodell-Elemente, die nur auf der Schnittstelle *IProductModelItem* basieren, sind atomare Elemente. Sie lassen sich daher nicht in weitere Produktmodell-Elemente zerlegen und können daher nur Ziel einer Kompositionsbeziehung (vgl. Abschnitt 4.2.3, S. 76ff.) sein (aber nicht Quelle). Um die Kompositionsbeziehung in komplexeren Produktmodellen auszunutzen, müssen zusammengesetzte Elementtypen die Schnittstelle *IComplexProductModelItem* implementieren. Dadurch können sie (über die Assoziation „consists of“) mit ihren Kindelementen in Beziehung gesetzt werden. Auch diese Assoziation ist ein virtueller Pfad, so dass die konkrete Struktur des Produktmodells hinter einer Baumstruktur versteckt wird.

Diese Baumstruktur muss sich jedoch manipulieren lassen, um z.B. Elemente oder Teilbäume einfügen, entfernen oder sogar austauschen zu können. Dazu muss der Entwickler das Verhalten der Methoden zum Hinzufügen (*addToContainedItems()*) bzw. Löschen (*removeFromContainedItems()*) von Kind-Elementen modellieren. Dabei muss für beide Methoden gelten, dass sie Seiteneffektfrei genutzt werden können, was sich wie folgt ausdrücken lässt: Sei $IComplexProductModelItem\ add(x : IComplexProductModelItem, y : IProductModelItem)$ die Operation, die ein Produktmodell-Item y einem komplexen Produktmodell-Item x hinzufügt. Analog sei $IComplexProductModelItem\ del(x : IComplexProductModelItem, y : IProductModelItem)$ die Operation, die ein Produktmodell-Item y aus einem komplexen Produktmodell-Item entfernt. Dann muss für jedes komplexe Produktmodell-Objekt o und Produktmodell-Item p gelten:

$$o = del(add(o, p), p)$$

Weiterhin verlangt der Produktmodell-Manager (*AbstractProductModelItemManager*) die Modellierung einer Methode zum Einfügen/Austauschen neuer Elemente (*insertItem()*).

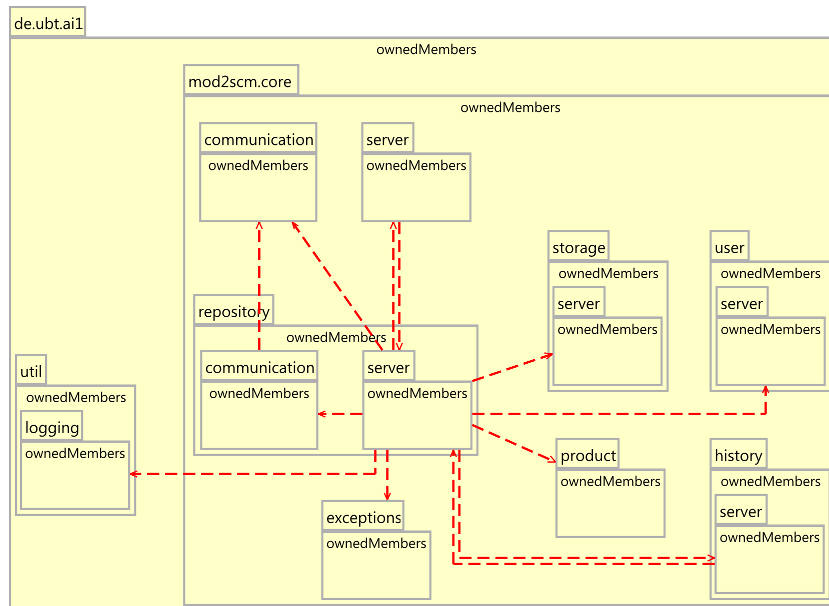


Abbildung 6.34.: Paketdiagramm: core.repository

Das abstrakte Produktmodell aus *core.product* stellt somit sicher, dass sich jedes Produktmodell als Menge von Elementen mit einer Objektkennung präsentiert. Zwischen diesen Elementen existiert evtl. noch eine Kompositionsbeziehung, die einen „Enthaltenseins-Baum" unter den Elementen aufspannt. Dieser Baum lässt sich mit Hilfe der Schnittstellen manipulieren. Beide Beziehungen sind dabei als virtueller Pfad modelliert, so dass sie mittels der Beziehungen des konkreten Produktmodells modelliert werden können. So bleiben die unterschiedlichen Produktmodell-Strukturen vor den restlichen Modulen verborgen (vgl. Abschnitt 6.2.1, S. 155ff.).

6.3.9. Kernmodul: core.repository

Überblick

core.repository ist das Kernmodul für Repositorien (vgl. Abschnitt 4.4.1, S. 91ff.). Es bietet Schnittstellen, um Produktmodell-Elemente einzuspielen (engl. commit) und abzufragen (engl. checkout). Aufgabe eines Repositorien-Moduls ist die Steuerung dieser Zugriffe, insbesondere beim parallelen Zugriff mehrerer Entwickler. Zur Speicherung der Produktmodell-Elemente dient jeweils ein Historien- und ein Speichermodul (vgl. Abschnitt 6.4.1, S. 213ff. und vgl. Abschnitt 6.4.21, S. 279ff.). In MOD2-SKM sollte jede Art von Repositorium auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.34 zeigt die Abhängigkeiten von *core.repository*. Das Kernmodul steuert sowohl Komponenten für das Server-Teilsystem als auch die Kommunikation bei, so dass

Abhängigkeiten zu den Kernmodulen beider Teilsysteme bestehen (*core.server* und *core.communication*). Während *core.repository.communication* keine weiteren Abhängigkeiten besitzt, veranschaulichen die Abhängigkeiten von *core.repository.server* die zentrale Rolle des Repositoriums: Jedes Repositorium verwendet eine Historie (*core.history*) und einen Speichermechanismus (*core.storage*), um Produktmodell-Elemente (*core.product*) zu versionieren. Diese Vorgänge lassen sich in ein Logbuch protokollieren (*util.logging*), wobei auch immer der zugreifende Benutzer (*core.user*) erfasst wird. Evtl. auftretende Probleme während eines Repositoriums-Zugriffs werden über Ausnahmen (*core.exceptions*) sowohl an das Logbuch als auch den Kommunikationspartner weitergeleitet. Alle Abhängigkeiten existieren nur zu Kernmodulen, so dass ein Repositorium mit unterschiedlichen konkreten Historien, Speicherverfahren, Benutzerverwaltungen und Produktmodellen verwendbar ist.

Detailbeschreibung

Das Kernmodul *core.repository* ist in zwei Unterpakete aufgeteilt, eines für das Server- (*core.repository.server*) und eines für das Kommunikations-Teilsystem (*core.repository.-communication*). Beide basieren auf den entsprechenden Schnittstellen aus den Kernpaketen *core.server* bzw. *core.communication*.

Server-Teilsystem: *core.repository.server*

Abbildung 6.35 (s. S. 193) zeigt die Schnittstellen für die beiden Komponenten des Server-Teilsystems. Zum einen handelt es sich um das Repositoriums-Modul (*AbstractRepositoryModule*), das die eingehenden Kommandos aus dem Kommunikations-Teilsystem verarbeitet (*processCommand()* bzw. *execute()*). Zum anderen um die Schnittstelle für einzelne Repositorien (*AbstractRepository*). Beide Schnittstellen bieten die Verwendung eines Logbuchs an („writes to"-Assoziation zu *ILogger*), um ihr Verhalten zu protokollieren.

Von Repositorien wird verlangt, dass sie einen eindeutigen Namen besitzen (Attribut *name* und qualifizierte „hasRepositories"-Assoziation). Weiterhin verwenden sie eine Menge von Historien („has Items"), die Produktmodell-Elemente mit gleicher Objektkennung gruppieren, und sie so als unterschiedliche Versionen des gleichen Elements auffassen (vgl. Abschnitt 6.3.4, S. 178ff.). Die Produktmodell-Elemente selbst sind dabei in einem, auf *core.storage* (vgl. Abschnitt 6.3.11, S. 200ff.) basierenden, Speicher abgelegt („stores in"). Alle Elemente liegen dabei in einem einzigen Speicher, so dass ein Speichermodul auch die Beziehungen zwischen den Produktmodell-Elementen (vgl. Abschnitt 4.2.3, S. 76ff.) ausnutzen kann (s. Modul *storage.complex*, vgl. Abschnitt 6.4.22, S. 280ff.). Beide Beziehungen bestehen zu den Schnittstellen der jeweiligen Kernmodule, so dass unterschiedliche Historien- und Speichermodule verwendet werden können (vgl. Abschnitt 6.4.1, S. 213ff. und vgl. Abschnitt 6.4.21, S. 279ff.).

core.repository betrachtet ein Repositorium lediglich als ungeordnete Menge von Elementen, die sich eindeutig über eine Tupel aus Objekt- und Versionskennung identifizieren lassen. So kann z.B. eine Liste von Elementen durch Angabe ihrer Tupel (*re-*

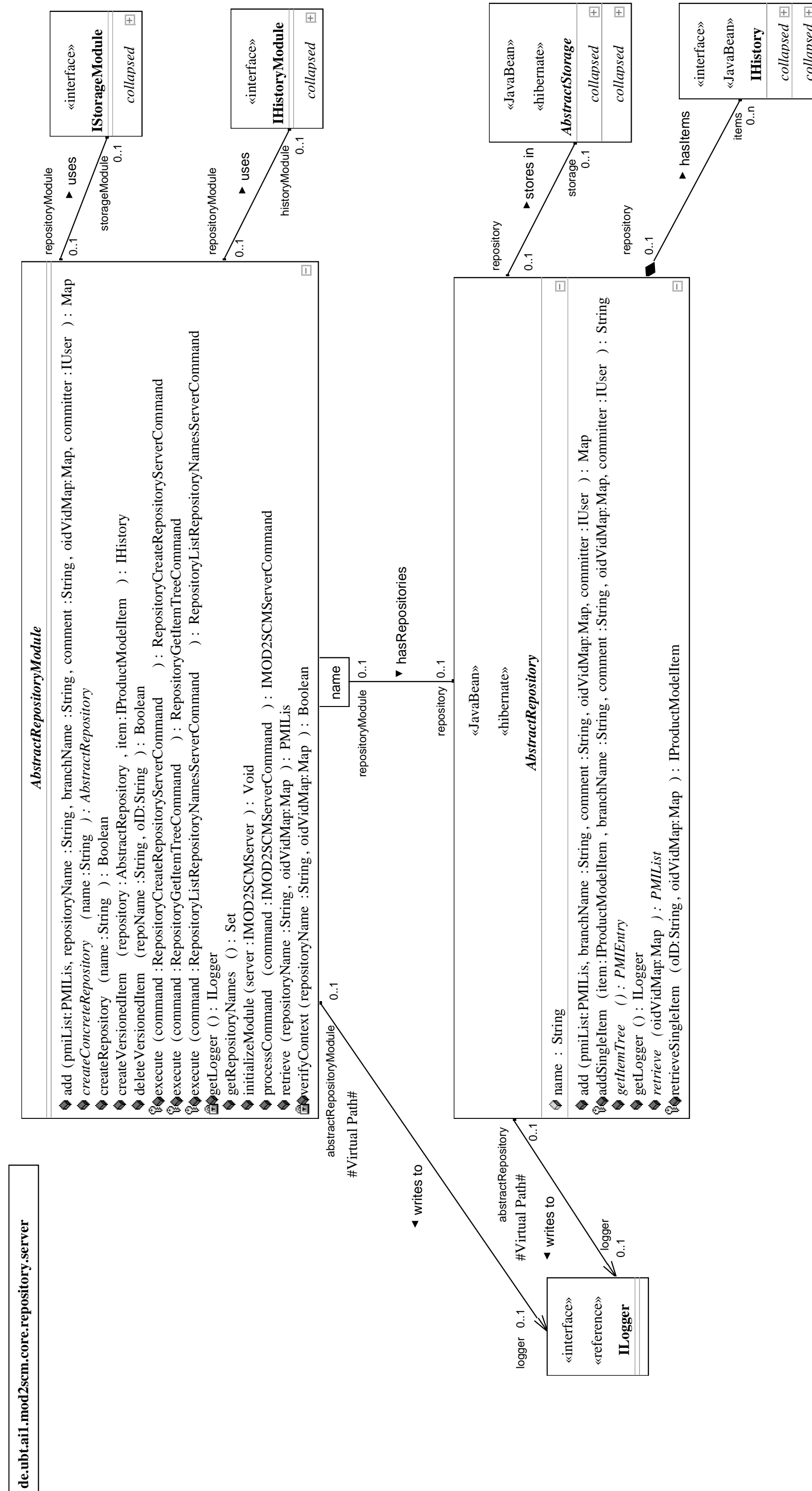


Abbildung 6.35.: Klassendiagramm: core.repository.server

trieve(oidVidMap : Map)) aus dem Repositorium ausgelesen werden. Ebenso lassen sich Elemente in das Repositorium einspielen (*add()*). Dies geschieht unter Angabe der Elemente und ihren Kennungen (Parameter *pmiList* und *oidVidMap*), sowie eines Kommentars (Parameter *comment*) und des verantwortlichen Benutzers (Parameter *committer*). Zwei weitere Methoden müssen in jedem Repositoriums-Modul noch modelliert werden: Der Zugriff auf die Baumrepräsentation seines Inhalts (*getItemTree()*) und der Zugriff auf eine Menge von Produktmodell-Elementen (*retrieve()*). Diese Methode wird vom Repositoriums-Browser des Eclipse-Client benötigt (vgl. Abschnitt 7, S. 315ff.), der in der aktuellen Implementierung nur die Repositoriumswurzel mit versionszentriert ausgewählten Kindern erwartet. In zukünftigen Arbeiten kann diese Schnittstelle erweitert werden, um (1) frühere Versionen der Repositoriumswurzel und (2) Kindelemente verwoben bzw. nach Auswahlkriterien bestimmt anzeigen zu können.

Die Schnittstelle des Repositoriums-Moduls bietet grundlegende Funktionen zur Verwaltung von Repositorien. Es lassen sich z.B. neue Repositorien anlegen (*createRepository()*), solange kein Repositorium unter diesem Namen existiert. Dabei wird die Beziehung zum Speichermodul („uses"-Assoziation zu *IStorageModule*) genutzt, um einen neuen Speicher anzulegen. Das Repositoriums-Modul löst auch den Repositoriumsnamen beim Einspielen (*add()*) bzw. Auslesen (*retrieve()*) von versionierten Produktmodell-Elementen auf. Im Vergleich zu den korrespondierenden Methoden im Repositorium ist die Signatur noch um einen Repositoriumsnamen erweitert (Parameter *repositoryName*). So leitet das Modul den Aufruf an die zugehörige Methode des genannten Repositoriums weiter (oder löst eine Ausnahmebehandlung aus, wenn das Repositorium nicht existiert vgl. Abschnitt 6.3.3, S. 177ff.).

Wird während eines *add*-Aufrufs festgestellt, dass für ein Produktmodell-Element noch keine zugehörige Historie existiert (da es neu unter Versionskontrolle gestellt wurde), ist das Repositoriums-Modul für das initialisieren der Historie zuständig. Dazu wird ein Historien-Modul verwendet („uses"-Assoziation zu *IHistoryModule*). Analog zum Speichermodul wird auch hier nur die Schnittstelle aus *core.history* verwendet, so dass verschiedene Historien-Module (vgl. Abschnitt 6.4.1, S. 213ff.) einsetzbar sind. Wie viele andere Kernmodule, verlangt auch das Repositoriums-Modul – gemäß dem Fabrik-Entwurfsmuster [FF04] – dass eine Methode zum Erzeugen des konkreten Repositoriums existiert (*createConcreteRepository()*).

Zusätzlich bietet das Repositoriums-Modul auch eine Schnittstelle, um Informationen über Repositorien zu erhalten. Zum einen liefert sie eine Namensliste der existierenden Repositorien (*getRepositoryNames()*), zum anderen eine Baumdarstellung des Inhalts eines Repositoriums (*getItemTree()*). Mit Hilfe dieser Methoden ist es möglich einen einfachen Repositorien-Browser zu modellieren, damit die Entwickler existierende Repositorien zum initialen Checkout ausfindig machen können (vgl. Abschnitt 7, S. 315ff.).

core.repository betrachtet – analog zum Modul *core.history* – die versionierte Objektbasis (vgl. Abschnitt 4.3.4, S. 81ff.) als ungeordnete Menge von versionierten Produktmodell-Elementen ohne Beziehungen. Erst die konkreten Repositorien-Module erwarten Beziehungen zwischen den Elementen, und modellieren so die drei unterschiedlichen Integrationsverfahren: produktzentriert, versionszentriert oder verschränkt (vgl. Abschnitt 6.4.16, S. 251ff.) [CW98]. Kernelement ist hier der Parameter *oidVidMap*, der eine Ob-

jektkennung mit einer Versionskennung in Verbindung bringt. Die Schnittstellen von Historien- und Speichermodul erwarten, dass alle betroffenen Elemente **explizit** in der Liste enthalten sind. Die Kennungsliste, die an das Repositorium übergeben wird, enthält diese Information jedoch nur **implizit**, da sie i.d.R. auf einer Benutzerauswahl basiert. Abhängig vom Integrationsverfahren wird nun diese Liste erweitert, bis sie alle betroffenen Elemente explizit aufzählt. So wird z.B. die „version proliferation“ [CW98] im versionszentrierten Ansatz realisiert (siehe Modul *repository.complexitem*, vgl. Abschnitt 6.4.16, S. 251ff.): Ein Benutzer spielt eine neue Version eines Elements ein, die Kennungsliste wird um die Elternelemente erweitert und anschließend an Historie und Speicher übermittelt.

Kommunikations-Teilsystem: *core.repository.communication*

Abbildung 6.36 (s. S. 196) zeigt die drei Kommandos der Kernkomponente. Sie basieren auf der Schnittstelle des Kommunikations-Kernmoduls (*core.communication*) und übertragen alle Daten, die zu ihrer Ausführung notwendig sind (vgl. Abschnitt 6.3.1, S. 173ff.). Der Kommando-Typ legt fest, welche *execute()*-Methode des Repositorien-Moduls (*AbstractRepositoryModule*) ausgeführt wird (siehe Abbildung 6.35, S. 193): Ein Kommando legt ein neues Repositorium mit einem eindeutigen Namen an (*RepositoryCreateRepositoryServerCommand*). Eines listet alle Repositorien auf, die das Modul verwaltet (*RepositoryListRepositoryNamesServerCommand*). Und eines liefert den Inhalt eines Repositoriums in einer Baumstruktur zurück (*RepositoryGetItemTreeCommand*). Diese Baumstruktur besteht aus Knoten des Typs *PMIEntry*, die lediglich die Objektkennung (*oid*) der versionierten Elemente im Repositorium enthalten. Die beiden letzten Kommandos sind notwendig, um Informationen über einen MOD2-SKM-Server anzeigen zu können, wenn z.B. ein Repositorium initial in den Arbeitsbereich ausgecheckt werden soll (vgl. Abschnitt 7, S. 315ff.).

6.3.10. Kernmodul: core.server

Überblick

core.server ist das Kernmodul für das Server-Teilsystem (vgl. Abschnitt 6.2.3, S. 161ff.). Es bietet Schnittstellen für die Modellierung eines Repositoriums-Servers, der Daten über das Kommunikations-Teilsystem *core.communication* empfangen und versenden kann (vgl. Abschnitt 6.3.1, S. 173ff.). Aufgabe von *core.server* ist es, die Daten an die unterschiedlichen Module, aus denen ein Server besteht, weiterzuleiten (z.B. an das Historien-Modul, das Speichermodul, usw.). Dazu werden nur Schnittstellen der Kernmodule benutzt, so dass sich die konkreten Module aus der Modulbibliothek kombinieren lassen. In MOD2-SKM sollte jeder Server – und jedes Modul, das auf einem Server zum Einsatz kommen soll – auf diesem Kernmodul basieren.

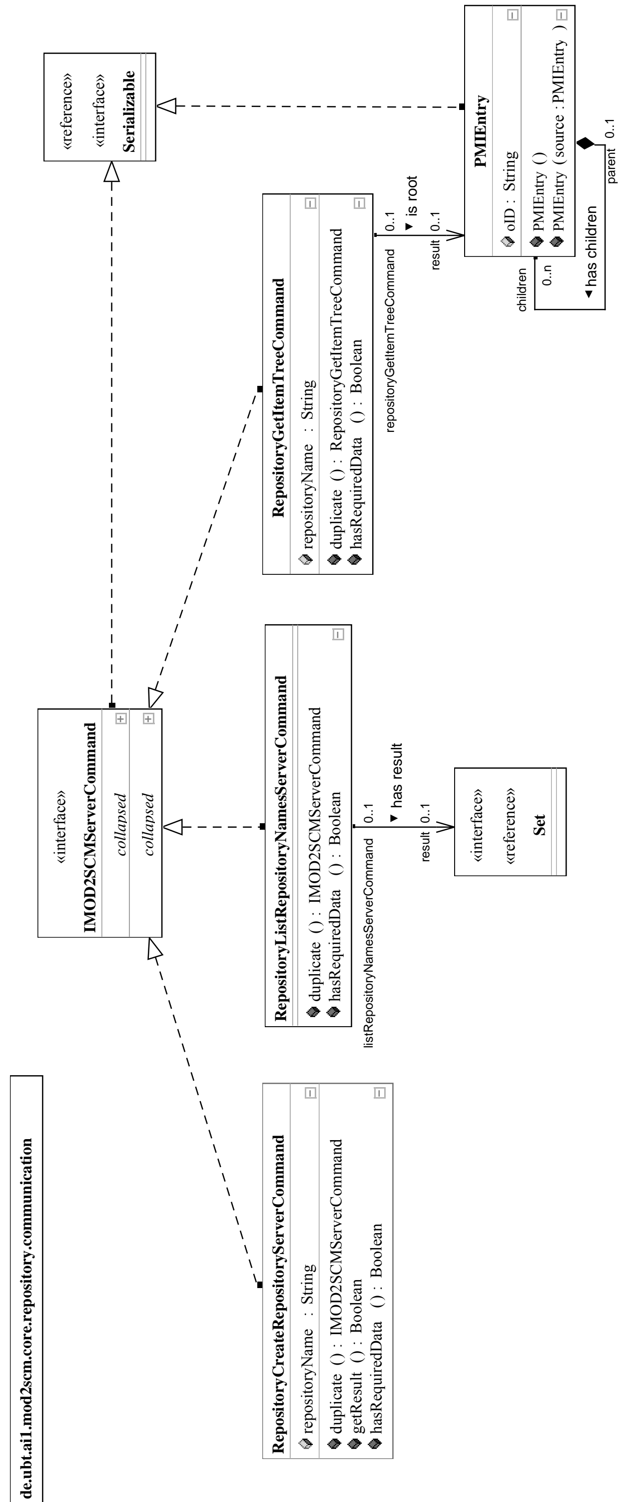


Abbildung 6.36.: Klassendiagramm: core.repository.communication

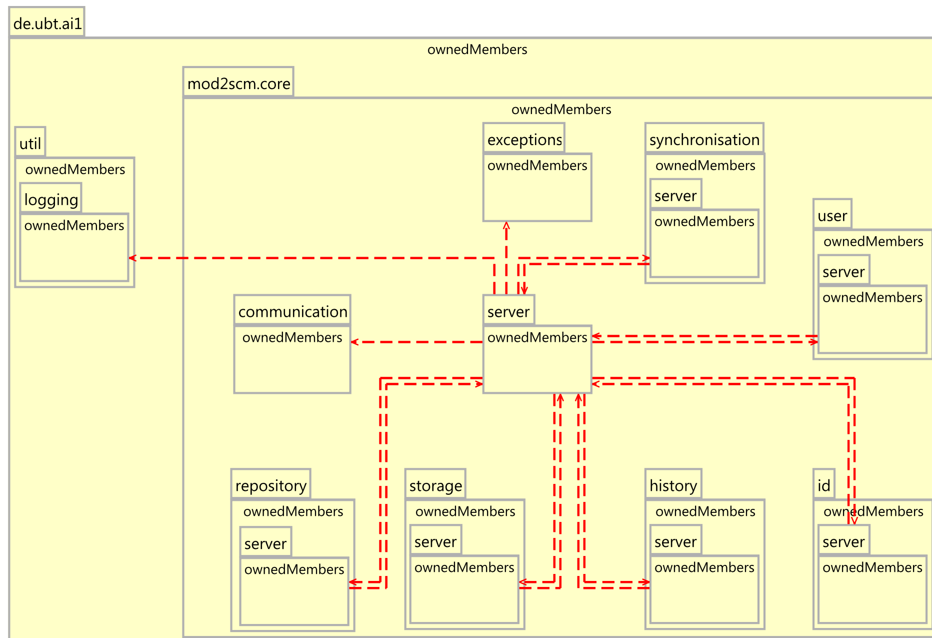


Abbildung 6.37.: Paketdiagramm: core.server

Abhängigkeiten

Abbildung 6.37 (s. S. 197) zeigt die Abhängigkeiten von *core.server*. Ein Server benötigt mehrere Komponenten, um überhaupt als Repositoriums-Server lauffähig zu sein: (1) Ein Repositorium, in das die versionierten Produktmodell-Elemente eingespielt werden (*core.repository*). Jedes Repositorium besteht (2) aus einer Historie (*core.history*) und (3) einem Speichermechanismus (*core.storage*). Zusätzlich lassen sich noch (4) Module zur Kennungsübersetzung benutzen (*core.id*), um z.B. Elemente mit Hilfe von Markierungen oder Auswahlregeln zu referenzieren. Für den Zugriff auf das Repositorium wird außerdem (5) eine Benutzerverwaltung (*core.user*) und (6) ein Synchronisationsverfahren für den Mehrbenutzerbetrieb benötigt (*core.synchronisation*). Die Abhängigkeit zu all diesen Modulen ist bidirektional, damit ein Modul – über den Server – mit anderen Modulen Daten austauschen kann. Alle Abhängigkeiten bestehen auch nur zu Kernmodulen. Ein konfigurierter MOD2-SKM-Server ist daher eine Kombination aus jeweils sechs entsprechenden Modulen der Modulbibliothek (s.S. 211ff.).

Weiterhin kann ein Server über das Kommunikations-Teilsystem (*core.communication*) angesprochen werden. Außerdem verwaltet jeder Server ein Logbuch, das von allem Modulen zum Protokollieren ihrer Operationen verwendet werden kann. Dies schließt auch die evtl. auftretenden Ausnahmen (*core.exceptions*) ein.

Detailbeschreibung

Das Kernmodul *core.server* definiert die Schnittstellen des Server-Teilsystems. Das Klassendiagramm in Abbildung 6.38 zeigt zum einen die Schnittstelle des Servers selbst

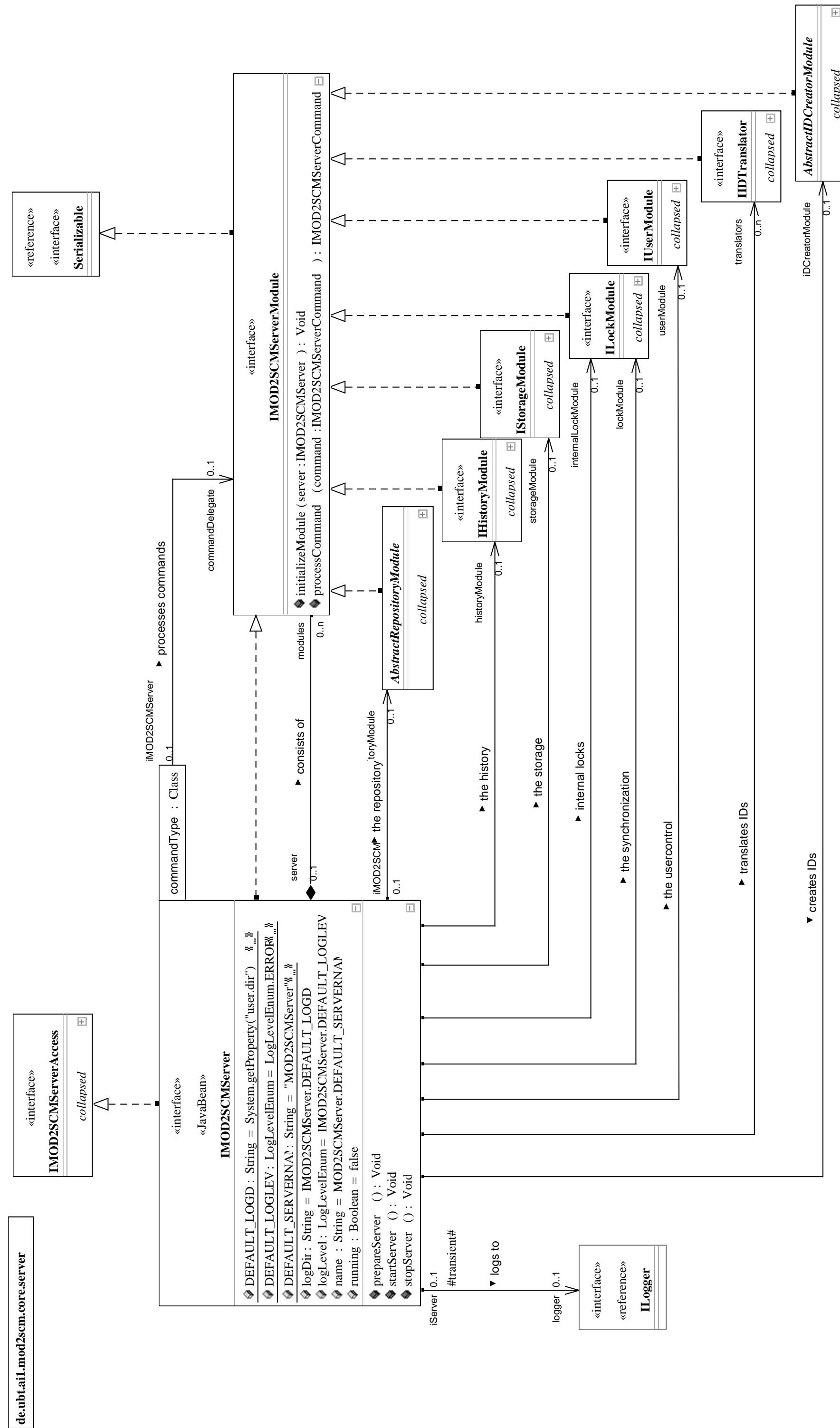


Abbildung 6.38.: Klassendiagramm: core.server

(*IMOD2SCMServer*) und zum anderen die Schnittstelle der Module (*IMOD2SCMServer-Module*), die auf einem MOD2-SKM-Server verwendet werden sollen.

Die Server-Schnittstelle (*IMOD2SCMServer*) bindet den Server an das Kommunikations-Teilsystem an (vgl. Abschnitt 6.2.3, S. 161ff.), indem sie die entsprechende Schnittstelle (*IMOD2SCMServerAccess*) aus dem Kernmodul *core.communication* implementiert. Hauptaufgabe ist die Verwaltung der Servermodule (*IMOD2SCMServerModule*) und ihre Anbindung an die Kommunikationsschicht. Zentral ist hier insbesondere die Zuordnung der eingehenden Kommandos an das zugehörige Modul.

Ein Kommando basiert auf der Schnittstelle *IMOD2SCMServerCommand* aus dem Kernmodul *core.communication* (vgl. Abschnitt 6.3.1, S. 173ff.). Ein konkretes Kommando ist jedoch nur für ein einziges Servermodul bestimmt, z.B. die Anfrage nach der Historie eines Elements an das Historien-Modul, oder das Anlegen einer Markierung (engl. tag) an das Tag-Kennungsmodul. Die Schnittstelle für Module (*IMOD2SCMServerModule*) verlangt, dass ein Modul beim initialisieren (*initializeModule()*) alle seine Kommandos registriert. Dies erfolgt über eine Verknüpfung mit Hilfe der qualifizierten Assoziation „processes commands“. Als Schlüssel dient der Typ eines Kommandos, so dass ein Modul für jedes seiner Kommandos eine Beziehung anlegt. So lässt sich ein eingehendes Kommando zur Ausführung an das entsprechenden Modul delegieren (*processCommand()*).

Die „processes commands“-Assoziation abstrahiert vollständig von der Funktion der einzelnen Module. Da ein MOD2-SKM-Server jedoch (1) Module für bestimmte Aufgabenbereiche benötigt und (2) nur ein Modul pro Aufgabenbereich einsetzen kann, bestehen noch weitere Beziehungen: Jeder Server benötigt ein Repositorium („the repository“-Assoziation zu *AbstractRepositoryModule* aus dem Modul *core.repository*, vgl. Abschnitt 6.3.9, S. 191ff.) mit Historie („the history“-Assoziation zu *IHistoryModule* aus *core.history*, vgl. Abschnitt 6.3.4, S. 178ff.) und Speicher („the storage“-Assoziation zu *IStorageModule* aus *core.storage*, vgl. Abschnitt 6.3.11, S. 200ff.), ein Modul zum Sperren der Elemente während einer laufenden Operation („internal locks“-Assoziation zu *ILockModule* aus *core.synchronisation*, vgl. Abschnitt 6.3.12, S. 202ff.), eine Benutzerverwaltung („the usercontrol“-Assoziation zu *IUserModule* aus *core.user*, vgl. Abschnitt 6.3.13, S. 203ff.) und ein Modul zum Erzeugen der Versionskennung („creates IDs“-Assoziation zu *AbstractIDCreatorModule* aus *core.id*, vgl. Abschnitt 6.3.5, S. 182ff.). Optional können auch noch Module zum Sperren von Elementen („the synchronization“-Assoziation zu *ILockModule* aus *core.synchronisation*, vgl. Abschnitt 6.3.12, S. 202ff.) und für erweiterte Kennungsverfahren eingesetzt werden („translates IDs“-Assoziation zu *IIDTranslator* aus *core.id.server*, vgl. Abschnitt 6.3.5, S. 182ff.), z.B. für Markierungen oder regelbasierte Auswahl. Alle Assoziationen bestehen zu Kernmodulen, und abstrahieren somit von den konkreten Implementierungen.

Ansonsten verlangt die Server-Schnittstelle *IMOD2SCMServer* noch, dass ein Server kontrolliert gestartet (*startServer()*) und beendet werden kann (*stopServer()*). Sie dient auch dazu, das Server-Logbuch zu verwalten („logs to“-Assoziation zu *ILogger* aus *util.logger*) und z.B. das Logverzeichnis (*logDir*) oder die Granularität der Logbucheinträge (*logLevel*)zu steuern.

Die Server-Schnittstellen stellen somit sicher, dass sich die Server-Module einheitlich über das Kommunikations-Teilsystem ansprechen lassen. Zusätzlich verlangen sie, dass

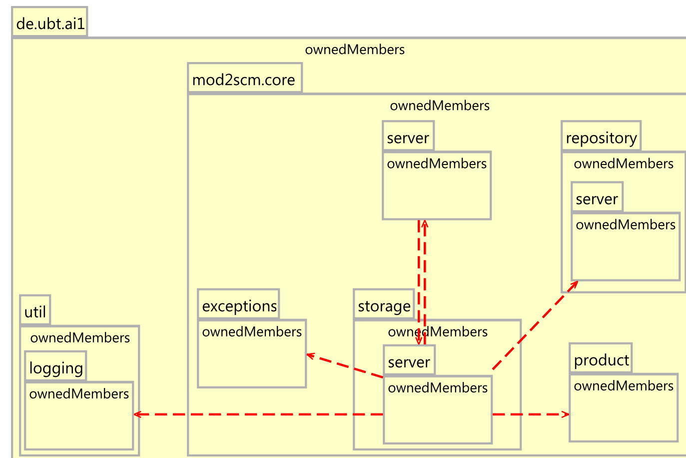


Abbildung 6.39.: Paketdiagramm: core.storage

für jeden Aufgabenbereich ein entsprechendes Modul vorhanden ist. Diese Abhängigkeiten bestehen jedoch auf Ebene der Kernmodule, so dass sich die jeweiligen konkreten Module der Modul-Bibliothek austauschen lassen.

6.3.11. Kernmodul: core.storage

Überblick

core.storage ist das Kernmodul für Speichermechanismen, um versionierte Produktmodell-Elemente effizient abzulegen (vgl. Abschnitt 4.3.6, S. 88ff.). Es bietet eine Schnittstelle, um Produktmodell-Elemente zu speichern und wieder zu laden. In MOD2-SKM sollte jeder Speichermechanismus auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.39 zeigt die Abhängigkeiten von *core.storage*. Ein Speichermechanismus wird im Server-Teilsystem (*core.server*) von Repositorien (*core.repository*) eingesetzt, um dort versionierte Produktmodell-Elemente und ihre Beziehungen (*core.product*) zu speichern. Das Speichern und Wiederherstellen lässt sich protokollieren (*core.util*), und über evtl. auftretende Probleme informieren Ausnahmen (*core.exceptions*). Alle Abhängigkeiten bestehen zu Kernmodulen, so dass ein Speicherverfahren unabhängig vom konkreten Produktmodell oder Repositorium modelliert werden kann.

Detailbeschreibung

Das Kernmodul *core.storage* definiert eine Schnittstelle zum Speichern und Wiederherstellen von Produktmodell-Elementen, die in Abbildung 6.39 gezeigt ist. Jedes Repositorium verwendet einen Speicher (vgl. Abschnitt 6.3.9, S. 191ff.), der über die Schnittstelle

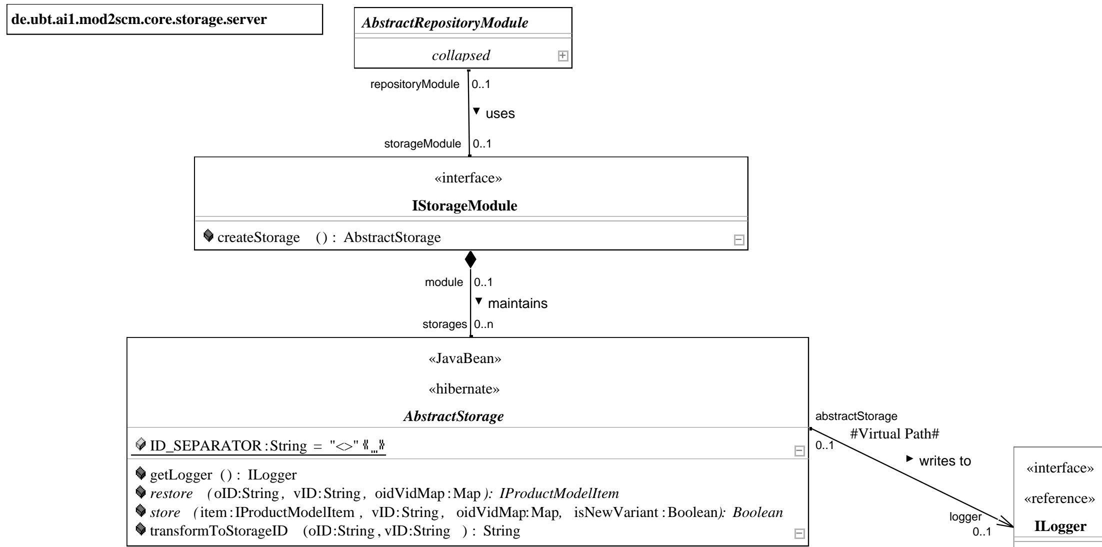


Abbildung 6.40.: Klassendiagramm: core.storage

des konkreten Speichermoduls (*IStorageModule*) erzeugt wird. Das Fabrik-Entwurfsmuster ermöglicht es, einen Speicher unabhängig vom konkreten Typ zu erzeugen [FF04].

Die Speicherschnittstelle (*AbstractStorage*) verlangt, dass eine Methode zum Speichern (*store()*) und Wiederherstellen (*restore()*) existiert. Dabei muss folgender Zusammenhang zwischen den beiden Methoden gewährleistet sein. Ein gespeichertes Element muss durch Angabe seiner Kennungen vollständig wiederherstellbar sein. D.h. beim Speichern eines Produktmodell-Elements o – mit Objektkennung oid und Versionskennung vid – muss gelten:

$$store(oid, vid, o) \rightarrow o = restore(oid, vid)$$

Intern verwendet der Speicher eine Speicherkennung (engl. storage id), die sich aus dem Tupel (oid, vid) zusammensetzt. Mit Hilfe der Methode *transformToStorageID()* lässt sich ein Tupel entsprechend umwandeln: Da es sich bei beiden Kennungen um eine Zeichenkette handelt, werden die beiden – getrennt durch die Konstante *ID_SEPARATOR* – konkateniert. Wie bereits in *core.history* angemerkt, ließe sich auch hier ein abstraktes Kennungselement verwenden (vgl. Abschnitt 6.3.4, S. 178ff.).

Von den gespeicherten Produktmodell-Elementen wird lediglich verlangt, dass sie die Schnittstellen aus *core.product* implementieren (Parameter *item* aus *store()*). Somit ist die Speicherschnittstelle unabhängig vom konkreten Produktmodell. Die Kennungen (Parameter *oid* und *vid* von *store()* bzw. *restore()*) sind Zeichenketten und ebenfalls unabhängig vom konkreten Produktmodell bzw. von der konkreten Historie. Lediglich die Information, ob ein Produktmodell-Element eine neue Variante ist (*isNewVariant*) (vgl. Abschnitt 4.3.3, S. 80ff.), wird nicht in jedem Fall benötigt. Dieser Parameter verhält sich wie *branchName* von *addVersion()* der Schnittstelle *IHistory*, d.h. die im Kernmodul *core.history* geschilderten Lösungsansätze ließen sich hier ebenfalls anwenden (vgl. Abschnitt 6.3.4, S. 178ff.).

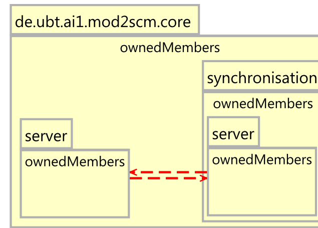


Abbildung 6.41.: Paketdiagramm: core.synchronisation

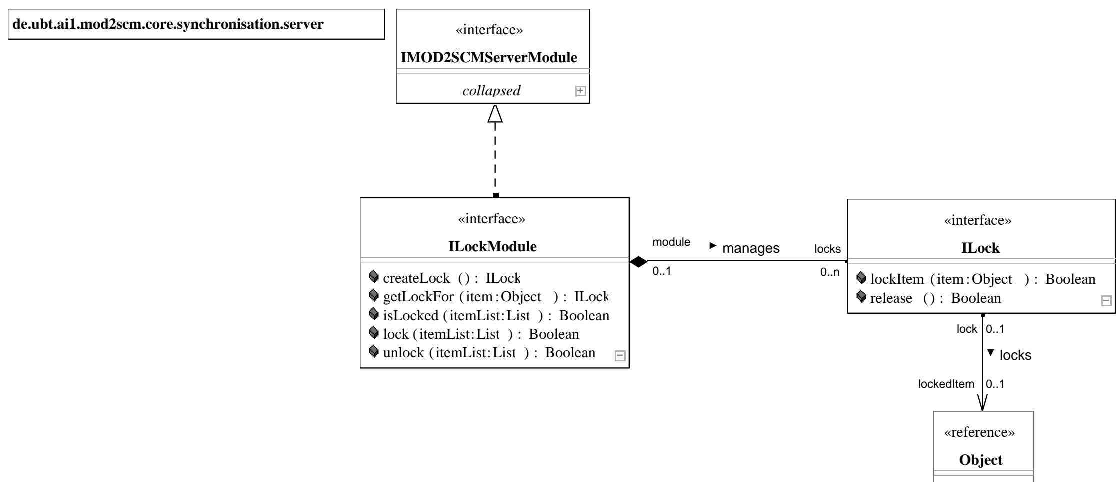


Abbildung 6.42.: Klassendiagramm: core.synchronisation

6.3.12. Kernmodul: core.synchronisation

Überblick

core.synchronisation ist das Kernmodul für Sperrverfahren zur Koordination konkurrierender Zugriffe. Es ermöglicht das Anlegen und Aufheben von Sperren (engl. locks) – sowohl für Nebenläufigkeiten paralleler Kommandos als auch für pessimistische Sperren durch Benutzer. In MOD2-SKM sollte jedes Modul für ein Synchronisationsverfahren auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.41 zeigt die Abhängigkeiten von *core.synchronisation*. Als Modul des Server-Teilsystems ist es vom entsprechenden Kernmodul (*core.server*) abhängig. Ansonsten bestehen keine weiteren Abhängigkeiten.

Detailbeschreibung

Das Kernmodul *core.synchronisation* definiert eine Schnittstelle, um konkurrierende Zugriffe auf ein Laufzeit-Objekt zu ordnen. Das Klassendiagramm in Abbildung 6.42 zeigt

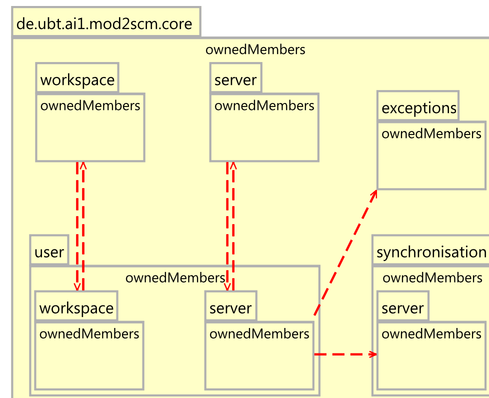


Abbildung 6.43.: Paketdiagramm: core.user

die Schnittstelle für das Servermodul (*ILockModule*) sowie die von ihm verwalteten Sperren (engl. lock) (*ILock*).

Einer Sperre kann (über die Methode *lockItem()*) ein beliebiges Objekt zugewiesen werden („locks"-Assoziation). Das bedeutet, dass dieses Objekt keiner weiteren Sperre zugeordnet werden darf, bis die Zuweisung wieder aufgehoben wird (*release()*). Die Zuweisung ist unidirektional, so dass die Objekte nicht auf ihre Sperren zugreifen können. Die Sperren werden vom zugehörigen Servermodul (*ILockModule*) verwaltet („manages"-Assoziation), über das Objekte gesperrt (*lock()*) und entsperrt (*unlock()*) werden können, sowie ihr Status abgefragt (*isLocked()*). Eine konkrete Sperre wird gemäß Fabrik-Entwurfsmuster [FF04] erzeugt (*createLock()*).

Die Schnittstellen von *core.synchronisation* abstrahiert vollständig von den zu sperrenden Objekten („locks"-Assoziation zu *java.lang.Object*) oder die tatsächliche Realisierung der Sperren. Insbesondere die Atomarität der Methoden *lock()* und *unlock()* muss von den konkreten Modulen modelliert werden (vgl. Abschnitt 6.4.24, S. 287ff.). Die Sperren lassen sich für pessimistische Synchronisation als auch für Behandlung paralleler Benutzeraufrufe verwenden. I.d.R. wird für letzteres jedoch die Transaktionsunterstützung des Persistenzmechanismus empfohlen (vgl. Abschnitt 6.3.7, S. 186ff.).

6.3.13. Kernmodul: core.user

Überblick

core.user ist das Kernmodul zur Modellierung bzw. Integration einer Benutzerverwaltung. Es bietet Schnittstellen zum Verwalten von Benutzern, sowie zum Sperren und Entsperren von Produktmodell-Elementen zur Realisierung pessimistischer Synchronisation (vgl. Abschnitt 4.4.1, S. 92ff.). In MOD2-SKM sollte jedes Benutzerverwaltungs-Modul auf diesem Kernmodul basieren.

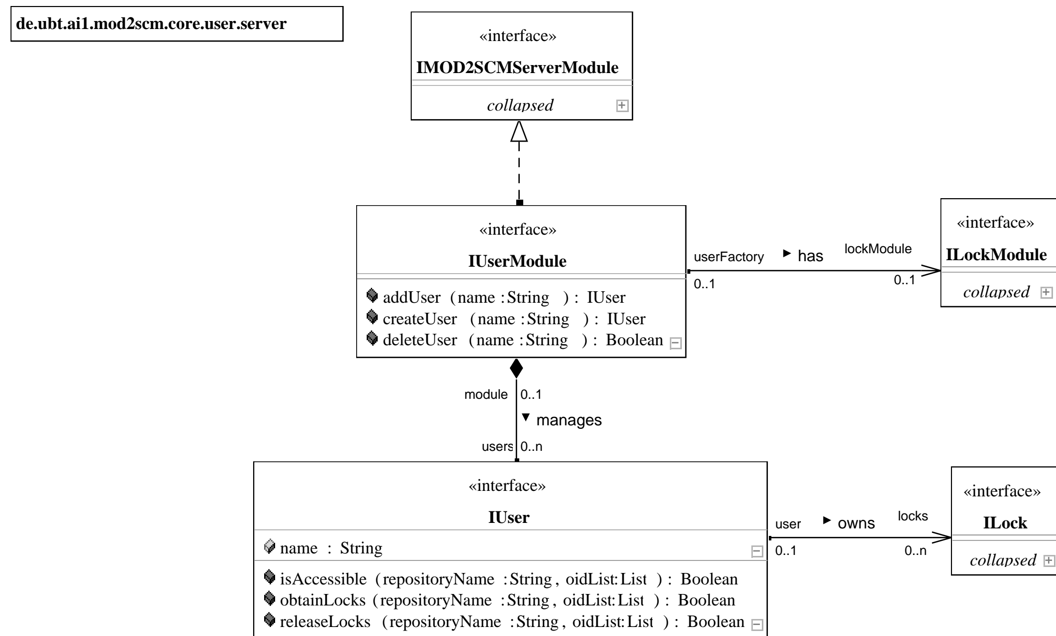


Abbildung 6.44.: Klassendiagramm: core.user.server

Abhängigkeiten

Abbildung 6.43 zeigt die Abhängigkeiten von *core.user*. Die Schnittstellen für das Server- (*core.user.server*) bzw. Arbeitsbereichs-Teilsystem (*core.user.workspace*) basieren auf ihrem jeweiligen Kernmodul (*core.server* bzw. *core.workspace*). Die Server-Schnittstellen bieten ebenfalls die Möglichkeit, Produktmodell-Elemente im Namen des Benutzers zu sperren, wofür ein Synchronisations-Modul genutzt wird (*core.synchronisation*). Dabei können, insbesondere beim Anfordern von Sperren, Ausnahmen (*core.exceptions*) auftreten.

Detailbeschreibung

Das Kernmodul *core.user* ist in zwei Unterpakete aufgeteilt, eines für das Server- (*core.user.server*) und eines für das Arbeitsbereich-Teilsystem (*core.user.workspace*). Beide basieren auf den entsprechenden Schnittstellen aus den Kernpaketen *core.server* bzw. *core.workspace*.

Server-Teilsystem: *core.user.server*

Abbildung 6.44 zeigt die Schnittstellen für die Benutzerverwaltung im Server-Teilsystem. Das Benutzerverwaltungs-Modul (*IUserModul*) verlangt Methoden zum Anlegen (*addUser()*) und Entfernen (*deleteUser()*) eines Benutzers. Dieser wird anhand eines Benutzernamens identifiziert (Parameter *name*). Das Modul verwaltet die einzelnen Benutzer-Objekte („manages“-Assoziation zu *IUser*), die sich – gemäß des Fabrik-Entwurfsmusters

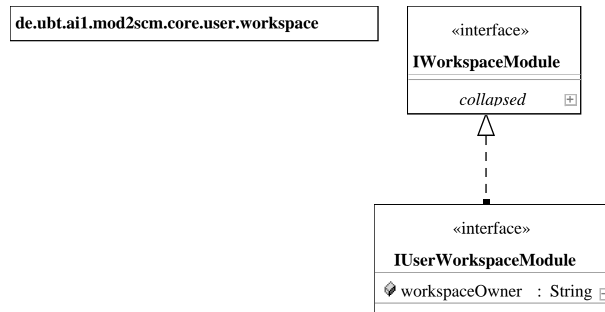


Abbildung 6.45.: Klassendiagramm: core.user.workspace

[FF04] – unabhängig vom konkreten Typ erzeugen lassen (*createUser()*).

Die Schnittstelle für einzelne Benutzer (*IUser*) verlangt, dass sie o.g. eindeutigen Namen besitzen (Attribut *name*). Außerdem lassen sich im Namen des Benutzers Sperren anlegen (*obtainLocks()*) und wieder aufheben (*releaseLocks()*), und es kann geprüft werden, ob der Benutzer auf eine Menge von Produktmodell-Elementen überhaupt zugreifen darf (*isAccessible()*). Dazu wird ein Synchronisations-Mechanismus benötigt. Da jedoch nur Abhängigkeiten zu den Schnittstellen aus dem Kernmodul *core.synchronisation* bestehen (unidirektional Assoziationen auf *ILockModule* bzw. *ILock*, vgl. Abschnitt 6.3.12, S. 202ff.), sind die konkreten Module austauschbar (vgl. Abschnitt 6.4.24, S. 287ff.).

Das Kernmodul abstrahiert von konkreten Benutzerverwaltungen, indem sie den Benutzer auf ein Objekt mit einem Namen reduziert, dessen Schnittstelle eine Methode besitzt (*isAccessible()*), die angibt, ob er auf ein oder mehrere Elemente zugreifen darf (Parameter *oidList*). So lässt sich entweder eine einfache Benutzerverwaltung selbst realisieren (vgl. Abschnitt 6.4.26, S. 290ff.) oder auch eine bereits bestehende Benutzerverwaltung integrieren.

Arbeitsbereich-Teilsystem: *core.user.workspace*

Abbildung 6.45 zeigt die Schnittstelle der Benutzerverwaltung im Arbeitsbereich (*IUserWorkspaceModule*). Sie verlangt lediglich, dass ein Benutzername als Besitzer des Arbeitsbereichs existiert (Attribut *workspaceOwner*). Über diesen lässt sich ein Arbeitsbereich seinem Benutzer-Objekt auf dem Server zuordnen (Parameter *name* von *IUser*, s. Abbildung 6.44).

6.3.14. Kernmodul: core.workspace

Überblick

core.workspace ist das Kernmodul für das Arbeitsbereichs-Teilsystem (vgl. Abschnitt 6.2.3, S. 161ff.). Es bietet Schnittstellen zur Modellierung eines Arbeitsbereichs, der Daten über das Kommunikations-Teilsystem (*core.communication*, vgl. Abschnitt 6.3.1, S. 173ff.) mit einem Repositoriums-Server (*core.server*, vgl. Abschnitt 6.3.10, S. 195ff.) austauschen kann. Aufgabe des Arbeitsbereichs ist es, eine Produktmodell-Instanz mit

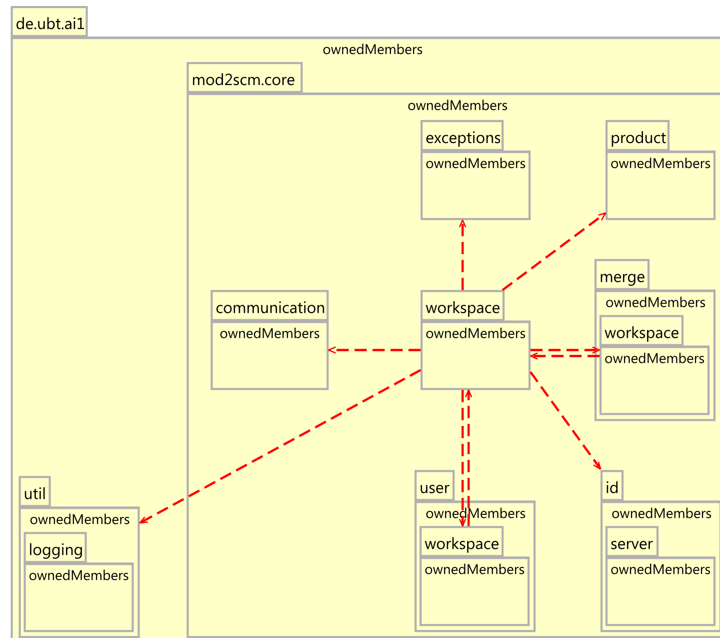


Abbildung 6.46.: Paketdiagramm: core.workspace

einem Repositorium zu synchronisieren, und die Auflösung von Konflikten bei konkurrierenden Änderungen zu unterstützen. Ein Arbeitsbereich stellt auch eine Anbindung für Software ohne SKM-Konzepte dar (vgl. Abschnitt 4.4.2, S. 93ff.), so dass die Produktmodell-Instanz auch auf externe Änderungen überwacht wird. In MOD2-SKM sollte jedes Modell eines Arbeitsbereichs auf diesem Kernmodul basieren.

Abhängigkeiten

Abbildung 6.46 zeigt die Abhängigkeiten von *core.workspace*. Als Schnittstelle des Arbeitsbereichs-Teilsystems müssen Daten über das Kommunikations-Teilsystem (*core.-communication*) gesendet und empfangen werden. Jeder Arbeitsbereich ist dabei einem Entwickler zugeordnet (*core.user*). Im Arbeitsbereich wird eine Produktmodell-Instanz (*core.product*) verwaltet, deren Elementen eine eindeutige Objektkennung (*core.id*) zugeordnet ist. Bei konkurrierenden Änderungen besteht u.U. der Bedarf, unterschiedliche Versionen eines Produktmodell-Elements zu verschmelzen (*core.merge*). Zur Laufzeit können auch Ausnahmen (*core.exceptions*) auftreten, die sich – wie auch der Standardablauf – in einem Logbuch (*util.logging*) protokollieren lassen. Alle Abhängigkeiten bestehen zu Kernmodulen, so dass ein Arbeitsbereich weder von einem konkreten Produktmodell noch einem konkreten Server abhängig ist.

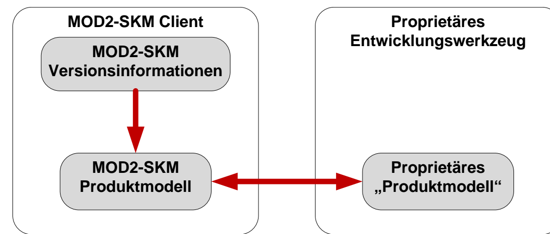


Abbildung 6.47.: Architektur des MOD2-SKM-Arbeitsbereich

Detailbeschreibung

Das Kernmodul *core.workspace* definiert die Schnittstellen des Arbeitsbereichs-Teilsystems. Dessen zentrale Komponente ist ein Arbeitsbereichs-Client, der die Versionsinformationen der Produktmodell-Instanz verwaltet, die im Arbeitsbereich vorliegt. Hierzu gehört insbesondere das Registrieren von Änderungen, da ein Entwickler die Produktmodell-Elemente mit Software-Werkzeugen verändert, die i.d.R. die Produktmodell-Instanz nur unversioniert bearbeiten (vgl. Abschnitt 4.4.2, S. 93ff.) [Dar90, CW98].

Abbildung 6.47 illustriert den Zusammenhang zwischen dem proprietären Datenmodell der Entwicklungswerkzeuge und dem MOD2-SKM Produktmodell: Die Entwickler besitzen Entwicklungswerkzeuge (rechter Kasten), die Daten in proprietären Datenstrukturen speichern. I.d.R. sind diese Daten unversioniert und sollen nun mit Hilfe eines MOD2-SKMS verwaltet werden. Damit bilden diese Daten ein unversioniertes, proprietäres „Produktmodell“. Aufgabe des MOD2-SKM Produktmodells ist es, von den proprietären Datenstrukturen zu abstrahieren und eine einheitliche Schnittstelle (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.) zu implementieren. Über diese Schnittstelle lässt sich das MOD2-SKM Produktmodell nun (1) vom Arbeitsbereich-Client (linker Kasten) verwalten und (2) mit Versionsinformationen anreichern. Diese werden getrennt vom Produktmodell verwaltet.

Grund für die Trennung ist Aufgabe des MOD2-SKM Produktmodells, das proprietäre Modell exakt abzubilden. Zur Laufzeit müssen beide Instanzen vollständig synchron gehalten werden. Änderungen durch Edieroperationen im Entwicklungswerkzeug müssen zum MOD2-SKM Client übertragen werden. Im Gegenzug müssen dann die Änderungen durch Synchronisation mit einem Repositorium an das Entwicklerwerkzeug übermittelt werden. Die Synchronisation ist Aufgabe des konkreten MOD2-SKM Produktmodells (z.B. beim Dateisystem-Produktmodell vgl. Abschnitt 6.4.14, S. 240ff.).

Beschreibung der Schnittstellen

Das Klassendiagramm in Abbildung 6.48 zeigt die Schnittstelle des Arbeitsbereichs-Clients (*AbstractWorkspaceManager*) und der einzelnen Arbeitsbereichs-Module (*IWorkspaceModule*), die in einem MOD2-SKM-Arbeitsbereich verwendet werden.

Der Arbeitsbereichs-Client (*AbstractWorkspaceManager*) bindet den Arbeitsbereich an das Kommunikations-Teilsystem an (*IMOD2SCMServerAccess* aus *core.communication*). Über diese Schnittstelle können die einzelnen Arbeitsbereichs-Module (*IWorkspace-*

Abbildung 6.48.: Klassendiagramm: core.workspace

Module) Daten und Anfragen an den Server senden. Jeder Arbeitsbereich ist dabei an ein Repositorium gebunden (Attribut *repositoryName*). Die Kommunikation erfolgt über ein Kommando, das über den Client versendet wird (*executeOnServer()*) (vgl. Abschnitt 6.3.1, S. 173ff.). Hier ist insbesondere die Methode *IMOD2SCMServerCommand.hasRequiredData()* von Bedeutung. Mit ihrer Hilfe kann der Client prüfen, ob ein Kommando unvollständig ist und so überflüssige Datenübertragungen über das Kommunikations-Teilsystem verhindern.

Jeder Arbeitsbereich-Client verwaltet eine Produktmodell-Instanz (*AbstractProductModelItemManager*) über die Schnittstellen des Produktmodell-Kernmoduls (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.), so dass der Client unabhängig vom konkreten Produktmodell verwendbar ist. Die Objektkennungen (vgl. Abschnitt 4.2.2, S. 76ff.) der Produktmodell-Elemente werden über ein Kennungsmodul erstellt. Auch hier wird durch Schnittstellen-Abstraktion (*IDCreator* aus *core.id*, vgl. Abschnitt 6.3.5, S. 182ff.) die Austauschbarkeit der konkreten Kennungsmodule erreicht.

Die Versionsinformationen über die einzelnen Produktmodell-Elemente sind in speziellen Datenobjekten (*WSInfo*) abgelegt. Jedes *WSInfo*-Objekt referenziert dabei sein zugehöriges Produktmodell-Element („observes"-Assoziation zu *IProductModelItem*) und speichert dessen Versionskennung (Attribut *currentVID*) und ob das Element durch den Entwickler verändert wurde (Attribut *changed*). Die Änderung wird dabei durch Überwachung des Produktmodell-Elements erfasst. Die Schnittstelle *IProductModelItem* ist Teil eines Überwachungs-Entwurfsmusters [FF04], und *WSInfo* implementiert die geeignete Schnittstelle (*IObserver*), um das Element zu überwachen. Im Falle einer Änderung, wird über die *update()*-Methode des Entwurfsmusters das *changed*-Attribut gesetzt. Im Falle von zusammengesetzten Produktmodell-Elementen (vgl. Abschnitt 6.3.8, S. 188ff.) ist es möglich, die Änderung an die *WSInfo*-Objekte der Elternelemente zu propagieren (*rippleChangeToRoot()*).

Nur Produktmodell-Elemente unter Versionskontrolle besitzen ein *WSInfo*-Objekt. Neu erstellte Element lassen sich über den Arbeitsbereichs-Client (*AbstractWorkspaceManager*) unter Versionskontrolle stellen. Dazu existieren drei Methoden, um einzelne Elemente (*addToVersionControlSingle()*), mehrere Elemente (*addToVersionControlMultiple()*) oder ganze Teilbäume zusammengesetzter Elemente (*addToVersionControlTree()*) hinzuzufügen. Analog existieren drei Methoden, um Elemente wieder aus der Versionskontrolle herauszunehmen (*removeFromVersionControl...*). Ob ein Element unter Versionskontrolle steht, lässt sich über (*isUnderVersionControl()*) ausgeben. Und es lässt sich eine Liste aller versionierten (*listVersionedItems()*) oder auch nur der aller geänderten Elemente (*listChangedItems()*) abfragen.

Produktmodell-Elemente lassen sich ebenfalls aus dem Arbeitsbereich ausschließen, d.h. sie werden von allen Operationen ignoriert. Der Client verwaltet dazu eine Liste der ignorierten Elemente („ignores"-Assoziation auf *IProductModelItem*), zu der sich Elemente hinzufügen (*ignore()*) oder entfernen lassen (*unignore()*). Wie bei Elementen unter Versionskontrolle, lässt sich auch hier der Status abfragen (*isIgnoring()*) bzw. eine Liste der ignorierten Elemente ausgeben (*listIgnoredItems()*).

In allen Fällen wird die Objektkennung (*oid*) bzw. eine Liste von Objektkennungen (*oidList*) zur Identifikation der betroffenen Produktmodell-Elemente verwendet. In

MOD2-SKM müssen Kennungen explizit aufgelistet werden (s. auch *core.repository*, vgl. Abschnitt 6.3.9, S. 191ff.). Um die Verwaltung zusammengesetzter Produktmodell-Elemente zu erleichtern, existieren daher noch drei Hilfsmethoden, um implizit angegebene Kennungen in eine explizite Liste umzuwandeln:

1. *expandOidList* erweitert eine Liste von Elementen um den transitiven Abschluss über alle Kindelemente. Dies wird u.a. benötigt, um Teilbäume auszuwählen.
2. *expandToRootOidList* erweitert eine Liste von Elementen um alle Elternelemente, die im Arbeitsbereich liegen. Dies wird u.a. benötigt, wenn ein Änderung an einem Element an alle Elternelemente propagiert wird.
3. *expandToVersionedParentOidList* erweitert eine Liste von Elementen um alle unversionierten Elternelemente. Dies wird u.a. benötigt, wenn ein Element unter Versionskontrolle gestellt wird. Dann werden auch seine Elternelemente unter Versionskontrolle gestellt.

Die Daten des Arbeitsbereichs müssen – wie der Server – persistent gespeichert werden. Dazu besitzt jeder Client ein Datenverzeichnis, in dem diese abgelegt werden. Ebenso besitzt er ein Verzeichnis, in dem sich weitere Informationen über den Arbeitsbereich ablegen lassen, wie z.B. die Protokoll-Datei des Logbuch-Mechanismus. Die Persistenzierung der Arbeitsbereichs-Informationen ist zur Zeit im Client integriert. Es ist jedoch wahrscheinlich, dass sich die auf *core.persistency* basierenden Persistenzmechanismen auch für diesen Zweck eignen. Dies bleibt jedoch zukünftigen Arbeiten überlassen.

Alle weiteren Operationen im Arbeitsbereich, wie z.B. das Einspielen und Auslesen von Elementen aus dem Repositorium, werden durch Arbeitsbereichs-Module (*IWorkspaceModule*) implementiert. So kann ein konkretes Modul zusätzliche Operationen und Datenstrukturen zum Arbeitsbereich hinzufügen. Die Schnittstelle für Arbeitsbereichs-Module stellt sicher, dass ein Modul über seine Initialisierung (*initializeModule()*) und eine Aktualisierung der Daten im Arbeitsbereich benachrichtigt wird (*refresh()*). Zwei spezielle Module, die ein Arbeitsbereich immer benötigt, sind ein Modul zur Benutzerverwaltung (*IUserWorkspaceModule* aus *core.user*, vgl. Abschnitt 6.3.13, S. 203ff.) und ein Modul zum Verschmelzen (*IMergeModule* aus *core.merge*, vgl. Abschnitt 6.3.6, S. 184ff.). In beiden Fällen sind die konkreten Module durch Schnittstellen aus den entsprechenden Kernmodulen abstrahiert und somit austauschbar.

Paarung von Server- und Arbeitsbereichs-Modulen

Bietet ein Servermodul spezielle Operationen im Arbeitsbereich an, so wird auch ein entsprechendes Arbeitsbereichs-Modul benötigt. Wie unter *core.communication* und *core.server* beschrieben, wird ein Kommando, das über das Kommunikations-Teilsystem eintrifft, anhand seines Typs dem jeweiligen Server-Modul zugeordnet (vgl. Abschnitt 6.3.10, S. 195ff.). Um Daten aus dem Arbeitsbereich an das Server-Modul zu senden, muss also im Arbeitsbereich ein Kommando des entsprechenden Typs erzeugt werden. Dies ist Aufgabe des zugehörigen Arbeitsbereichs-Moduls. Dies führt zu einer Paarbildung in vielen konkreten Modulen der Modulbibliothek (vgl. Abschnitt 6.4, S. 211ff.).

Um die Abhängigkeiten zwischen konkreten Modulen zu minimieren, wurden die zugehörigen Paare nur paarweise betrachtet und modelliert. Es bleibt daher offen, ob sich Teile der Arbeitsbereichs-Module oder Kommandos über Module hinweg wiederverwenden lassen. Dies ließe sich z.B. durch die Einführung abstrakter Module realisieren, die selbst nicht instanziert, aber von mehreren konkreten Modulen verwendet werden.

Modulare Benutzerschnittstelle

Zur Verwaltung des Arbeitsbereichs benötigt der Entwickler eine Benutzerschnittstelle, z.B. in Form einer Kommandozeile oder einer grafischen Benutzeroberfläche (vgl. Abschnitt 5.3.2, S. 111ff.). Die Entwicklung einer modularen Benutzeroberfläche, die mit Hilfe des MODPL-Werkzeugkastens konfiguriert werden kann, ist nicht mehr Teil dieser Arbeit. Daher wurde lediglich ein vorkonfigurierter grafischer Client in die Entwicklungsumgebung „Eclipse" integriert. Dieser ist in Abschnitt 7 (s. S. 315) detailliert beschrieben.

6.4. Die MOD2-SKM Modulbibliothek

Die Modulbibliothek modelliert die variablen Komponenten der MOD2-SKM Produktlinie. Wie bei den Kernmodulen (vgl. Abschnitt 6.3, S. 171ff.), beginnen alle Paketnamen mit dem Präfix *de.ubt.ai1.mod2scm*, das aus Gründen der Übersichtlichkeit weggelassen wird. Jedes Modul implementiert gezielt die Schnittstelle eines Kernmoduls. Diese Abhängigkeit ist über die Paketnamen erkennbar. Der Paketname eines Bibliotheksmoduls beginnt – nach dem Präfix – mit dem letzten Teil des Paketnamens seines Kernmoduls, z.B. beginnt der Paketname aller Module, die vom Kernmodul *core.history* abhängen, mit *history*: *history.base*, *history.flat* und *history.tree*.

Ein Modul steuert i.d.R. Komponenten zu ein oder mehreren Teilsystemen der Client-Server-Architektur (vgl. Abschnitt 6.2.3, S. 161ff.) bei. Diese Zuordnung ist – analog zu den Kernmodulen – an den Unterpaketen der einzelnen Module erkennbar. Komponenten für das Kommunikations-Teilsystem sind im Unterpaket *communication* zu finden, und Komponenten des Client-Teilsystems im Unterpaket *client*. Komponenten des Server-Teilsystems sind direkt im Modulpaket enthalten und nicht – wie in der Modulbibliothek – in einem *server*-Unterpaket. Das Verschieben der Elemente in ein entsprechendes Unterpaket ist erwünscht, würde aber die grafische Darstellung der Diagramme zerstören. Es erfordert ca. 30-40 Personenstunden, um alle Paketdiagramme manuell neu zu erstellen. Diese hinderliche Einschränkung der grafischen Komponente des Paketdiagrammeditors wird in Abschnitt 8.4.4 (s. S. 342) genauer beschrieben.

Abbildung 6.49 zeigt die Abhängigkeiten zwischen den Modulen der Modulbibliothek anhand ihrer Paketimporte (vgl. Abschnitt 6.2, S. 155ff.). Diese werden in den nun folgenden Abschnitten für jedes Bibliotheksmodul detailliert beschrieben. Das Paketdiagramm zeigt alle ausgehenden Import-Beziehungen zu anderen Modulen aus der Bibliothek, d.h. es existieren keine weiteren Import-Beziehungen zu anderen Paketen der Modulbibliothek.

Die Abbildung zeigt auch die Merkmalsmarkierungen auf Paketebene, d.h. ein Paket ist nur dann Teil eines konfigurierten Systems, wenn die entsprechende Merkmalsmarkie-

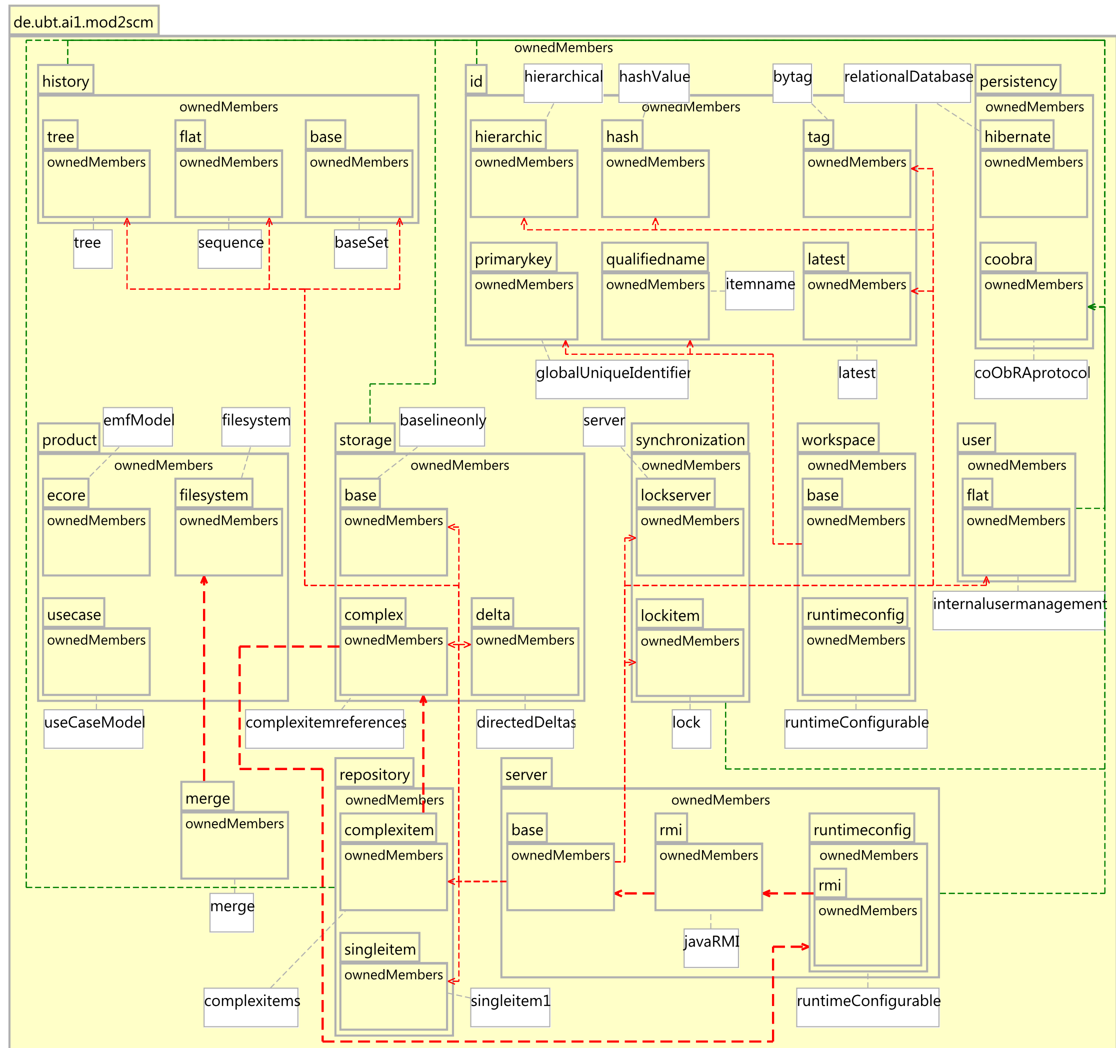


Abbildung 6.49.: Paketdiagramm der Modulbibliothek

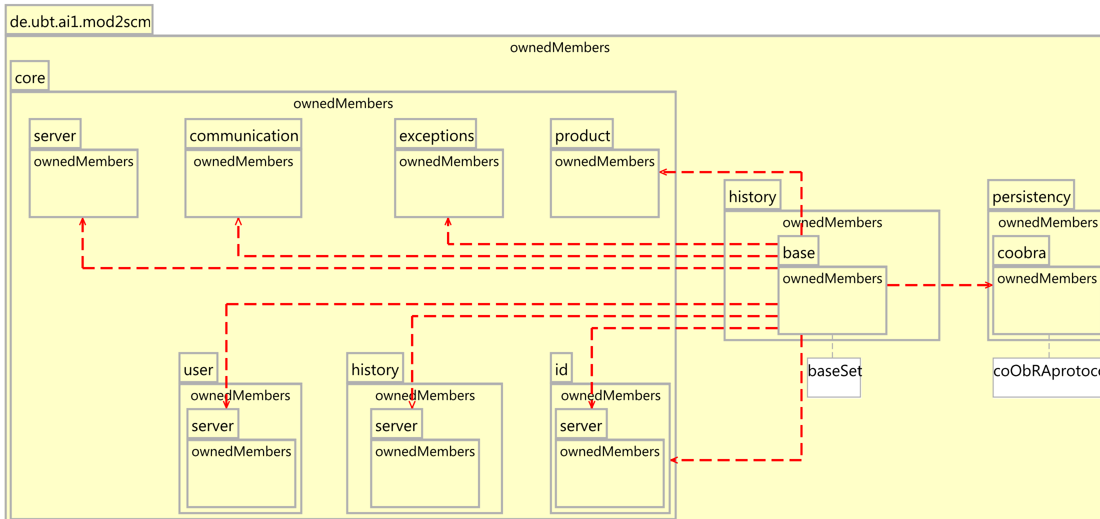


Abbildung 6.50.: Paketdiagramm: history.base

rung in der Konfiguration enthalten ist. Eine Importbeziehung zwischen zwei markierten Modulen deutet auch auf eine Abhängigkeit zwischen den verwendeten Merkmalen hin. In Abschnitt 6.5.3 (s. S. 300) werden die bibliotheksinternen Abhängigkeiten gezielt untersucht.

6.4.1. Modul: history.base

Überblick

Das Modul *history.base* ist das Modell einer unstrukturierten Historie, d.h. es existieren keine expliziten Beziehungen im Versionsraum. Sie können jedoch implizit über die Versionskennung realisiert werden. Dieses Modul ist damit die primitivste Realisierung einer Historie. Es sind keine Module für das Arbeitsbereichs- oder Kommunikations-Teilsystem implementiert. Ein vollständiges Modul enthielte – analog zu *history.tree* (vgl. Abschnitt 6.4.3, S. 216ff.) – die Unterpakete *communication* und *workspace*.

Abhängigkeiten

Abbildung 6.50 zeigt die Abhängigkeiten von *history.base*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Einfache Menge“ (D.1.2.1.2) (Markierung: *baseSet*) zur Konfiguration gehört. In der Modulbibliothek existiert eine Abhängigkeit zum CoObRA2-Persistenzmodul (*persistency.coobra*, vgl. Abschnitt 6.4.11, S. 235ff.) – falls dieser Mechanismus über das Merkmal „CoObRA Protokoll“ (O.4.1) (Markierung: *coObRAprotocol*) ausgewählt ist. Die restlichen Abhängigkeiten bestehen zu Kernmodulen. Als Historie bestehe eine Abhängigkeit zu *core.history* (vgl. Abschnitt 6.3.4, S. 178ff.), das u.a. eine Fabrik für Elemente des Server-Teilsystems verlangt (*core.server*, vgl. Abschnitt 6.3.10, S. 195ff.). Weiterhin speichert eine Historie Versionen

von Produktmodell-Elementen (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.), die eine Versionskennung erhalten (*core.id*, vgl. Abschnitt 6.3.5, S. 182ff.) und von einem Benutzer eingespielt werden (*core.user*, vgl. Abschnitt 6.3.13, S. 203ff.). Außerdem existieren Kommandos, um Informationen der Historie abzufragen (*core.communication*, vgl. Abschnitt 6.3.1, S. 173ff.). Es existiert jedoch kein Arbeitsbereich-Modul, da diese aus Zeitgründen vor allem für die vom Eclipse-Client (vgl. Abschnitt 7, S. 315ff.) verwendeten Module modelliert wurden (s. Arbeitsbereich-Modul von *history.tree*, vgl. Abschnitt 6.4.3, S. 216ff.).

Detailbeschreibung

Das Modul *history.base* modelliert eine Historie ohne explizite Beziehungen. Es ist damit die einfachste Implementierung der Schnittstellen aus *core.history* (vgl. Abschnitt 6.3.4, S. 178ff.), wie auch das Klassendiagramm in Abbildung 6.51 zeigt. Es existiert eine Modul-Klasse (*BaseVersionsModule*), die als Fabrik für die Historien-Objekte (*BaseVersionSet*) dient und auch die, an das Modul delegierten, Kommandos verarbeitet (*processCommand()*, vgl. Abschnitt 6.3.10, S. 195ff.).

Eine Historie (*BaseVersionSet*) dieses Moduls ist eine ungeordnete Menge von Versionen (*BaseVersion*) eines Produktmodell-Elements (modelliert durch die Komposition „has versions“). Die Elemente können aus einem beliebigen Produktmodell stammen, da der Parameter *item* der Methoden *addVersion()* und *retrieveVersion()* nur die Schnittstelle *IProductModelItem* aus dem Kernmodule *core.product* erwartet (s. *core.history*, vgl. Abschnitt 6.3.4, S. 178ff.). Eine Version (*BaseVersion*) besitzt keine weiteren Informationen – außer der Versionskennung, die das Kernmodul (*AbstractVersion*, vgl. Abschnitt 6.3.4, S. 178ff.) verlangt. Dieses Modul ist die primitivste Implementierung einer Historie, da keine Verknüpfungen zwischen den Versionen existieren. Diese lassen sich lediglich über die Versionskennung nachbilden, wie z.B. in den SKMS „SCCS“ oder „CVS“ (vgl. Abschnitt 4.3.5, S. 85ff.).

6.4.2. Modul: history.flat

Überblick

Das Modul *history.flat* ist das Modell einer Historie mit expliziter Vorgänger-Nachfolger-Beziehung, d.h. die Versionen bilden eine einzige Sequenz, können aber nicht als Varianten (über Verzweigungen) gespeichert werden (vgl. Abschnitt 4.3.3, S. 80ff.). Wie bei *history.base* sind keine Module für das Arbeitsbereichs- oder Kommunikations-Teilsystem implementiert. Ein vollständiges Modul enthielte – analog zu *history.tree* (vgl. Abschnitt 6.4.3, S. 216ff.) – die Unterpakete *communication* und *workspace*.

Abhängigkeiten

Abbildung 6.52 zeigt die Abhängigkeiten von *history.flat*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Sequenz“ (D.1.2.1.4) (Markierung: *sequence*) zur Konfiguration gehört. Die abhängigen Module und die Gründe für die Abhängigkeiten sind die selben wie bei *history.base* in Abschnitt 6.4.1.

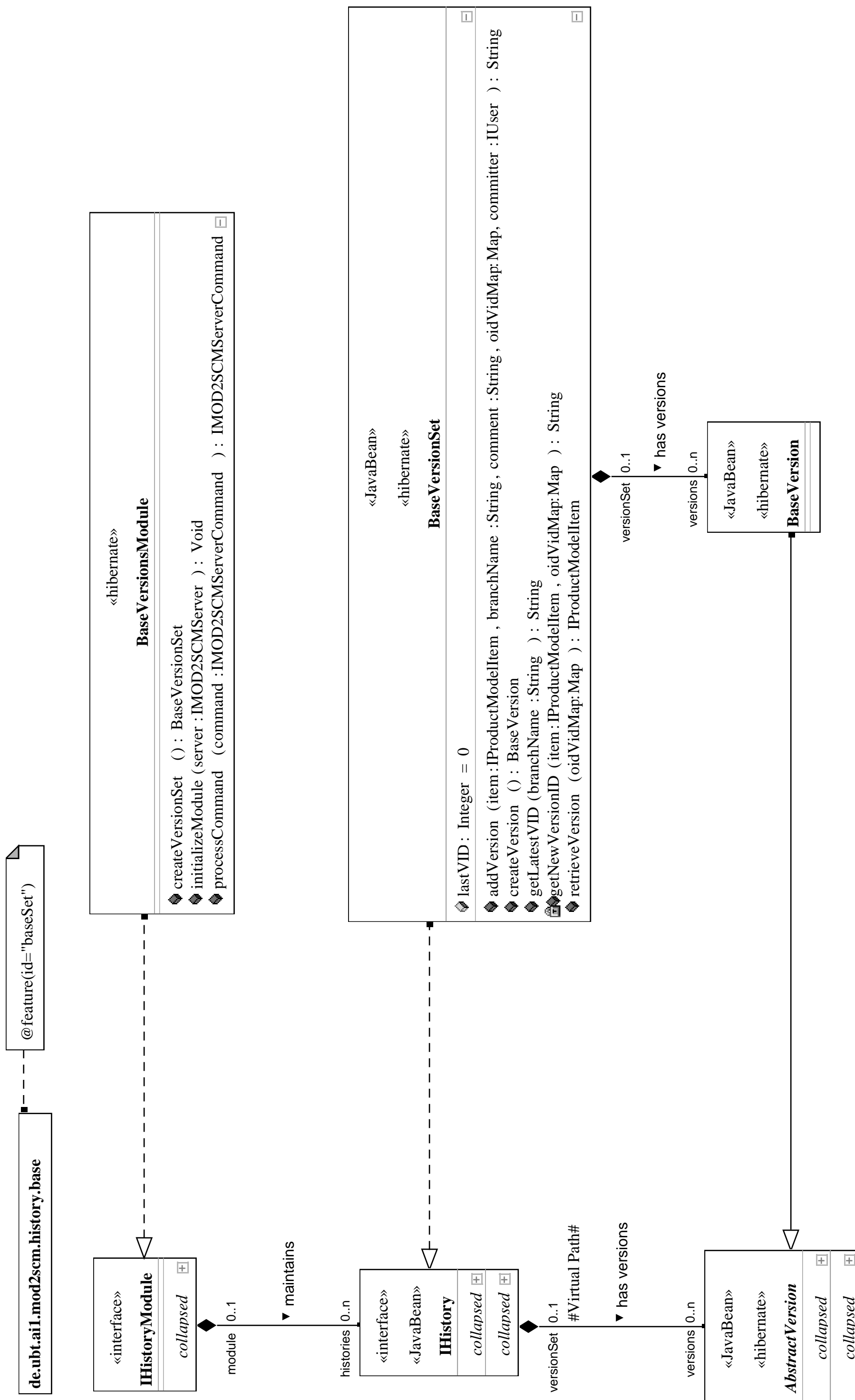


Abbildung 6.51.: Klassendiagramm: history.base

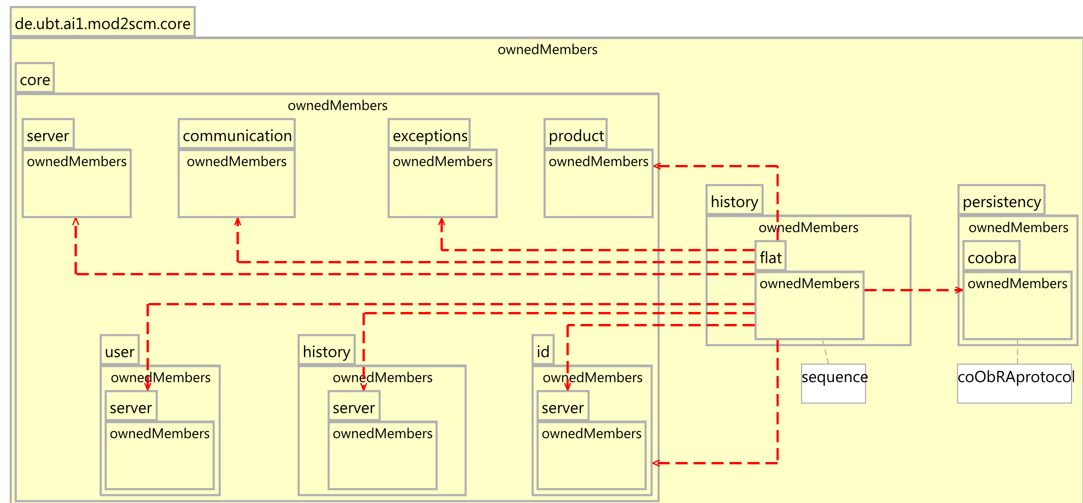


Abbildung 6.52.: Paketdiagramm: history.flat

Detailbeschreibung

Das Modul *history.flat* modelliert eine Historie mit Vorgänger-Nachfolger-Beziehung (vgl. Abschnitt 4.3.3, S. 80ff.). Gemäß des Kernmoduls für Historien (*core.history*, vgl. Abschnitt 6.3.4, S. 178ff.), existiert eine Fabrik-Klasse, die auch die zugehörigen Kommandos verarbeitet (*FlatHistoryModule*) und eine, vom konkreten Produktmodell unabhängige, Historie (*FlatHistory*), die ein oder mehrere Versionen eines Produktmodell-Elements verwaltet (*Version*). Im Gegensatz zur unstrukturierten Historie (*history.base*, vgl. Abschnitt 6.4.1, S. 213ff.) existiert jedoch eine explizite Assoziation „has Next“, um die Vorgänger-Nachfolger-Beziehung zu modellieren.

6.4.3. Modul: history.tree

Überblick

Das Modul *history.tree* ist das Modell einer Historie mit zwei Arten von expliziter Beziehung: Vorgänger-Nachfolger und Verzweigung. Damit lassen sich auch mehrere Varianten einer Version speichern (vgl. Abschnitt 4.3.3, S. 80ff.). Im Gegensatz zu *history.base* und *history.flat* enthält diese Modul auch Komponenten im Arbeitsbereichs- und Kommunikations-Teilsystem, d.h. es zeigt den vollständigen Umfang, den ein Historien-Modul letztendlich besitzt.

Abhängigkeiten

Abbildung 6.54 zeigt die Abhängigkeiten von *history.tree*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Baum“ (D.1.2.1.1) (Markierung: *tree*) zur Konfiguration gehört. Die abhängigen Module und die Gründe für die Abhängigkeiten sind die selben wie bei *history.base* in Abschnitt 6.4.1.

Abbildung 6.53.: Klassendiagramm: history.flat

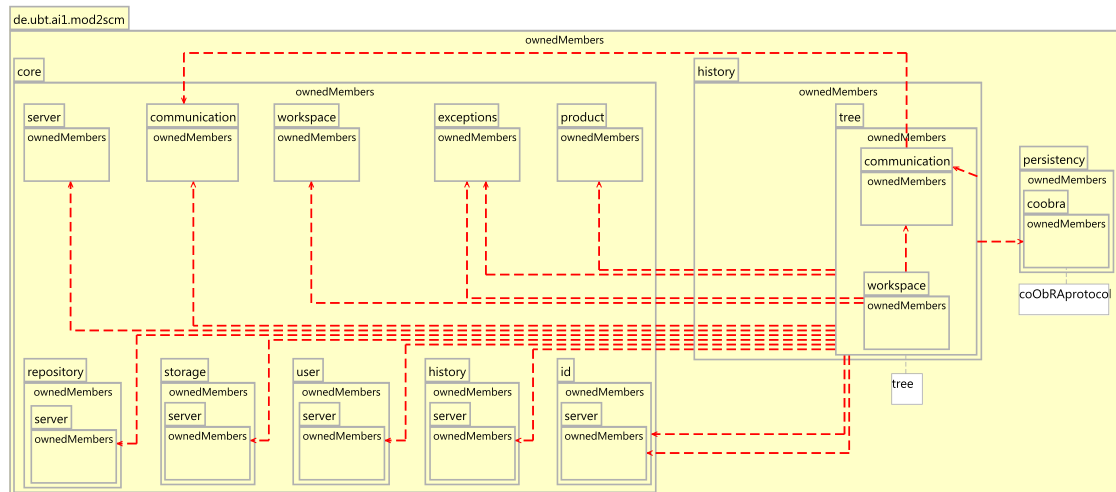


Abbildung 6.54.: Paketdiagramm: history.tree

Im Gegensatz zu den anderen beiden Historien-Modulen existieren jedoch auch Abhängigkeiten zum Arbeitsbereich, da für *history.tree* auch ein Arbeitsbereich-Modul modelliert wurde (*history.tree.workspace*, s. Abb. 6.54).

Detailbeschreibung

Das Modul *history.tree* modelliert eine Baum-Historie, d.h. es existieren sowohl Vorgänger-Nachfolger- als auch Varianten-Beziehungen (vgl. Abschnitt 4.3.3, S. 80ff.). Abbildung 6.55 zeigt das Klassendiagramm des Moduls. Wie bei den beiden vorherigen Historien existieren auch hier ein Fabrik-Modul, das die Kommandos aus *history.tree.-communication* verarbeitet (*TreeHistoryModule*). Die Historie (*TreeHistory*) verwaltet die Versionen eines einzigen Produktmodell-Elements. Die Baum-Historie ist mittels Revisionen (*Revision*) und Zweigen (*Variant*) modelliert. Sie besteht aus mindestens einem Zweig („has Branch“-Komposition auf *Variant*) und über die „has trunk“-Assoziation wird einer der Zweige als Hauptzweig (engl. trunk) referenziert.

Die Schnittstelle des Kernmoduls *core.history* kennt jedoch keine Varianten, sondern erwartet lediglich eine ungeordnete Versionsmenge (vgl. Abschnitt 6.3.4, S. 178ff.). *Variant*-Objekte referenzieren daher kein Produktmodell-Element, sondern verweisen lediglich auf die erste Revision des Zweigs („has Revision“-Komposition). Die Versionsmenge wird nur aus Revisionen gebildet, die dementsprechend die *AbstractVersion*-Klasse aus *core.history* erweitern. Mit Hilfe des *TreeVersionIterator* wird dann die *iteratorOfVersions*-Methode des virtuellen Pfads „has versions“ modelliert, so dass sich jede Baumstruktur aus Revision- und Variant-Objekten über die *core.history*-Schnittstelle navigieren lässt.

Abbildung 6.55.: Klassendiagramm: history.tree

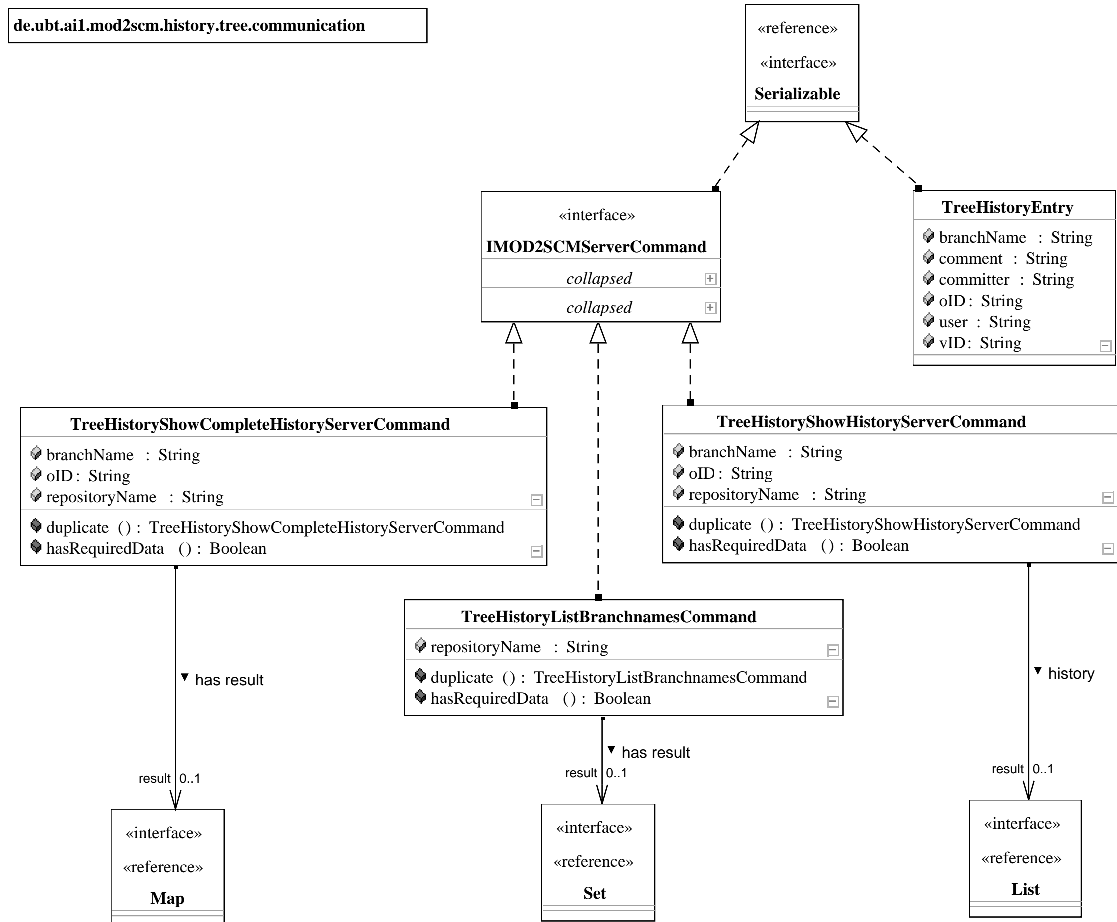


Abbildung 6.56.: Klassendiagramm: history.tree.communication

Kommunikations-Teilsystem: *history.tree.communication*

Abbildung 6.56 zeigt das Klassendiagramm des Kommunikations-Teilsystems von *history.tree*. Basierend auf der Schnittstelle des Kernmoduls *core.communication*, existieren ein Kommando zum Auflisten der Zweignamen (*TreeHistoryListBranchnamesCommand*) und zwei Kommandos zum Anzeigen der Historie (*TreeHistoryShowHistoryServerCommand* und *TreeHistoryShowCompleteHistoryServerCommand*). In beiden Fällen werden die Versionshistorien eines einzelnen Produktmodell-Elements (Parameter: *oid*) zurückgeliefert. *TreeHistoryShowHistoryServerCommand* gibt jedoch nur die Liste der Versionskennungen zurück, während *TreeHistoryShowCompleteHistoryServerCommand* detaillierte Informationen zu jeder Version liefert (*TreeHistoryEntry*).

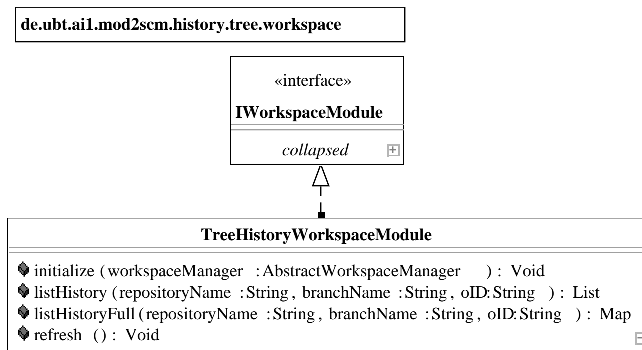


Abbildung 6.57.: Klassendiagramm: history.tree.workspace

Objektkennung	Versionskennung	Zweigname
id.hash	Objektkennung	erforderlich
id.primarykey	id.hierarchic, id.primarykey	optional
id.qualifiedname	id.hierarchic, id.primarykey	optional

Tabelle 6.2.: Abhängigkeiten zwischen Kennungen und Zweignamen

Arbeitsbereichs-Teilsystem: *history.tree.workspace*

Abbildung 6.57 zeigt das Arbeitsbereichs-Modul (*TreeHistoryWorkspaceModule*) von *history.tree*. Es bietet Zugriff auf die Versionsinformationen der Historie einzelner Produktmodell-Elemente, indem es die o.g. Kommandos an das Server-Teilsystem sendet. Diese Informationen werden z.B. in der Historienansicht des MOD2-SKM-Plugins angezeigt (vgl. Abschnitt 7, S. 315ff.).

Abhängigkeiten zwischen Kennungen und Zweignamen

Ein Tupel aus Objekt- und Versionskennung (oid, vid) identifiziert eine Version eines Produktmodell-Elements eindeutig (vgl. Abschnitt 4.3.2, S. 79ff.). In der Baumhistorie existiert noch ein Zweigname als weiteres Identifikationsmerkmal. Daher stellt sich die Frage: Ist der Zweigname Teil der Versionskennung? Solange die Versionskennung die Eigenschaften einer eindeutigen Kennung aufweist (vgl. Abschnitt 4.2.2, S. 76ff.), ist der Zweigname zur Identifikation nicht unbedingt erforderlich. Erst wenn die Kennung keine singuläre Referenz mehr ist, muss der Zweigname Teil der Versionskennung sein.

Tabelle 6.2 zeigt, dass der Zweigname nur bei Hash-Kennungen zur Identifikation notwendig ist (vgl. Abschnitt 6.4.4, S. 222ff.). Wird z.B. von einem unveränderten Element eine Variante angelegt, dann verändert sich der Hashwert nicht. Die Abhängigkeit lässt sich auflösen, indem der Zweigname der Baumhistorie in jedem Fall an die generierte Versionskennung angehängt wird. Im Gegensatz zu den beiden vorigen Historien *history.base* und *history.flat* wird hier die, durch ein Kennungsmodul (vgl. Abschnitt 6.3.5, S. 182ff.) generierte, Versionskennung noch durch die Historie verändert.

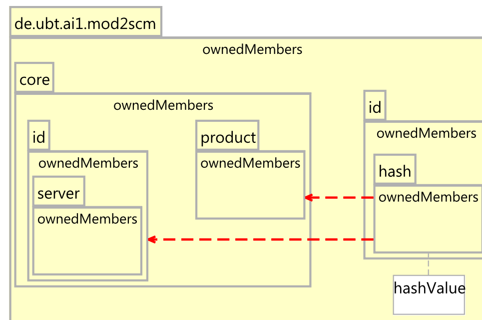


Abbildung 6.58.: Paketdiagramm: id.hash

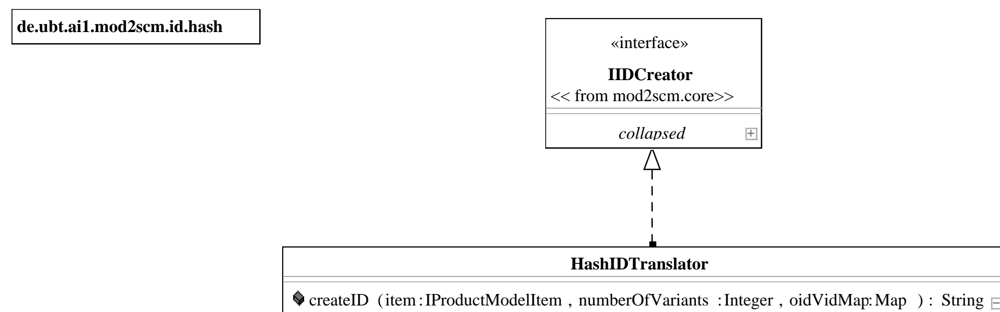


Abbildung 6.59.: Klassendiagramm: id.hash

6.4.4. Modul: id.hash

Überblick

Das Modul *id.hash* erzeugt eine Kennung für ein Produktmodell-Element mit Hilfe einer Hashfunktion (vgl. Abschnitt 4.2.4, S. 77ff.).

Abhängigkeiten

Abbildung 6.58 zeigt die Abhängigkeiten von *id.hash*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Hashwert“ (D.8.1.3) (Markierung: *hashValue*) zur Konfiguration gehört. Das Modul basiert auf der Schnittstelle des Kernmoduls (*core.id*) und berechnet die Kennung für ein beliebiges Produktmodell-Element (*core.product*).

Detailbeschreibung

Abbildung 6.59 zeigt das Klassendiagramm von *id.hash*. Der *HashIDCreator* implementiert die Schnittstelle *IIDCreator* aus *core.id* und liefert den Hashwert des übergebenen Produktmodell-Elements (Parameter: *item*).

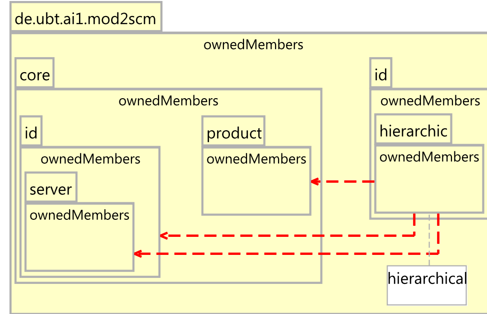


Abbildung 6.60.: Paketdiagramm: id.hierarchic

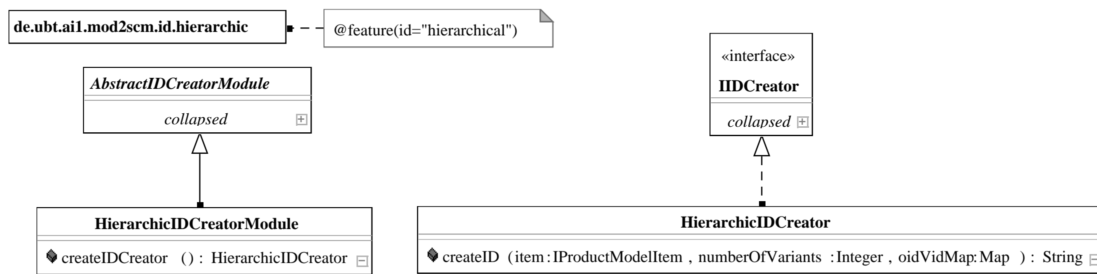


Abbildung 6.61.: Klassendiagramm: id.hierarchic

6.4.5. Modul: id.hierarchic

Überblick

Das Modul *id.hierarchic* erzeugt eine hierarchische numerische Versionskennung eines Produktmodell-Elements anhand einer Vorgängerkennung und der Anzahl bereits existierenden Varianten (vgl. Abschnitt 4.2.4, S. 77ff.).

Abhängigkeiten

Abbildung 6.60 zeigt die Abhängigkeiten von *id.hierarchic*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Hierarchisch“ (D.2.2.2.2) (Markierung: *hierarchical*) zur Konfiguration gehört. Das Modul basiert auf der Schnittstelle des Kernmoduls (*core.id*) und berechnet die Kennung für ein beliebiges Produktmodell-Element (*core.product*).

Detailbeschreibung

Abbildung 6.61 zeigt das Klassendiagramm von *id.hierarchic*. Der *HierarchicIDCreator* implementiert die Schnittstelle *IIDCreator* aus *core.id* und liefert eine hierarchische numerische Kennung für das übergebene Produktmodell-Element (Parameter: *item*). Der Parameter „oidVidMap“ enthält die Vorgängerkennung. Ist der Parameter „numberOfVariants“ gleich 0, wird die letzte Stelle der Vorgängerkennung um 1 erhöht, z.B. wird „1.2“

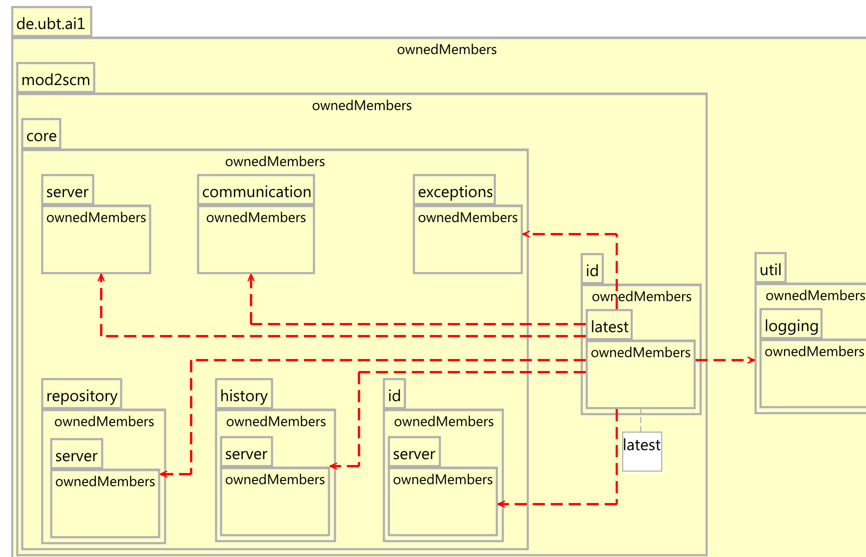


Abbildung 6.62.: Paketdiagramm: id.latest

zu „1.3“. Ist der Parameter „numberOfVariants“ größer als Null, wird an die Vorgängerkennung ein weiterer zweistelliger Zähler angehängt, z.B. wird bei 3 Varianten „1.2“ zu „1.2.6.1“[4].

6.4.6. Modul: id.latest

Überblick

Das Modul *id.latest* ermöglicht die Identifikation der zuletzt eingespielten Version eines Produktmodell-Elements. Dieses Modul bietet damit ein ergänzendes Identifikationsmerkmal an, das sich zusätzlich zur Versionskennung verwenden lässt.

Abhängigkeiten

Abbildung 6.62 zeigt die Abhängigkeiten von *id.latest*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Neuste Version“ (D.2.1.3) (Markierung: *latest*) zur Konfiguration gehört. Als zusätzliche Identifikationsmöglichkeit (*core.id*) ist dieses Modul ein austauschbares Servermodul (*core.server* und *core.communication*). Um die letzte Version zu finden, ist es notwendig, über die Repositorien zur entsprechenden Historie zu navigieren (*core.history* und *core.repository*). Die Anfragen werden protokolliert (*util.logging*) und, bei Bedarf, Ausnahmen geworfen (*core.exceptions*).

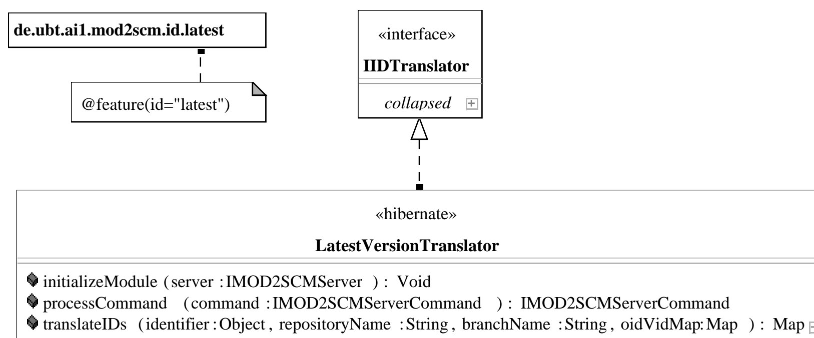


Abbildung 6.63.: Klassendiagramm: id.latest

Detailbeschreibung

Abbildung 6.63 zeigt das Klassendiagramm des Moduls *id.latest*. Es implementiert die Schnittstelle für zusätzliche Identifikations-Module (*IIDTranslator*, vgl. Abschnitt 6.3.5, S. 182ff.). Enthält der Parameter *identifier* den Wert „true“, dann wird für alle Objektkennungen des Parameters *oidVidMap* die zuletzt eingespielte Versionskennung eingetragen und zurückgegeben. Damit entspricht der Rückgabewert einer Liste von (oid, vid)-Tupeln, die mit den Kernmodul-Schnittstellen von Historie und Repositorium kompatibel ist (vgl. Abschnitt 6.3.4, S. 178ff. und vgl. Abschnitt 6.3.9, S. 191ff.).

6.4.7. Modul: id.primarykey

Überblick

Das Modul *id.primarykey* erzeugt eine Kennung für ein Produktmodell-Element anhand einer laufenden Nummer (vgl. Abschnitt 4.2.4, S. 77ff.).

Abhängigkeiten

Abbildung 6.64 zeigt die Abhängigkeiten von *id.primarykey*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Global eindeutige ID“ (D.8.1.2) (Markierung: *globalUniqueIdentifier*) zur Konfiguration gehört. Das Modul basiert auf der Schnittstelle des Kernmoduls (*core.id*) und berechnet die Kennung für ein beliebiges Produktmodell-Element (*core.product*).

Detailbeschreibung

Abbildung 6.65 zeigt das Klassendiagramm von *id.primarykey*. Der *PrimaryKeyIDCreator* implementiert die Schnittstelle *IIDCreator* aus *core.id* und liefert eine laufende Nummer zurück. Diese wird nach jeder Anfrage um eins erhöht und gespeichert (Variable: *primaryKey*).

[4] In Anlehnung an CVS ist die erste Stelle eines neuen Zählers immer eine gerade Zahl, so dass der Wert von *numberOfVariants* verdoppelt wird $2 * 3 = 6$

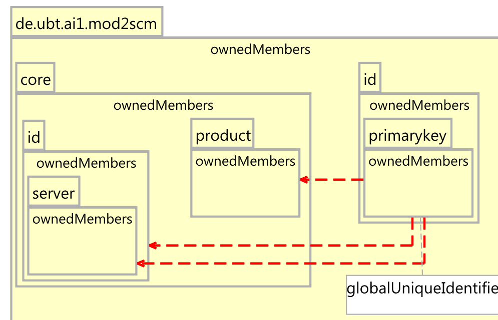


Abbildung 6.64.: Paketdiagramm: id.primarykey

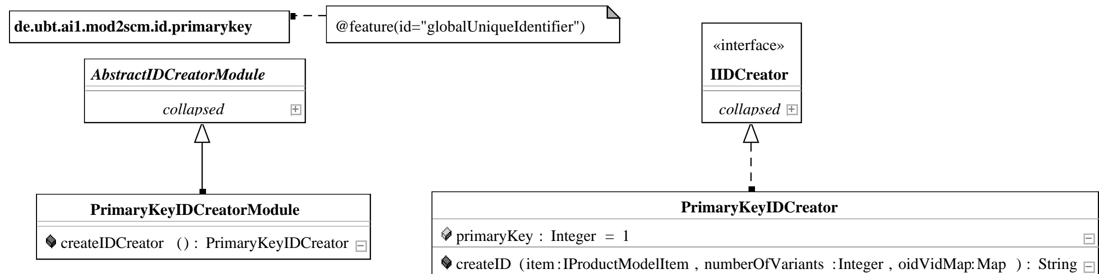


Abbildung 6.65.: Klassendiagramm: id.primarykey

6.4.8. Modul: id.qualifiedname

Überblick

Das Modul *id.qualifiedname* erzeugt eine Kennung eines Produktmodell-Elements anhand seines Namens (vgl. Abschnitt 4.2.4, S. 77ff.).

Abhängigkeiten

Abbildung 6.66 zeigt die Abhängigkeiten von *id.qualifiedname*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Elementname“ (D.8.1.1) (Markierung: *itemname*) zur Konfiguration gehört. Das Modul basiert auf der Schnittstelle des Kernmoduls (*core.id*) und berechnet die Kennung für ein beliebiges Produktmodell-Element (*core.product*).

Detailbeschreibung

Abbildung 6.67 zeigt das Klassendiagramm von *id.qualifiedname*. Der *QualifiedNameIDCreator* implementiert die Schnittstelle *IIDCreator* aus *core.id* und liefert den Namen des übergebenen Produktmodell-Elements (Parameter: *item*) zurück.

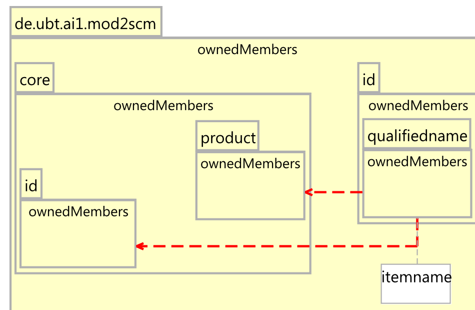


Abbildung 6.66.: Paketdiagramm: id.qualifiedname

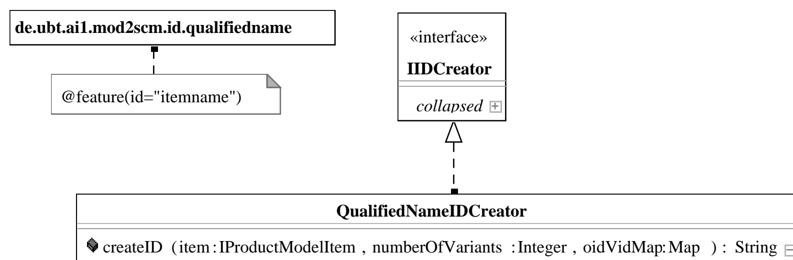


Abbildung 6.67.: Klassendiagramm: id.qualifiedname

6.4.9. Modul: id.tag

Überblick

Das Modul *id.tag* unterstützt das Markieren einer Produktmodell-Instanz mit einer Zeichenkette, um die Instanz durch Angabe dieses Namens wiederherstellen zu können. Dieses Modul bietet damit ein ergänzendes Identifikationsmerkmal an, das sich zusätzlich zur Versionskennung verwenden lässt.

Abhängigkeiten

Abbildung 6.68 zeigt die Abhängigkeiten von *id.tag*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Nach Markierung“ (D.2.1.2) (Markierung: *bytag*) zur Konfiguration gehört. Als zusätzliche Identifikationsmöglichkeit (*core.id*), ist dieses Modul ein austauschbares Servermodul (*core.server* und *core.communication*), das, bei Bedarf, Ausnahmen wirft (*core.exceptions*).

Detailbeschreibung

Das Tag-Modul steuert, im Gegensatz zu den anderen Kennungsmodulen, nicht nur Komponenten zum Server-Teilsystem, sondern auch Elemente zum Kommunikations- und Arbeitsbereich-Teilsystem bei. Grund sind die zusätzlichen Kommandos zum Eingeben und Anzeigen der Markierungen im Arbeitsbereich.

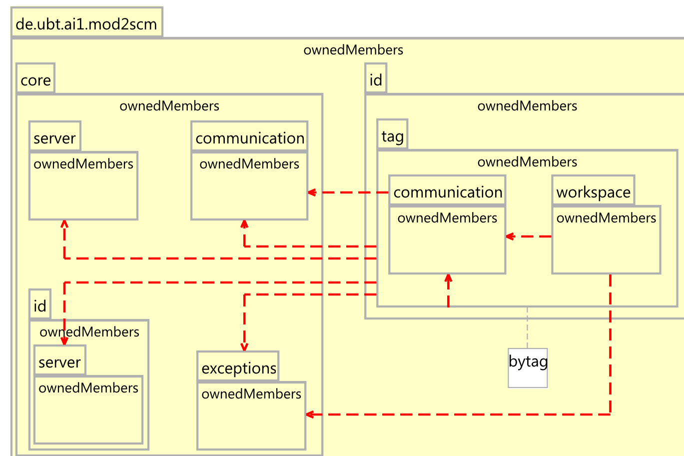


Abbildung 6.68.: Paketdiagramm: id.tag

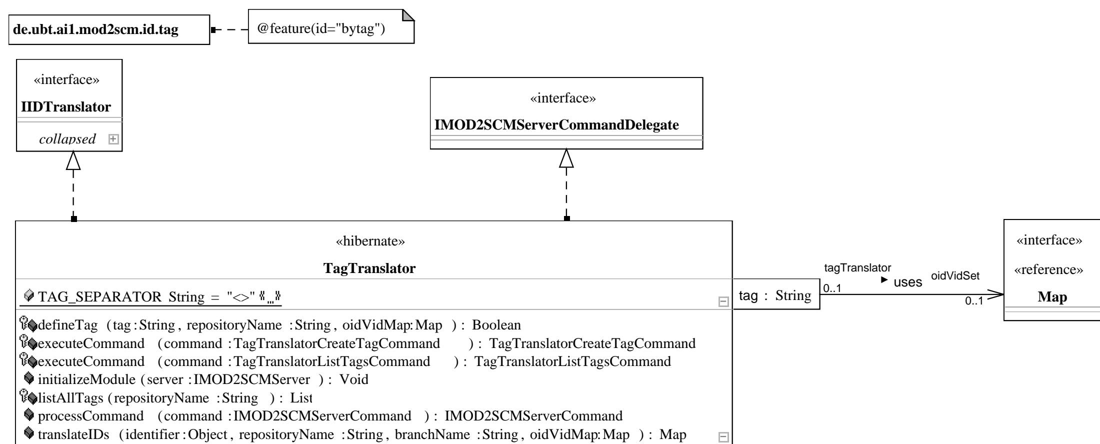


Abbildung 6.69.: Klassendiagramm: id.tag

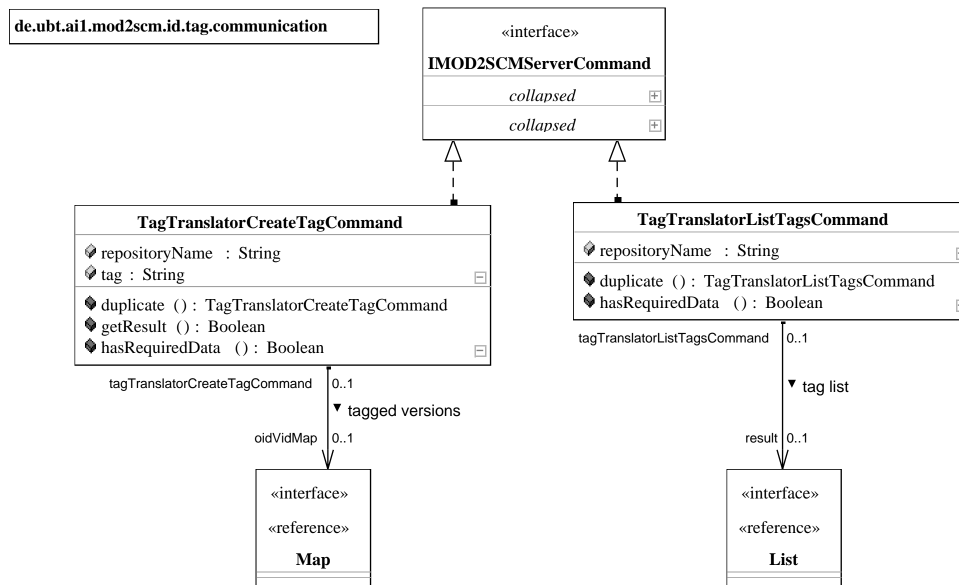


Abbildung 6.70.: Klassendiagramm: id.tag.communication

Abbildung 6.69 zeigt das Klassendiagramm des Server-Teilsystems von *id.tag*. Das Markierungs-Modul (*TagTranslator*) übernimmt sowohl die Speicherung der Markierungen als auch ihre Übersetzung in eine Liste von (oid, vid)-Tupeln, die von den Schnittstellen von *core.history* und *core.repository* verarbeitet werden kann. Beim Anlegen einer Markierung (*defineTag()*) wird unter einem Namen (Parameter: *tag*) eine Liste von (oid, vid)-Tupeln abgelegt (Parameter: *oidVidMap*) und mit Hilfe der qualifizierten Assoziation „uses" gespeichert. Die Methode *listAllTags()* listet alle diese gespeicherten Namen auf.

TagTranslator implementiert die Schnittstelle für zusätzliche Identifikationsmethoden (*IIDTranslator* aus *core.id*, vgl. Abschnitt 6.3.5, S. 182ff.) und damit die Methode *translateIDs()*. Enthält ihr Parameter identifier eine Zeichenkette, versucht diese Modul unter diesem Namen eine Liste aus (oid, vid)-Tupeln zu lokalisieren. Existiert diese Liste, wird sie zurückgegeben und ersetzt die ursprünglich verwendete Liste (Parameter: *oidVidMap*).

Abbildung 6.70 zeigt das Klassendiagramm des Kommunikations-Teilsystems von *id.tag*. Es existieren ein Kommando zum Anlegen (*TagTranslatorCreateTagCommand*) und eines zum Auflisten (*TagTranslatorListTagsCommand*) der Markierungen. *TagTranslatorListTagsCommand* liefert eine Liste mit allen bisher angelegten Markierungen zurück (Assoziation „tag list"). *TagTranslatorCreateTagCommand* speichert unter einer Markierung (Attribut: *tag*) – sofern sie nicht schon existiert – den aktuell im Arbeitsbereich vorhandenen Stand als (oid, vid)-Tupel ab. Das Erstellen dieser Liste übernimmt das Arbeitsbereichs-Modul (*TagWorkspaceModule*) aus Abbildung 6.71 in der Methode *tagWorkspace()*.

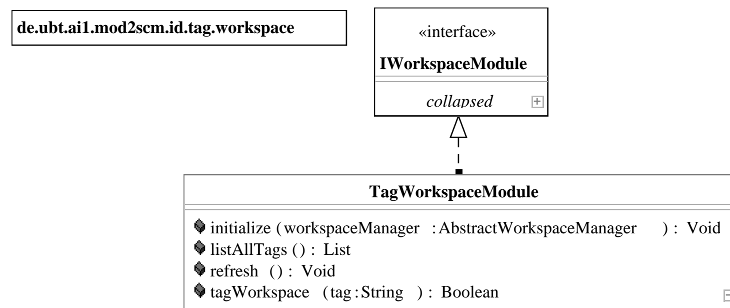


Abbildung 6.71.: Klassendiagramm: id.tag.workspace

6.4.10. Modul: merge

Überblick

Das Modul *merge* von Christopher Bär ermöglicht das Verschmelzen von Produktmodell-Elementen. Die Verschmelzungsverfahren wurden jedoch nicht neu modelliert; vielmehr wurden zwei existierende Verschmelzungswerkzeuge integriert: Das Eclipse-Compare-Rahmenwerk [ecl10] und das Guiffy-Werkzeug [GS10]. Dieses Modul ist in der Bachelorarbeit von Christopher Bär ausführlich beschrieben [Bär10].

Abhängigkeiten

Abbildung 6.72 zeigt die Abhängigkeiten von *merge*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Verschmelzen" (D.11.2.3) (Markierung: *merge*) zur Konfiguration gehört. Das Modul steuert Komponenten zu allen drei Teilsystemen bei und ist daher von *core.server*, *core.communication* und *core.workspace* abhängig. Da Produktmodell-Elemente verschmolzen werden, besteht auch eine Abhängigkeit zu *core.product*. Ob entweder das Untermodul *merge.eclipse* oder *merge.guiffy* Teil einer Konfiguration ist, steuert das Merkmal „Integration in Eclipse" (O.2.3) (Markierung: *clientEclipseIntegration* bzw. *nOTClientEclipseIntegration*). Beide Untermodule besitzen Abhängigkeiten zu den jeweiligen integrierten Rahmenwerken und – da die Rahmenwerke nur Dateien verschmelzen können – auch zum Dateisystem-Produktmodell (*product.filesystem*).

Detailbeschreibung

Abbildung 6.73 zeigt das Klassendiagramm des Server-Teilsystems *merge.server*. Das Verschmelzen-Modul (*MergeServerModule*) implementiert die Schnittstelle für Server-Module (*IMOD2SCMServerModule*, vgl. Abschnitt 6.3.10, S. 195ff.) und modelliert eine Methode zum Lokalisieren des nächsten gemeinsamen Vorgängers zweier Versionen (*fetchAncestor()*). Dieser wird beim Drei-Wege-Verschmelzen benötigt [CW98, Bär10]. Die Ausgangsversionen werden anhand ihres Tupels ($oldOID, oldVID$) bzw. ($newOID, newVID$) identifiziert. Die Tupel werden mittels des in Abbildung 6.74 gezeigten Kommandos

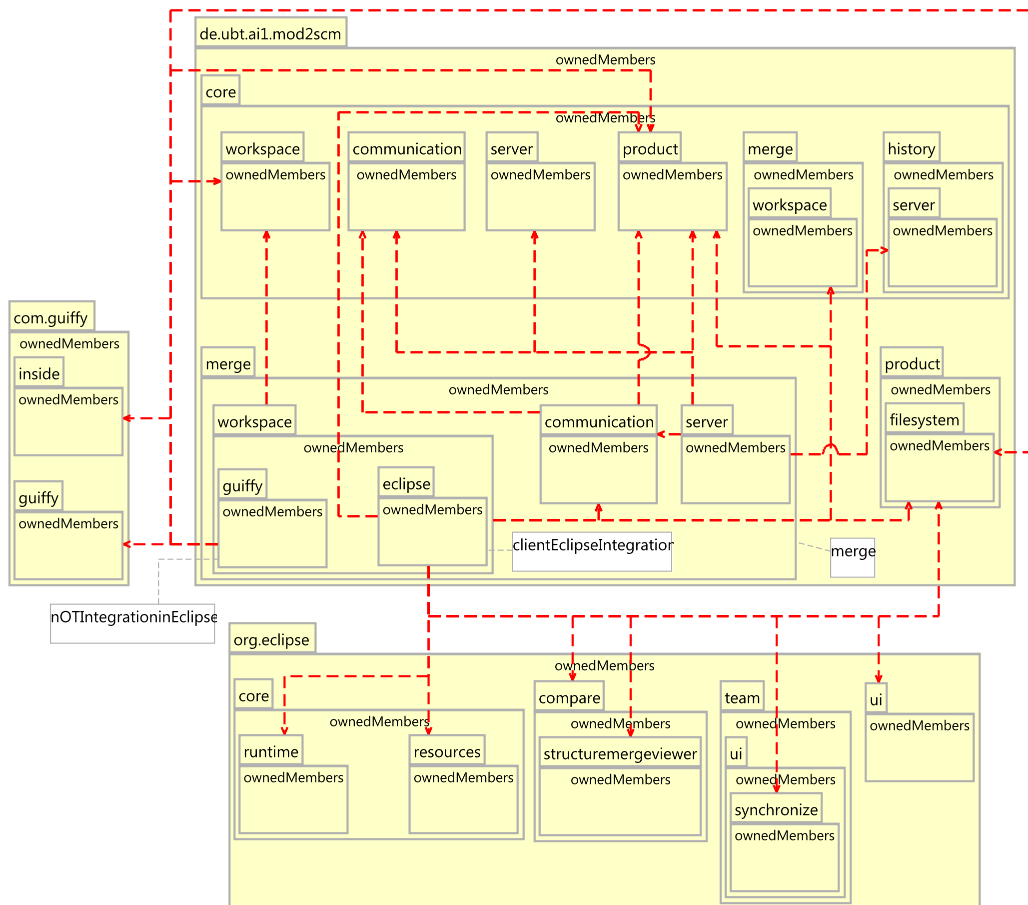


Abbildung 6.72.: Paketdiagramm: merge

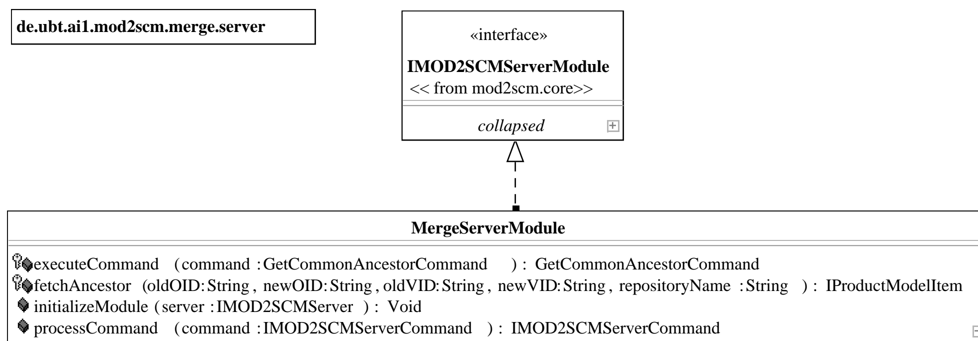


Abbildung 6.73.: Klassendiagramm: merge.server

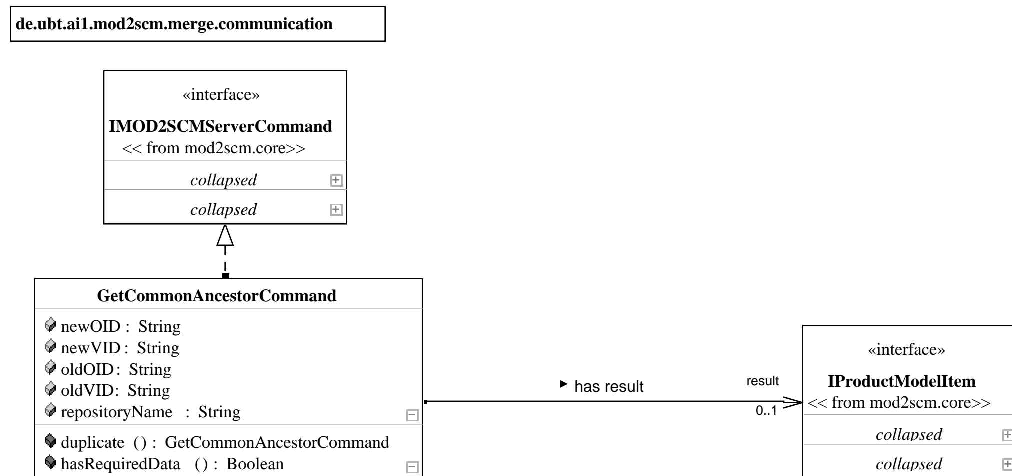


Abbildung 6.74.: Klassendiagramm: merge.communication

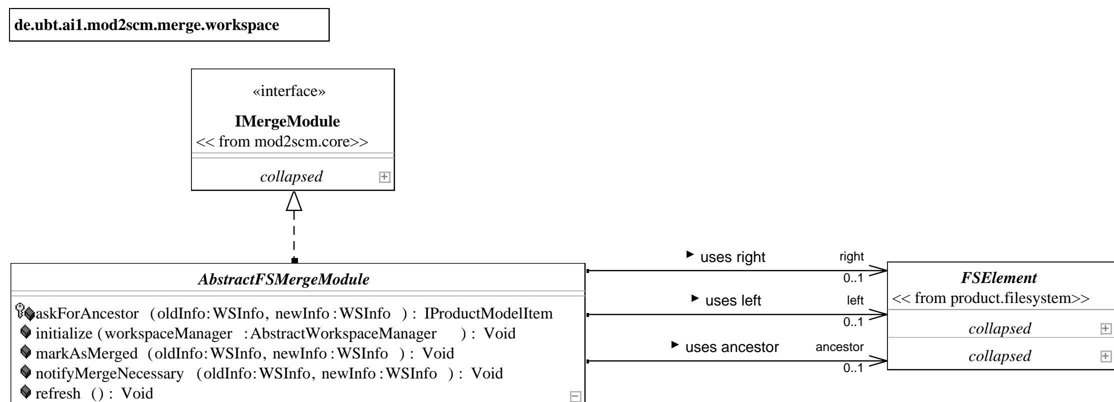


Abbildung 6.75.: Klassendiagramm: merge.workspace

(*GetCommonAncestorCommand* aus *merge.communication*) über das Kommunikations-Teilsystem übermittelt.

Abbildung 6.75 zeigt das Klassendiagramm des Arbeitsbereichs-Teilsystems *merge.-workspace*. Das Verschmelzen-Modul im Arbeitsbereich (*AbstractFSMergeModule*) bietet zunächst eine Methode zur Abfrage des nächsten gemeinsamen Vorfahren (*askForAncestor()*), die mit dem oben beschriebenen Server-Modul in Verbindung tritt. Zusätzlich implementiert *AbstractFSMergeModule* die Schnittstelle *IMergeModule* aus dem Kernmodul *core.merge*. Diese verlangt eine Methode *notifyMergeNecessary()*, die beim Feststellen einer konkurrierenden Änderung während eines Aktualisierungsvorgangs aufgerufen wird (vgl. Abschnitt 6.4.18, S. 254ff.). Es ist nun Aufgabe des Verschmelzungsmoduls, die Produktmodell-Elemente und ihre Metadaten explizit zu sichern. Sind die Elemente verschmolzen, so wird dies über die Methode *markAsMerged()* bekannt gegeben, und das Verschmelzen-Modul löscht die Ursprungsversionen und integriert das neue Element in

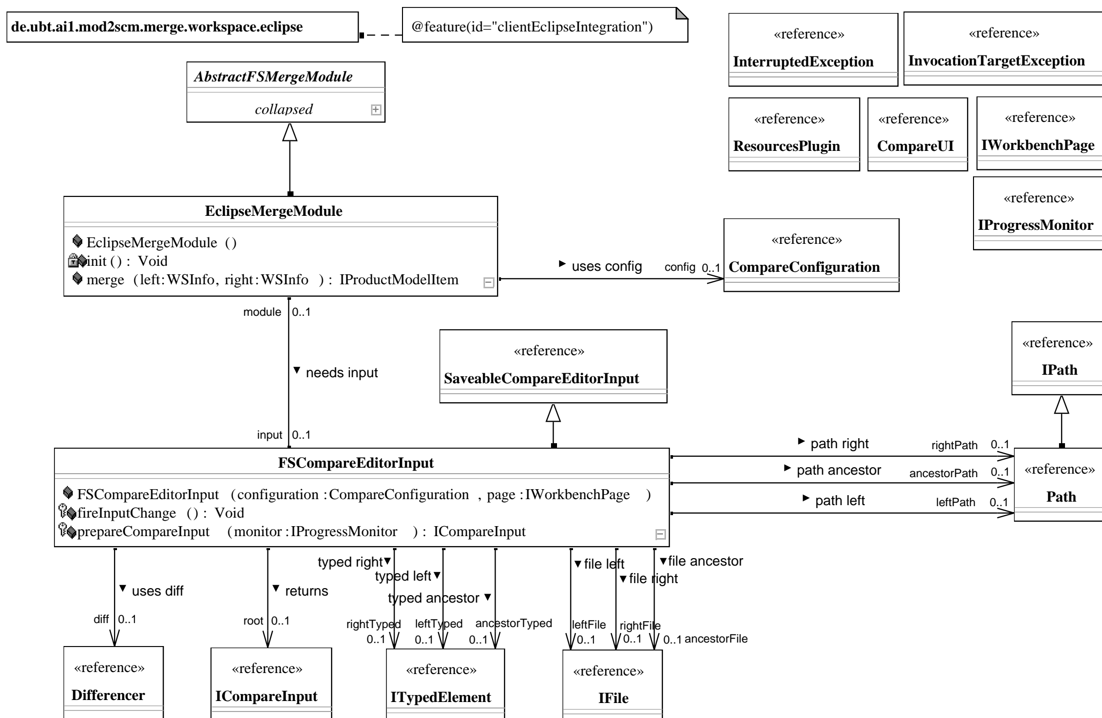


Abbildung 6.76.: Klassendiagramm: merge.workspace.eclipse

den Arbeitsbereich. Das Modul selbst ist abstrakt, denn das Verschmelzen wird in den jeweiligen Untermodulen realisiert [Bär10].

Untermodul: merge.eclipse

Abbildung 6.76 zeigt das Klassendiagramm des Untermoduls *merge.workspace.eclipse*, das die Verschmelzungsverfahren der Eclipse-Entwicklungsumgebung in den MOD2-SKM Arbeitsbereich integriert. Es ist nur Teil einer Konfiguration, wenn das Merkmal „Integration in Eclipse“ (O.2.3) (Markierung: *clientEclipseIntegration*) ausgewählt ist. Das *EclipseMergeModule* modelliert die Methode *merge()*, die zwei Produktmodell-Elemente miteinander verschmilzt. Eclipse beschränkt das Produktmodell dabei auf das Dateisystem (*product.filesystem*, vgl. Abschnitt 6.4.14, S. 240ff.) [Bär10, ecl10].

Das Klassendiagramm in Abbildung 6.76 demonstriert, wie die Integration eines proprietären Rahmenwerks in das Domänenmodell aussehen kann. Die benötigten Klassen und Schnittstellen des Eclipse-Compare-Rahmenwerks sind als Referenzen deklariert (vgl. Abschnitt 6.1, S. 144ff.) und über unidirektionale Assoziationen referenziert. Die Integration ermöglicht den Einsatz der grafischen Oberfläche des Eclipse-Verschmelzungs-Editors und das manuelle Verschmelzen zweier Dateien während des Aktualisierungsvorgangs. Für eine detaillierte Beschreibung sei auf [Bär10] verwiesen.

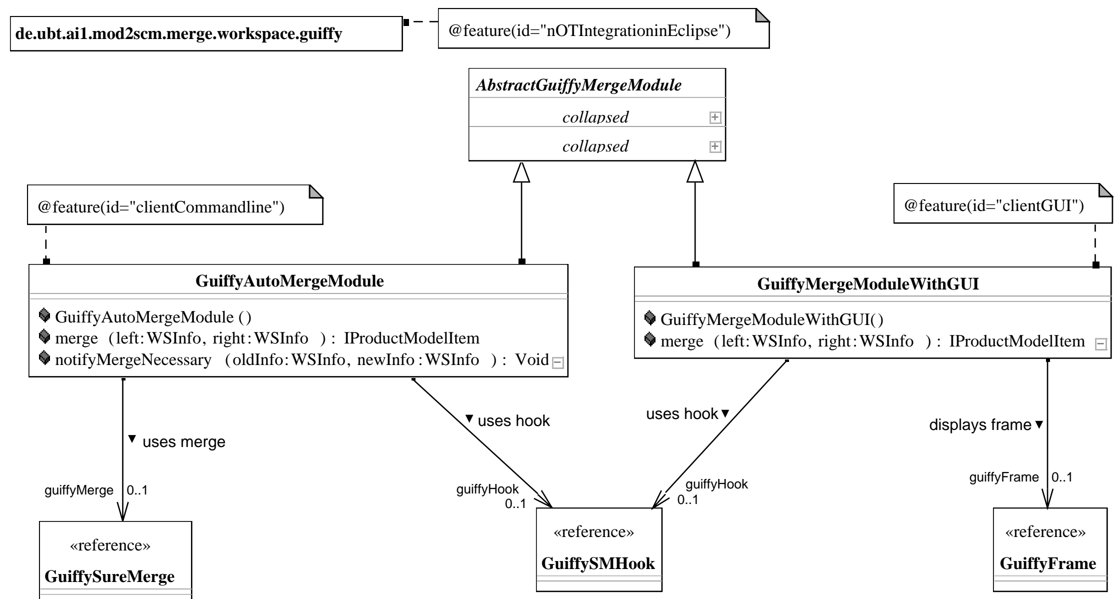


Abbildung 6.77.: Klassendiagramm: merge.workspace.guiffy

Untermodul: merge.guiffy

Abbildung 6.77 zeigt das Klassendiagramm des Untermoduls *merge.workspace.guiffy*, das die Verschmelzungsverfahren des Guiffy-Verschmelzungswerkzeugs [GS10] in den MOD2-SKM Arbeitsbereich integriert. Es ist nur Teil einer Konfiguration wenn das Merkmal „Integration in Eclipse“ (O.2.3) (Markierung: *nOTClientEclipseIntegration*) deselektiert ist. Basierend auf dem Guiffy-Verschmelzungs-Werkzeug, existieren zwei Untermodule. Beide implementieren die *merge()*-Methode, um zwei Produktmodell-Elemente zu verschmelzen. Wie Eclipse beschränkt Guiffy das Produktmodell auf das Dateisystem.

Das *GuiffyAutoMergeModule* verschmilzt zwei Dateien automatisch. Muss der Konflikt manuell aufgelöst werden, fügt es beide Varianten in die Datei ein und markiert die Konfliktstellen. Die Anwender müssen diese nun manuell editieren und anschließend das Modul über die Auflösung unterrichten (Aufruf von *markAsMerged()* aus Abbildung 6.75). Das *GuiffyMergeModuleWithGUI* verschmilzt die Dateien automatisch und öffnet, bei Bedarf, einen grafischen Verschmelzungs-Editor. Nach dem Speichern einer manuell verschmolzenen Datei, muss der Entwickler noch das erfolgreiche Verschmelzung melden. Wie beim Eclipse-Modul, sind auch die Guiffy-Klassen und -Schnittstellen als Referenzen im Domänenmodell modelliert. Im Vergleich benötigt das Guiffy-Modul deutlich weniger Referenzen als das Eclipse-Modul, so dass sich Ersteres deutlich leichter integrieren ließ. Für eine detaillierte Beschreibung sei hier ebenfalls auf [Bär10] verwiesen.

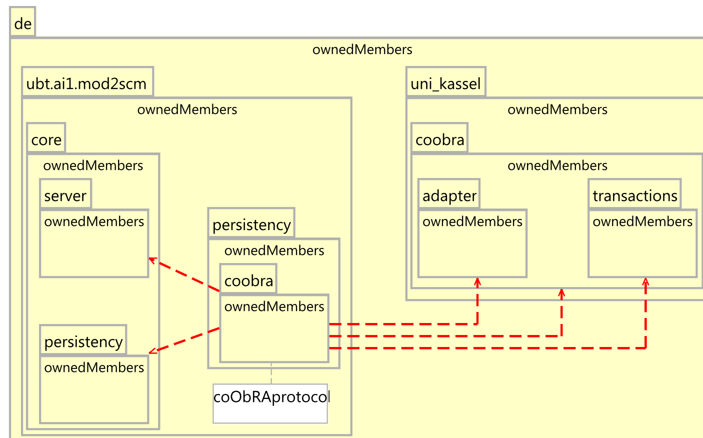


Abbildung 6.78.: Paketdiagramm: persistency.coobra

6.4.11. Modul: persistency.coobra

Überblick

Das Modul *persistency.coobra* integriert den CoObRA2-Persistenzmechanismus [Sch07] zur Persistenzierung des Server-Teilsystems. Das Modul kann prinzipiell auch den Arbeitsbereichs persistenzieren, wird aber lediglich auf dem Server eingesetzt (vgl. Abschnitt 6.4.27, S. 292ff.).

Abhängigkeiten

Abbildung 6.78 zeigt die Abhängigkeiten von *persistency.coobra*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „CoObRA Protokoll" (O.4.1) (Markierung: *coObRAprotocol*) zur Konfiguration gehört. Das Modul implementiert die Schnittstelle aus dem Persistenz-Kernmodul *core.persistency*. Da es das Server-Teilsystem persistenziert, ist es auch von *core.server* abhängig. Zusätzlich existieren auch Abhängigkeiten zu proprietären Komponenten von CoObRA2.

Detailbeschreibung

Abbildung 6.79 zeigt das Klassendiagramm des Moduls *persistency.coobra*. Die Klasse *CoobraPersistencyModule* implementiert die Schnittstelle aus *core.persistency* und modelliert Methoden zum Speichern und Laden des Server-Teilsystems (*store()* und *restore()*, vgl. Abschnitt 6.3.7, S. 186ff.) und zum Verwalten von Transaktionen (*beginTransaction()*, *cancelTransaction()* und *finishTransaction()*, vgl. Abschnitt 6.3.7, S. 186ff.). In ihren Story-Diagrammen delegieren sie die entsprechenden Aufrufe an das CoObRA2-Repositorium bzw. den CoObRA2-Transaktionsmechanismus weiter. Da CoObRA2 die Änderungen an registrierten Objekten protokolliert, dient die store()-Methode lediglich zur Registrierung. Anschließend werden alle Änderungen direkt nach Erfolgen in das

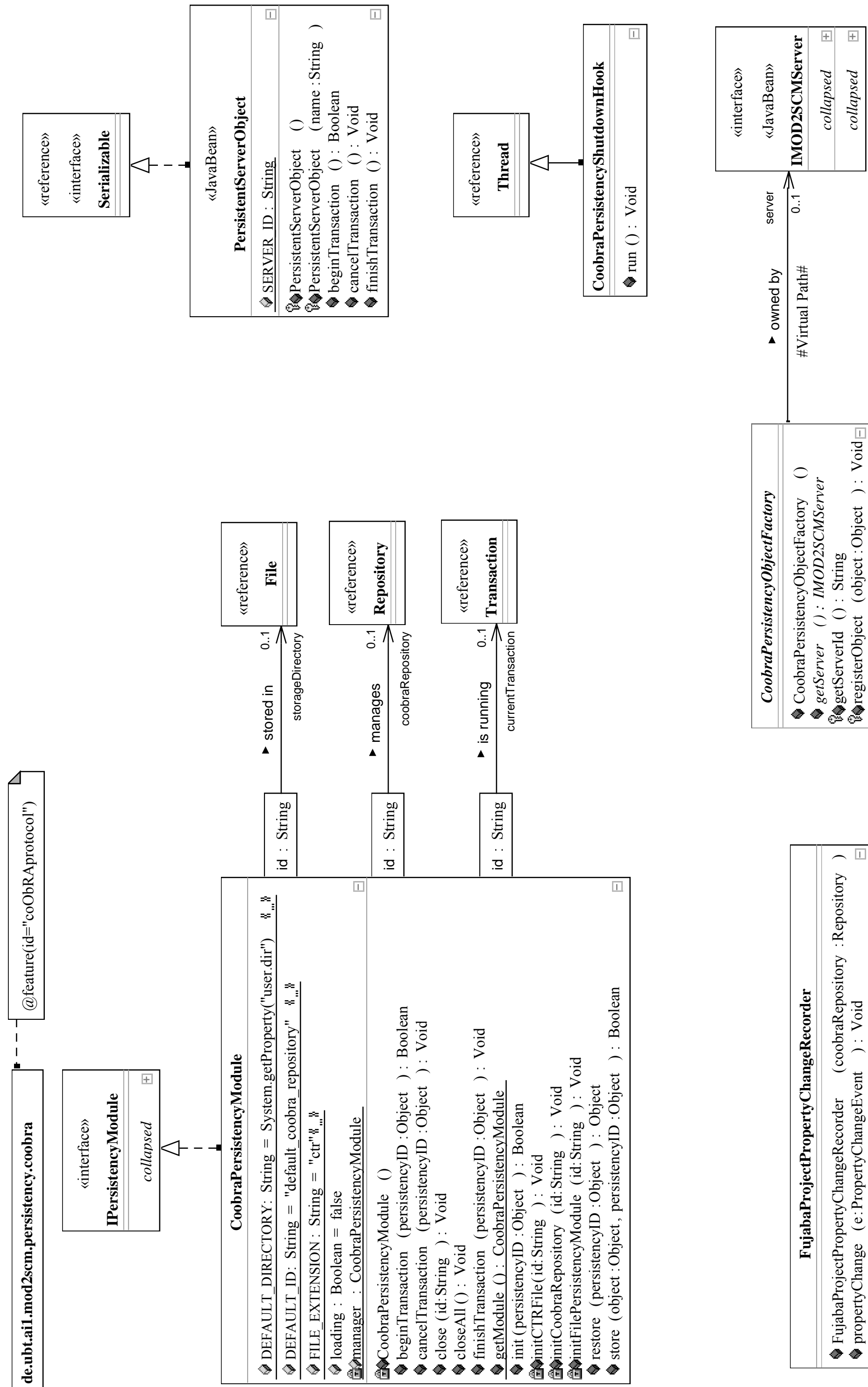


Abbildung 6.79.: Klassendiagramm: persistency.coobra

Protokoll eingetragen. Für eine detaillierte Beschreibung von CoObRA2 sei auf [Sch07] verwiesen.

Ein MOD2-SKM-Server ist immer einem CoObRA2-Repositorium zugeordnet. Die Zuordnung erfolgt über die MOD2-SKM-Server-Kennung (siehe qualifizierte Assoziationen „stored in", „manages" und „is running"), da CoObRA2 nur ein Repositorium pro virtueller Java-Maschine unterstützt. Die Funktion der CoObRA2-Komponenten ist in [Sch07] beschrieben. Die Registrierung eines Objektes findet i.d.R. bei der Konstruktion statt. Da in MOD2-SKM häufig das Fabrik-Entwurfsmuster zum Einsatz kommt, erfolgt die Registrierung dort, und es existiert eine abstrakte Fabrik-Klasse (*CoobraPersistencyObjectFactory*), die alle notwendigen Methoden zur Registrierung bereitstellt. Alle Fabrik-Klassen des Server-Teilsystem müssen diese Klasse implementieren. Diese Beziehung entfällt durch Propagation der Merkmalsmarkierungen, sobald ein SKMS ohne CoObRA2-Persistenzmechanismus erstellt wird.

Im Vergleich zum Hibernate-Persistenzmechanismus, ist die Integration von CoObRA2 komplexer und stellt mehr Anforderungen an die Klassen des Domänenmodells: Jedes neue Modul muss CoObRA2-kompatibel gemacht werden, indem (1) die Fabrik-Klasse von *CoobraPersistencyObjectFactory* erbt, (2) persistenzierte Klassen entweder (a) den Stereotyp „Java Bean" erhalten, oder (b) die Schnittstelle *java.io.Serializable* implementieren. Weiterhin ist CoObRA2 nicht mit allen Arten der Deltaspeicherung kompatibel (vgl. Abschnitt 6.4.23, S. 283ff.) und bei Einsatz der CoObRA2-Transaktionen zeigt sich, dass die Schnittstelle zwar entsprechende Methoden besitzt, diese aber nicht implementiert sind.

6.4.12. Modul: persistency.hibernate

Überblick

persistency.hibernate integriert das Hibernate-Persistenzrahmenwerk [BHRS07] zur Persistenzierung des Server-Teilsystems. Das Modul kann prinzipiell auch den Arbeitsbereichs persistenzieren, wird aber lediglich auf dem Server eingesetzt (vgl. Abschnitt 6.4.27, S. 292ff.).

Abhängigkeiten

Abbildung 6.80 zeigt die Abhängigkeiten von *persistency.hibernate*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Hibernate ORM" (O.4.2) (Markierung: *relationalDatabase*) zur Konfiguration gehört. Das Modul implementiert die Schnittstelle des Persistenz-Kernmoduls (*core.persistency*) und ist ansonsten nur von seinen proprietären Komponenten abhängig.

Detailbeschreibung

Abbildung 6.81 zeigt das Klassendiagramm des Moduls *persistency.hibernate*. Die Klasse *HibernatePersistencyModule* implementiert die Schnittstelle aus *core.persistency* und

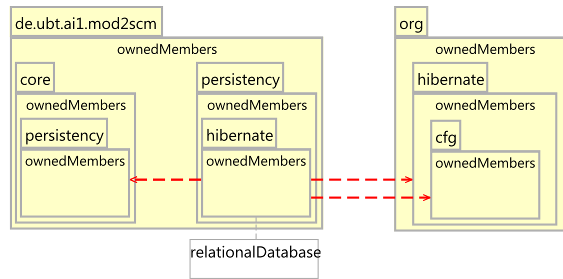


Abbildung 6.80.: Paketdiagramm: persistency.hibernate

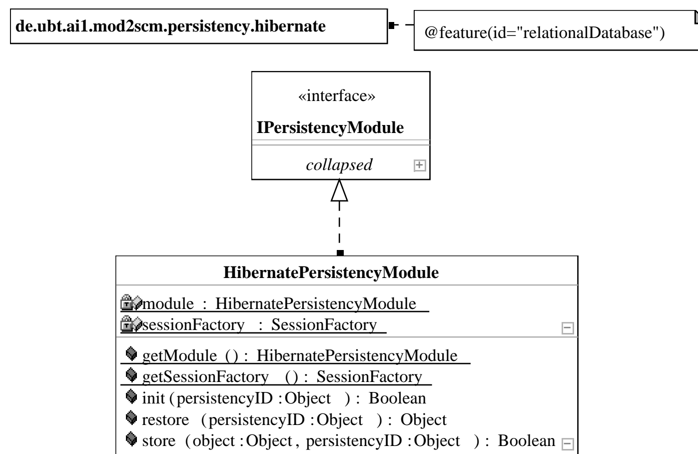


Abbildung 6.81.: Klassendiagramm: persistency.hibernate

modelliert Methoden zum Speichern und Laden des Server-Teilsystems (*store()* und *restore()*, vgl. Abschnitt 6.3.7, S. 186ff.) und zum Verwalten von Transaktionen (*beginTransaction()*, *cancelTransaction()* und *finishTransaction()*, vgl. Abschnitt 6.3.7, S. 186ff.). In ihren Story-Diagrammen delegieren sie die entsprechenden Aufrufe an die *SessionFactory* des Hibernate-Rahmenwerks weiter. Siehe [BHRS07] für eine detaillierte Beschreibung.

Alle Klassen, die mit Hibernate persistenziert werden sollen, müssen mit dem Stereotyp „hibernate“ gekennzeichnet werden. Damit wird ihr generierter Quellcode um Hibernate-Annotationen erweitert [Öhm10]. Diese Auszeichnung lässt sich nicht konfigurieren, da es nicht möglich ist, eine Merkmalsmarkierung an Stereotypen anzubringen. Auch das Umschalten der Code-Erzeugung auf die Hibernate-Erweiterung von Stefan Oehme [Öhm10] erfolgt manuell durch die Entwickler.

Bevor das Server-Teilsystem des generierten SKMS lauffähig ist, muss (1) ein Datenbank-Server gestartet und (2) ein Datenbank-Schema für das Server-Teilsystem erzeugt werden. Dieses lässt sich anhand der o.g. Hibernate-Annotationen im Quellcode generieren [BHRS07]. Für detailliertere Informationen sei auf [Öhm10] verwiesen. Eine Integration dieser Operationen in *persistency.hibernate*, zur Vereinfachung der Inbetriebnahme eines Hibernate-basierten SKMS, bleibt zukünftigen Arbeiten überlassen.

6.4.13. Modul: product.ecore

Die Artefakte der modellgetriebenen Software-Entwicklung sind Modelle und ihre Elemente – und nicht Dateien und ihre Zeilen. Trotzdem werden zu ihrer Versionierung Dateisystem-basierte SKMS eingesetzt. Daher stellt sich die Frage: „Welche Vorteile bieten auf Modelle spezialisierte SKMS?“ [AKK+08] zeigt bereits, dass ein Produktmodell für Ecore-Modelle des Eclipse Modelling Framework (EMF) [SBPM09] implementiert werden kann.

Da die MOD2-SKM Produktlinie speziell den Austausch des Produktmodells unterstützt (vgl. Abschnitt 6.3.8, S. 188ff.), untersuchte Udo Pöhner im Rahmen dieser Arbeit die Modellierung eines Ecore-Produktmodells für die MOD2-SKM-Modulbibliothek. Sein Ergebnis ist jedoch kein vollständiges Produktmodell, sondern evaluiert lediglich Integrations-Ansätze des Ecore-Metamodells in die MOD2-SKM-Produktlinie. Er wirft folgende Fragen auf, die vor einer endgültigen Modellierung geklärt werden sollten:

1. Sind Produktmodell-Instanzen (a) Instanzen des Ecore-Metamodells oder (b) Instanzen eines domänenspezifischen Modells, das mittels Ecore definiert wurde?
2. Erfolgt der Abgleich zwischen unversioniertem und versioniertem Produktmodell synchron oder asynchron?
3. Inwieweit lassen sich bestehende Ecore-Editoren bzw. generierte domänenspezifische Editoren mit SKMS-Operationen erweitern?

Für eine detaillierte Beschreibung sei auf [Pöh09] verwiesen. Auf Grund des bereits existierenden MOD2-SKM-Plugins für Eclipse (vgl. Abschnitt 7, S. 315ff.) und der Verbreitung und Stabilität von EMF [SBPM09], empfiehlt sich die Entwicklung eines Ecore-Produktmodells in zukünftigen Arbeiten.

6.4.14. Modul: product.filesystem

Überblick

Das Modul *product.filesystem* modelliert ein Dateisystem-Produktmodell, das synchron mit einem Teilbereich des Betriebssystem-Dateisystem gehalten wird. Dieses Modul ist das am ausführlichsten modellierte und getestete Produktmodell-Modul der Produktlinie und bietet auch mehrere Verfahren zur Delta-Berechnung von Dateien. Dieses Untermodul ist in der Bachelorarbeit von Stefan Matthaei ausführlich beschrieben [Mat09].

Abhängigkeiten

Abbildung 6.82 zeigt die Abhängigkeiten von *product.filesystem*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Dateisystem“ (B.2.2) (Markierung: *filesystem*) zur Konfiguration gehört. Als Produktmodell implementiert es die Schnittstellen des Kernmoduls *core.product* und gehört zum Arbeitsbereichs-Teilsystem (*core.workspace*). Die Überwachung des Betriebssystem-Dateisystems erfolgt mittels der *jnotify*-Bibliothek und kann mit Hilfe von *util.logging* protokolliert werden. Das Untermodul *product.filesystem.delta* modelliert Verfahren zur Delta-Berechnung und -Anwendung, und basiert auf dem Kernmodul *core.delta*. Einige dieser Verfahren verwenden proprietäre Algorithmen zur Delta-Berechnung (*com.google.code*).

Detailbeschreibung

Abbildung 6.82 zeigt das Paket *product.filesystem*, das ein Modell eines Dateisystem-Produktmodells enthält. Zusätzlich gibt es vier Unterpakete: *product.filesystem.events* enthält alle Ereignisse, die bei Überwachung des physikalischen Dateisystems auftreten. *product.filesystem.delta* modelliert Datenstrukturen zur Erstellung und Anwendung von Deltas von Dateien. Die dazu verwendeten Algorithmen stammen entweder aus proprietäre Bibliotheken oder dem Unterpaket *product.filesystem.diff*. Dieses enthält von Stefan Matthaei modellierte Algorithmen zur Delta-Berechnung. Anhand seiner Arbeit lässt sich auch die Modellierung von effizienten Algorithmen mittels Story-Diagrammen evaluieren [Mat09].

Modellierung des Dateisystem in *product.filesystem*

Abbildung 6.83 zeigt die Vererbungshierarchie der Klassen von *product.filesystem*. Jedes Element des Dateisystems (*FSElement*) ist auch Element des Produktmodells und implementiert die Schnittstelle *IProductModelItem* (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.). Dateisystem-Elemente sind entweder Verzeichnisse (*FSDirectory*) oder Dateien (*FSFile*). Erstere sind zusammengesetzte Produktmodell-Elemente und implementieren daher die Schnittstelle *IComplexProductModelItem*. Für *FSFile* existieren Delta-Algorithmen, die über die Schnittstelle *IDeltifiableItem* aufgerufen werden können (*core.delta*, vgl. Abschnitt 6.3.2, S. 175ff.). Die verwendeten Delta-Algorithmen sind jedoch

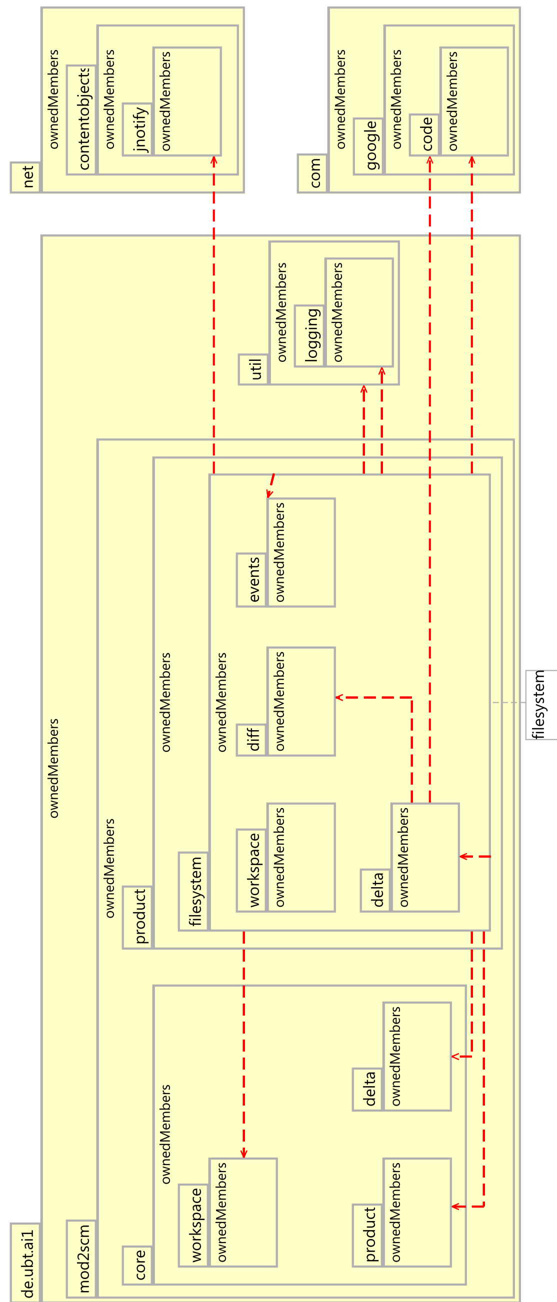


Abbildung 6.82.: Paketdiagramm: product.filesystem

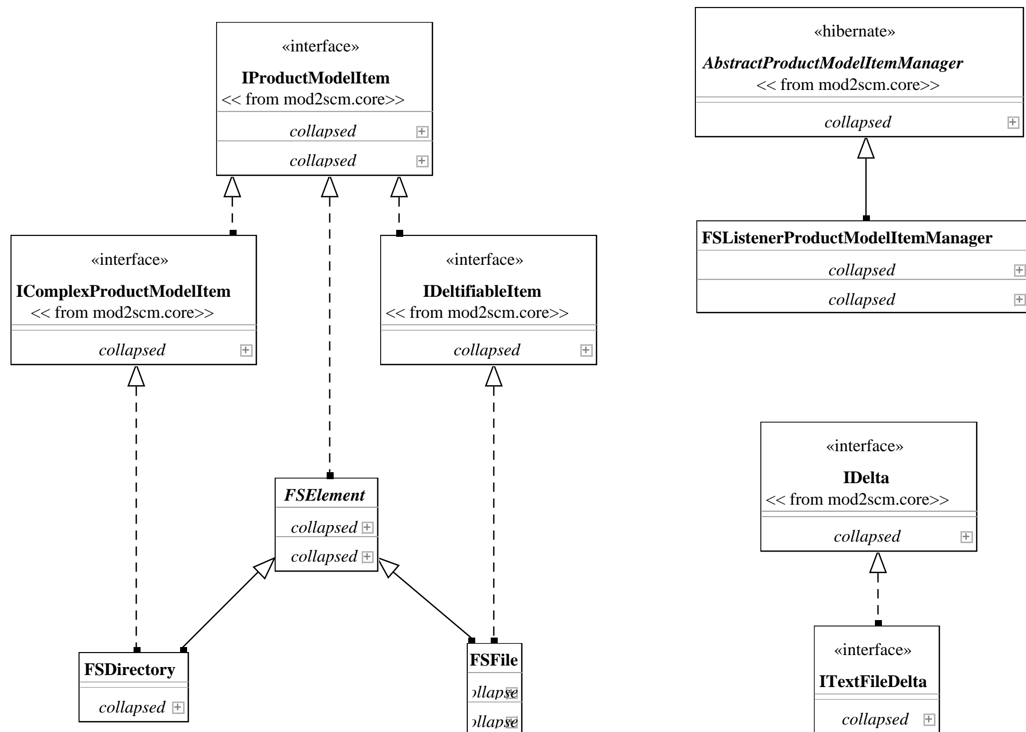


Abbildung 6.83.: Klassendiagramm: Vererbungshierarchie in product.filesystem

auf Textdateien beschränkt (*ITextFileDelta*). Und gemäß der Schnittstellen im Kernmodul *core.product*, existiert mit *FSListenerProductModelItemManager* eine Realisierung von *AbstractProductModelItemManager*.

Abbildung 6.84 zeigt die Klassen des Dateisystem-Produktmodells und ihre Beziehungen untereinander. Die Schnittstellen aus dem Produktmodell-Kernmodul *core.product* stellen sicher, dass über den Produktmodell-Manager (*FSListenerProductModelItemManager*) auf jedes Dateisystem-Element direkt über seine Objektkennung zugegriffen werden kann (qualifizierte Assoziation „has Items“), eine Beziehung zwischen Verzeichnissen und ihren enthaltenen Elementen besteht („contains“-Assoziation) und Methoden zum Zerlegen und Zusammensetzen der Verzeichnisse und ihres Inhalts existieren (*addToContainedItem()* und *removeFromContainedItem()* in *FSDirectory*) (vgl. Abschnitt 6.3.8, S. 188ff.).

Die Methoden der Dateisystem-Elemente (*FSElement*) sind typische Dateisystem-Operationen. So lässt sich ein Element kopieren (*copy()*), umbenennen (*rename()*), verschieben(*move()*) oder löschen (*delete()*). Einige dieser Methoden sind abstrakt, da sie für Dateien und Verzeichnisse unterschiedlich modelliert sind. Verzeichnisse (*FSDirectory*) bieten zusätzlich eine Operation zum Anlegen neuer Kindelemente (*createChild()*), während Dateien (*FSFile*) die durch das Delta-Kernmodul vorgegebene *compareTo()*-Methode implementieren und ein Textdatei-Delta (*ITextFileDelta*) zurückliefern. Daher wird der Inhalt einer Datei (*FSFile*) als Zeichenkette behandelt (Variable: *content*).

Abbildung 6.84.: Klassendiagramm: product.filesystem

Dateisystem-Deltas

Abbildung 6.85 zeigt die Klassen des Delta-Untermoduls *product.filesystem.delta*. Das Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Gerichtete Deltas“ (D.4.1.1) (Markierung: *directedDeltas*) zur Konfiguration gehört. Es stehen zwei Delta-Berechnungsverfahren zur Auswahl: Entweder die proprietäre *com.google.code*-Bibliothek oder die mit Fujaba modellierten *FSFileDeltas*. Letztere lassen sich noch mit dem Merkmal „Kompression“ (D.4.2) konfigurieren (Markierung: *compression* bzw. *nOTCompression*)[Mat09].

Für beide Verfahren existiert eine entsprechende Delta-Factory (*GoogleFileDeltaFactory* bzw. *FSFileDeltaFactory*). Diese implementiert die vom Delta-Kernmodul geforderte *IDeltaModule*-Schnittstelle mit einer Methode zum Erzeugen eines Deltas zwischen zwei Produktmodell-Elementen. Diese gibt ein Delta-Objekt zurück (*GoogleFileDelta* bzw. *FSFileDelta*), das die Schnittstelle *IDelta* implementiert und daher eine Methode zum Anwenden des Deltas besitzt (*apply()*) (vgl. Abschnitt 6.3.2, S. 175ff.). Beide Methoden kapseln die unterschiedliche Modellierung der jeweiligen Delta-Verfahren.

Synchronisation mit dem Betriebssystem-Dateisystem

Jede Instanz des Dateisystem-Produktmodells ist eine Abbildung eines Teils des Betriebssystem-Dateisystems. Der Dateisystem-Manager (*FSListenerProductModelItemManager*) referenziert dazu ein Verzeichnis als Wurzelverzeichnis („has root“-Assoziation in Abb. 6.84). Es kann immer nur genau ein Wurzelverzeichnis geben. Alle direkt und indirekt in ihm enthaltenen Dateisystem-Objekte (*FSElement*) bilden diesen Teilbaum des Betriebssystem-Dateisystems nach und verweisen auf ihre zugehörige Systemdatei (unidirektionale „has System File“-Assoziation).

Die Synchronisierung zwischen Betriebssystem-Dateisystem und Produktmodell-Instanz erfolgt bidirektional. Änderung am Dateisystem werden registriert und in die Produktmodell-Instanz übernommen, z.B. wenn die Entwickler Dateien im Arbeitsbereich editieren. Gleichzeitig werden Änderungen an der Produktmodell-Instanz in das Dateisystem übertragen, z.B. bei Aktualisierung des Arbeitsbereichs.

Betriebssystem-Dateisystem → Produktmodell-Instanz

Jede Instanz eines Dateisystem-Produktmodells überwacht den ihr zugeordneten Teilbaum des Betriebssystem-Dateisystems mittels der frei verfügbaren Bibliothek JNotify. Jede Instanz besitzt daher einen *FSJNotifyAdapter*, der bei Änderungen in einem überwachten Verzeichnis eine (asynchron abgesendete) Meldung empfängt. Insgesamt existieren vier verschiedene Meldungen, anhand derer eine vorgegebene Callback-Methode aufgerufen wird (s. Tab. 6.3).

Eine Callback-Methoden erzeugt ein Ereignis und fügt dies an das Ende der Ereignis-Warteschlange, *FSEventQueue*, des JNotify-Adapters (*FSJNotifyAdapter*) an. Für jede der vier Meldungen existiert ein eigener Ereignis-Typ (s. Tab. 6.3).

Abbildung 6.86 zeigt die vier Ereignis-Klassen. Jedes Ereignis speichert die Daten, die an seine Callback-Methode übergeben wurden (z.B. den Namen der betroffenen Da-

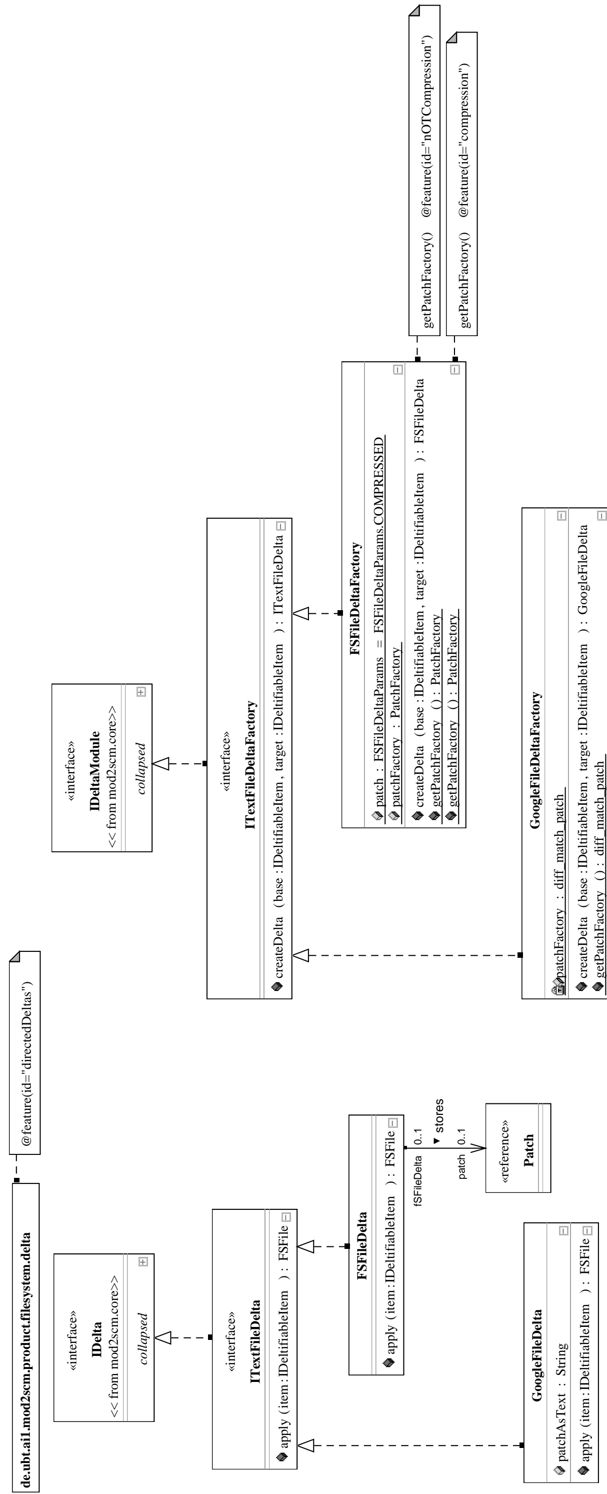


Abbildung 6.85.: Klassendiagramm: product.filesystem.delta

Meldung	Callback-Meth.	Ereignis-Typ	Manager-Methode
Element erzeugt	fileCreated()	FSCreateEvent	createFSElement()
Element umbenannt	fileRenamed()	FSRenameEvent	renameFSElement()
Element modifiziert	fileModified()	FSModifyEvent	modifyFSElement()
Element gelöscht	fileDeleted()	FSDeleteEvent	deleteFSElement()

Tabelle 6.3.: Meldungen und Ereignisse der Dateisystem-Überwachung

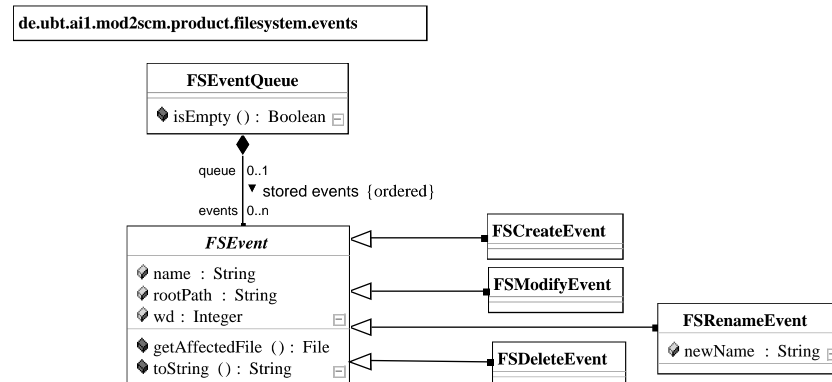


Abbildung 6.86.: Klassendiagramm: product.filesystem.events

tei). Für die Verarbeitung eines Ereignisses ist der Produktmodell-Manager (*FSListener-ProductModelItemManager*) zuständig. Seine *processEvent()*-Methode wertet den Typ und die Daten eines Ereignisses aus und passt die Produktmodell-Instanz an. Für jeden Ereignis-Typ existiert eine eigene Manager-Methode zur Umsetzung der Änderung (s. Tab. 6.3).

Eine Operation im Betriebssystem-Dateisystem kann bis zu drei Meldungen auslösen, z.B. erzeugt das Anlegen einer neuen Datei folgende drei Meldungen: (1) Datei erzeugt, (2) Datei modifiziert und (3) Elternverzeichnis modifiziert. Die Verarbeitung der Ereignisse erfolgt nicht automatisch, sondern nur nach Aufruf von *processNextEvent()* bzw. *processAllEvents()*, um das nächste bzw. alle Ereignisse abzuarbeiten. Damit ist es Aufgabe der SKMS-Client-Entwickler, den Modus für die Aktualisierung der Produktmodell-Instanz zu wählen, z.B. automatisiert über einen Zeitgeber oder auf Benutzeranfrage über eine Client-Operation.

Produktmodell-Instanz → Betriebssystem-Dateisystem

Eine Änderung an der Produktmodell-Instanz soll auch das Dateisystem verändern, d.h. jede Dateisystem-Operation der Produktmodell-Instanz besteht aus zwei Teiloperationen: Erstens der Änderung der Produktmodell-Instanz selbst und zweitens der Änderung am Betriebssystem-Dateisystem. Dabei lässt sich ausnutzen, dass eine Änderung am Dateisystem – über die Manager-Methoden aus Tabelle 6.3 – die Produktmodell-Instanz anpasst. Eine synchrone Änderung lässt sich mit ihrer Hilfe wie folgt erreichen:

Synchrone Operation	Dateisystem-Methode	Manager-Methode
createDirectory(), createFile()	createSystemFile()	createFSElement()
deleteElement()	deleteSystemFile()	deleteFSElement
renameElement()	renameSystemFile()	renameFSElement
setAndWriteContent()	writeContentToSystemFile()	modifyFSElement()

Tabelle 6.4.: Synchronisierte Methoden der Produktmodell-Instanz

1. Synchrone Operation aufrufen
2. Dateisystem-Methode aufrufen
3. Solange die Instanz nicht aktualisiert wurde:
   a) Nächstes Ereignis verarbeiten (*processNextEvent()*)
   b) Manager-Methode ausführen

Tabelle 6.4 zeigt die synchronen Operationen, die Dateisystem-Methode zur Veränderung des Dateisystems sowie die durch die Änderung ausgelöste Manager-Methode. Wichtigste Aufgabe der synchronen Operation ist es, abzuwarten, bis die Änderung am Betriebssystem-Dateisystem gemeldet und verarbeitet wurde. Die Methode befinden sich größtenteils in der Klasse *FSListenerProductModelItemManager*, nur die Methoden zum Schreiben des Dateiinhalts befinden sich in *FSFile*.

6.4.15. Modul: product.usecase

Überblick

Das Modul *product.usecase* modelliert ein Metamodell für UML-Anwendungsfall-Diagramme als MOD2-SKM-Produktmodell. Es existiert bisher keine Anbindung an einen Editor, da es als Studie zur Modellierung des Ecore-Produktmodells diente (vgl. Abschnitt 6.4.13, S. 239ff.).

Abhängigkeiten

Abbildung 6.82 zeigt die Abhängigkeiten von *product.usecase*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Anwendungsfall-Modell“ (B.2.1) (Markierung: *useCaseModel*) zur Konfiguration gehört. Als Produktmodell implementiert es die Schnittstellen des Kernmoduls *core.product* und gehört zum Arbeitsbereichs-Teilsystem (*core.workspace*). Änderungen an den Instanzen lassen sich mit Hilfe von *util.logging* protokollieren.

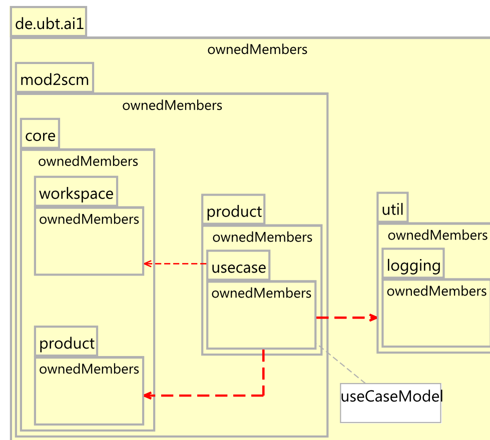


Abbildung 6.87.: Paketdiagramm: product.usecase

Detailbeschreibung

Abhängigkeiten

Abbildung 6.88 zeigt das Klassendiagramm von *product.usecase*. Ein Anwendungsfall-Diagramm (*UseCaseDiagram*) besteht aus unterschiedlichen Elementen („contains"-Komposition zu *Element*). Dabei kann es sich entweder um benannten Knoten (*NamedElement*) oder unbenannte Kanten (*Relationship*) handeln. Alle drei Klassen sind abstrakt, da es sich bei den Knoten letztendlich entweder um Aktoren (*Actor*) oder Anwendungsfälle (*UseCase*) handelt. Jede Kante ist entweder eine Erweitert-Beziehung (*ExtendsRelationship*), Enthält-Beziehung (*IncludesRelationship*) oder eine Assoziation (*Association*). Die ersten beiden Kantentypen sind gerichtete Beziehungen zwischen zwei Anwendungsfällen, was jeweils durch die beiden Assoziationspaare „to ... UC" modelliert wird. Der dritte Kantentyp ist eine gerichtete Beziehung zwischen Aktor und Anwendungsfall. Für eine detaillierte Beschreibung von Anwendungsfall-Diagrammen sei auf [HK99] verwiesen.

Anpassung an das Produktmodell-Kernmodul

Das Kernmodul für Produktmodelle *core.product* (vgl. Abschnitt 6.3.8, S. 188ff.) erwartet eine Klasse zur Verwaltung der Produktmodell-Instanzen (*AbstractProductModelItemManager*). Daher existiert in Abbildung 6.88 noch zusätzlich die Klasse *UseCaseItemManager*, welche lediglich Anwendungsfall-Diagramme verwaltet („manages"-Assoziation). Es muss also immer ein Diagramm existieren, in dem die anderen Elemente enthalten sind.

In *core.product* existieren Schnittstellen für atomare (*IProductModelItem*) und zusammengesetzte Elemente (*IComplexProductModelItem*). Indem nun die einzelnen Element-Klassen diese Schnittstellen implementieren, wird die Granularität des Produktmodells verfeinert, so dass sich einzelne Elemente oder Teildiagramme versionieren lassen. Abbildung 6.89 zeigt, welche Elemente als komplexe oder atomare Produktmodell-Elemente

Abbildung 6.88.: Klassendiagramm: product.usecase

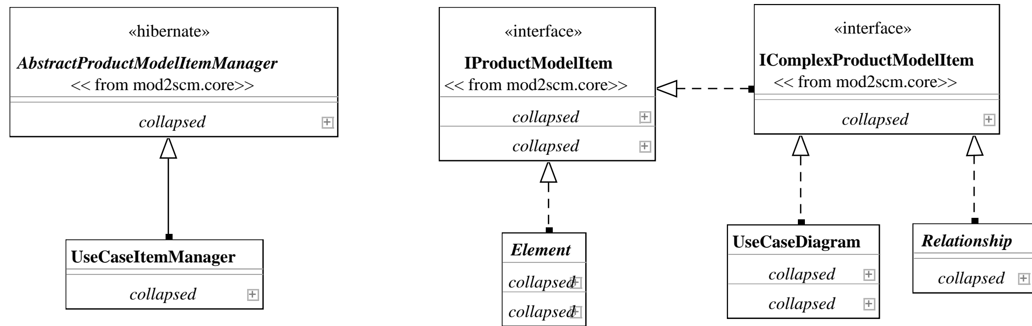


Abbildung 6.89.: Klassendiagramm: Vererbungshierarchie von product.usecase

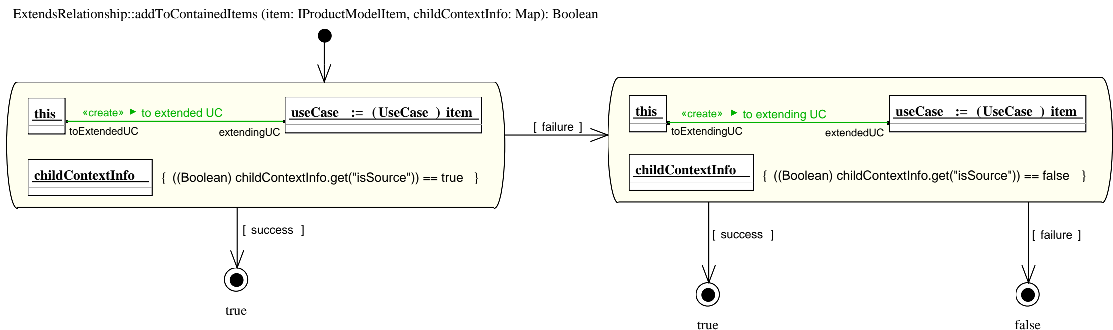


Abbildung 6.90.: Storydiagramm: Zusammensetzen einer Extends-Beziehung

modelliert sind: Das Diagramm (*UseCaseDiagram*) und alle Kantentypen (*Relationship*) sind zusammengesetzte Elemente. Die restlichen Elemente (*Element*), d.h. die Knoten (*NamedElement*), sind atomar.

Die Schnittstellen erwarten eine Methode zum Einfügen von Elementen (*insertItem()* in *UseCaseItemManager*) sowie zwei Methoden zum Zerlegen und Zusammenfügen zusammengesetzter Elemente (*addToContainedItems()*, *removeFromContainedItems()* in *UseCaseDiagram* und allen *Relationship*-Unterklassen) (vgl. Abschnitt 6.3.8, S. 188ff.). Während sich das Zerlegen leicht modellieren lässt, erweist sich das Zusammensetzen als problematisch. Das Kern-Produktmodell betrachtet die Kindelemente zusammengesetzter Elemente als ungeordnete Menge, so dass bei Beziehungen zwischen Anwendungsfällen die Information „verloren geht“, ob es sich bei einem Anwendungsfall um Start oder Ziel der Beziehung handelt. Bei der *Association*-Beziehung lassen sich Quelle und Ziel auf Grund des unterschiedlichen Typs unterscheiden.

Dieses Zuordnungsproblem ist bei allen Produktmodellen mit gerichteten Beziehungen zwischen Elementen des gleichen Typs zu erwarten, so dass sich beim Speichern zusammengesetzter Elemente zusätzliche Kind-Kontext-Informationen als Schlüssel-Wert-Paar angeben lassen (vgl. Abschnitt 6.4.22, S. 280ff.). Im Falle von *createChildContextInfo()* in *ExtendsRelationship* bzw. *IncludesRelationship* wird unter dem Schlüssel „isSource“ ein Wahrheitswert gespeichert. Dieser Wert kann dann beim Zusammensetzen ausgewertet

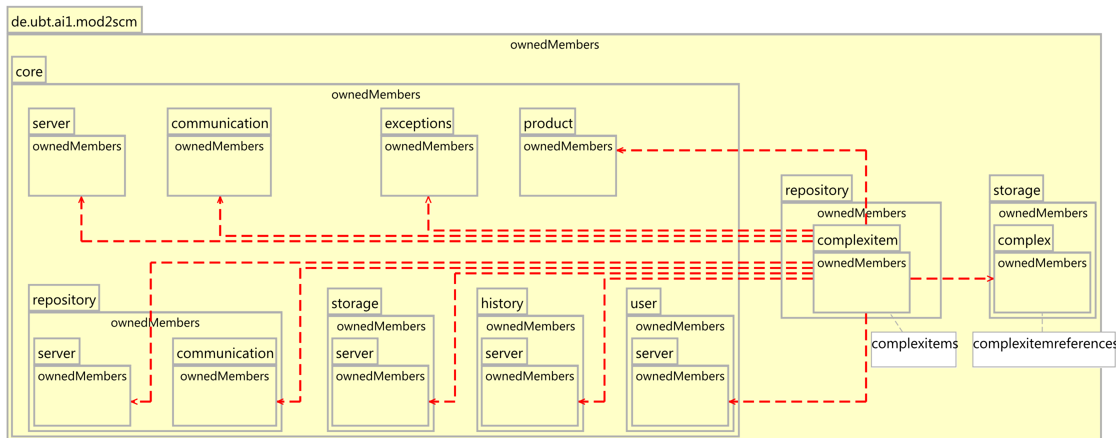


Abbildung 6.91.: Paketdiagramm: repository.complexitem

werden, so dass sich ein Anwendungsfall korrekt zuordnen lässt, wie das Storydiagramm Abbildung 6.90 zeigt: Abhängig vom Wahrheitswert, wird entweder die linke oder die rechte Storyaktivität ausgeführt, und so entweder die Beziehung zum Start- oder zum Zielknoten angelegt.

6.4.16. Modul: repository.complexitem

Überblick

Das Modul *repository.complexitem* modelliert ein Repositorium, das sowohl atomare als auch zusammengesetzte Produktmodell-Elemente verwalten kann (vgl. Abschnitt 4.2.3, S. 76ff.). Dadurch ist die versionszentrierte bzw. verwobene Integration von Produkt- und Versionsraum möglich (vgl. Abschnitt 4.3.4, S. 81ff.). Das Modul bildet jedoch nur die Voraussetzung für beide Integrationsarten. Das Verhalten selbst ist im zugehörigen Speichermodul *storage.complex* modelliert (vgl. Abschnitt 6.4.22, S. 280ff.).

Abhängigkeiten

Abbildung 6.82 zeigt die Abhängigkeiten von *repository.complexitem*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Komplexe Elemente“ (D.5.2) (Markierung: *complexitems*) zur Konfiguration gehört. Als Repositorium implementiert es die Schnittstellen des Kernmoduls *core.repository* und gehört zum Server-Teilsystem (*core.server*). Es lässt sich mit jedem Produktmodell verwenden (*core.product*) und verarbeitet die durch das Kommunikations-Modul (*core.communication*) übertragenen Produktmodell-Listen (*PMIList*, vgl. Abschnitt 6.3.1, S. 173ff.). Laufzeitfehler werden über Ausnahmen (*core.exceptions*) mitgeteilt. *repository.complex* kann mit beliebigen Benutzerverwaltungen (*core.user*) und Historien (*core.history*) kombiniert werden, ist jedoch beim Speicher (*core.storage*) auf den Speicher für zusammensetzte Elemente beschränkt (*storage.complex*). Die aussagenlogischen Einschränkungen im Merkmalsmo-

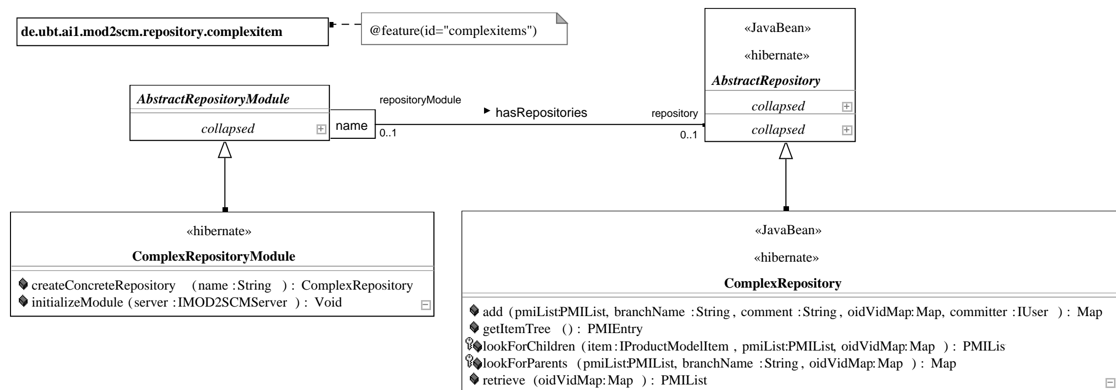


Abbildung 6.92.: Klassendiagramm: repository.complexitem

dell stellen daher sicher, dass gleichzeitig das Merkmal „Referenzen komplexer Elemente“ (D.4.3) (Markierung: *complexitemreferences*) ausgewählt ist (vgl. Abschnitt 5.6.2, S. 137ff.).

Detailbeschreibung

Abbildung 6.92 zeigt das Klassendiagramm von *repository.complex*. Das *ComplexRepositoryModule* implementiert die Fabrik-Methode zum Anlegen eines neues Repositoriums (*createConcreteRepository()*), wie vom Kernmodul verlangt (vgl. Abschnitt 6.3.9, S. 191ff.). In ein Repositorium (*ComplexRepository*) lassen sich neue Elemente einfügen (*add()*), indem eine Liste von Produktmodell-Elemente (Parameter: *pmiList*) sowie eine Liste von (oid, vid)-Tupeln (Parameter: *oidVidMap*) übergeben wird. Diese Liste benennt die Vorgänger-Version jedes übertragenen Produktmodell-Elements. Beim Auslesen (*retrieve()*) beschreiben die Tupel dagegen die angeforderten Elemente, die als Liste (*PMIList*) zurückgeliefert werden. Die Methode *getItemTree()* liefert den aktuellen Stand des Repositoriums als Baum aus Objektkennung zurück. So kann das Repositorium in der Server-Ansicht des Eclipse-Clients dargestellt werden (vgl. Abschnitt 7.3, S. 322ff.).

Die Historien-, Speicher-, und Arbeitsbereich-Module erwarten, dass Element- und Tupel-Liste explizit alle betroffenen Elemente enthalten. Das Repositorium stellt daher sicher, dass die beiden Listen auch explizit alle Elemente aufzählen. Beim Hinzufügen (*add()*) wird daher geprüft, ob auch alle Elternelemente der übertragenen Produktmodell-Elemente enthalten sind. Wenn nicht, werden sie den Listen hinzugefügt (*lookForParents()*). Beim Auslesen (*retrieve()*) wird dagegen geprüft, ob auch alle Kindelemente explizit angefordert wurden, ansonsten werden sie ebenfalls in die Listen eingetragen (*lookForChildren()*). Insbesondere die Methode *lookForParents()* ermöglicht die Verwendung von Teilbäumen des Produktmodells in einem Arbeitsbereich, da sie die nicht im Arbeitsbereich vorhandenen Elternelemente ergänzt.

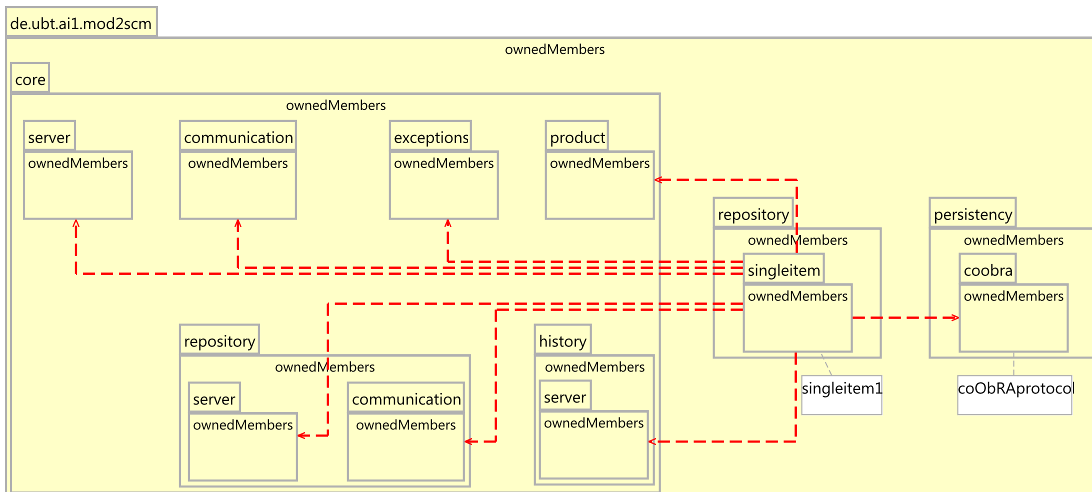


Abbildung 6.93.: Paketdiagramm: repository.singleitem

6.4.17. Modul: repository.singleitem

Überblick

Das Modul *repository.singleitem* modelliert ein Repositorium, das lediglich für atomare Produktmodell-Elemente verwendet werden kann (vgl. Abschnitt 4.2.3, S. 76ff.). Somit ist es auch auf eine produktzentrierte Integration von Produkt- und Versionsraum beschränkt (vgl. Abschnitt 4.3.4, S. 81ff.).

Abhängigkeiten

Abbildung 6.82 zeigt die Abhängigkeiten von *repository.singleitem*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Atomare Elemente“ (D.5.1) (Markierung: *singleitem1*) zur Konfiguration gehört. Als Repositorium implementiert es die Schnittstellen des Kernmoduls *core.repository* und gehört zum Server-Teilsystem (*core.server*). Es lässt sich mit jedem Produktmodell verwenden (*core.product*) und verarbeitet die durch das Kommunikations-Modul (*core.communication*) übertragenen Produktmodell-Listen (*PMIList*, vgl. Abschnitt 6.3.1, S. 173ff.). Es kann mit beliebigen Benutzerverwaltungen (*core.user*) und Historien (*core.history*) kombiniert werden. Laufzeitfehler werden über Ausnahmen (*core.exceptions*) mitgeteilt. In der Modulbibliothek existiert eine Abhängigkeit zum CoObRA2-Persistenzmodul (*persistency.coobra*, vgl. Abschnitt 6.4.11, S. 235ff.) – falls dieser Mechanismus über das Merkmal „CoObRA Protokoll“ (O.4.1) (Markierung: *coObRAprotocol*) ausgewählt ist.

Detailbeschreibung

Abbildung 6.94 zeigt das Klassendiagramm von *repository.singleitem*. Das *AtomicRepositoryModule* implementiert die Fabrik-Methode zum Anlegen eines neues Repositori-

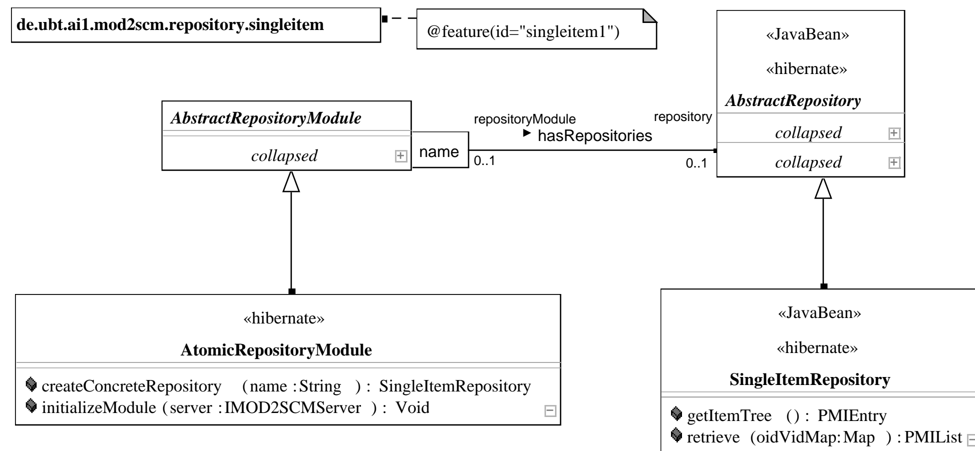


Abbildung 6.94.: Klassendiagramm: repository.singleitem

ums (*createConcreteRepository()*), wie vom Kernmodul verlangt (vgl. Abschnitt 6.3.9, S. 191ff.). In ein Repositorium (*SingleItemRepository*) lassen sich neue Elemente einfügen (*add()* aus *AbstractRepository*, vgl. Abschnitt 6.3.9, S. 191ff.), indem eine Liste von Produktmodell-Elemente (Parameter: *pmiList*), sowie eine Liste von (oid, vid)-Tupeln (Parameter: *oidVidMap*) übergeben wird. Diese Liste benennt die Vorgänger-Version jedes übertragenen Produktmodell-Elements. Beim Auslesen (*retrieve()*) beschreiben die Tupel dagegen die angeforderten Elemente, die als Liste (*PMIList*) zurückgeliefert werden. Die Methode *getItemTree()* liefert den aktuellen Stand des Repositoriums als Baum aus Objektkennung zurück. So kann das Repositorium in der Server-Ansicht des Eclipse-Clients dargestellt werden (vgl. Abschnitt 7.3, S. 322ff.).

6.4.18. Modul: server.base

Überblick

server.base modelliert einen MOD2-SKM Repositoriums-Server. Es ist damit die zentrale Komponente des Server-Teilsystems, welche die Kommandos aus dem Kommunikation-Teilsystem empfängt und an die entsprechenden Module delegiert. Der Server modelliert außerdem die Methoden zum Einspielen und Auslesen von versionierten Produktmodell-Elementen, d.h. das Commit- und das Checkout-Verhalten (vgl. Abschnitt 4.4.1, S. 91ff.).

Abhängigkeiten

Abbildung 6.82 zeigt die Abhängigkeiten von *server.base*. Für die Commit- und Checkout-Operationen steuert es auch Komponenten zum Kommunikations- (*server.base.communication*) und Arbeitsbereich-Teilsystem (*server.base.workspace*) bei. Daher ist der Server auch von allen drei Teilsystemen (*core.server*, *core.communication*, *core.workspace*) abhängig. Damit die Server-Module der Modulbibliothek austauschbar bleiben, verwendet der Server nur die Schnittstellen ihrer Kernmodule. *server.base* benötigt ein Repo-

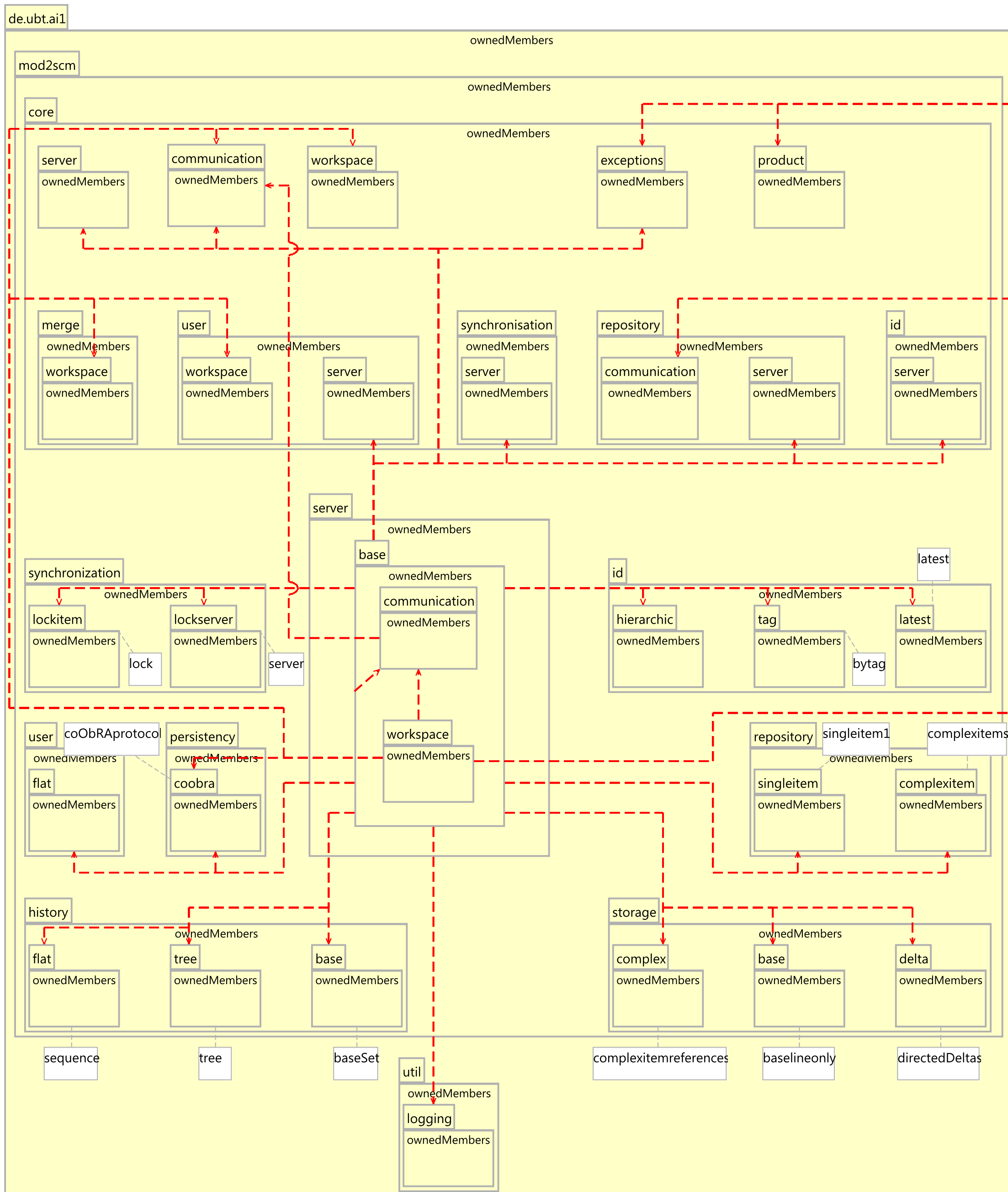


Abbildung 6.95.: Paketdiagramm: server.base

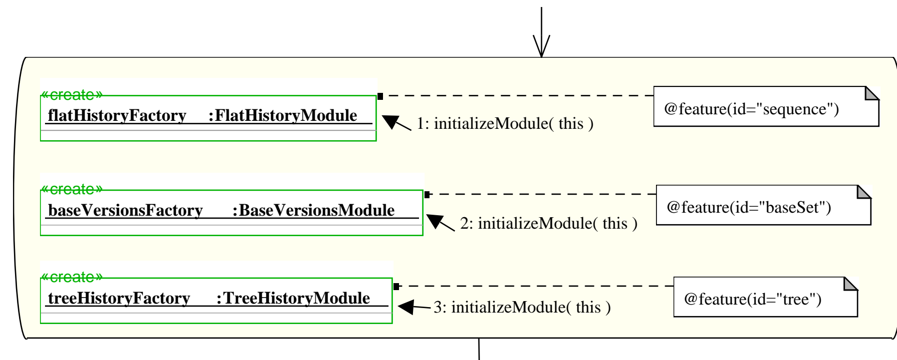


Abbildung 6.96.: Storydiagramm: Ausschnitt aus initServer()

sitorium (*core.repository*), eine Historie (*core.history*) und einen Speichermechanismus (*core.storage*), sowie ein Modul für die Benutzerverwaltung, zur Erzeugung von Versionskennungen (*core.id*), für Informationen bei Verschmelzung-Operationen (*core.merge*) und für die interne Synchronisation (*core.synchonisation*). Die Operationen zum Einspielen und Auslesen sind vom konkreten Produktmodell unabhängig (*core.product*) und behandeln Laufzeitfehler mit Ausnahmen (*core.exceptions*).

Aufgabe des Server ist es, die konkreten Fabrik-Module zu initialisieren. Die Komponenten eines Moduls, wie z.B. die Historie oder der Speicher, lassen sich mit Hilfe des Fabrik-Entwurfsmusters unabhängig vom konkreten Typ erzeugen [FF04]. Die Fabrik selbst muss jedoch zunächst instanziert werden, so dass der Server eine Abhängig zu jedem konkreten Modul besitzt. Die entsprechenden Abhängigkeiten sind in Abbildung 6.95 erkennbar. Sie zeigen alle auf Pakete, die mit Merkmalsmarkierungen versehen sind. Die Methode zur Instanzierung der Fabriken ist so modelliert, dass sie alle Fabriken dieser Module erzeugt (d.h. aller Historien, aller Repositorien, usw.). Das Merkmalsmodell stellt jedoch sicher, dass eine Konfiguration immer nur genau einen Typ Historie, eine Art Repositorium, etc. enthält (vgl. Abschnitt 5.3, S. 104ff.). Um diese Abhängigkeit explizit zu zeigen, ist jede Instanzierung einer Fabrik mit dem entsprechenden Merkmal seines Paketes markiert (s. Abb. 6.96), obwohl die Paket-Markierung bis zu den Objekten im Story-Diagramm propagiert wird [BD09a, Buc10]. Letztendlich werden nur die Fabriken instanziert, die auch zur Konfiguration gehören. Der Server eines konfigurierten SKMS besitzt also nur Abhängigkeiten zu den Modulen, die er auch verwendet.

server.base ist mit keinem Merkmal markiert und damit Teil jedes konfigurierten SKMS. Dennoch ist es ein Modul der Modulbibliothek. Zum einen wegen der Abhängigkeiten zu den anderen Modulen der Bibliothek und zum anderen, weil *server.base* nur eines von vielen möglichen Server-Modellen darstellt. Es sind sowohl Server mit anderen Kombinationen von Kernmodulen als auch unterschiedlichem Commit- und Checkout-Verhalten möglich.

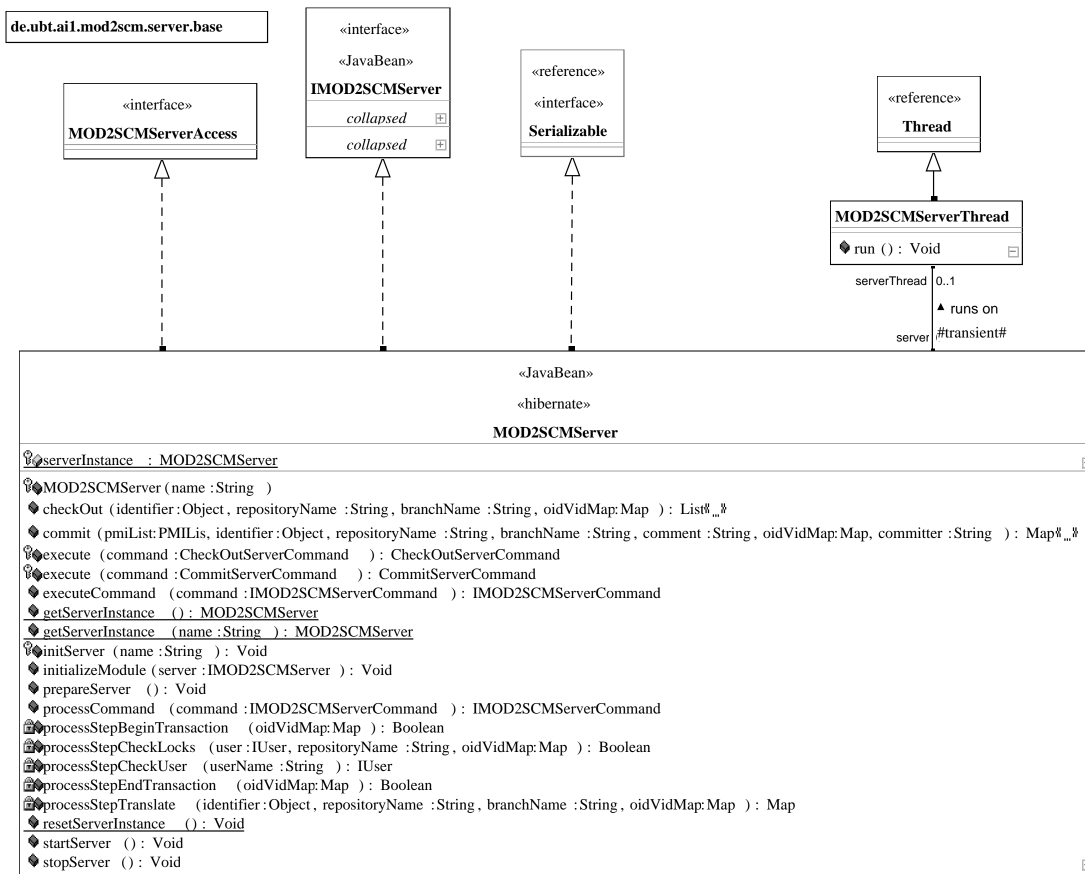


Abbildung 6.97.: Klassendiagramm: server.base

Detailbeschreibung

server.base steuert Komponenten zu allen drei Teilsystemen (Server, Kommunikation und Arbeitsbereich, vgl. Abschnitt 6.2.3, S. 161ff.) bei. Das Paket *server.base* selbst enthält die Klassen des Server-Teilsystems und seine Unterpakete, *server.base.communication* und *server.base.workspace*, die Klassen der anderen beiden.

Server-Teilsystem: *server.base*

Abbildung 6.97 zeigt das Klassendiagramm des Server-Teilsystems von *server.base*. Die Hauptkomponente ist ein modularer SKM-Server (*MOD2SCMServer*). Er implementiert sowohl die Schnittstelle des Kernmoduls für modulare SKM-Server (*IMOD2SCMServer* aus *core.server*, vgl. Abschnitt 6.3.10, S. 195ff.) als auch die Schnittstelle des Kernmoduls des Kommunikations-Teilsystems (*IMOD2SCMServerAccess* aus Modul *core.communication*, vgl. Abschnitt 6.3.1, S. 173ff.). Es gibt in jeder Laufzeitumgebung nur eine Server-Instanz, so dass MOD2SCMServer dem Singleton-Entwurfsmuster entspricht (statische Methode: *getServerInstance()*) [FF04].

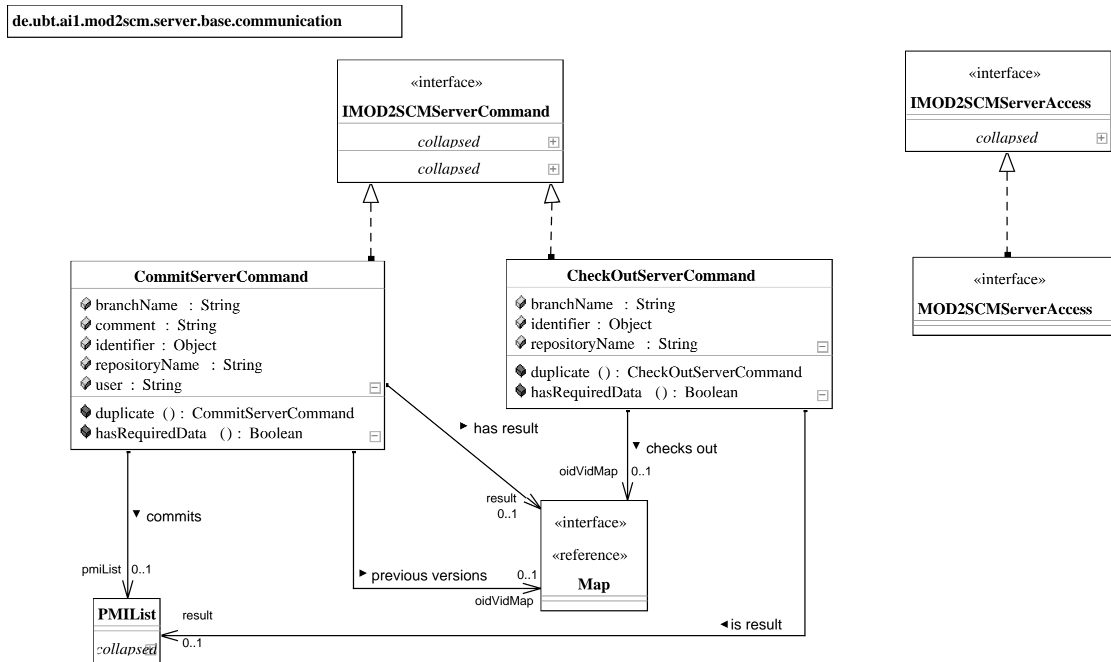


Abbildung 6.98.: Klassendiagramm: server.base.communication

Nach dem Instanzieren genügt ein Aufruf der Methode *startServer()*, um den Server zu starten und um ihn später wieder mit *stopServer()* anzuhalten. Beim Startvorgang wird u.a. überprüft, ob bereits ein persistenzierte Server existiert. Denn nur beim allerersten Start wird die o.g. Methode *initServer()* aufgerufen, um die Modul-Fabriken zu instanzieren. Ansonsten wird der bereits persistenzierte Server geladen (s. *core.persistency*, vgl. Abschnitt 6.3.7, S. 186ff.). Nach dem Start wartet jeder Server auf eingehende Kommandos des Kommunikations-Teilsystems. Um das Betriebssystem nicht unnötig zu belasten, läuft er in seinem eigenen Thread (*MOD2SCMServerThread*) [Ull09].

Kommunikations-Teilsystem: *server.base.communication*

Abbildung 6.98 zeigt die beiden Kommandos, die der *MOD2SCMServer* verarbeitet: Das Kommando zum Einspielen (engl. commit) (*CommitServerCommand*) und Auslesen (engl. checkout) (*CheckOutServerCommand*). Das Commit-Kommando überträgt eine Liste von Produktmodell-Elementen („commits"-Assoziation auf *PMIList*) und die Versionskennung ihrer Vorgängerversionen („previous versions"-Assoziation auf *Map*, d.h. als Liste von (oid, vid)-Tupeln) an den Server. Zusätzlich enthält es noch Informationen über das Ziel-Repositorium (Parameter: *repositoryName*), den verantwortlichen Benutzer (Parameter: *user*) und seinen Kommentar (Parameter: *comment*). Je nach Server-Konfiguration können auch ein Zweigname (Parameter: *branchName*) oder ein erweitertes Identifikationsmerkmal (Parameter: *identifier*), z.B. eine Markierung (vgl. Abschnitt 6.4.9, S. 227ff.) oder die Anfrage nach der letzten Version (vgl. Abschnitt 6.4.6, S. 224ff.)

vorliegen. Als Ergebnis liefert das Commit-Kommando eine Liste von (*oid*, *vid*)-Tupeln zurück, die jeder Objektkennung die neu zugeordnete Versionskennung zuweist.

Das Checkout-Kommando (*CheckOutServerCommand*) übermittelt eine Liste von (*oid*, *vid*)-Tupeln, die aus dem Repositorium ausgelesen werden sollen. Dazu ist auch die Angabe des Ziel-Repositoriums nötig (Parameter: *repositoryName*). Wie beim Commit-Kommando können hier, je nach Konfiguration, noch Zweignamen (Parameter: *branchName*) oder erweiterte Identifikationsmerkmale (Parameter: *identifier*) übergeben werden.

Commit- und CheckOut-Verhalten des Servers

Ein eingehendes Commit-Kommando führt letztendlich zur Ausführung der Methode *commit()* (*MOD2SCMServer* in Abb. 6.97). Die Parameter der Methode entsprechen dabei den Attribut-Werten des Kommandos (*CommitServerCommand* in Abb. 6.98). Abbildung 6.99 zeigt das Storydiagramm des Commit-Vorgangs. Die durchnummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Der Aufruf der Methode wird protokolliert.

2. Die Methode *processStepCheckUser()* prüft die Existenz und Berechtigungen des Benutzers anhand des Namens (Parameter: committer).

3. Schlägt die Prüfung fehl, wird eine Ausnahme geworfen und der Vorgang abgebrochen.

4. Ist der Benutzer berechtigt, prüft die Methode *processStepCheckLocks()* die pessimistischen Sperren für den Benutzer und die einzuspielenden Elemente (Parameter: *oidVidMap*). Außerdem beginnt sie eine Transaktion durch Aufruf der Methode *processStepBeginTransaction()*.

5. Hat ein anderer Benutzer ein einzuspielendes Element gesperrt (Lokale Variable: *noLocks*) oder lässt sich die Transaktion nicht starten (Lokale Variable: *blockSuccessful*), wird die Transaktion abgebrochen (*cancelTransaction()*) und eine Ausnahme geworfen.

6. Die Methode *processStepTranslate()* übergibt die Liste der einzuspielenden (*oid*, *vid*)-Tupel und den Parameter für erweiterte Identifikation (Parameter: *identifier*) an die zusätzlichen Identifikations-Module (basierend auf der Schnittstelle *IDTranslator* aus *core.id*, vgl. Abschnitt 6.3.5, S. 182ff.). Hier wird z.B. eine Markierung (engl. tag) durch *id.tag* (vgl. Abschnitt 6.4.6, S. 224ff.) und die zuletzt eingespielte Version durch *id.latest* (vgl. Abschnitt 6.4.6, S. 224ff.) aufgelöst.

7. Das Repositoriums-Modul wird über seine Kernmodul-Schnittstelle angesprochen (vgl. Abschnitt 6.3.9, S. 191ff.).

8. Die einzuspielenden Elemente (Parameter: *pmiList*) und die, evtl. veränderte, Liste der (*oid*, *vid*)-Tupel (Parameter: *oidVidMap*) werden eingespielt. Das Repositorien-Modul delegiert den Aufruf an das entsprechende Repositorium (Parameter: *repo-*

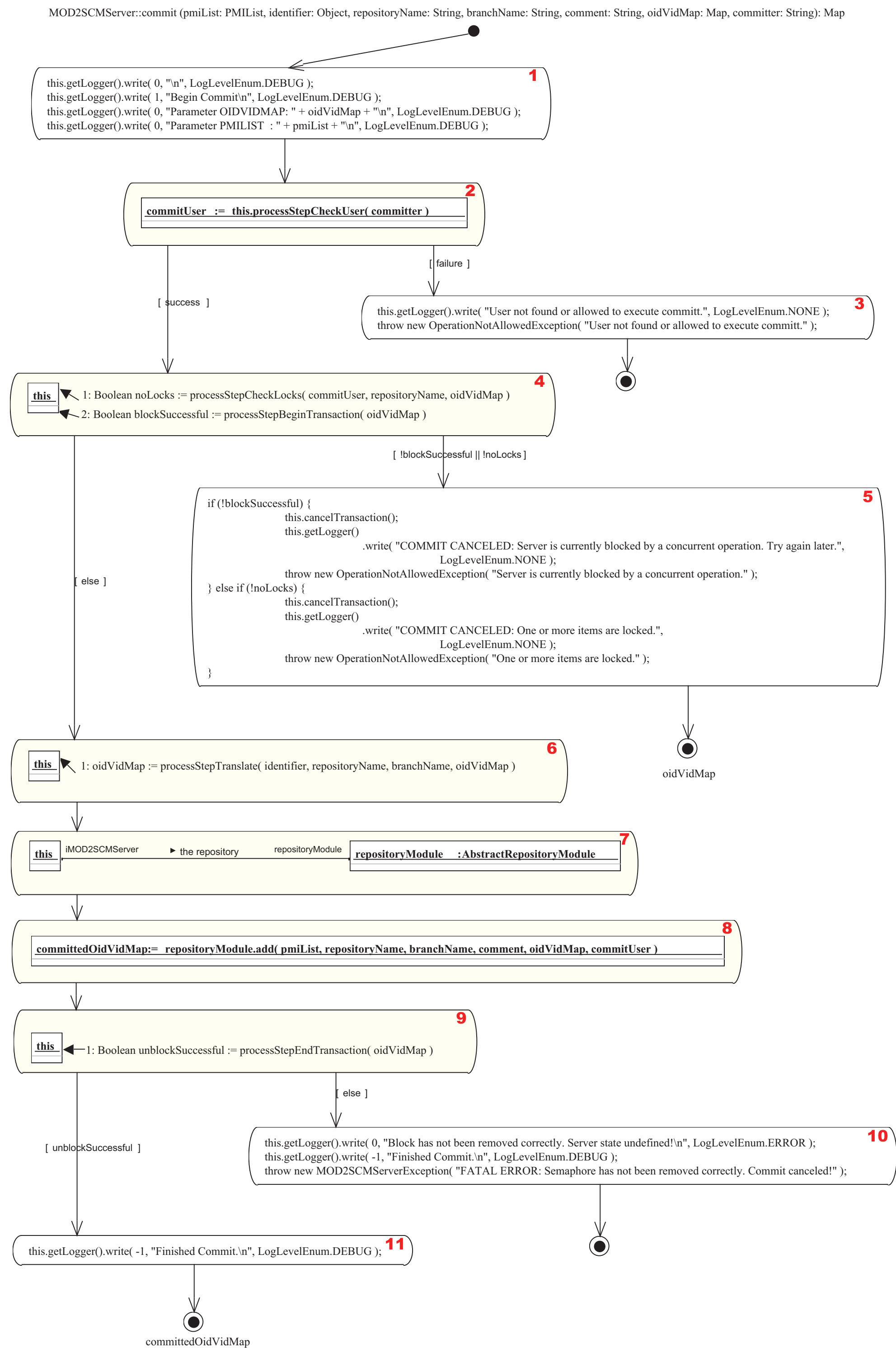


Abbildung 6.99.: Storydiagramm: commit() von MOD2SCM-Server

sitoryName). Repositorium (*core.repository*, vgl. Abschnitt 6.3.9, S. 191ff.), Historie (*core.history*, vgl. Abschnitt 6.3.4, S. 178ff.) und Speicher (*core.storage*, vgl. Abschnitt 6.3.11, S. 200ff.) werden dabei nur über ihre Kernmodul-Schnittstelle angesprochen. Das Ergebnis ist eine Liste der (*oid*, *vid*)-Tupel – mit den neu zugewiesenen Versionskennungen (Lokales Objekt: *committedOidVidMap*).

9. Abschließen wird über *processStepEndTransaction()* die Transaktion beendet.

10. Schlägt dies fehl, wird die Transaktion abgebrochen, was beim CoObRA2-Persistenzmechanismus leider keine Auswirkung hat (*coobra.persistency*, vgl. Abschnitt 6.4.11, S. 235ff.), und den Server in einen undefinierten Zustand versetzt. Die vollständige Integration des Hibernate-Rahmenwerks ist unbedingt erforderlich (*persistency.hibernate*, vgl. Abschnitt 6.4.12, S. 237ff.).

11. Eine erfolgreiche Transaktion wird protokolliert und die Liste der (*oid*, *vid*)-Tupel – mit den neu zugewiesenen Versionskennungen – zurückgegeben (Lokales Objekt: *committedOidVidMap*).

Analog startet ein eingehendes Checkout-Kommando die Ausführung der Methode *checkOut()* (*MOD2SCMServer* in Abb. 6.97). Jegliche Anfrage nach versionierten Elementen des Repositoriums-Servers wird als Checkout behandelt, d.h. auch die Anfragen im Rahmen einer Arbeitsbereich-Aktualisierung. Die Unterscheidung zwischen Update und (initialem) Checkout existiert nur im Arbeitsbereich-Teilsystem (s.u.). Die Parameter der Methode *checkOut()* entsprechen dabei den Attribut-Werten des Kommandos (*CheckOutServerCommand* in Abb. 6.98). Abbildung 6.100 zeigt das Storydiagramm des Checkout-Vorgangs. Die durchnummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Der Aufruf der Methode wird protokolliert.

2. Durch Aufruf von *processStepBeginTransaction()* wird – wie in Aktivität vier von *commit()* – versucht, eine Transaktion zu starten.

3. Ist dies nicht möglich, wird eine Ausnahme geworfen.

4. Nach erfolgreichem Start erfolgt – analog zur sechsten Aktivität in *commit()* – eine Auswertung des Parameters für zusätzliche Identifikations-Module (Parameter: *identifier*) (vgl. Abschnitt 6.3.5, S. 182ff.).

5. Wie im siebten Schritt von commit() wird das Repositoriums-Modul über seine Kernmodul-Schnittstelle angesprochen (vgl. Abschnitt 6.3.9, S. 191ff.).

6. Die , evtl. verändert, Liste der angeforderten (*oid*, *vid*)-Tupel (Parameter: *oidVidMap*) und das zuständige Repositorium (Parameter: *repositoryName*) werden vom Repositorien-Modul verarbeitet. Dabei werden die Parameter über die Kernmodul-Schnittstellen an Repositorium (*core.repository*, vgl. Abschnitt 6.3.9, S. 191ff.), Historie (*core.history*, vgl. Abschnitt 6.3.4, S. 178ff.) und Speicher (*core.storage*, vgl.

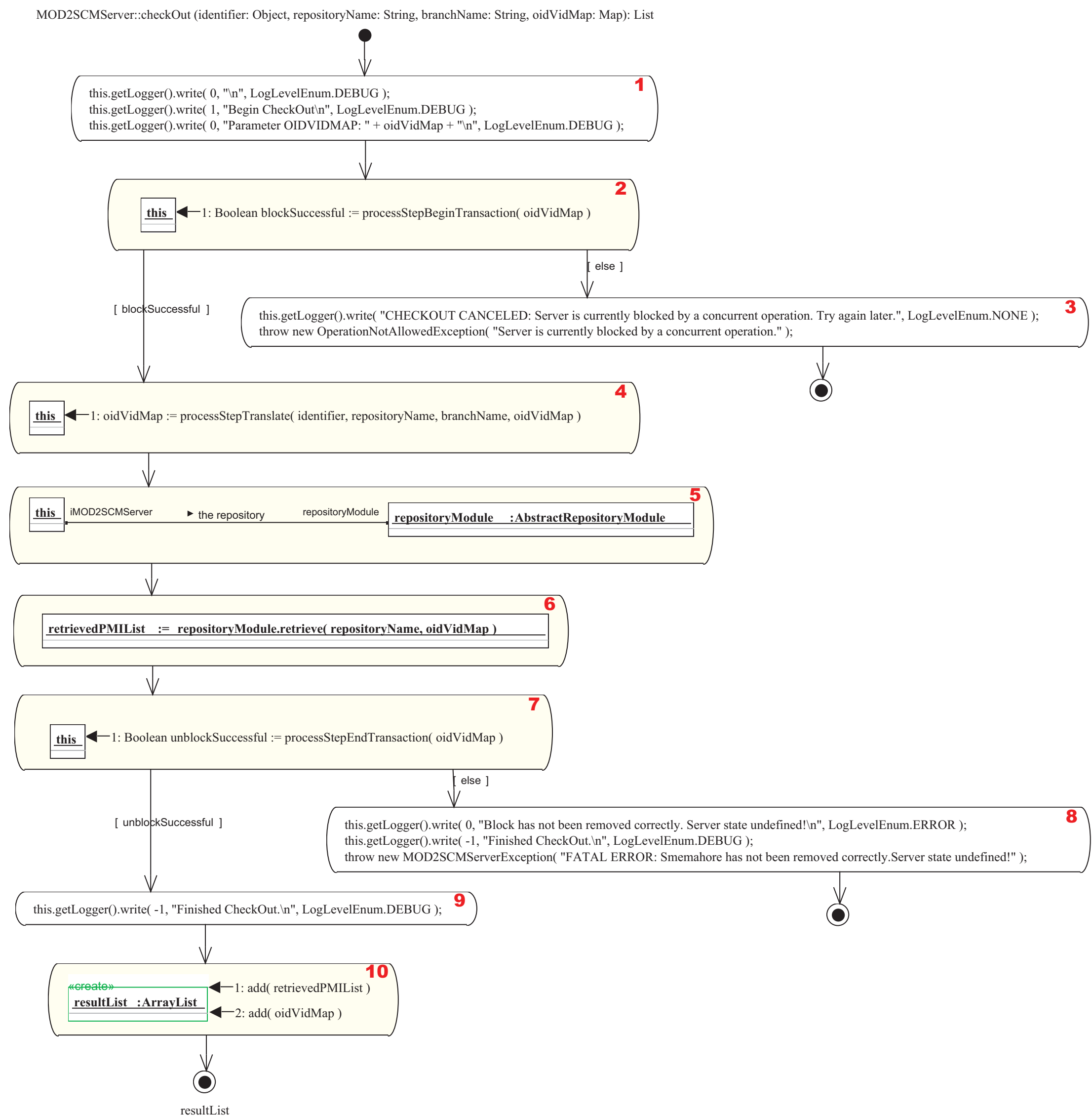


Abbildung 6.100.: Storydiagramm: checkout() von MOD2SCM-Server

Abschnitt 6.3.11, S. 200ff.) delegiert. Rückgabewert ist eine Liste von Produktmodell-Elementen (Lokales Objekt: *retrievedPMIList*). Im Falle von zusammengesetzten Elementen sind auch die Beziehungen unter ihnen wiederhergestellt und evtl. vorhandene Kind-Elemente in die List mit aufgenommen worden. In letzterem Fall wurde auch die Liste der (oid, vid)-Tupel erweitert (Parameter: *oidVidMap*).

7. Analog zum neunten Schritt in *commit()* wird nun die Transaktion beendet.

8. Auch hier führt ein Scheitern zu einem Abbruch der gesamten Transaktion – und bei CoObRA2 zu einem undefinierten Server-Zustand.

9. Ein erfolgreicher Abschluss der Transaktion wird protokolliert.

10. Anschließend die Liste der angeforderten Produktmodell-Elemente (Parameter: *retrievedPMIList*) sowie die Liste der angeforderten (oid, vid)-Tupel (Parameter: *oidVidMap*) zurückgeben. Letzteres ist auf Grund der Schritte vier und sechs nötig, da beide diese Liste verändern bzw. erweitern können.

Arbeitsbereich-Teilsystem: *server.base.workspace*

Abbildung 6.101 zeigt das Klassendiagramm des Arbeitsbereichs-Teilsystems. Es besteht aus zwei Komponenten. Erste ist die Klasse *LocalServerWorkspaceModule*, die ein Protokoll für die Übertragung der Kommando-Objekte modelliert. Dieses kann die Kommandos nur an einen MOD2-SKM-Server in der gleichen Laufzeitumgebung übertragen. Das Modul *server.rmi* modelliert ein leistungsfähigeres Protokoll, das Daten auch über Systemgrenzen hinweg überträgt (vgl. Abschnitt 6.4.19, S. 271ff.). *LocalServerWorkspaceModule* kann jedoch im Kontext von Peer-to-Peer-SKMS zum Einsatz kommen, da hier ein lokales Repositorium verwendet wird (vgl. Abschnitt 6.2.2, S. 157ff.).

Zweite Komponente ist das *MOD2SCMServerWorkspaceModule*, die „Gegenstelle“ für *MOD2SCMServer* aus dem Server-Teilsystem (s. Abb. 6.97). Sie erstellt und sendet Commit- und Checkout-Kommandos an das Server-Teilsystem (*commit()* und *update()*) und integriert die Ergebnisse in den Arbeitsbereich (*CommitServerCommand* und *CheckOutServerCommand* in Abb. 6.98). Es existieren mehrere Methoden für Commit und Update. Sie modellieren unterschiedliche Auswahlstrategien, um den Benutzern die Auswahl der betroffenen Elemente zu erleichtern (vgl. Abschnitt 5.3.1, S. 106ff.). Sie setzen Untermerkmale des Merkmals „Commit/Update“ (B.1.1) um, und sind daher mit entsprechenden Merkmalsmarkierungen versehen. Einzelne Elemente lassen sich immer einspielen und aktualisieren (*commitSingle()* und *updateSingle()*), doch mehrere Elemente auf einmal (*commitMultiple()* und *updateMultiple()*) oder sogar ganze Teilbäume (*commitSubtree()*) nur, wenn jeweils „Mehrere Elemente“ (B.1.1.3.2) (Markierung: *multipleitems*), „Mehrere Elemente“ (B.1.1.8.2) (Markierung: *multipleitems0*) oder „Vollständige Rekursion“ (B.1.1.3.4) (Markierung: *completesubtree*) Teil der Konfiguration des SKMS sind. Ihr Aufgabe ist es, eine Liste der betroffenen Elemente zu erstellen. Der weitere Ablauf beim Einspielen und Aktualisieren ist unabhängig von der Auswahlstrategie modelliert und erwartet lediglich die entsprechende Liste (Parameter *oidList* von *commit()*, Parameter *oidVidMap* von *update()*).

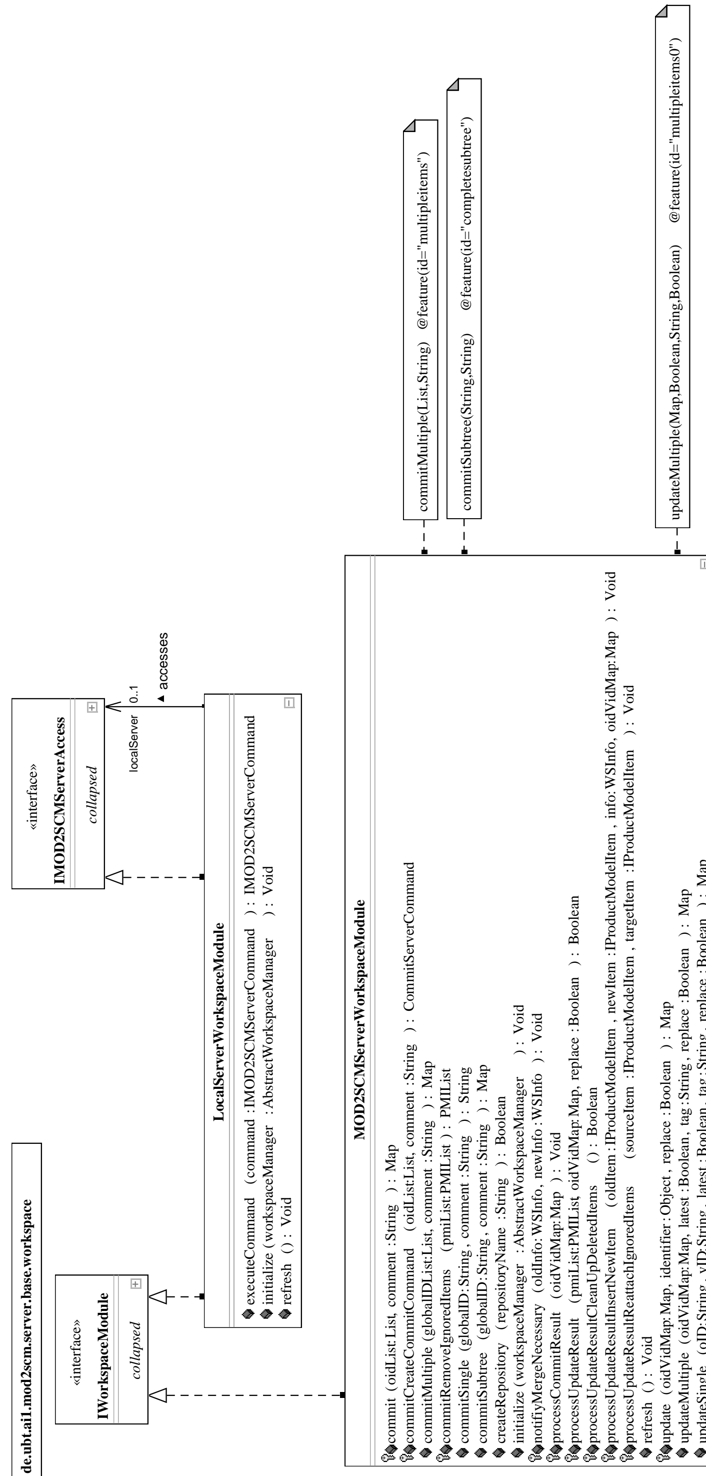


Abbildung 6.101.: Klassendiagramm: server.base.workspace

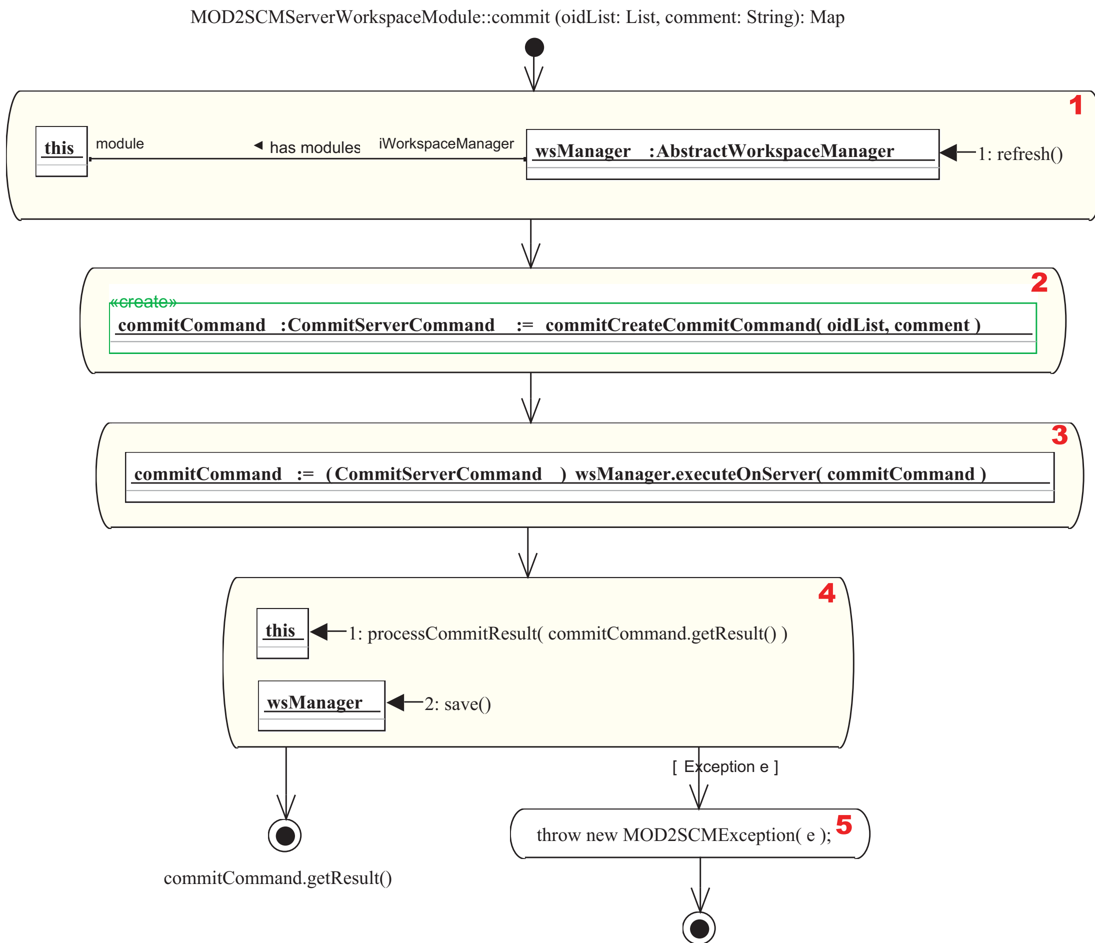


Abbildung 6.102.: Storydiagramm: commit() des MOD2-SKM-Arbeitsbereichs

Commit- und Update-Verhalten des Arbeitsbereichs

Abbildung 6.102 zeigt das Verhalten der *commit()*-Methode. Sie erhält – basierend auf den durch Benutzer und Auswahlstrategie bestimmten Elementen – die o.g. Liste der einzuspielenden Elemente (Parameter: *oidList*) und evtl. einen Kommentar (Parameter: *comment*). Die nummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Der Arbeitsbereich wird über seine Kernmodul-Schnittstelle (*AbstractWorkspace-Manager* aus *core.workspace*, vgl. Abschnitt 6.3.14, S. 206ff.) zu einer Aktualisierung aufgefordert, der dies an seine Module weiterleitet. Dies hat nicht in jedem Modul eine Auswirkung, ist aber besonders beim synchronen Abgleich zwischen Produktmodell und ihrer Betriebssystem-Repräsentation notwendig (*product.filesystem*, vgl. Abschnitt 6.4.14, S. 240ff.).

2. Ein Commit-Kommando (*CommitServerCommand*, s. Abb. 6.98) wird instanziert.

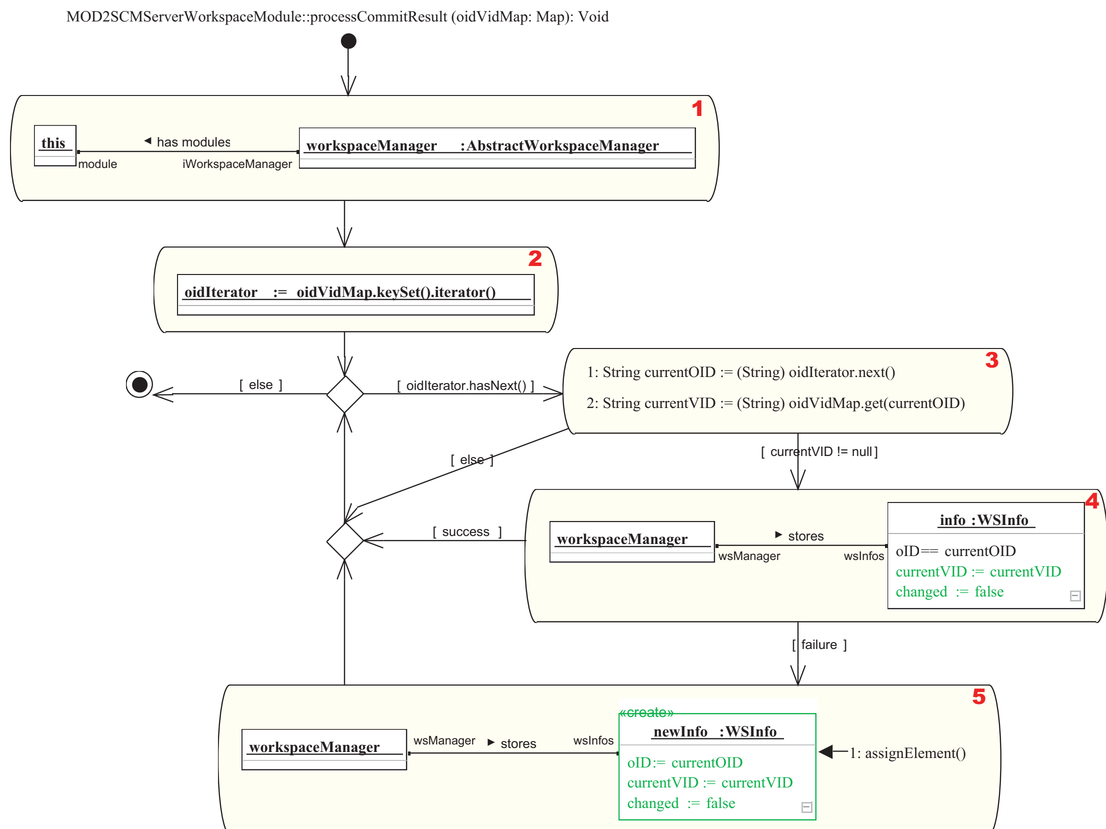


Abbildung 6.103.: Storydiagramm: processCommitResult() des MOD2-SKM-Arbeitsbereichs

3. Das Kommando wird über das Protokoll-Modul (z.B. *LocalServerWorkspaceModule* in Abb. 6.101) an das Server-Teilsystem gesendet und von ihm verarbeitet.

4. Die Informationen des Arbeitsbereichs werden aktualisiert. Dazu wird das Ergebnis des Kommandos (*commitCommand.getResult()*) in *processCommitResult()* verarbeitet. Dieser Vorgang wird im Anschluss noch genauer beschrieben. Abschließend werden die Versionsinformationen des Arbeitsbereichs persistenziert (*save()*).

5. Bei evtl. auftretenden Laufzeitfehlern, während der Ausführung des Kommandos oder der Aktualisierung des Arbeitsbereichs, wird eine Ausnahme geworfen und die Operation abgebrochen.

Abbildung 6.103 zeigt das Verhalten der Methode *processCommitResult()*, die, nach erfolgreicher Ausführung eines *commit()*-Kommandos, die Versionsinformationen des Arbeitsbereichs aktualisiert (Schritt vier in Abb. 6.102). Sie erhält die Liste der (oid, vid)-Tupel mit den neu zugewiesenen Versionskennungen der eingespielten Elemente (Parameter: *oidVidMap*). Die nummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Die Arbeitsbereichs-Verwaltung wird über ihre Kernmodul-Schnittstelle angesprochen (*AbstractWorkspaceManager* aus Kernmodul *core.workspace*, vgl. Abschnitt 6.3.14, S. 206ff.).
2. Es wird über alle Objektkennungen der (oid, vid)-Tupel iteriert. Für jedes Element werden die Schritte drei bis fünf ausgeführt.
3. Objekt- und Versionskennung werden ausgelesen. Die Versionskennung entspricht der im Repositorium zugewiesenen Versionskennung.
4. Die Versionsinformationen des Elements werden aktualisiert (*WSInfo* aus *core.-workspace*, vgl. Abschnitt 6.3.14, S. 206ff.), d.h., das eingespielte Objekt entspricht der neuen Versionskennung (*currentVID:=currentVID*) und gilt wieder als unverändert (*changed:=false*).
5. Sollte kein *WSInfo*-Objekt existieren, dann ist das Produktmodell-Element noch nicht unter Versionskontrolle. Es wird ein *WSInfo*-Objekt angelegt, das auf das neue Element verweist (*assignElement()*).

Abbildung 6.104 zeigt das Verhalten der *update()*-Methode. Sie erhält – basierend auf den durch Benutzer und Auswahlstrategie bestimmten Elementen – die o.g. Liste der zu aktualisierenden Elemente als (oid, vid)-Tupel (Parameter: *oidVidMap*), und evtl. ein zusätzliches Identifikationsmerkmal (Parameter: *identifier)*, wie z.B. eine Markierung oder die Anfrage nach der zuletzt eingespielten Version. Weiterhin legt der Parameter *replace* fest, ob geänderte Elemente überschrieben werden sollen (*replace:=true*) oder nicht (*replace:=false*). Die nummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Analog zu Schritt eins der *commit()*-Methode wird der Arbeitsbereich aktualisiert.
2. Anschließend wird über die Objektkennungen der angeforderten (oid, vid)-Tupel iteriert und Schritt drei für alle Kennungen ausgeführt.
3. Es wird geprüft, ob im Arbeitsbereich unter der Objektkennung bereits ein unversioniertes Produktmodell-Element existiert.
4. Fall ja, wird eine Ausnahme geworfen und die Aktualisierung abgebrochen.
5. Gibt es keine Konflikte mit unversionierten Elementen, wird ein Checkout-Kommando erzeugt (*CkeckOutServerCommand*, s. Abb. 6.98) und die Tupel-Liste als Parameter angehängt.
6. Analog zur dritten Storyaktivität von *commit()* wird das Checkout-Kommando vom Server verarbeitet.
7. Als Ergebnis liefert das Kommando die Liste der angeforderten Elemente (Lokales Objekt: *pmiList*) sowie eine evtl. veränderte Liste von (oid, vid)-Tupel zurück (Lokales Objekt: *oidVidMap*). Die Veränderung erfolgt bei Verwendung von zusätzlichen Identifikations-Merkmalen oder zusammengesetzten Elementen (s. Aktivität sechs und zehn in *checkOut()*, Abb. 6.100).

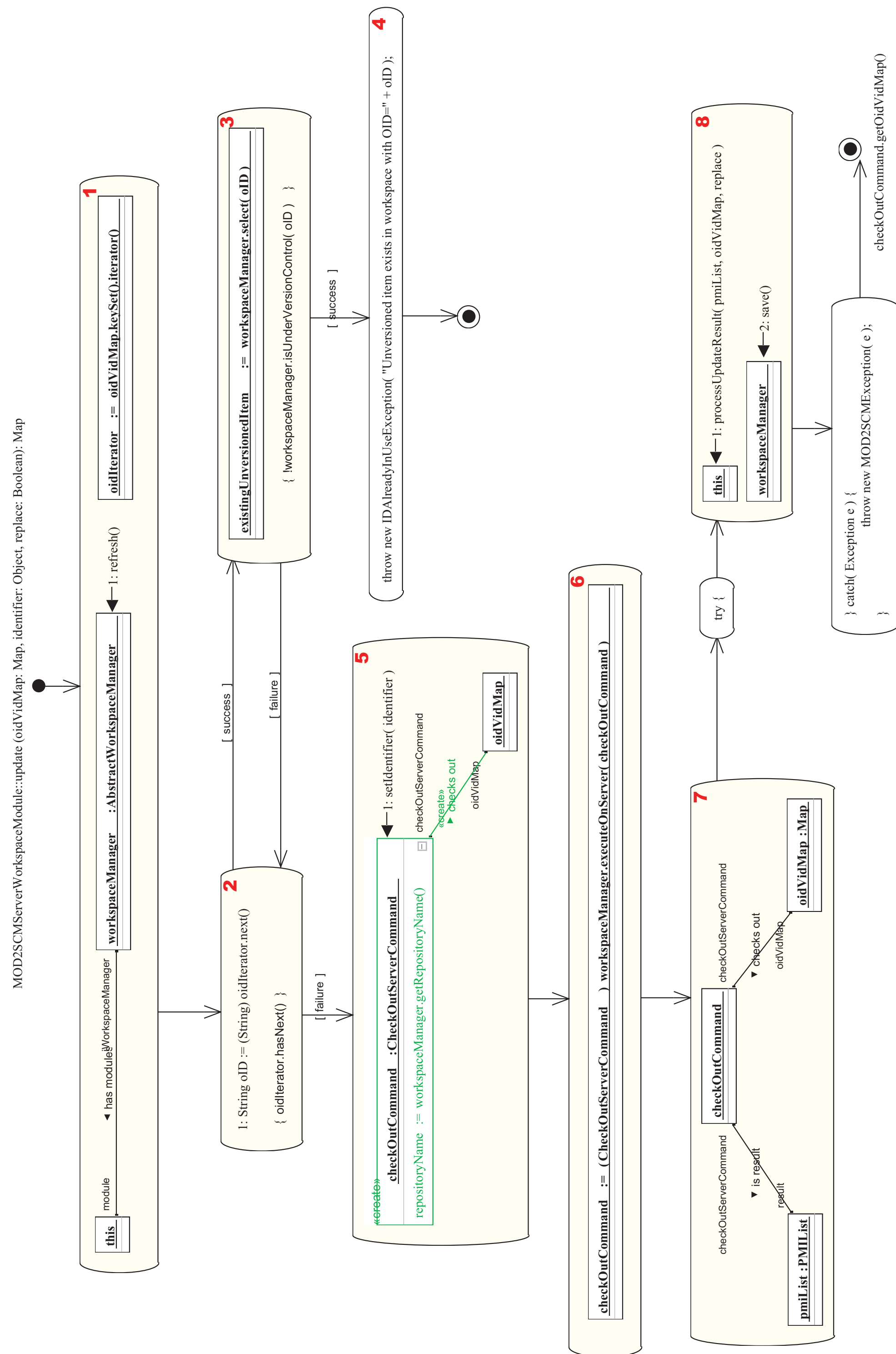


Abbildung 6.104.: Storydiagramm: update() des MOD2-SKM-Arbeitsbereichs

8. Beide Listen werden an *processUpdateResult()* übergeben, welche die Versionsinformationen des Arbeitsbereichs aktualisiert und die neuen Produktmodell-Elemente in die bestehende Produktmodell-Instanz integriert. Dieser Vorgang wird anschließend noch detaillierter beschrieben. Zuletzt werden, wie in Schritt vier von *commit()*, die Versionsinformationen persistenziert. Im Falle von Laufzeitfehlern wird eine Ausnahme geworfen.

Abbildung 6.105 zeigt das Verhalten der Methode *processUpdateResult()*, die nach erfolgreicher Ausführung eines *update()*-Kommandos die Versionsinformationen und die Produktmodell-Instanz des Arbeitsbereichs aktualisiert (Schritt acht in Abb. 6.104). Sie erhält die Listen der aktualisierten Produktmodell-Elemente (Parameter: *pmiList*) und der (oid, vid)-Tupel mit den angeforderten Versionskennungen der Elemente (Parameter: *oidVidMap*). Der Parameter *replace* bestimmt, ob geänderte Produktmodell-Elemente ersetzt werden sollen. Die nummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Die Arbeitsbereichs-Verwaltung wird über die Schnittstelle ihres Kernmoduls angesprochen (*AbstractWorkspaceManager* aus *core.workspace*, vgl. Abschnitt 6.3.14, S. 206ff.).

2. Es wird über die Objektkennungen der aktualisierten Elemente iteriert und die Schritte vier bis 11 für jedes Element ausgeführt.

3. Die nächste Objektkennung wird gesucht. Gibt es keine mehr, ist die Aktualisierung des Arbeitsbereichs abgeschlossen.

4. Die Objektkennung wird gelesen (Lokale Variable: *oid*).

5. Auswahl des aktualisierten Produktmodell-Elements mit der Objektkennung *oid* (Lokales Objekt: *newItem*).

6. Suche der Versionsinformationen im Arbeitsbereich über eine bereits existierende Version des Elements (Lokales Objekt: *info*).

7. Existieren keine Versionsinformationen, werden diese angelegt.

8. Das neue Produktmodell-Element (*newItem*) wird in die Produktmodell-Instanz des Arbeitsbereichs eingefügt. Dann geht es bei Schritt drei weiter.

9. Findet Storyaktivität sechs Versionsinformationen, existiert bereits eine Version des aktualisierten Elements. Dieses soll **nicht** überschrieben werden (Einschränkung: *!replace*), falls es bereits geändert wurde (*changed==true*). In diesem Fall werden die neuen Versionsinformationen getrennt abgelegt (Lokales Objekt: *newInfo*) und mit den alten Informationen (*info*) an das Verschmelzen-Modul weitergeleitet (*notifiyMergeNecessary()*, vgl. Abschnitt 6.3.6, S. 184ff.).

10. Sollen die lokalen Änderung dagegen überschrieben werden, wird der Status auf *changed:=false* zurückgesetzt, und Schritt 11 trotz der Änderungen ausgeführt.

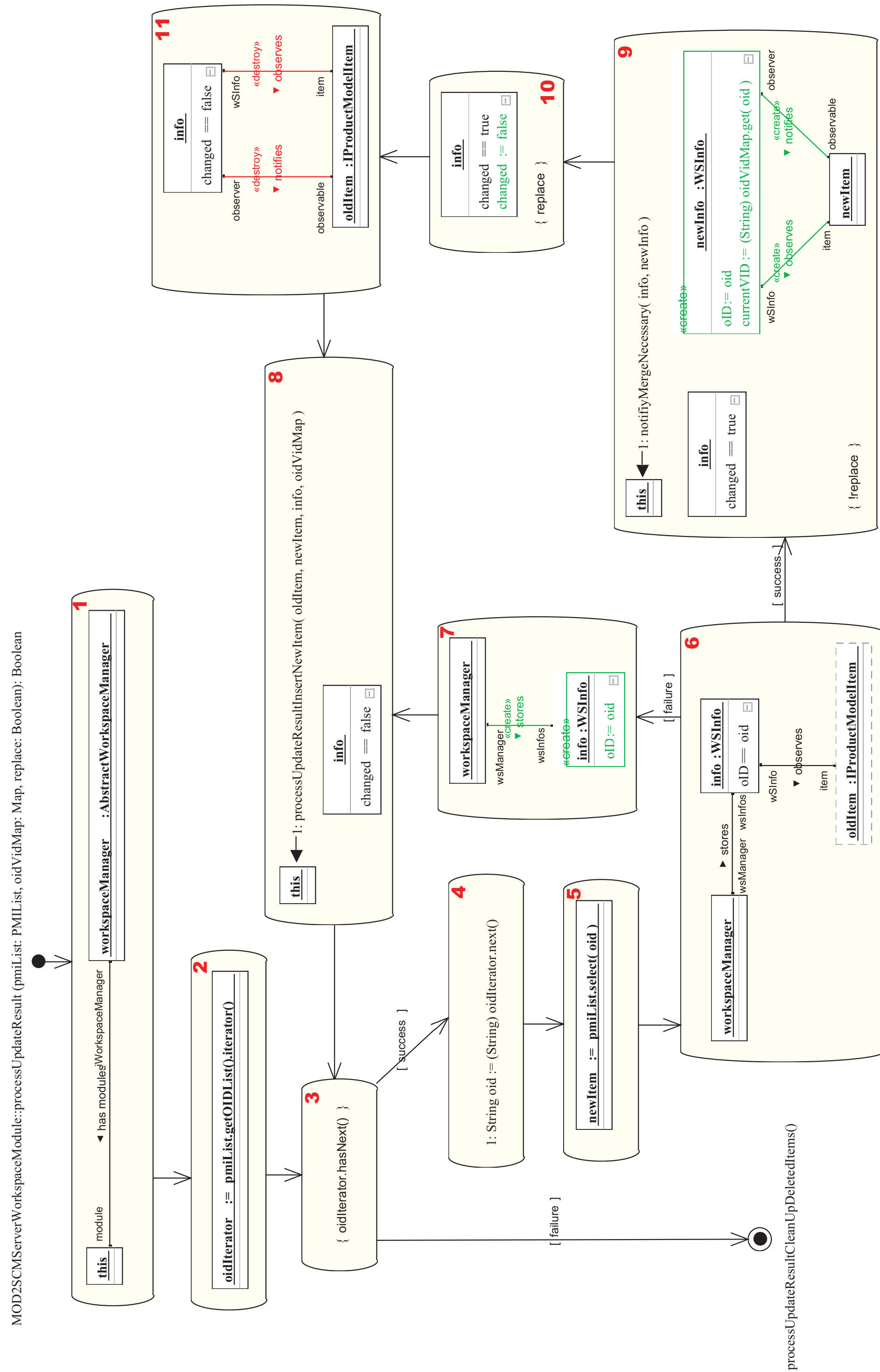


Abbildung 6.105.: Storydiagramm: processUpdateResult() des MOD2-SKM-Arbeitsbereichs

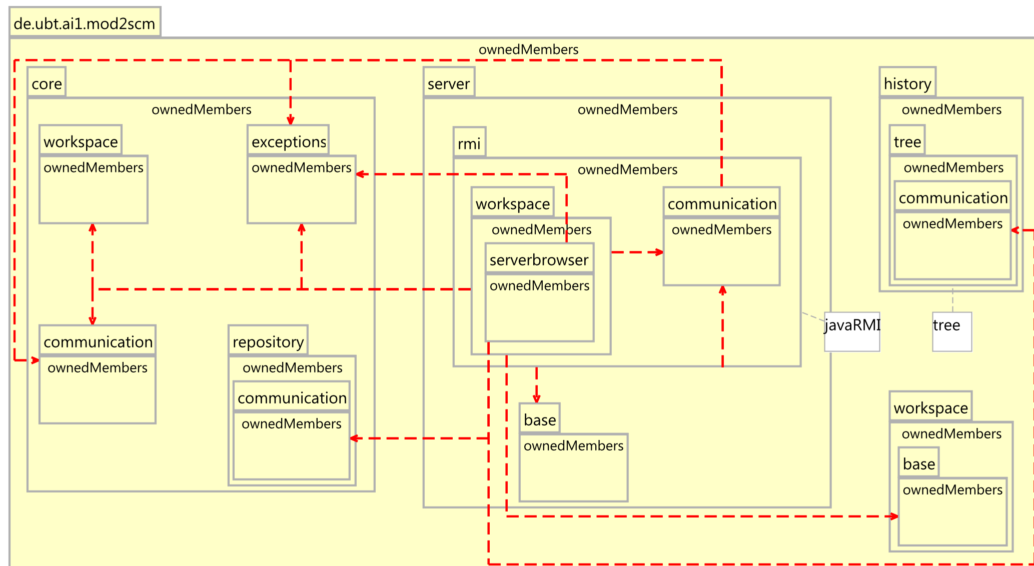


Abbildung 6.106.: Paketdiagramm: server.rmi

11. Die Versionsinformationen (*info*) werden vom lokal existierenden Produktmodell-Element gelöst (*oldItem*). Damit ist der gleiche Zustand erreicht, wie beim Einfügen eines vollständig neuen Produktmodell-Elements und es geht weiter bei Schritt acht.

6.4.19. Modul: server.rmi

Überblick

Das Modul *server.rmi* erweitert das Kommunikations-Teilsystem um die Datenübertragung über Java-RMI. Server- und Arbeitsbereich-Teilsystem können dann auf unterschiedlichen Computersystemen laufen. Das Verhalten des Servers wird fast vollständig aus *server.base* wiederverwendet, und lediglich das „Java RMI“-Protokoll in die Server-Schnittstelle integriert.

Abhängigkeiten

Abbildung 6.106 zeigt die Abhängigkeiten von *server.rmi*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Java RMI“ (I.3.2) (Markierung: *javaRMI*) zur Konfiguration gehört. Das Modul basiert auf dem Kernmodul des Kommunikations-Teilsystems (*core.communication*). Außerdem müssen noch die zusätzlichen Ausnahmen (*core.exceptions*) bei der Datenübertragung im Arbeitsbereich (*core.workspace*) verarbeitet werden. Das restliche Verhalten kann durch Vererbung aus dem eben beschriebenen *server.base* übernommen werden. Die MOD2-SKM Erweiterung für Eclipse bietet einen „Server-Browser“, um ein Repositorium für einen initialen Checkout auszuwählen (vgl. Abschnitt 7, S. 315ff.). Diese Komponente (*workspace.serverbrowser*) ist direkt in *server.rmi* integriert und sendet Anfragen an die Repositorien (*core.repository*) und die

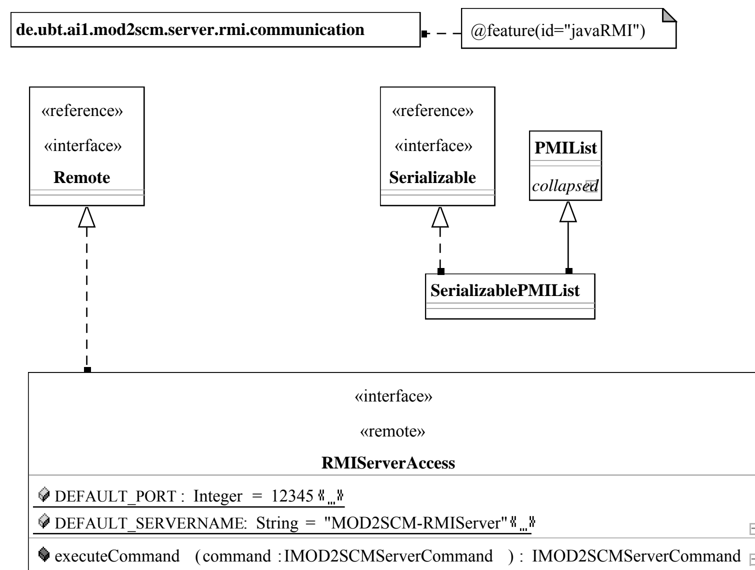


Abbildung 6.107.: Klassendiagramm: server.rmi.communication

Baumhistorie (*history.tree*). Die letzte Abhängigkeit zeigt, dass in zukünftigen Arbeiten die Schnittstelle aus *core.history* noch an das Konzept eines Server-Browser angepasst werden sollte.

Detailbeschreibung

Die Abbildung 6.107 zeigt die erweiterte Schnittstelle des Kommunikations-Teilsystems (*RMIServerAccess*), basierend auf der Schnittstelle *java.rmi.Remote* [Ull09]. Die Signatur der Methode aus *core.communication* (vgl. Abschnitt 6.3.1, S. 173ff.) hat sich kaum verändert, außer das der Stereotyp „remote“ die Ausnahme „RemoteException“ hinzufügt. Zusätzlich muss die Liste der Produktmodell-Elemente serialisierbar sein, um über RMI gesendet werden zu können (*SerializablePMIList*).

Abbildung 6.108 zeigt die Klasse des Server-Teilsystems, *RMIServer*. Sie implementiert die eben beschriebene Schnittstelle *RMIServerAccess* und erbt die Methoden von *MOD2-SCMServer* aus *server.base*, u.a. das Commit- und Checkout-Verhalten (vgl. Abschnitt 6.4.18, S. 254ff.). Beim Starten und Anhalten des Servers ist es notwendig, das Server-Objekt für RMI zu registrieren (*registerRMI()*) bzw. zu deregistrieren (*unregisterRMI()*). Dies geschieht durch Überladen der Methoden *prepareServer()* bzw. *stopServer()*.

Abbildung 6.109 zeigt das *RMIWorkspaceModule*, d.h. die Gegenstelle zu *RMIServer* (s. Abb. 6.108). Sie ersetzt das in *server.base* definierte Protokoll *LocalServerWorkspace-Module*, in dem sie, für die Übertragung eines Kommando-Objekts, eine RMI-Verbindung aufbaut (*connect()*) und nach Erhalt des Rückgabewerts beendet (*disconnect()*). Beide Protokolle nutzen die gleiche Schnittstelle wie ein Server (*IMOD2SCMServerAccess*), d.h., sie verbergen die Details des Kommunikations-Teilsystems vor den restlichen Arbeitsbereichs-Modulen.

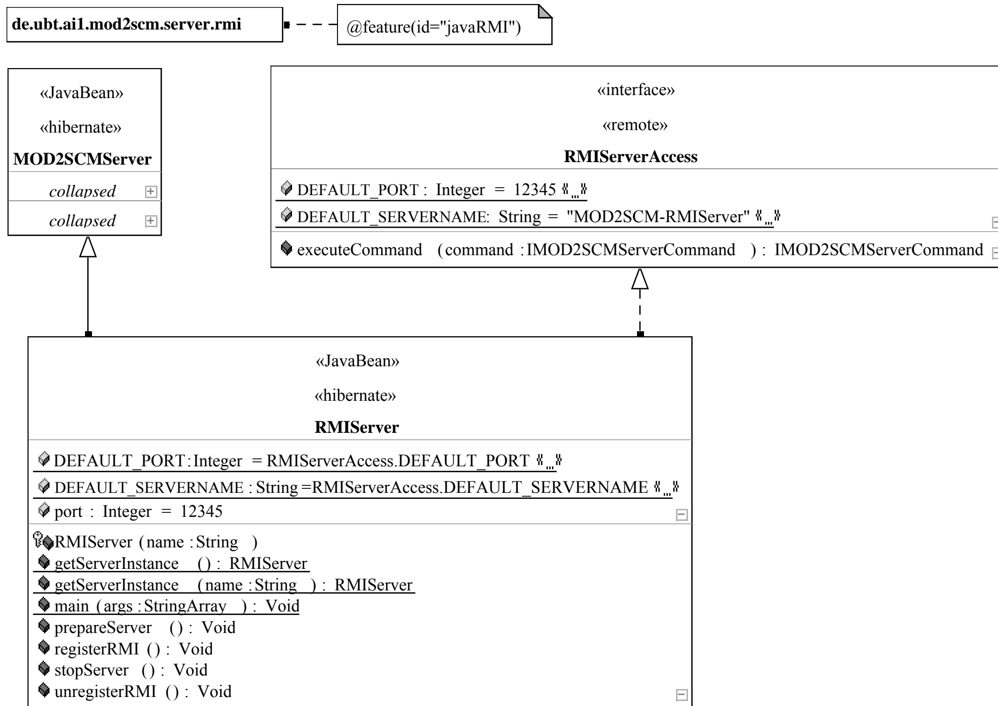


Abbildung 6.108.: Klassendiagramm: server.rmi

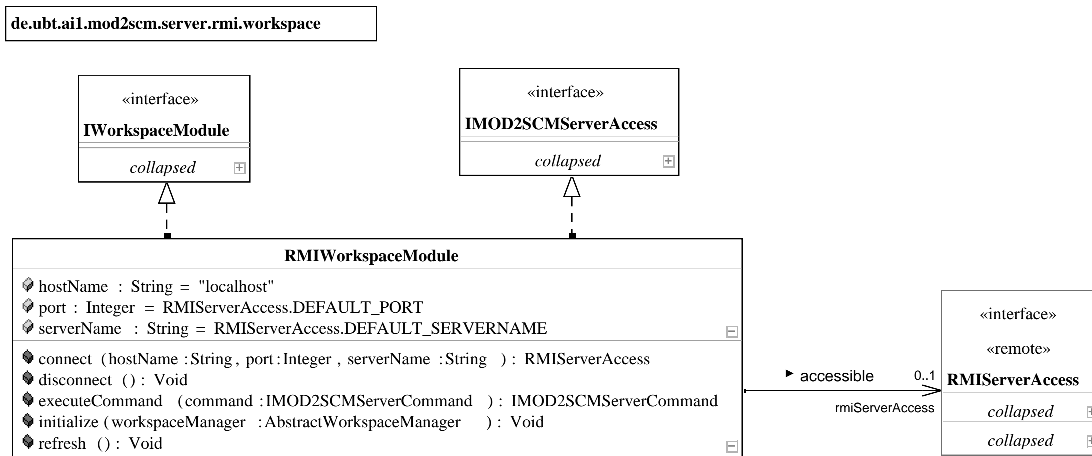


Abbildung 6.109.: Klassendiagramm: server.rmi.workspace

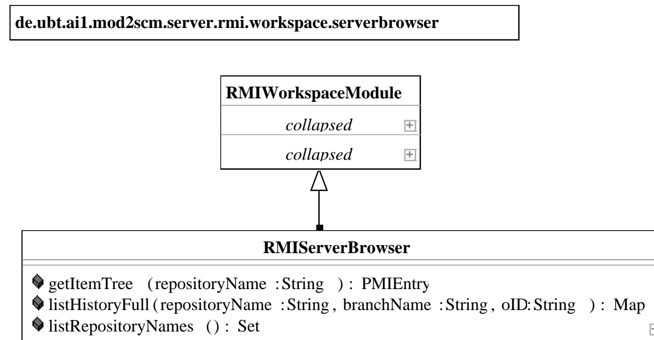


Abbildung 6.110.: Klassendiagramm: server.rmi.workspace.serverbrowser

Abbildung 6.110 zeigt das Klassendiagramm des Server-Browsers (*RMIServerBrowser*). Dabei handelt es sich um eine Erweiterung des Kommunikations-Protokolls (*RMIWorkspaceModule*), die zusätzliche Methoden anbietet. Diese benötigt die MOD2-SKM-Erweiterung für Eclipse, um (1) die Namen aller Repositorien (*listRepositoryNames()*), (2) die aktuellste Version ihres Inhalts (*getItemTree()*) und (3) die Historie eines Elements (*listFullHistory()*) anzeigen zu können (vgl. Abschnitt 7, S. 315ff.).

6.4.20. Modul: server.runtimeconfig

Überblick

Das Modul *server.runtimeconfig* modelliert einen MOD2-SKM-Server, der zur Laufzeit konfiguriert werden kann. Mit ihm lassen sich unterschiedliche Konfigurationen des Server-Teilsystems testen, indem die Testfälle die Konfiguration übernehmen. Struktur und Verhalten werden aus server.base und server.rmi wiederverwendet, und es kommen auch die gleichen Module aus der Modulbibliothek zum Einsatz. Die Laufzeitkonfiguration erfolgt dynamisch, durch Zuweisung der Modulfabriken mit Hilfe einer externen Server-Fabrik, anstatt durch Ausführen der statisch konfigurierten Methode *initServer()* (s. Abb. 6.96).

Abhängigkeiten

Abbildung 6.111 zeigt die Abhängigkeiten von *server.runtimeconfig*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Laufzeitkonfigurierbarer Server“ (I.2) (Markierung: *runtimeConfigurable*) zur Konfiguration gehört. Als Server basiert das Modul auf dem Kernmodul des Server-Teilsystems (*core.server*). Verhalten und Anbindung an das Kommunikations-Teilsystem werden aus *server.rmi* wiederverwendet.

Detailbeschreibung

Die Modul-Fabriken des MOD2-SKM-Servers werden in *server.base* in der Methode *initServer()* angelegt (vgl. Abschnitt 6.4.18, S. 254ff.). Den Typ der instanzierten Fabrik

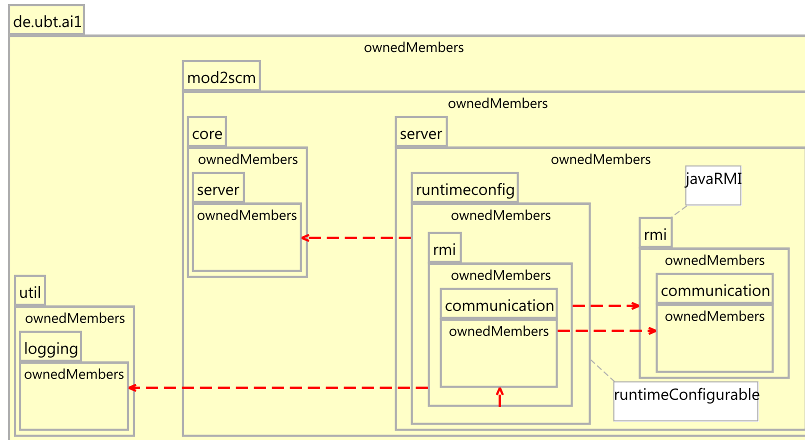


Abbildung 6.111.: Paketdiagramm: server.runtimeconfig

legt letztendlich ein Merkmal aus der Konfiguration fest (s. Abb. 6.96, S.256). In *server.runtimconfigurable* existiert eine Server-Fabrik, welche die Fabrik-Module anhand einer Konfiguration instanziert und dann einem Server zuweist.

Abbildung 6.112 zeigt das Klassendiagramm der Server-Fabrik. Es definiert eine abstrakte Basisklasse für Server-Fabriken (*AbstractRuntimeConfigurableServerFactory*), die unterschiedlich konfigurierte Server erzeugt (*createServer()*), indem sie ihnen anhand einer Konfiguration (Parameter: *configuration*) ihre Modul-Fabriken zuweist (*assignFactories()*). Die Server-Fabrik kann daher nur Merkmale konfigurieren, die über eine Modul-Fabrik modelliert wurden.

Eine Konfiguration besteht aus einer Menge von Aufzählungs-Datentypen (*FeatureEnum*). Jedes Element einer Aufzählung entspricht einem Merkmal aus der Merkmalskonfiguration (z.B. entspricht *HistoryEnum.TREE* dem Merkmal „Baum“ (D.1.2.1.1)). Analog zu einer Merkmalsgruppe (vgl. Abschnitt 3.2, S. 54ff.), gruppiert eine Aufzählung mehrere Elemente. Z.B. gruppiert die Aufzählung *HistoryEnum* ihre Elemente analog zu „Graphbasiert“ (D.1.2.1).

Abbildung 6.113 zeigt die Erweiterung der Schnittstelle zum Kommunikations-Teilsystem (*RuntimeConfigurableRMIAccess*). Sie erweitert die Schnittstelle des RMI-Servers (*RMIServerAccess* aus *server.rmi*, vgl. Abschnitt 6.4.19, S. 271ff.) um eine Methode zum Abfragen der Modul-Fabrik-Konfiguration (*getConfiguration()*).

Abbildung 6.114 zeigt schließlich die Realisierung einer Server-Fabrik für laufzeitkonfigurierbare RMI-Server. Die Server-Fabrik (*RuntimeConfigurableRMIServerFactory*) basiert auf der, in Abb. 6.112 gezeigten, abstrakten Fabrik (*AbstractRuntimeConfigurableServerFactory*). Sie erzeugt unterschiedliche konfigurierte RMI-Server (*RuntimeConfigurableRMIServer*), welche die Schnittstelle *RuntimeConfigurableRMIAccess* aus Abb.6.113 implementiert und so Auskunft über seine Konfiguration geben kann.

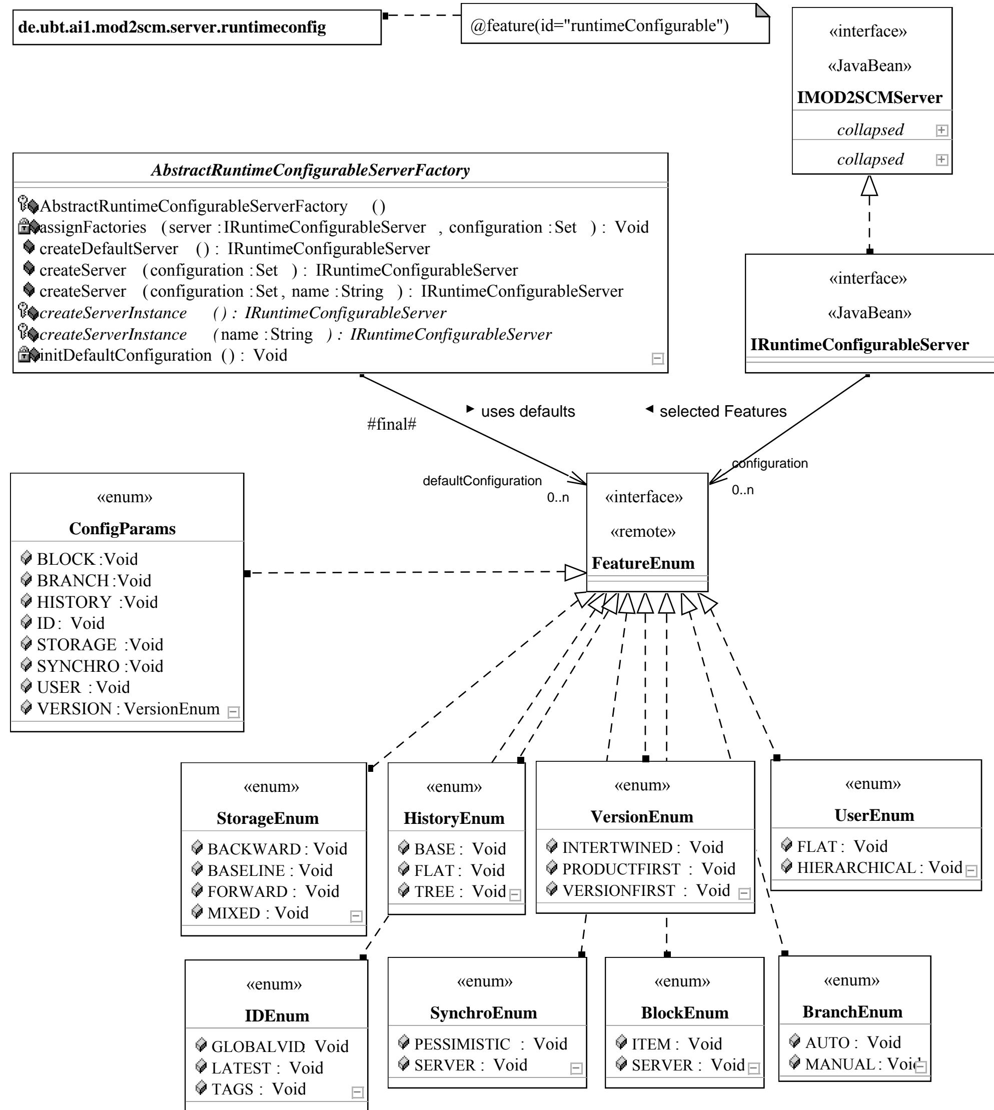


Abbildung 6.112.: Klassendiagramm: server.runtimeconfig

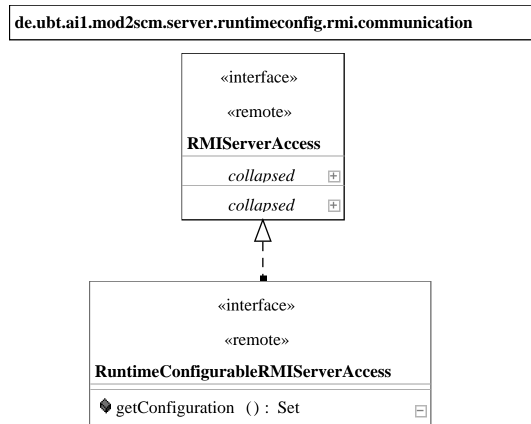


Abbildung 6.113.: Klassendiagramm: server.runtimeconfig.rmi.communication

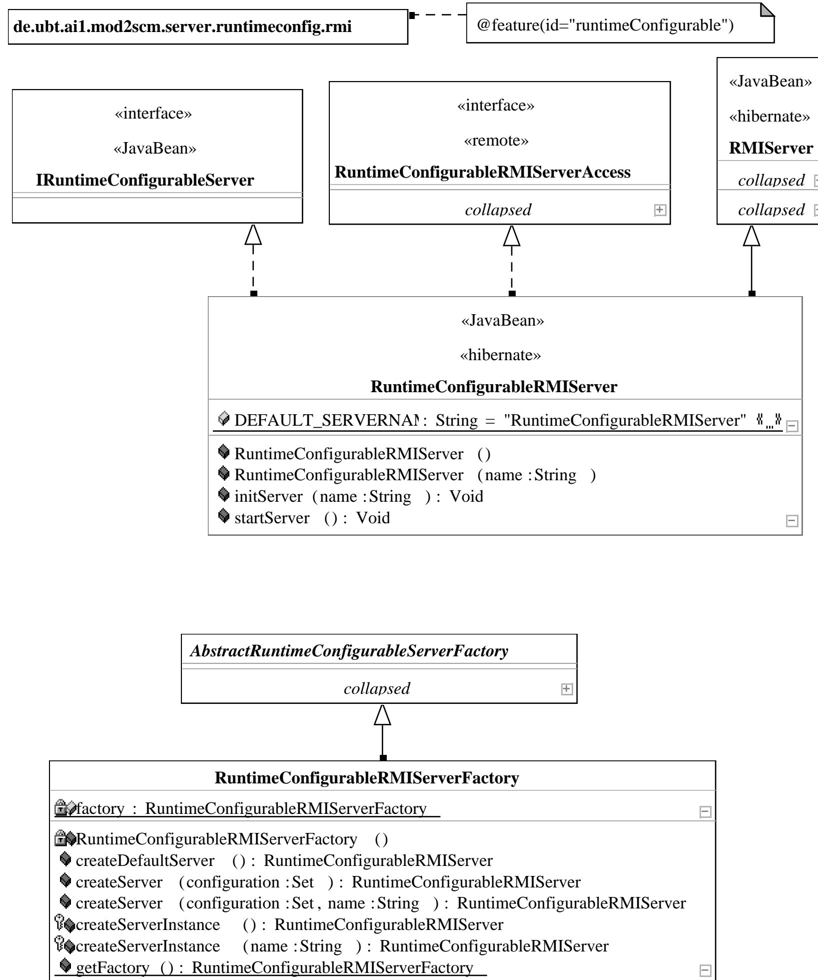


Abbildung 6.114.: Klassendiagramm: server.runtimeconfig.rmi

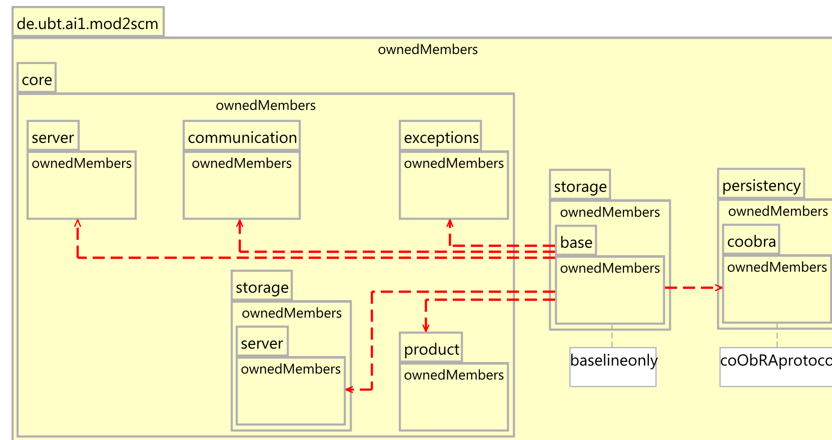


Abbildung 6.115.: Paketdiagramm: storage.base

6.4.21. Modul: storage.base

Überblick

Das Modul *storage.base* modelliert einen Speicher für Produktmodell-Elemente. Es handelt sich um die einfachste Speichermethode, als Menge von Elementen ohne Beziehungen.

Abhängigkeiten

Abbildung 6.115 zeigt die Abhängigkeiten von *storage.base*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn in der Konfiguration das Merkmal „Deltas“ (D.4.1) und das Merkmal „Referenzen komplexer Elemente“ (D.4.3) **fehlen** (Markierung: *baselineonly*). Als Speichermodul basiert es auf dem Kernmodul *core.storage* und ist als Komponente des Server-Teilsystems (*core.server*) in der Lage, Kommandos aus dem Kommunikations-Teilsystem entgegenzunehmen (*core.communication*). Speicherinhalt sind Elemente beliebiger Produktmodelle (*core.product*). Laufzeitfehler werden durch Ausnahmen behandelt (*core.exceptions*).

Detailbeschreibung

Abbildung 6.116 zeigt das Klassendiagramm von *storage.base*. Wie in *core.storage* verlangt (vgl. Abschnitt 6.3.11, S. 200ff.), existiert ein Fabrik-Modul zum Erzeugen eines Speichers vom Typ *BaseStorageModule* für ein Repositorium. Eingespielte Produktmodell-Elemente werden in ihm unter ihrer Speicherkennung (Parameter *storageID* der qualifizierten Assoziation „stores“) abgelegt. Dies wird in der abstrakten Oberklasse aus Objekt- und Versionskennung erstellt (*AbstractStorage* aus *core.storage*, vgl. Abschnitt 6.3.11, S. 200ff.).

Ein zusammengesetztes Element würde in *storage.base* vollständig unter der Kennung seines obersten Elements gespeichert. Die Kindelemente und ihre Versionen sind damit

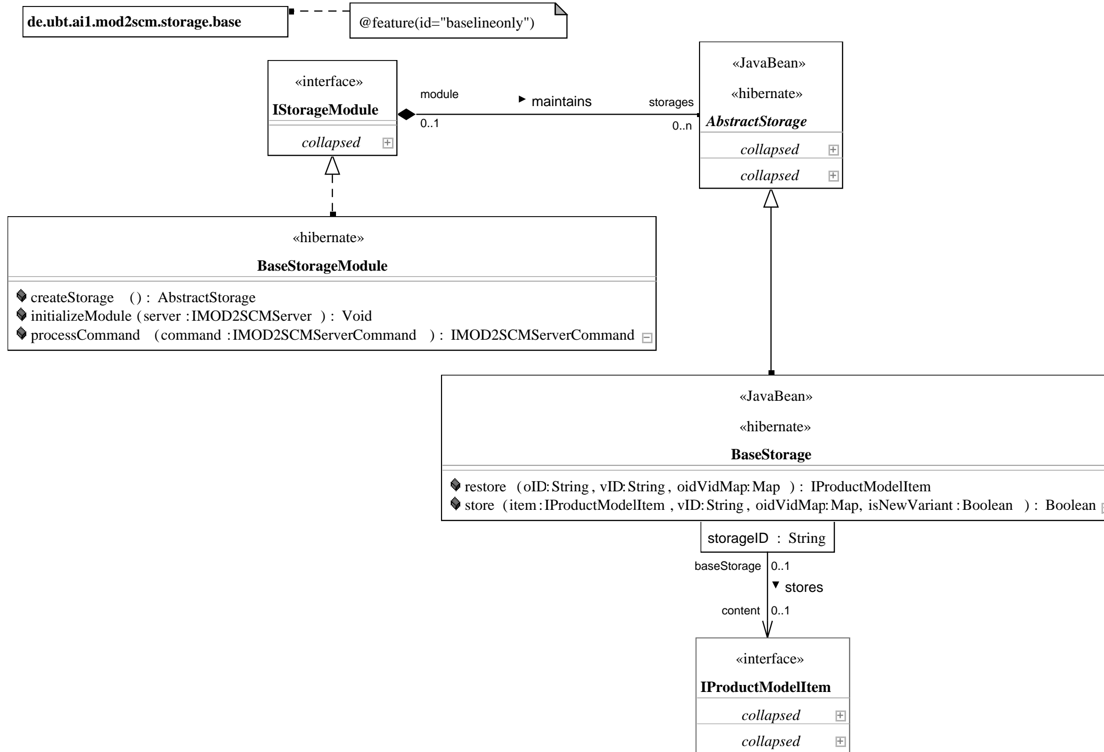


Abbildung 6.116.: Klassendiagramm: storage.base

implizit über das Elternelement festgelegt. Auch würde das Element jedes Mal vollständig gespeichert, auch wenn ganze Teilbäume von Kindelementen unverändert geblieben sind. Das Merkmalsmodell stellt daher über aussagenlogische Einschränkungen sicher, dass dieser Speicher nur mit atomaren Produktmodell-Elementen verwendet wird (vgl. Abschnitt 5.6.2, S. 137ff.).

6.4.22. Modul: storage.complex

Überblick

Das Modul *storage.complex* modelliert einen Speicher für die effiziente Speicherung zusammengesetzter Produktmodell-Elemente. Eine seiner Methoden trifft die Unterscheidung zwischen versionszentrierter und verwobener Integration von Produkt- und Versionsraum (vgl. Abschnitt 4.3.4, S. 81ff.).

Abhängigkeiten

Abbildung 6.117 zeigt die Abhängigkeiten von *storage.base*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Referenzen komplexer Elemente“ (D.4.3) (Markierung: *complexitemreferences*) Teil der Konfiguration ist. Als Speichermodul basiert es auf dem Kernmodul *core.storage* und ist als Komponente des Server-

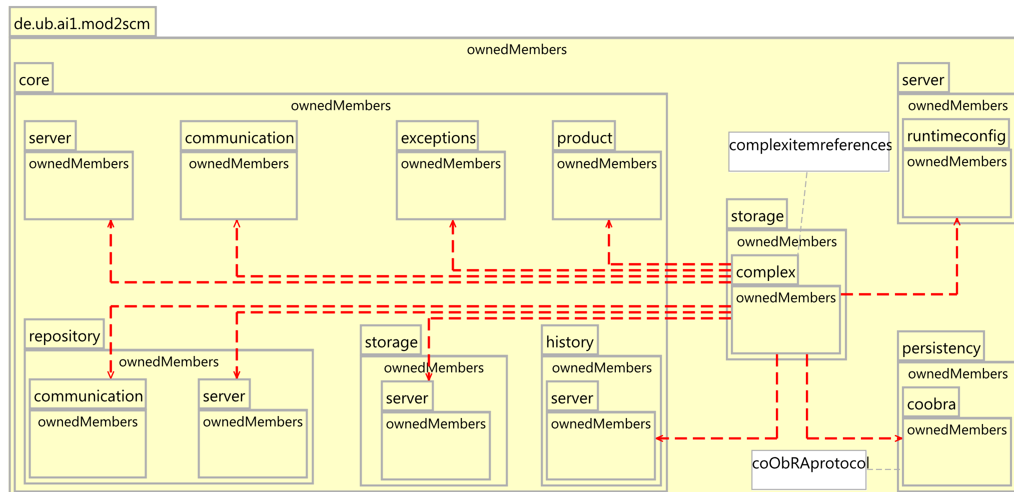


Abbildung 6.117.: Paketdiagramm: storage.complex

Teilsystems (*core.server*) in der Lage, Kommandos aus dem Kommunikations-Teilsystem entgegenzunehmen (*core.communication*). Speicherinhalt sind Elemente beliebiger Produktmodelle (*core.product*). Um unveränderte Kind-Elemente zu referenzieren, bzw. zusammengesetzte Elemente verwoben wiederherstellen zu können (vgl. Abschnitt 4.3.4, S. 83ff.), sind auch Informationen des Repositoriums (*core.repository*) und der Historie (*core.history*) notwendig. Laufzeitfehler werden durch Ausnahmen behandelt (*core.exceptions*). Um auf dem laufzeitkonfigurierbaren Server (*server.runtimeconfig*, vgl. Abschnitt 6.4.20, S. 273ff.) zwischen versionszentrierter und verwobener Integration wechseln zu können, musste eine spezielle Methode modelliert werden.

Detailbeschreibung

Abbildung 6.118 zeigt das Klassendiagramm von *storage.complex*. Wie vom Speicher-Kernmodul verlangt (*IStorageModule* aus *core.storage*, vgl. Abschnitt 6.3.11, S. 200ff.), existiert ein Fabrik-Modul zum Anlegen eines komplexen Speichers für ein Repositorium (*ComplexStorageModule*). Ein komplexer Speicher speichert nur die zusammengesetzten Produktmodell-Elemente unter einer Speicherkennung (*ComplexStorageItem* mit „hasContent“-Assoziation auf *IComplexProductModelItem*). Für die atomaren Kindelemente wird ein weiteres Speichermodul genutzt („hasChildFactory“-Assoziation von *ComplexStorageModule*), wofür dann z.B. *storage.base* (vgl. Abschnitt 6.4.21, S. 279ff.) oder *storage.delta* (vgl. Abschnitt 6.4.23, S. 283ff.) genutzt werden können.

Beim Speichern wird die Kernmodul-Schnittstelle für zusammengesetzte Produktmodell-Elemente benötigt (*IComplexProductModelItem* aus Kernmodul *core.product*, vgl. Abschnitt 6.3.8, S. 188ff.). Sie verlangt, dass jedes zusammengesetzte Produktmodell-Element zerlegt und zusammengesetzt werden kann. Beim Speichern wird daher jedes Element zerlegt und einzeln gespeichert – je nach Typ im Speicher für komplexe oder atomare Elemente. Für jede Eltern-Kind-Beziehung wird eine *ComplexChildReference*

Abbildung 6.118.: Klassendiagramm: storage.complex

zwischen dem gespeicherten Elternelement („hasChildren"-Assoziation von *ComplexStorageItem*) und dem Speicher des Kindes („hasChildren"-Assoziation auf *AbstractStorage*) erzeugt. Das Referenzobjekt speichert die Speicherkennung des Kindes, und kann so das Kindelement jederzeit wiederherstellen.

Die o.g. Art der Wiederherstellung entspricht der versionszentrierten Integration von Produkt- und Versionsraum: Durch Auswahl des Elternelements wird die genaue Speicherkennung – und damit auch die Version – des Kindelements festgelegt (vgl. Abschnitt 4.3.4, S. 81ff.). Alternativ kann auch eine vom Benutzer festgelegte Versionskennung verwendet werden. Dazu reicht es aus, die Speicherkennung neu zusammenzusetzen, anstatt sie über das Referenzobjekt wiederherzustellen. Daher bietet der komplexe Speicher (*ComplexStorage*) mehrere alternative Implementierung der Methode *restoreChildrenGetStorageID()* an. Über Merkmalsmarkierungen wird gesteuert, welche Methode im konfigurierten SKMS enthalten ist und ob eine versionszentrierte (Merkmal „Versionszentriert" (D.7.2), Markierungen: *nOTRuntimeConfigurable* und *versionfirst*) oder eine verwobene (Merkmal „Verwoben" (D.7.3), Markierungen: *nOTRuntimeConfigurable* und *intertwined*) Integration von Produkt- und Versionsraum vorliegt. Für den Einsatz im laufzeitkonfigurierbaren Server (*server.runtimeconfig*, vgl. Abschnitt 6.4.20, S. 273ff.) existiert eine dritte Implementierung (Merkmal „Laufzeitkonfigurierbarer Server" (I.2), Markierung: *runtimeConfigurable*), die anhand der aktuellen Server-Konfiguration entweder das Verhalten der ersten oder der zweiten emuliert. Diese Abfrage erzeugt die Abhängigkeit zum Paket *server.runtimeconfig* (s. Abb. 6.117).

Die Speichern zusammengesetzter Elemente erforderte mehrere aufwändig modellierte private Methoden (*expandToRootItem()*, *findParentItem()*, usw.), die ich aus Platzgründen hier nicht im Detail vorstelle. Dafür ist der Speichermechanismus jedoch vollständig vom konkreten Produktmodell entkoppelt und basiert nur auf den Schnittstellen des Kernmoduls *core.product*. Dies bedeutet, dass sich beliebige Produktmodell-Elemente sowohl Speicherplatz-effizient als auch versionszentriert bzw. verwoben integriert ablegen lassen. Zusätzlich können für ausgewählte atomare Elemente noch Delta-Algorithmen modelliert und über den Speicher für Kindelemente eingebunden werden. Die Referenz-Objekte (*ComplexChildReference*) können aus der Speicherkennung des Kindes noch beliebige Schlüssel-Wert-Paare speichern („additional Info"-Assoziation). Dies ist für Produktmodelle mit gerichteten Kanten notwendig, wie z.B. das Anwendungsfall-Produktmodell (*product.usecase*, vgl. Abschnitt 6.4.15, S. 247ff.).

6.4.23. Modul: storage.delta

Überblick

Das Modul *storage.delta* modelliert drei unterschiedliche Verfahren zur Speicherung gerichteter Deltas von Produktmodell-Elementen: Vorwärts-Deltas, Rückwärts-Deltas und gemischte Deltas (vgl. Abschnitt 4.3.6, S. 89ff.).

Die Verfahren zur Delta-Berechnung und -Anwendung sind im konkreten Produktmodell modelliert und alle drei Verfahren erwarten, dass sie diese Verfahren über die Schnittstelle des Kernmoduls *core.delta* (vgl. Abschnitt 6.3.2, S. 175ff.) ansprechen können. Im

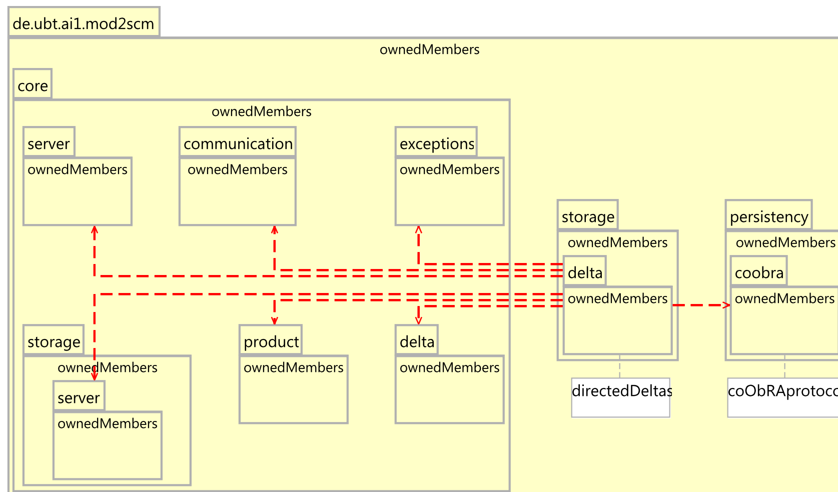


Abbildung 6.119.: Paketdiagramm: storage.delta

Dateisystem-Produktmodell *product.filesystem* sind mehrere Delta-Verfahren modelliert (vgl. Abschnitt 6.4.14, S. 240ff.).

Abhängigkeiten

Abbildung 6.119 zeigt die Abhängigkeiten von *storage.delta*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Deltas" (D.4.1) (Markierung: *directedDeltas*) Teil der Konfiguration ist. Als Speichermodul für Deltaspeicherung basiert es auf den Kernmodulen *core.storage* und *core.delta*. Als Komponente des Server-Teilsystems (*core.server*) ist es in der Lage, Kommandos aus dem Kommunikations-Teilsystem entgegenzunehmen (*core.communication*). Speicherinhalt sind Elemente beliebiger Produktmodelle (*core.product*). Laufzeitfehler werden durch Ausnahmen behandelt (*core.exceptions*).

Detailbeschreibung

Das Modul *storage.delta* modelliert drei unterschiedliche Arten der gerichteten Deltas: Vorwärts-Deltas, Rückwärts-Deltas und gemischte Deltas (vgl. Abschnitt 4.3.6, S. 89ff.). Dementsprechend sind die jeweiligen Klassen mit dem entsprechenden Merkmal markiert:

- „Vorwärts-Deltas" (D.4.1.1.3) (Markierung: *forwardDeltas*),
- „Rückwärts-Deltas" (D.4.1.1.2) (Markierung: *backwardDeltas*), oder
- „Gemischte Deltas" (D.4.1.1.1) (Markierung: *mixedDeltas*).

Abbildung 6.120 zeigt das Klassendiagramm von *storage.delta*. Wie vom Speicher-Kernmodul *core.storage* vorgegeben (vgl. Abschnitt 6.3.11, S. 200ff.), existiert für jede Art

Abbildung 6.120.: Klassendiagramm: storage.delta

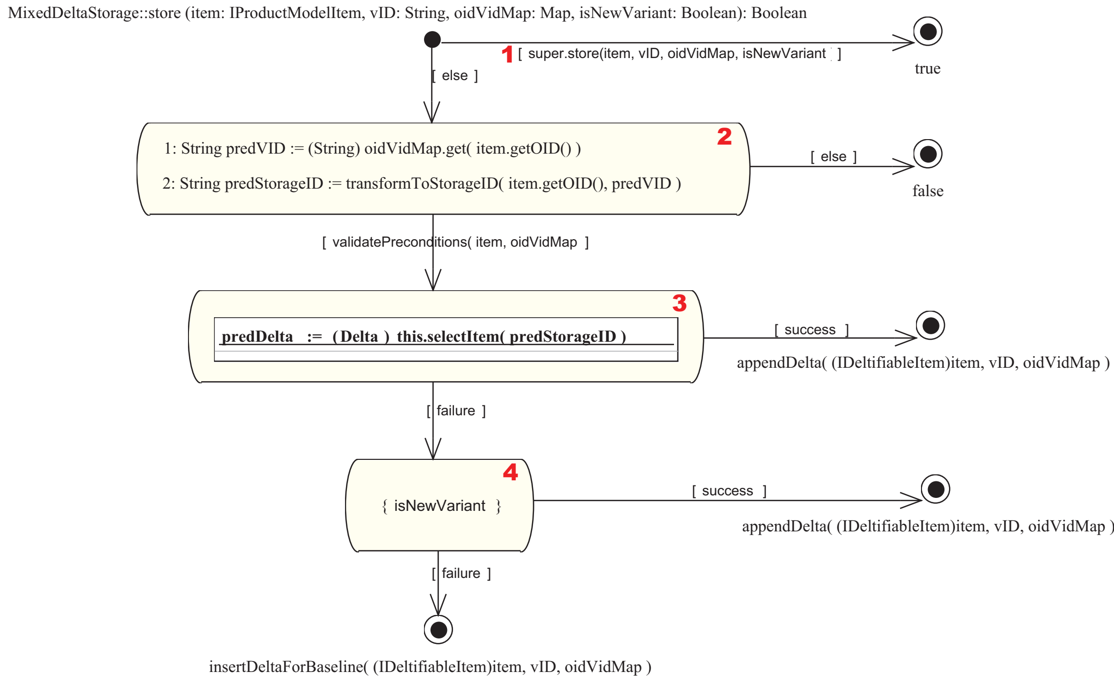


Abbildung 6.121.: Storydiagramm: store() des Speichers für gemischte Deltas

gerichtetes Delta ein eigenes Fabrik-Modul (*ForwardDeltaStorageModule*, *BackwardDeltaStorageModule* und *MixedDeltaStorageModule*), das für ein Repository den entsprechenden Delta-Speicher (*ForwardDeltaStorage*, *BackwardDeltaStorage* und *MixedDeltaStorage*) erzeugt (*createStorage()*). Jede dieser Klassen ist mit dem entsprechenden Merkmal markiert. Die drei Merkmale sind Teil der exklusiven Merkmalsgruppe „Gerichtete Deltas" (D.4.1.1) (Markierung: *directedDeltas*), so dass pro SKMS nur eine einzige Delta-Art verwendet werden kann. Unterschiedliche Repositorien mit verschiedenen Delta-Arten, auf einem MOD2-SKM Server, werden nicht unterstützt.

Die gemeinsamen Datenstrukturen und das gemeinsame Verhalten aller drei Delta-Arten ist durch die abstrakte Oberklasse *AbstractDeltaStorage* modelliert. Sie zeigt, dass sich die drei Verfahren sehr ähneln und nur beim Speichern neuer Produktmodell-Elemente (*store()*) unterscheiden. Jeder Delta-Speicher legt seine Elemente unter einer Speicherkennung ab (*AbstractStorageItem*) - entweder als Grundfassung (*Baseline*) oder als Delta (*Delta*). Jedes Delta besitzt ein Speicherelement als Vorgänger („hasNext"-Assoziation), entweder ein weiteres Delta oder eine Grundfassung. Eine Grundfassung verweist auf ihr Produktmodell-Element und lässt sich direkt wiederherstellen (*restore()*). Bei Deltas wird über die „hasNext"-Kanten die nächste Grundfassung gesucht und dann sukzessive die Deltas angewendet.

Die *store()*-Methode der drei Delta-Speicher unterscheidet sich daher bzgl. der Richtung, in der die „hasNext()"-Kanten angelegt werden. Bei den Vorwärts-Deltas übernimmt die ältere Version die „pred"-Rolle und die neuere die „succ"-Rolle. Bei den Rückwärts-

Deltas ist es genau umgekehrt. Bei den gemischten Deltas muss die Richtung abhängig vom Zweig bestimmt werden. Abbildung 6.121 zeigt das Storydiagramm zur Speicherung gemischter Deltas. Die durchnummerierten Storyaktivitäten modellieren folgendes Verhalten:

1. Die *store()*-Methode der Oberklasse wird aufgerufen. Sie modelliert den Sonderfall, dass noch keine Elemente im Speicher vorhanden sind. In diesem Fall legt sie eine initiale Grundfassung an und liefert den Wert „true“ zurück. Der Speichervorgang ist damit abgeschlossen.

2. Existieren schon Elemente im Speicher, wird die Speicherkennung der Vorgängerversion bestimmt (vgl. Abschnitt 6.3.11, S. 200ff.). Existiert der Vorgänger nicht oder implementiert das zu speichernde Elemente die Schnittstelle *IDeltifiable* (vgl. Abschnitt 6.3.2, S. 175ff.) nicht, bricht der Speichervorgang ab (*validatePreconditions()*).

3. Ansonsten wird geprüft, ob es sich beim Vorgänger-Element um ein Delta oder eine Grundfassung handelt. Bei einem Delta steht fest, dass ein neuer Zweig angelegt, bzw. ein existierender Zweig vergrößert wird. Ein Vorwärts-Delta wird angehängt (*appendDelta()*).

4. Handelt es sich dagegen um eine Grundfassung, ist nicht eindeutig festgelegt, ob die Grundfassung durch ein Rückwärtsdelta ersetzt (*insertDeltaForBaseline()*) oder ein neuer Zweig angelegt (*appendDelta()*) werden soll. Dies kann nur anhand des Parameters *isNewVariant* bestimmt werden, der von der Historie an den Speicher übergeben werden muss.

6.4.24. Modul: synchronisation.lockitem

Überblick

Das Modul *synchronisation.lockitem* modelliert einen Mechanismus zum Sperren ein oder mehrerer Produkt-Modellelemente (vgl. Abschnitt 4.4.1, S. 92ff.). Es kann sowohl für pessimistische Sperren als auch die interne Synchronisation parallel eintreffender Operationen eingesetzt werden.

Abhängigkeiten

Abbildung 6.122 zeigt die Abhängigkeiten von *synchronisation.lockitem*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Sperren“ (D.11.2.2) (Markierung: *lock*) Teil der Konfiguration ist. Als Synchronisations-Modul (*core.synchronisation*) ist es Komponente des Server-Teilsystems (*core.server*) und kann, bei Bedarf, Kommandos über das Kommunikations-Teilsystem (*core.communication*) empfangen.

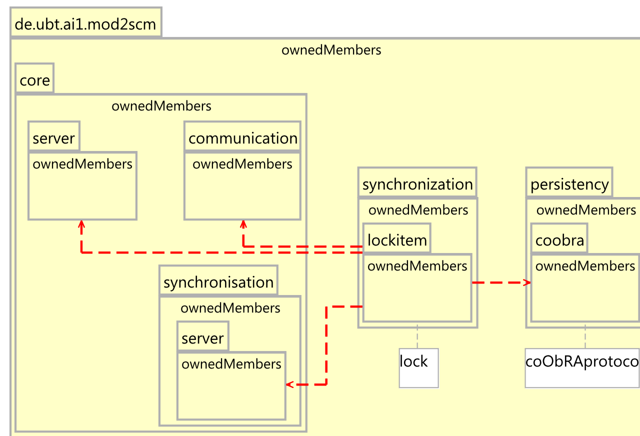


Abbildung 6.122.: Paketdiagramm: synchronization.lockitem

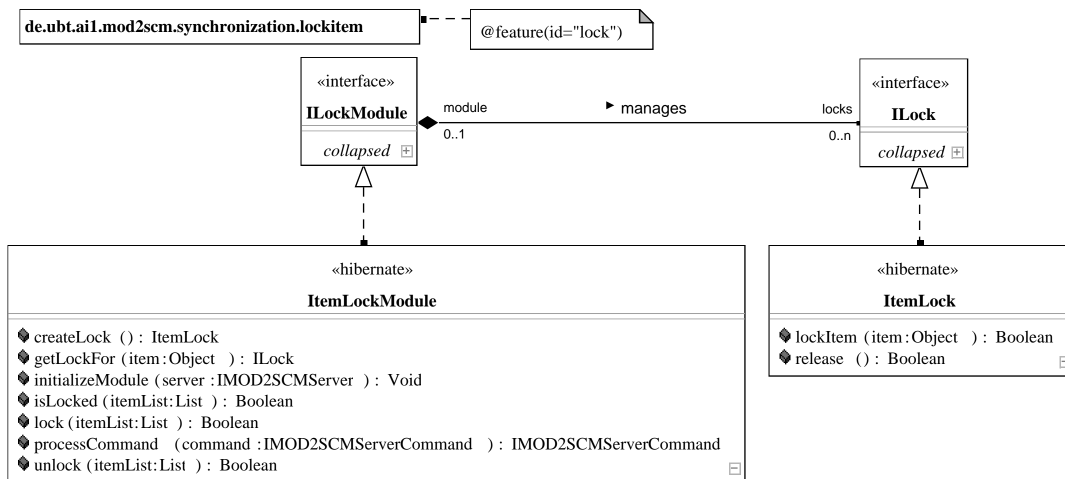


Abbildung 6.123.: Klassendiagramm: synchronization.lockitem

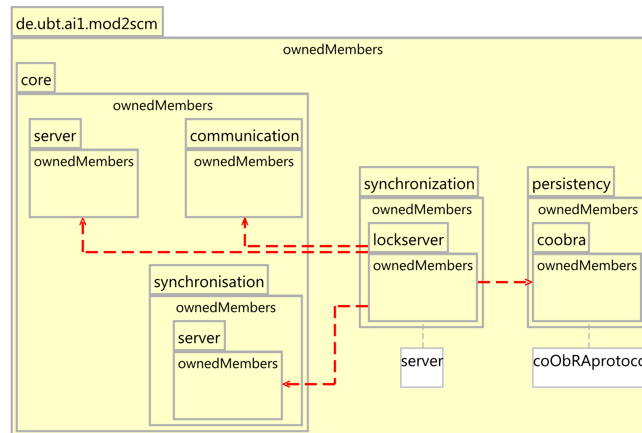


Abbildung 6.124.: Paketdiagramm: synchronization.lockserver

Detailbeschreibung

Abbildung 6.123 zeigt das Klassendiagramm von *synchronisation.lockitem*. Wie von den Schnittstellen des Kernmoduls *core.synchronisation* verlangt, existiert eine Modul-Fabrik für die Sperren (*ItemLockModule*). Sie bietet Operationen an, um eine Liste von Elemente zu sperren (*lock()*), ihren Status zu prüfen (*isLocked()*) und wieder zu entsperren (*unlock()*). Dies geschieht durch Anlegen (*lockItem()*) und Entfernen (*release()*) von Sperr-Objekten (*ItemLock*), die auf die entsprechenden Elemente verweisen. Im Falle von Produktmodell-Elementen wird die Objektkennung an dieses Modul übergeben.

6.4.25. Modul: synchronisation.lockserver

Überblick

Das Modul *synchronisation.lockserver* modelliert einen Mechanismus zum Sperren eines MOD2-SKM Servers. Es kann sowohl für pessimistische Sperren (vgl. Abschnitt 4.4.1, S. 92ff.) als auch die interne Synchronisation parallel eintreffender Operationen eingesetzt werden.

Abhängigkeiten

Abbildung 6.124 zeigt die Abhängigkeiten von *synchronisation.lockserver*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Server“ (D.11.1.2) (Markierung: *server*) Teil der Konfiguration ist. Als Synchronisations-Modul (*core.synchronisation*) ist es Komponente des Server-Teilsystems (*core.server*) und kann, bei Bedarf, Kommandos über das Kommunikations-Teilsystem (*core.communication*) empfangen.

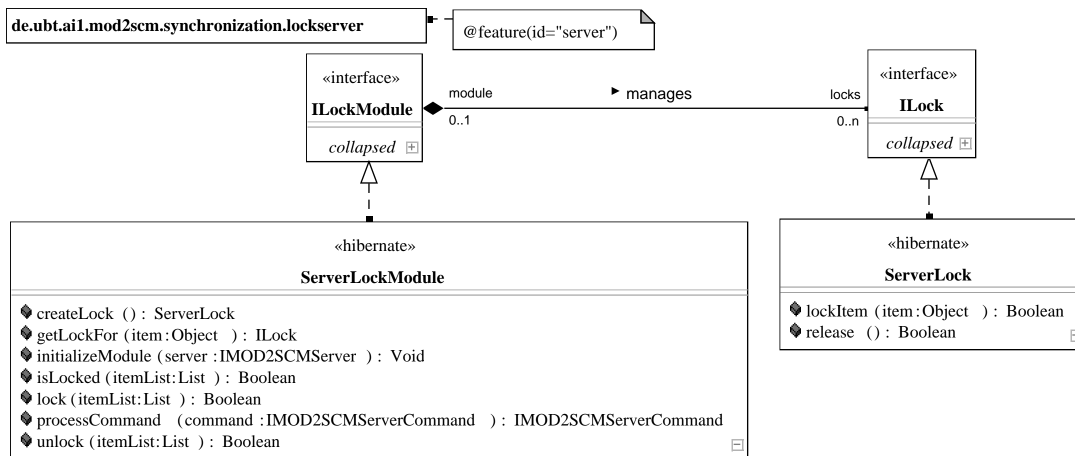


Abbildung 6.125.: Klassendiagramm: synchronization.lockserver

Detailbeschreibung

Abbildung 6.125 zeigt das Klassendiagramm von *synchronisation.lockserver*. Die Funktionalität entspricht der von *synchronisation.lockitem* (vgl. Abschnitt 6.4.24, S. 287ff.), nur müssen keine Elemente an das Modul übergeben werden. Stattdessen referenziert die Server-Sperre (*ServerLock*) das Server-Objekt, dem die Modul-Fabrik *ServerLockModule* zugeordnet ist („consists of"-Assoziation aus *core.server*, vgl. Abschnitt 6.3.10, S. 195ff.).

6.4.26. Modul: user.flat

Überblick

Das Modul *user.flat* modelliert eine einfache interne Benutzerverwaltung ohne Hierarchie oder Vergabe von Benutzerrechten. Es bleibt zukünftigen Arbeiten überlassen, Module für komplexere Verwaltungen oder die Integration externen Benutzerverwaltungen zu modellieren.

Abhängigkeiten

Abbildung 6.126 zeigt die Abhängigkeiten von *user.flat*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Intern" (B.5.1.2) (Markierung: *internalusermanagement*) Teil der Konfiguration ist. Als Benutzerverwaltungs-Modul (*core.user*) ist es Komponente des Server-Teilsystems (*core.server*), doch es existiert auch ein Modul im Arbeitsbereich (*core.workspace*) zur Verwaltung gesperrter Produktmodell-Elemente (*core.product*). Dieses kommuniziert über das Kommunikations-Teilsystem (*core.communication*) mit dem Server, um dort die Sperren (*core.synchronisation*) zentral zu verwalten. Laufzeitfehler werden über Ausnahmen behandelt (*core.exceptions*).

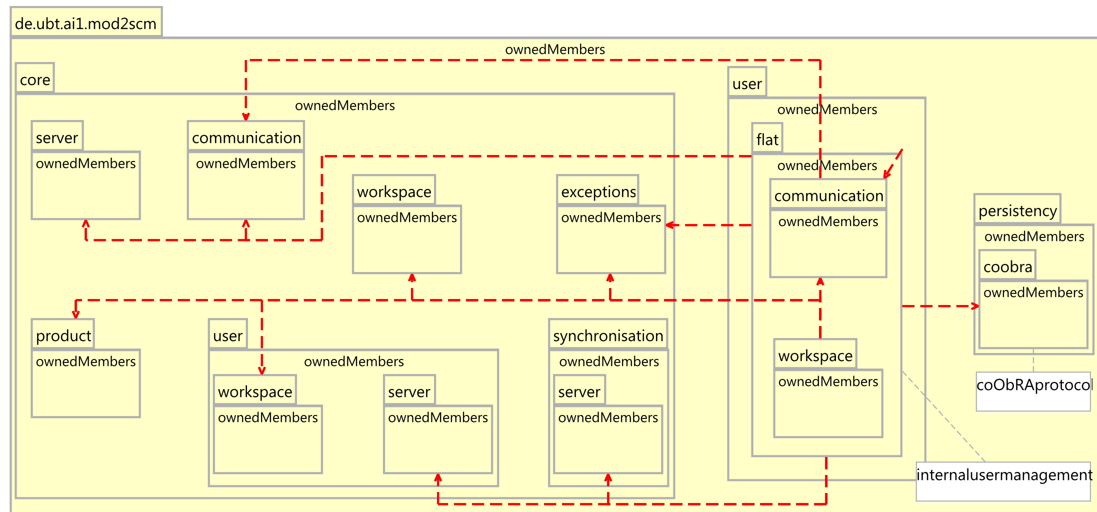


Abbildung 6.126.: Paketdiagramm: user.flat

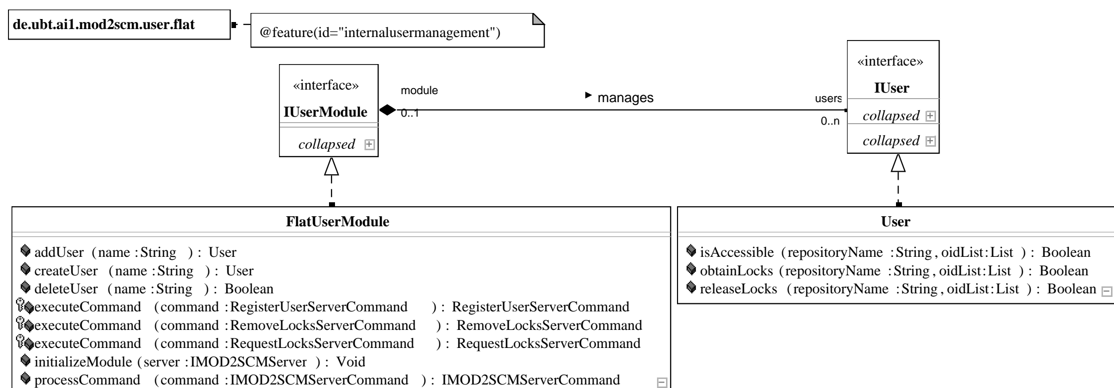


Abbildung 6.127.: Klassendiagramm: user.flat

Detailbeschreibung

Abbildung 6.127 zeigt das Klassendiagramm des Server-Teilsystems von *user.flat*. Das Fabrik-Modul (*FlatUserModule*) verwaltet die Benutzer-Objekte (*User*) und modelliert die vom Kernmodul (*core.user*, vgl. Abschnitt 6.3.13, S. 203ff.) verlangten Methoden zum Anlegen (*addUser()*) und Löschen (*deleteUser()*) von Benutzer. Jedes Benutzer-Objekt (*User*) bietet Operationen zum Anlegen (*obtainLocks()*), Prüfen (*isAccessible()*) und Entfernen (*releaseLocks()*) von pessimistischen Sperren. Die Operationen nutzen dafür das vom Server-Teilsystem verwendete Synchronisations-Modul über seine Kernmodul-Schnittstelle *core.synchronisation* (vgl. Abschnitt 6.3.12, S. 202ff.).

Die Verwaltung der Benutzer bzw. Sperren erfolgt über Kommandos des Kommunikations-Teilsystems (*core.communication*, vgl. Abschnitt 6.3.1, S. 173ff.). Abbildung 6.128 zeigt die drei Kommandos von *user.flat*. *RegisterUserServerCommand* registriert, auf dem

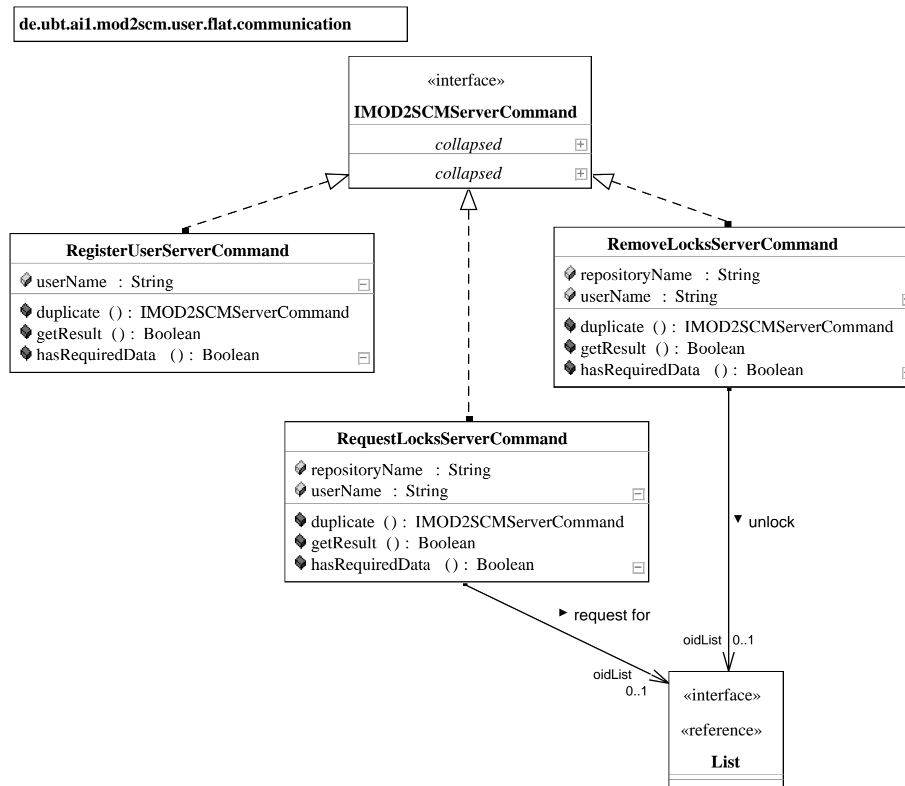


Abbildung 6.128.: Klassendiagramm: user.flat.communication

Server, den Benutzernamen (Attribut: *userName*) eines Entwicklers im Arbeitsbereich-Teilsystem. *RequestLocksServerCommand* versucht, eine Liste von Objektkennungen („request for“-Assoziation) für einen bestimmten Benutzer (Attribut: *userName*) zu sperren, während *RemoveLocksServerCommand* sie wieder freigibt.

Abbildung 6.129 zeigt das Klassendiagramm des Arbeitsbereich-Teilsystems. Das Arbeitsbereichs-Modul (*FlatUserWorkspaceModule*) sendet die o.g. Kommandos an das Server-Teilsystem, sobald der Entwickler Sperren anfordert (*requestLocks()*) oder freigibt (*removeLocks()*). Zusätzlich referenziert es alle Elemente der Produktmodell-Instanz im Arbeitsbereich, die zur Zeit gesperrt worden sind („is locking“-Assoziation). Diese werden nach Genehmigung einer Sperren-Anfrage an den Server (*requestLocks()* gibt „true“ zurück) angelegt (*processGrantedLockRequest()*).

6.4.27. Modul: workspace.base

Überblick

workspace.base modelliert einen MOD2-SKM Arbeitsbereich. Analog zum Modul *server.base* (vgl. Abschnitt 6.4.18, S. 254ff.), ist es die zentrale Komponente des Arbeitsbereich-Teilsystems und auch für die Instanzierung seiner Module zuständig.

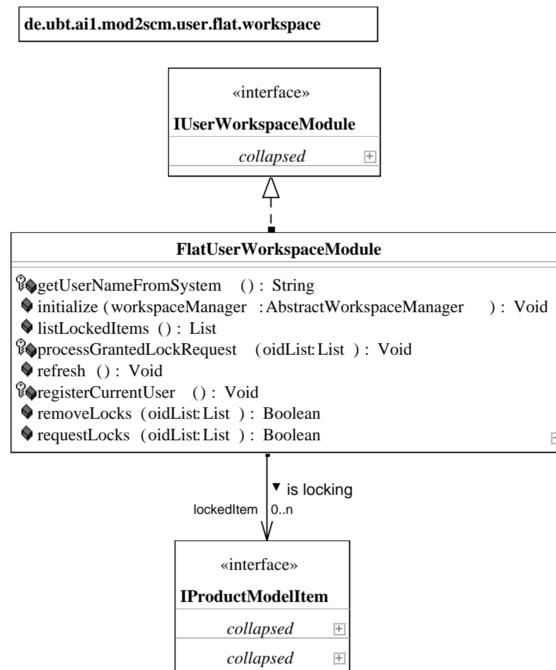


Abbildung 6.129.: Klassendiagramm: user.flat.workspace

Abhängigkeiten

Abbildung 6.130 zeigt die Abhängigkeiten von *workspace.base*. Es basiert auf dem Kernmodul für Arbeitsbereiche (*core.workspace*) und bietet eine Methode, die eine Anfrage an das Produktmodell (*core.product*) weiterleitet. Laufzeitfehler werden durch Ausnahmen (*core.exceptions*) behandelt. Außerdem ist *workspace.base* für die Initialisierung des Objektkennungs-Generators (*id*-Modul) zuständig.

Detailbeschreibung

Abbildung 6.131 zeigt das Klassendiagramm von *workspace.base*. Ein Großteil des Verhaltens wird durch Vererbung aus dem Kernmodul (*AbstractWorkspaceManager* aus *core.workspace*, vgl. Abschnitt 6.3.14, S. 206ff.) wiederverwendet. Lediglich die Persistenzierung der Versionsinformationen (*load()* und *save()*), sowie die Instanzierung der Arbeitsbereichs-Module (*initWorkspace()*) wird noch modelliert. Analog zu *server.base* (*initServer()*, vgl. Abschnitt 6.4.18, S. 254ff.), bestimmen die Merkmalsmarkierungen, welche Konstruktoraufrufe letztendlich in der konfigurierten Methode enthalten sind (s. Abb. 6.96, S. 256).

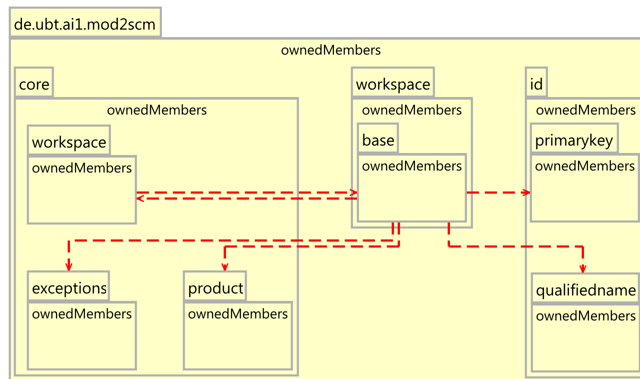


Abbildung 6.130.: Paketdiagramm: workspace.base

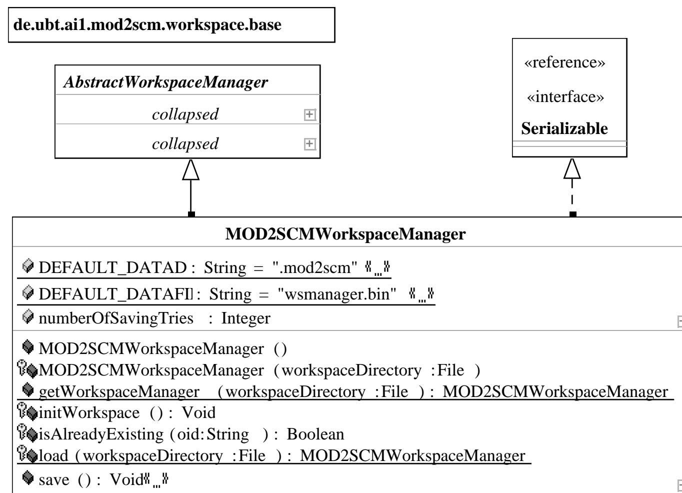


Abbildung 6.131.: Klassendiagramm: workspace.base

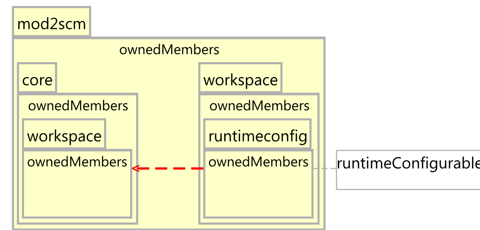


Abbildung 6.132.: Paketdiagramm: workspace.runtimeconfig

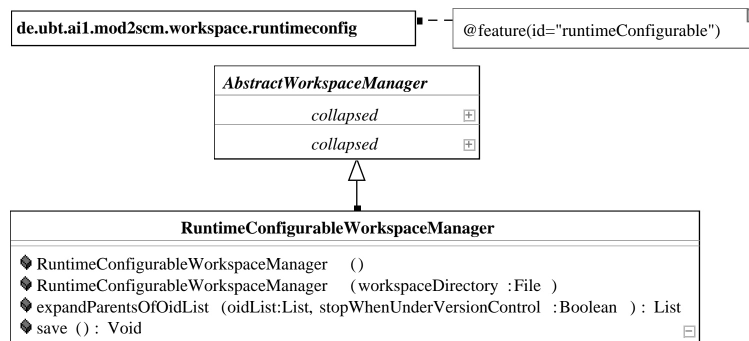


Abbildung 6.133.: Klassendiagramm: workspace.runtimeconfig

6.4.28. Modul: workspace.runtimeconfig

Überblick

Das Modul *workspace.runtimeconfig* modelliert einen Arbeitsbereich, um beim Testen auf einen laufzeitkonfigurierbaren Server zugreifen zu können (vgl. Abschnitt 6.4.20, S. 273ff.).

Abhängigkeiten

Abbildung 6.132 zeigt die Abhängigkeiten von *workspace.runtimeconfig*. Das komplette Paket ist nur Teil eines konfigurierten SKMS, wenn das Merkmal „Laufzeitkonfigurierbarer Server" (I.2) (Markierung: *runtimeConfigurable*) zur Konfiguration gehört. Als Arbeitsbereich basiert das Modul auf *core.workspace*.

Detailbeschreibung

Abbildung 6.133 zeigt das Klassendiagramm von *workspace.runtimeconfig*. Da dieser Arbeitsbereich nur in Testfällen verwendet wird, ist im Vergleich zu *workspace.base* wenig modelliert. Lediglich die Methode *expandParentsOfOidList()* passt ihr Verhalten an die aktuelle Art der Integration von Produkt- und Versionsraum an (vgl. Abschnitt 6.4.27, S. 292ff.).

6.5. Untersuchung der Architektur

Bei näherer Betrachtung der Architektur der MOD2-SKM Produktlinie stellen sich folgende Fragen:

1. Basiert jede Kopplung auf einer Abhängigkeit zwischen Konzepten der SKM-Domäne (und ist damit probleminhärent), oder existieren (vermeidbare) Kopplungen?
2. Werden Kopplungen des Domänenmodells auch im Merkmalsmodell beachtet, oder lassen sich Konfigurationen erstellen, welche die Kopplungen verletzen?
3. Ist die Architektur der MOD2-SKM Produktlinie modular, d.h., wurde das Architekturprinzip der losen Kopplung umgesetzt?
4. Enthält jedes Modul nur die Elemente, die es benötigt, oder sind voneinander unabhängige Komponenten in einem Modul zusammengefasst, d.h., wurde das Architekturprinzip der hohen Kohäsion umgesetzt?

Existieren keine unmotivierten Kopplungen, lässt sich folgende Annahme treffen:

> Durch Umsetzung des Prinzips „Separation of Concerns“ ist jedem Modul ein SKM-Konzept zugeordnet. Die Abhängigkeiten zwischen zwei Modulen – und auch ihr Kopplungsgrad – lassen sich auf die zugehörigen SKM-Konzepte übertragen: Je enger die Kopplung zwischen den Modulen, desto enger ist auch die Kopplung der Konzepte (und umgekehrt).

Damit ließen sich die Abhängigkeiten zwischen den SKM-Konzepten anhand der existierenden Kopplungen erkennen und analysieren.

Ignoriert das Merkmalsmodell die Kopplungen im Domänenmodell, dann lassen sich Konfigurationen erstellen, aus denen kein lauffähiges SKMS generiert werden kann. Zu einer Abhängigkeit zwischen zwei markierten Modell-Elementen im Domänenmodell sollte daher auch immer eine korrespondierende Abhängigkeit zwischen den entsprechenden Merkmalen existieren.

Um die Kopplung bzw. Kohäsion bewerten zu können, ist entweder ein Vergleich zu anderen Produktlinien oder ein absoluter Messwert notwendig. Ersteres ist sowohl auf Grund mangelnder Vergleichssysteme als auch des hohen Aufwands nicht möglich. Daher böte sich die Untersuchung anhand von Kopplungs- und Kohäsionsmetriken an.

Zunächst ist jedoch zu klären, wann überhaupt eine Kopplung vorliegt. Im allgemeinen wird jede „Beziehung zwischen Systembausteinen“ als Kopplung bezeichnet [VAC+09]. Für eine systematische Untersuchung der Architektur ist daher eine spezifischere Definition der Kopplung notwendig. Auch hier können die o.g. Kopplungsmetriken helfen, da zur Bewertung von Kopplungen eine exakte Definition notwendig ist [BDW98, VAC+09].

6.5.1. Existierende Metriken für Kopplung

Modularität und Kopplung wurde bereits in nicht objektorientierten Systemen analysiert (siehe z.B. [Par72, SMC74]). Die dort verwendeten Metriken sind daher ausführlich untersucht, ignorieren aber spezielle objektorientierte Konzepte wie z.B. Vererbung. Zu diesem

Zweck wurden objektorientierte Metriken entwickelt (s. [BDW98] für einen Überblick). Da MOD2-SKM objektorientierte Konzepte zur Modularisierung einsetzt, sollte eine objektorientierte Kopplungsdefinition als Grundlage dienen. Dabei stellten sich folgende Fragen: (1) Inwieweit sind existierende Metriken auf die Architektur von Modellen anwendbar? Und (2), sind evtl. für die Messung der Kopplung in Modellen eigene Metriken notwendig – ähnlich wie für objektorientierte Software-Systeme?

Die bereits existierenden objektorientierten Metriken erfassen qualitativ und quantitativ eine Reihe von Kriterien und präsentieren so unterschiedliche Sichten auf die Kopplung eines Software-Systems [BDW98]. Die folgenden beiden Aspekte einer Metrik sind zentral für die o.g. Fragestellungen bzgl. der Kopplungen in MOD2-SKM:

- Kopplungsmechanismus
- Kopplungsstärke

Ein **Kopplungsmechanismus** definiert, wann eine Kopplung vorliegt. Dies ist zum einen von den Komponenten des Software-Systems abhängig, zwischen denen die Kopplung untersucht wird. Zum anderen von den (syntaktischen) Beziehungen zwischen ihnen, die als Kopplung gewertet werden. I.d.R. basieren objektorientierte Kopplungsmechanismen auf Klassen und ihrer Verwendung in anderen Klassen, z.B. als Attributs- und Parameter-Typ oder als Ziel eines Methodenaufrufs. Die **Kopplungsstärke** bestimmt den Grad der Kopplung. Sie wird sowohl anhand der Anzahl der Beziehungen als auch des Typs der Beziehung gemessen. Für eine umfassendere Untersuchung und Klassifizierung von Kopplungsmetriken sei auf [BDW98] verwiesen.

Existierende Messwerkzeuge

Existierende Werkzeuge zur Berechnung der Metriken eines Software-Systems [BDW98] basieren i.d.R. auf dessen Quellcode. D.h. es ist notwendig, ihn zu analysieren und die Abhängigkeiten aus seinen Ausdrücken abzuleiten. Im Falle des MOD2-SKM Domänenmodells existieren die Abhängigkeiten bereits im Fujaba-Modell. Daher liegt es nahe, das Domänenmodell direkt zu analysieren, anstatt den generierten Quellcode zu verwenden. Für Fujaba existiert mit „Reclipse“ zwar ein Werkzeug zur Metrikberechnungen, jedoch nicht für Kopplungsmetriken [MN05].

6.5.2. Kopplung von Paketen

Existierende Metriken basieren i.d.R. auf Klassen und ihren Abhängigkeiten [BDW98]. Die Abhängigkeiten im MOD2-SKM Domänenmodell sind jedoch anhand der Pakete und ihren privaten Importbeziehungen beschrieben (vgl. Abschnitt 6.3, S. 171ff. und vgl. Abschnitt 6.4, S. 211ff.). Daher wäre es wünschenswert, eine Metrik zu benutzen, die Kopplungen auf Basis der Pakete untersucht. Dazu wird ein Paket a als Menge seiner Klassen $A_i, i \in \mathbb{N}$ aufgefasst:

$$a = \{A_1, A_2, \ldots, A_n\}$$

Die Mächtigkeit $|a|$ des Paketes a entspricht damit der Anzahl seiner Klassen. Eine Abhängigkeit zwischen zwei Paketen a und b lässt sich nun Hilfe der im Paket enthaltenen Klassen definieren:

> Ein Paket a ist genau dann von einem Paket b abhängig, wenn mindestens eine Klasse $A_i \in a, 1 \leq i \leq |a|$ existiert, die von einer beliebigen Klasse $B_j \in b, 1 \leq j \leq |b|$ abhängig ist.

Der **Kopplungsmechanismus** lässt sich somit auf Abhängigkeiten zwischen Klassen zurückführen. Diese können analog zu den Abhängigkeiten definiert werden, die für die Existenz privater Paketimport-Beziehung zuständig sind, da sich diese unmittelbar auf die im Paket enthaltenen Klassen zurückführen lassen [Buc10]:

> Eine Abhängigkeit einer Klasse A von einer Klasse B liegt genau dann vor, wenn
>
> 1. B ein Attributtyp von A ist,
> 2. B ein Rückgabetyp einer Methode von A ist (einschl. Ausnahmen),
> 3. B ein Parametertyp einer Methode von A ist,
> 4. B direkte Oberklasse von A ist,
> 5. B als Objekttyp in einem Storydiagramm von A verwendet wird.

Letzter Punkt deckt auch die Fälle ab, dass die paketfremden Klasse B der Typ einer lokalen Variablen in einer Methode von A ist. Die **Kopplungsstärke** $k(A, B)$ zwischen einer Klasse A und einer Klasse B lässt sich wie folgt berechnen:

$$\begin{aligned} k(A,B) &= |B \textit{ Attributtyp in } A| \\ &+ |B \textit{ Rückgabetyp in } A| \\ &+ |B \textit{ Parametertyp in } A| \\ &+ |B \textit{ Oberklasse von } A| \\ &+ |B \textit{ Objekttyp in } A| \end{aligned}$$

Wenn z.B. Klasse P drei Attribute vom Typ Q besitzt, dann beträgt die Kopplungsstärke $k(P, Q) = 3$. Diese Definition der Kopplungsstärke lässt sich nun auf die Kopplung von Paketen erweitern, indem die Abhängigkeiten zwischen allen Klassen dieser Pakete aufsummiert werden. Die Kopplungsstärke zwischen zwei Paketen a und b lässt sich damit wie folgt definieren:

$$k(a,b) = \sum_{x=1}^{|a|} \sum_{y=1}^{|b|} k(A_x, B_y)$$

Messen der Kohäsion

Außer einer losen Kopplung der Pakete, soll auch eine hohe Kohäsion vorliegen, d.h. die Elemente im Inneren der Pakete sollen „zusammengehören“. [VAC+09]. Die Kohäsion $K(a)$ eines Paketes a lässt sich ebenfalls mit Hilfe der Kopplungsstärke seiner Klassen definieren:

$$K(a) \quad = \quad k(a,a)$$

Die Kohäsion ist damit als Abhängigkeiten der Klassen eines Paketes untereinander definiert. Insbesondere werden hier auch die Abhängigkeiten einer Klasse von sich selbst gezählt.

Vergleich zu existierenden Metriken

Diese Messung der Abhängigkeiten ist hauptsächlich eine Zählung der Klassenverwendung. Dadurch lässt sich vor allem die Existenz von Kopplungen zeigen. Zusätzlich existiert die Möglichkeit einer rudimentären Erfassung der Kopplungsstärke. Im Vergleich zu existierenden objektorientierten Metriken werden mehrere Punkte ignoriert. Erstens fehlt eine Gewichtung der Stärke nach Beziehungstyp (Methodenaufruf, Vererbung, etc.) und zweitens wird auch die Kopplungsstärke nicht mit der Klassen- oder Paketgröße in Verbindung gebracht (vgl. [BDW98]). Im Rahmen dieser Arbeit kann jedoch aus Zeitgründen keine vollständige Metrik entwickelt und implementiert werden.

Durchführung der Messung und Grenzen

Da die Abhängigkeiten zwischen den Klassen analog zu den Voraussetzungen für private Paketimporte definiert wurden, kann der MODPL-Paketdiagrammeditor diese Daten erheben. So können die Messwerte anschließend aus dem Paketmodell ausgelesen werden. Dazu wird ein neues Paketmodell aus dem Fujaba-Modell erzeugt, das alle Pakete des Domänenmodells enthält. Als Nächstes werden die privaten Paketimporte automatisch erzeugt. Dabei wird auch – mit Hilfe einer manuellen Erweiterung des MODPL-Paketdiagrammeditors – die Kopplungsstärke für jede Abhängigkeit berechnet.

Das ursprüngliche Verfahren aus [Buc10] prüft für jede Beziehung zwischen zwei Klassen, ob bereits ein privater Paketimport vorhanden ist und legt diesen bei Bedarf an. Die Erweiterung weist neuen Paketimporten eine Kopplungsstärke von 1 zu. Und im Falle eines existierenden Imports erhöht sie dessen Kopplungsstärke um 1. Hierbei ist jedoch zu beachten, das die Sichtbarkeitsregeln einen Teil der Abhängigkeiten verdecken. Für Beziehungen zu Klassen aus Elternpaketen und dem Paket selbst sind keine privaten Paketimporte notwendig. Daher gehört zu der o.g. manuellen Erweiterung des MODPL-Paketdiagrammeditors auch eine Deaktivierung der Sichtbarkeitsregeln.

Abbildung auf private Paketimporte

Die Paketmodelle, die mit dem erweiterten Paketdiagrammeditor erstellt werden, lassen sich dennoch auf ein Paketdiagramm abbilden. Sie besitzen jedoch private Paketimporte, die auf Grund der Sichtbarkeitsregeln nicht notwendig sind. Allerdings lassen sich an solch einem Diagramm die Abhängigkeiten zwischen den Modulen einfach ablesen. Zusätzlich lässt sich noch die Kopplungsstärke auf die Liniendicke abbilden, d.h. ein Abhängigkeitspfeil ist umso dicker, je höher die Kopplungsstärke ist. Diese Kopplungsdiagramme werden im Folgenden zur Untersuchung und Visualisierung der Abhängigkeiten verwendet.

Unsichtbare textuelle Eingaben

Hier ist wichtig, anzumerken, dass die Analyse nicht alle Abhängigkeiten erkennen kann, da textuelle Eingaben in Storydiagrammen nicht analysiert werden. So können z.B. die Beziehungen zu Ausnahmen in „catch“-Anweisungs-Aktivitäten nicht erkannt werden (vgl. Abschnitt 6.1, S. 148ff.). Die Kopplung wird zwar erkannt – „geworfene“ Ausnahmen sind auch immer Teil der Methodensignatur – doch die Konstruktoraufrufe in den Anweisungen fließen nicht in die Berechnung der Kopplungsstärke mit ein. Das MOD2-SKM ist jedoch so modelliert, dass Anweisungs-Aktivitäten auf ein Minimum reduziert sind. Im Anbetracht dieser Einschränkung ist die Kopplungsstärke lediglich als Näherung zu betrachten. Doch die Implementierung einer eigenen Analyse-Software bzw. einer Anpassung des Fujaba-Metamodells liegen außerhalb des Rahmens dieser Arbeit.

Auf Grund der Einschränkungen lassen diese Werte keine absolute Bewertung der Architektur zu. Mit ihrer Hilfe lassen sich lediglich auffällige Pakete im MOD2-SKM Domänenmodell identifizieren. Die Untersuchung und Entwicklung von geeigneten modellgestützten Metriken und zugehörigen Evaluationswerkzeugen ließ sich im Rahmen dieser Arbeit nicht leisten. Sie sind jedoch zur Bewertung von Architekturen und Modellen unbedingt erforderlich und eine Aufgabe für zukünftige Arbeiten.

Grenzen quantitativer Messungen

Im Folgenden werden die Module und ihre Abhängigkeiten präsentiert und untersucht. Die o.g. Messwerte dienen dabei lediglich als Indikator für „interessante“ oder „kritische“ Abhängigkeiten. In MOD2-SKM existieren z.B. die Schnittstellen der Kernmodule, die speziell für Kopplungen vorgesehen sind. Eine Abhängigkeit dorthin ist z.B. einer Kopplung zwischen zwei Modulen der Modulbibliothek vorzuziehen. Es kann also zwischen „problematischen“ und „unproblematischen“ Kopplungen unterschieden werden. Diese Klassifikation ist von Architektur abhängig und erfordert ein Verständnis des Entwurfs. Diese Überlegungen müssten ebenfalls in das Messergebnis einfließen.

6.5.3. Module und ihre Abhängigkeiten

Die Messwerte im MOD2-SKM Domänenmodell basieren auf insgesamt 43 Modulen – 14 Kernmodulen und 29 Modulen aus der Modulbibliothek. Zwischen diesen existieren ins-

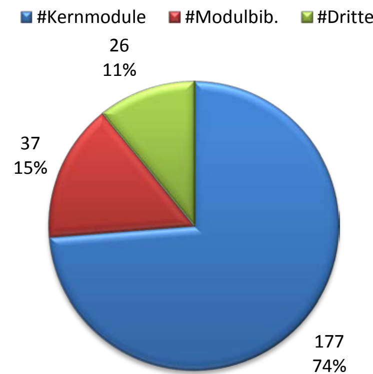


Abbildung 6.134.: Kategorie der abhängigen Module

gesamt 244 Abhängigkeiten. Abbildung 6.134 zeigt, dass 74% aller Abhängigkeiten von einem Kernmodul bestehen und 11% proprietäre Bibliotheken, die von Dritten implementiert wurden. 37 Abhängigkeiten (15%) sind Beziehungen zwischen Modulen der Modulbibliothek. Eine solche Abhängigkeit bedeutet: Obwohl beide Module die Schnittstelle zweier unterschiedlicher Kernmodule realisieren, lassen sie sich nicht mehr unabhängig voneinander gegen anderen Realisierungen der gleichen Schnittstelle austauschen. Da die Kernmodule gemäß den Aufgabenbereichen eines SKMS gebildet wurden (vgl. Abschnitt 4.1.2, S. 71ff.), ist eine Abhängigkeit zwischen zwei konkreten Modulen ein Hinweis auf eine unmittelbare Abhängigkeit zwischen den entsprechenden SKM-Konzepten. Sie sollte daher auch immer im Merkmalsmodell sichtbar sein (z.B. über aussagenlogische Einschränkungen, vgl. Abschnitt 5.2.1, S. 102ff.).

Abbildung 6.135 bildet die Abhängigkeiten zwischen den Modulen der Modulbibliothek ab. Sie zeigt, dass folgende Module eine direkte Abhängigkeit zu einem Modul aus der Modulbibliothek besitzen[5]:

history.base (1)
history.flat (1)
history.tree (1)
merge.workspace (1)
repository.complexitem (2)
repository.singleitem (1)
server.base (15)
server.rmi (3)
server.runtimeconfig (3)
storage.base (1)
storage.complex (2)
storage.delta (1)
synchronization.lockitem (1)
synchronization.lockserver (1)
user.flat (1)
workspace.base (2)

Das Servermodul *server.base* besitzt Abhängigkeiten zu 15 verschiedenen Modulen (in Abb. 6.135 mit „a)“ gekennzeichnet). Grund ist die Initialisierung der Fabrikklasse des jeweiligen Moduls, so dass sich alle anderen Objekte über deren Schnittstelle erzeugen lassen (vgl. Abschnitt 6.4.18, S. 254ff.). Das Servermodul selbst, und alle seine Module, die persistenziert werden sollen, müssen bei Verwendung des CoObRA2-Persistenzmechanismus (*persistency.coobra*), von einer speziellen Basisklasse abgeleitet

[5] Die Zahl in Klammern gibt die Anzahl der Abhängigkeiten an.

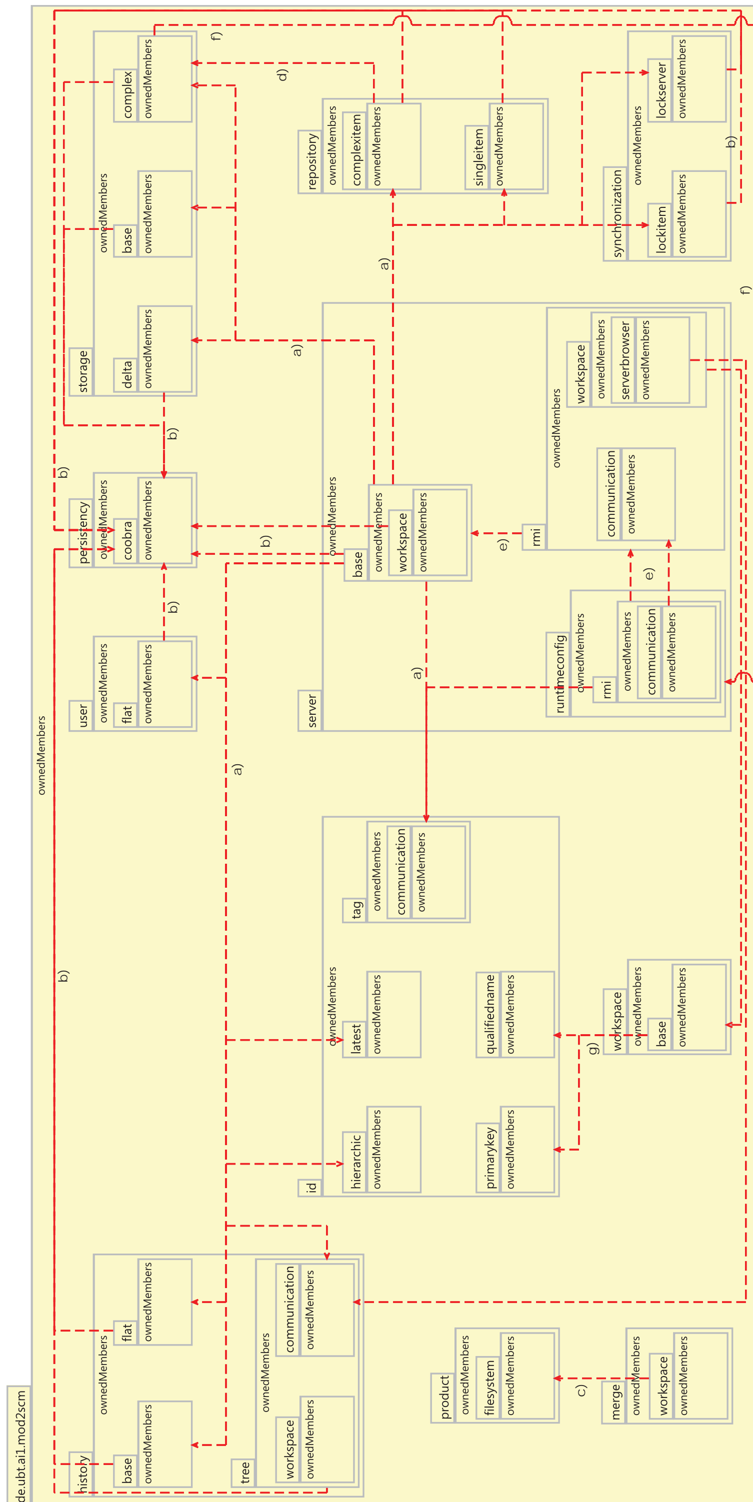


Abbildung 6.135.: Abhängigkeiten in der Modulbibliothek

werden (vgl. Abschnitt 6.4.11, S. 235ff.). Daher sind folgende Module vom CoObRA-Modul abhängig (in Abb. 6.135 mit „b)“ gekennzeichnet): Die Historien-Module (*history.base*, *history.flat*, *history.tree*), die Repositoriums-Module (*repository.complexitem*, *repository.singleitem*), die Speichermodule (*storage.base*, *storage.complex*, *storage.delta*), die Synchronisations-Module (*synchronization.lockitem*, *synchronization.lockitem*) und die Benutzerverwaltung (*user.flat*). Damit bleiben nur 9 Abhängigkeiten, die nicht mit dem Server- oder dem CoObRA-Modul zu tun haben. Die Module, von denen diese Abhängigkeiten ausgehen, sind in der obigen Aufzählung unterstrichen.

Das Verschmelzungs-Modul (*merge.workspace*, vgl. Abschnitt 6.4.10, S. 233ff.) ist vom Dateisystem-Produktmodell (*product.filesystem*, vgl. Abschnitt 6.4.14, S. 240ff.) abhängig („c)“ in Abb. 6.135). Grund sind die Verschmelzungs-Verfahren, die nur auf konkrete Produktmodell-Elemente (bzw. sogar nur bestimmten Untertypen) anwendbar sind [Bär10]. Eine ähnliche Abhängigkeit vom Typ des Produktmodell-Elements, existiert auch bei den Delta-Algorithmen (vgl. Abschnitt 6.3.2, S. 175ff.), so dass die Abhängigkeit auch analog aufgelöst werden kann: In zukünftigen Arbeiten sollte daher eine Schnittstelle *IMergeable* eingeführt werden. Diese verlangt eine Methode zum Verschmelzen, die dann von allen Produktmodell-Elementen implementiert wird und den passenden Algorithmus enthält.

Das Repositorium für zusammengesetzte Elemente (*repository.complexitem*, vgl. Abschnitt 6.4.16, S. 251ff.) ist vom Speicher für zusammengesetzte Elemente (*storage.complexitem*, vgl. Abschnitt 6.4.22, S. 280ff.) abhängig („d)“ in Abb. 6.135). Zusammengesetzte Elemente lassen sich zwar auch in einem Speicher für atomare Elemente ablegen, doch dann wird das Element, mit all seinen Teilelementen, als unteilbares Ganzes betrachtet. Die Teilelemente lassen sich so nicht mehr explizit referenzieren, was aber für eine versionszentrierte oder verschränkte Integration von Produkt- und Versionsraum nötig ist (vgl. Abschnitt 4.3.4, S. 81ff.). Daher muss in diesem Fall immer das Speichermodul *storage.complex* verwendet werden. Diese Abhängigkeit ist über aussagenlogische Einschränkungen im Merkmalsmodell modelliert (*Klausel 11* und *12*,vgl. Abschnitt 5.6.2, S. 137ff.).

Der RMI-Server (*server.rmi*, vgl. Abschnitt 6.4.19, S. 271ff.) und der laufzeitkonfigurierbare Server (*server.runtimeconfig*, vgl. Abschnitt 6.4.20, S. 273ff.) sind beide vom einfachen Servermodul (*server.base*, vgl. Abschnitt 6.4.18, S. 254ff.) abhängig („e)“ in Abb. 6.135). Ein RMI-Server unterscheidet sich lediglich bei der Erstellung des Server-Objektes und beim Ausführen der Kommandos von einem einfachen Server. Auch für die Laufzeitkonfiguration ist lediglich eine neue Methode zur Initialisierung der Module notwendig, so dass die restlichen Methoden durch Vererbung wiederverwendet werden (vgl. Abschnitt 6.4.19, S. 271ff. und vgl. Abschnitt 6.4.20, S. 273ff.). Der komplexe Speicher (*storage.complex*, vgl. Abschnitt 6.4.22, S. 280ff.) besitzt wiederum eine Abhängigkeit („f)“ in Abb. 6.135) zum laufzeitkonfigurierbaren Server (*server.runtimeconfig*, vgl. Abschnitt 6.4.20, S. 273ff.), da dort zwischen versionszentrierter und verschränkter Integration unterschieden wird. Im Falle einer Laufzeitkonfiguration ist daher die Information nötig, für welche Integrationsform der Server gerade konfiguriert ist. Über das gleiche Merkmal, das den laufzeitkonfigurierbaren Server auswählt (Merkmal „Laufzeitkonfigurierbarer Server“ (I.2)) wird auch gesteuert, ob die laufzeitkonfigurierbare Variante des

komplexen Speichers verwendet wird.

Das einfache Arbeitsbereich-Modul (*workspace.base*, vgl. Abschnitt 6.4.27, S. 292ff.) muss – analog zum Server-Modul *server.base* (s.o.) – ebenfalls Fabriken für seine Identifikations-Module anlegen, so dass hier eine Abhängigkeit zu jedem Identifikations-Modul existiert („g)“ in Abb. 6.135). Auch hier könnte dann, zur Minimierung der Kopplungsstärke, eine spezielle Fabrik für Identifikations-Module eingeführt werden.

Nur zwei der oben erläuterten Abhängigkeiten schränken die Kombinationsmöglichkeiten der Module ein: Erstens die Beziehung zwischen Repositorium und Speicher für komplexe Elemente („d)“ in Abb. 6.135), und zweitens die Beziehung zwischen Verschmelzungsmodul und dem Modul für das Dateisystem-Produktmodell („c)“ in Abb. 6.135). Ansonsten lassen sich alle Realisierungen eines Kernmoduls unabhängig von den anderen Modulen austauschen. Erstere ist im Merkmalsmodell modelliert (*Klausel 11* und *12*,vgl. Abschnitt 5.6.2, S. 137ff.), letztere muss noch hinzugefügt werden.

Die Abhängigkeit bei den zusammengesetzten Elementen ist notwendig, um beim Einspielen von Elementen in das Repositorium sicherzustellen, dass auch evtl. existierende Elternelemente eine neue Versionsnummer erhalten. Eine Auflösung dieser Abhängigkeit ist nur dann möglich, wenn sichergestellt ist, dass immer das Wurzelelement des Repositoriums bei einer Änderungen übertragen wird. Dann darf jedoch kein Arbeitsbereich existieren, der nur einen Teilbaum der gespeicherten Daten enthält. Dieser Unterschied betrifft z.B. die SKMS „Subversion“ und „GIT“ [CSFP08, Ca09]. Die Abhängigkeit zwischen Verschmelzungsmodul und Produktmodell entsteht durch die Verschmelzungs-Algorithmen, die unmittelbar vom Inhalt des Elements abhängen und so – ähnlich wie die Delta-Algorithmen – für jeden Typ von Produktmodell-Element implementiert werden müssen.

Untersuchung der Messergebnisse

Die geringe Anzahl von Abhängigkeiten zwischen den Modulen der Modulbibliothek zeigt, dass nur wenig Kopplungen zwischen den unterschiedlichen Realisierungen der Kernmodule bestehen. Mit Hilfe der in Abschnitt 6.5.2 beschriebenen Messungen lassen sich nun die Beziehungen zwischen den konkreten Modulen und den Kernmodulen, sowie die Abhängigkeiten der Kernmodule untereinander untersuchen.

Abbildung 6.136 und Abbildung 6.137 zeigen ein Balkendiagramm, das folgende drei Werte pro Modul darstellt: (1) Die Anzahl der Abhängigkeiten (*blau*), (2) die maximale externe Kopplungsstärke (*rot*) und (3) die interne Kopplungsstärke (*grün*).

Die **Anzahl der Abhängigkeiten** $k_{anz}(a)$ eines Moduls a wird berechnet, indem alle Module b gezählt werden, zu denen eine Abhängigkeit existiert (d.h. es existiert mindestens eine Abhängigkeit zwischen einer Klasse des Moduls a und des Moduls b, einschließlich Klassen in Unterpaketen):

$$k_{anz}(a) \quad = \quad |\{b \in Module | k(a,b) > 0\}|$$

Die maximale externe Kopplungsstärke $k_{max}(a)$ eines Moduls a wird berechnet, indem das Maximum aus allen Kopplungsstärken zu beliebigen anderen Modulen b, $k(a,b)$,

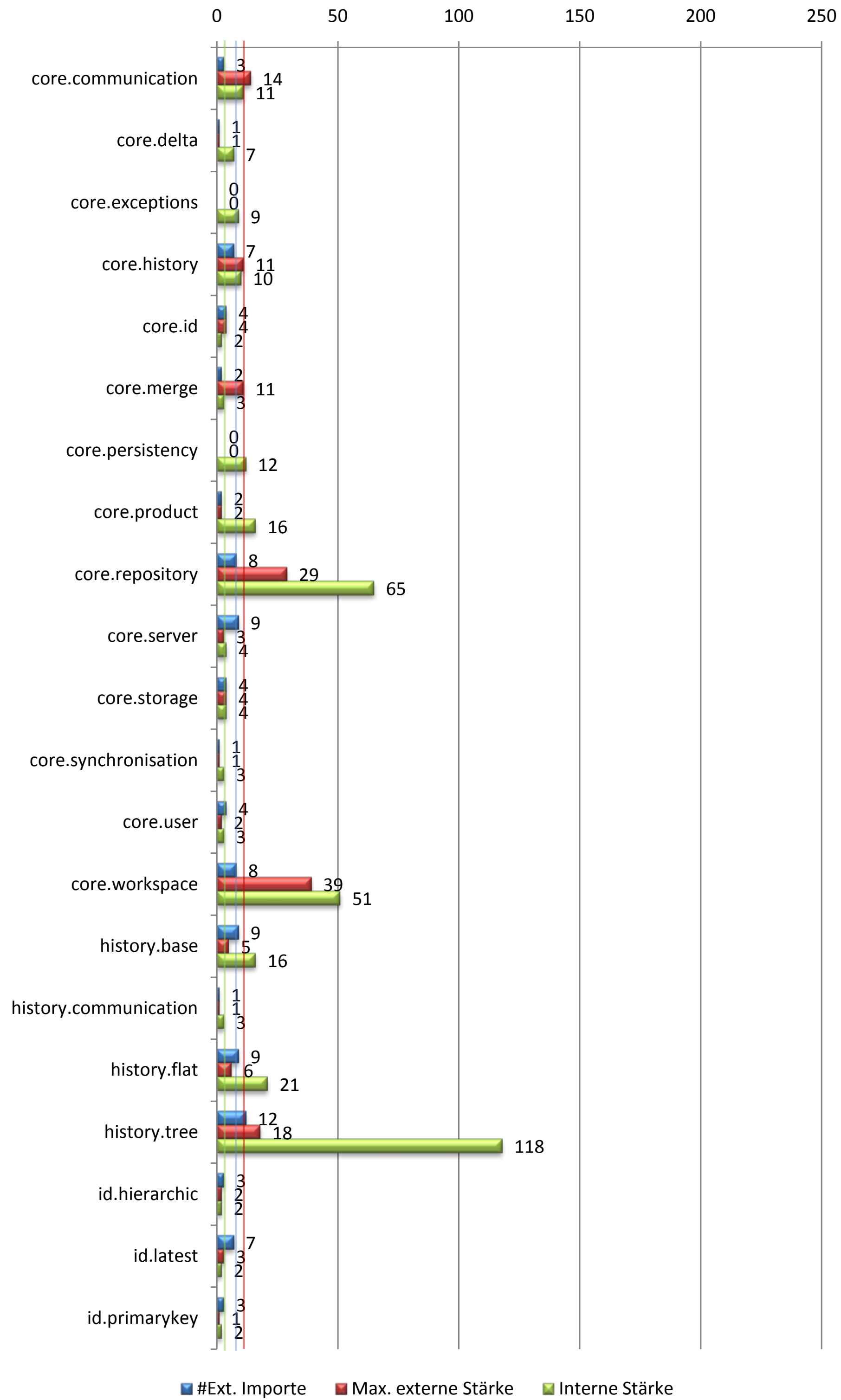


Abbildung 6.136.: Kopplung und Kohäsion je Modul – Teil 1

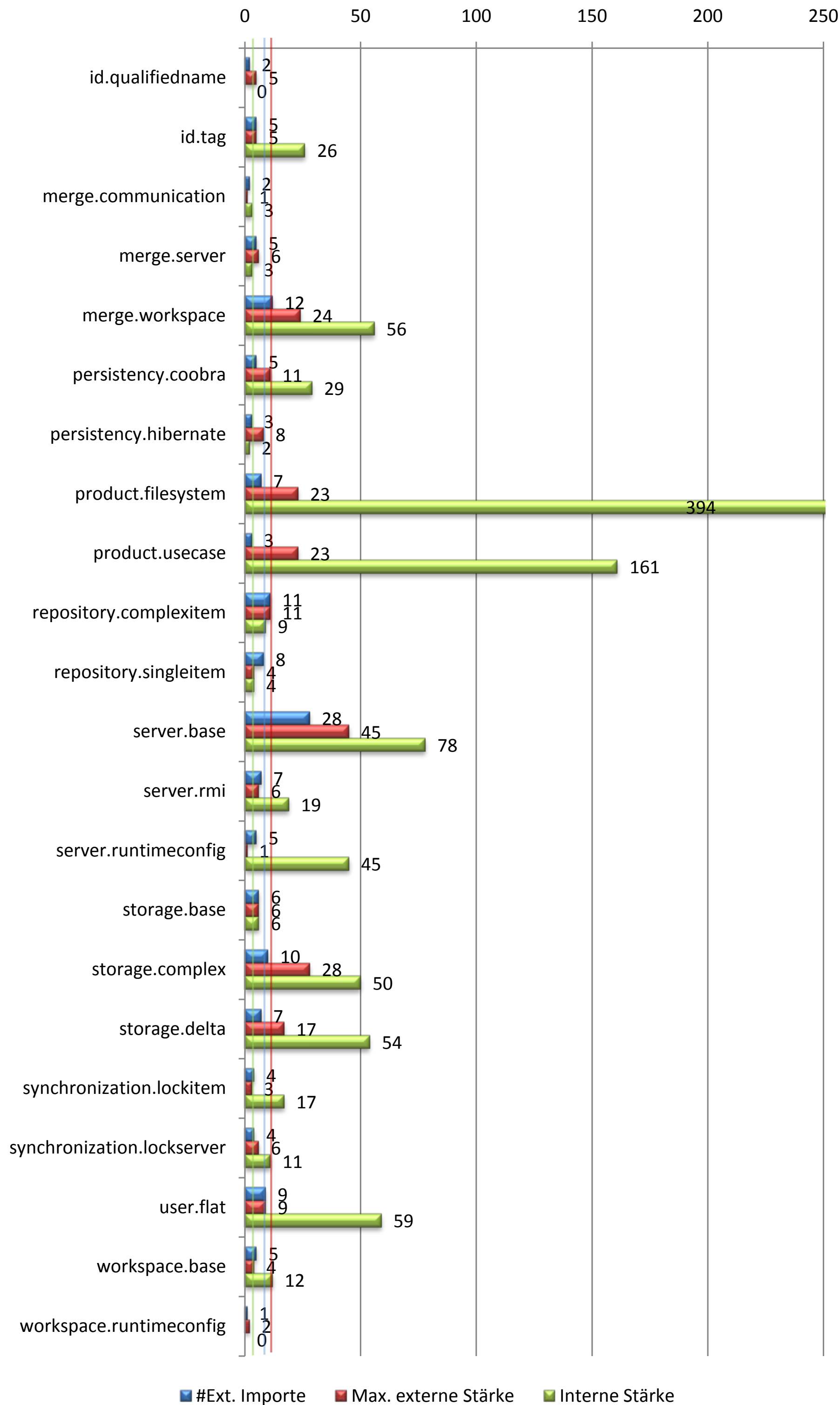


Abbildung 6.137.: Kopplung und Kohäsion je Modul – Teil 2

Maß	1. Quartil	2. Quartil (Median)	3. Quartil	4. Quartil
#Ext. Abhängigkeiten	3	5	8	28
Max. ext. Stärke	2	5	11	45
Interne Stärke	3	11	37	394

Tabelle 6.5.

gewählt wird:

$$k_{max}(a) \quad = \quad max(\{k(a,b)|a \neq b,\ b \in Module\})$$

Die **interne Kopplungsstärke** (Kohäsion) $K(a)$ eines Moduls a wird wie in Abschnitt 6.5.2 berechnet, indem alle Abhängigkeiten zwischen den Klassen des Pakets gezählt werden (einschließlich Klassen in Unterpaketen):

$$K(a) \quad = \quad k(a,a)$$

Die Module sind in aufsteigender Reihenfolge alphabetisch sortiert. Dabei sind Pakete externer Bibliotheken zwar als Ziel von Abhängigkeiten mitgezählt worden, aber nicht im Balkendiagramm dargestellt, da diese nur als Referenz im Fujaba-Modell existieren (vgl. Abschnitt 6.1, S. 144ff.) und die Beziehungen ihrer Elemente nicht Teil des Domänenmodells sind. Das Gleiche gilt für zwei Module des Domänenmodells, die in eigenen Projekten entwickelt wurden (*id.hash* und *storage.blobtree*) und sich auf Grund beschädigter Modelldateien nicht evaluieren ließen. Beide Projekte können nur zur Codeerzeugung genutzt werden.

Anhand der Messwerte ist es nicht möglich, eine Klassifizierung für „gute“ oder „schlechte“ Werte anzugeben, da weder eine Skala noch Vergleichsprojekte existieren. Ein statistischer Vergleich der Kopplung des MOD2-SKM Domänenmodells mit anderen Domänenmodellen liegt außerhalb des Rahmens meiner Arbeit. Daher vergleiche ich lediglich die Messwerte für Kopplung und Kohäsion der einzelnen Module untereinander. So lassen sich Extremwerte identifizieren und diese Auffälligkeiten untersuchen: Diese fallen in den Abbildungen 6.136 und 6.137 bereits optisch durch ihre hohe Balkenlänge auf.

Zur systematischen Untersuchung der Messwerte lassen sich die Quartile der jeweiligen Messwerte einsetzen. Sie zeigen die Intervallgrenzen, zwischen denen sich der Messwert für jeweils ein Viertel der Module bewegt. Tabelle 6.5 zeigt die Quartile aller drei Messwerte. Das letzte Intervall ist deutlich höher als die vorhergehenden, d.h. ein großer Teil der Module besitzt relativ ähnliche Messwerte, von denen einige wenige Module deutlich abweichen (was den optische Eindruck des Balkendiagramms bestätigt).

Als Grenze für Extremwerte der Kopplung wurde das dritte Quartil[6] gewählt, d.h. Module, die diesen Wert überschreiten, werden nun genauer untersucht, da sie entweder

[6] 75% der Werte sind kleiner

von vielen Modulen abhängen oder an ein oder mehrere Module stark gekoppelt sind. Die 3. Quartile sind auch in Abbildung 6.136 und 6.137 als blaue (externe Abhängigkeiten) bzw. rote, (Max. externe Kopplungsstärke) senkrechte Linie eingetragen.

Bei der Untersuchung der Kohäsion wird im Gegenzug das 1. Quartil als Grenzwert gewählt, da hier Module mit besonders niedriger Kohäsion untersucht werden sollen. D.h. es sind besonders die Module interessant, bei denen intern weniger als drei Beziehungen existieren. Das 1. Quartil ist als grüne, senkrechte Linie in den Balkendiagrammen eingezeichnet.

Identifikation der auffälligen Module

Bei der Kopplung lässt sich zum einen die Anzahl der Kopplungen und zum anderen die Stärke der Kopplungen untersuchen. Im ersten Fall besitzt ein Modul auffällig viele Import-Beziehungen zu anderen Modulen, im zweiten eine extrem starke Beziehung zu (mindestens) einem anderen Modul. Folgende Module besitzen mehr als acht Abhängigkeiten von anderen Modulen (und überschreiten damit das 3. Quartil der Anzahl der Abhängigkeiten):

core.server (9)
history.base (9)
history.flat (9)
history.tree (13)
merge.workspace (12)
repository.complexitem (11)
server.base (28)
storage.complex (10)
user.flat (9)

Unterstrichene Module besitzen sowohl auffällig viele als auch besonders starke Kopplungen. Die Module mit einer auffällig hohen externen Kopplungsstärke (größer als 11) sind:

core.communication (14)
core.repository (29)
core.workspace (39)
history.tree (18)
merge.workspace (24)
product.filesystem (23)
product.usecase (23)
server.base (45)
storage.complex (28)
storage.delta (17)

Außerdem existiert eine Reihe von Modulen, deren Kohäsion geringer als das 1. Quartil von drei ist:

core.id
id.hierarchic
id.latest
id.primarykey
id.qualifiedname
persistency.hibernate

Viele und starke Kopplungen: *server.base*

Das Modul *server.base* (vgl. Abschnitt 6.4.18, S. 254ff.) ist mit 28 externen Abhängigkeiten und einer maximalen Kopplungsstärke von 45 das mit Abstand komplexeste Modul des MOD2-SKM Domänenmodells. Dies zeigt auch dessen Schnittstelle *core.server*, die als einziges Kernmodul durch viele Abhängigkeiten auffällt. Abbildung 6.138 zeigt das

Paketdiagramm mit den Abhängigkeiten von *server.base*. Die Linienstärke ist abhängig von der Kopplungsstärke, d.h. eine dickere Linie bedeutet mehr Beziehungen, die den Import benötigen. Die stärkste Abhängigkeit besteht zwischen den Arbeitsbereich-Komponenten und dem zugehörigen Kernmodul *core.workspace* (vgl. Abschnitt 6.3.14, S. 206ff.). Auch zu den Kernpaketen für das Produktmodell (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.) und für die Ausnahmen (*core.exceptions*, vgl. Abschnitt 6.3.3, S. 177ff.) sind starke Abhängigkeiten zu erkennen. Auffällig ist die große Anzahl an schwachen Abhängigkeiten zu den 15 Modulen der Modulbibliothek. Grund hierfür ist die Initialisierung der Modulfabriken, die die Objekte des Moduls unabhängig vom konkreten Typ erzeugen: Die Fabrik muss als konkreter Typ erzeugt werden (s. Methode *initServer()*, vgl. Abschnitt 6.4.18, S. 254ff.). Eine mögliche Reduzierung dieser Abhängigkeiten bestünde im Modellieren einer eigenen Fabrikklasse pro Kernmodul, die dann jeweils für die entsprechenden konkreten Module zuständig ist, d.h. eine Fabrik für die *history*-Module der Modulbibliothek, eine für die *storage*-Module, usw.

Kopplung in Modulbibliothek: *repository.complex* und *storage.complex*

Auch der Speichermechanismus für zusammengesetzte Produktmodell-Elemente (*storage.complex*, (vgl. Abschnitt 6.4.22, S. 280ff.)) fällt durch besondere viele und starke Kopplung auf – ebenso wie das zugehörige Repositorium (*repository.complex*, (vgl. Abschnitt 6.4.16, S. 251ff.)). Das Repositorium ist auch direkt an das konkrete Modul für den Speichermechanismus gekoppelt (s. Abb.6.135, S. 302) und daher besitzen beide auch mehr Kopplungen als andere Speichermodule.

Zentrale Datenstruktur: *history*-Module

Alle drei Historien-Module (*history.base*, *history.flat* und *history.tree*) besitzen viele Abhängigkeiten. Grund ist die Funktion der Historie als zentrale Datenstruktur des Versionsraums (vgl. Abschnitt 4.3.1, S. 79ff.). Jede Historie benötigt Elemente in allen drei Teilsystemen – Arbeitsbereich, Kommunikation und Server. Im Arbeitsbereich muss ein Modul existieren, mit dem die Entwickler Informationen aus der Historie abrufen können. Dazu existieren Kommandos, die an den Server übertragen werden und aus den dort vorhandenen Datenstrukturen die Informationen auslesen. Zusätzlich ist es noch notwendig Informationen aus anderen Kernmodulen in der Historie zu speichern, wie z.B. den Benutzernamen (vgl. Abschnitt 6.3.4, S. 178ff.).

Häufig importiert: *core.exceptions* und *core.product*

Abbildung 6.139 zeigt alle Abhängigkeiten, deren Kopplungsstärke über 11 liegt (dem 3. Quartil). Die Abbildung zeigt, dass ein Kernmodul meist Ziel mehrerer starker Importe ist, d.h. die Elemente des Zielmoduls werden von mehreren Modulen benötigt. Hierbei fallen insbesondere das Ausnahmen-Kernmodul (*core.exceptions*, vgl. Abschnitt 6.3.3, S. 177ff.) und das Produktmodell-Kernmodul (*core.product*, vgl. Abschnitt 6.3.8, S. 188ff.) auf. Ersteres wird von allen Modulen benötigt, bei denen Ausnahmen zur

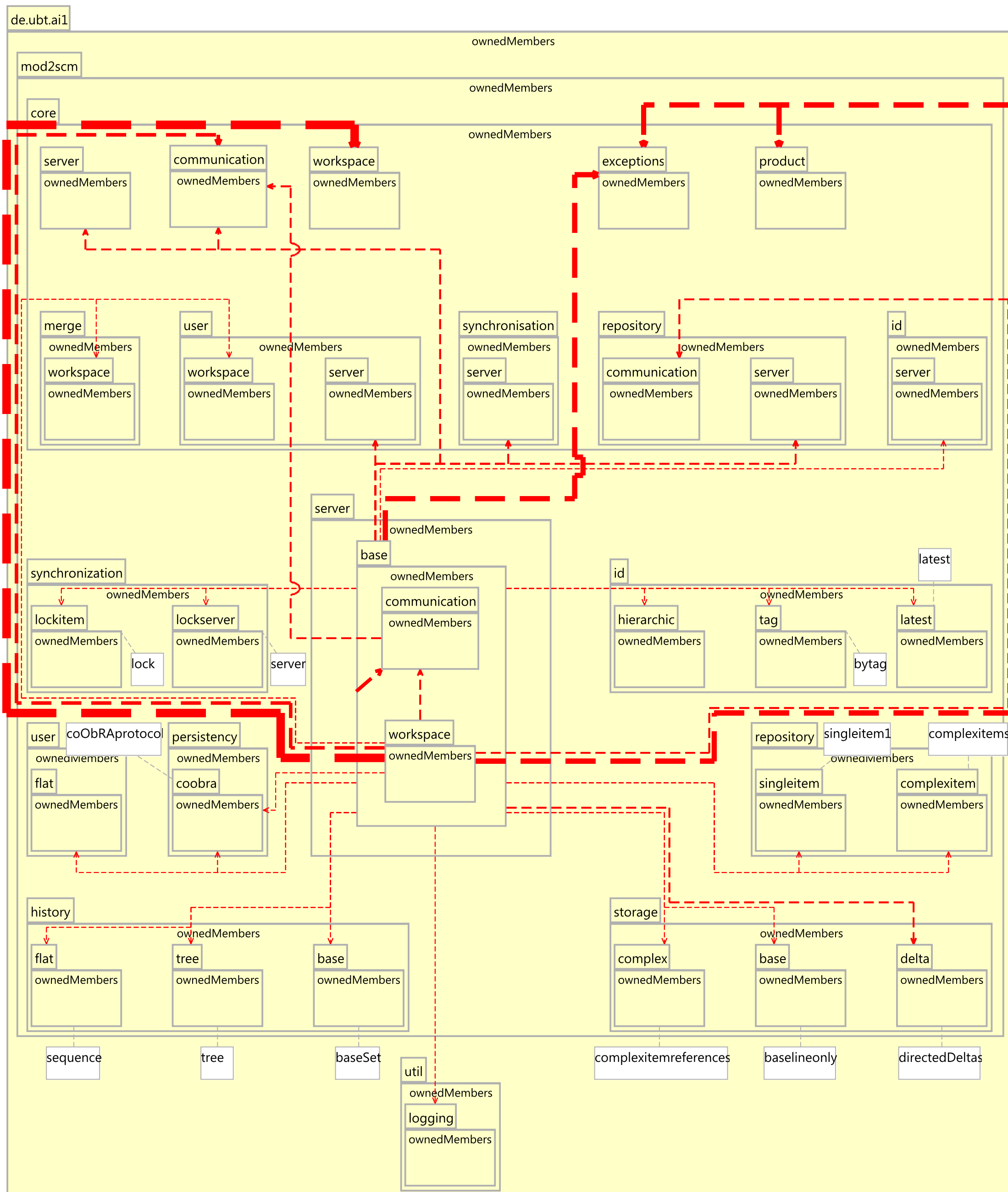


Abbildung 6.138.: Abhängigkeiten von *server.base*

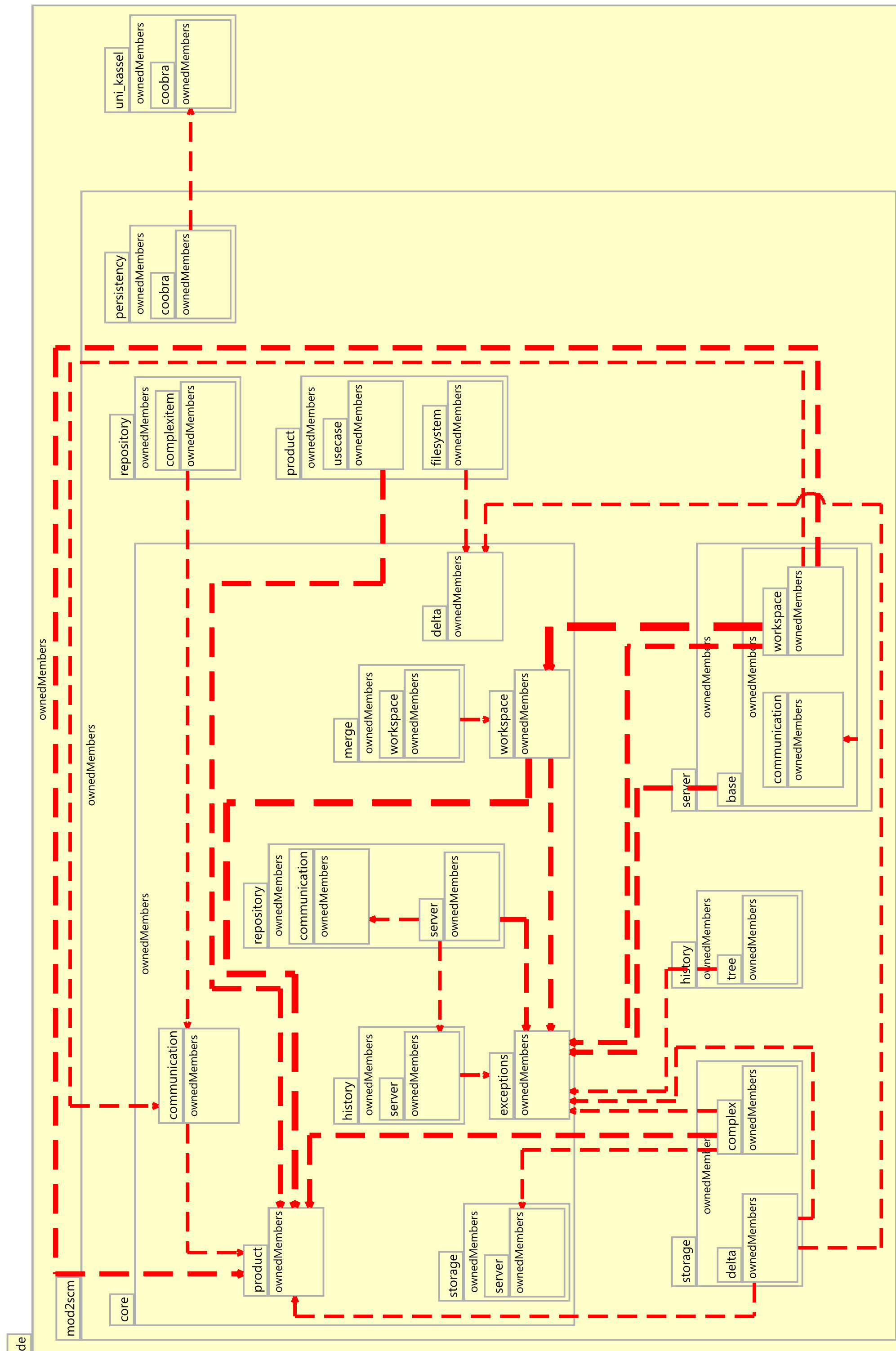


Abbildung 6.139.: Stärkste Abhängigkeiten des Domänenmodells

Laufzeit auftreten. Zweiteres zeigt, dass die Schnittstelle des Produktmodells die zentrale Schnittstelle eines SKMS ist, da sie von anderen Modulen häufig verwendet wird. Die Abstraktion vom konkreten Produktmodell bedeutet daher eine deutlich geringere Kopplung für ein SKMS.

In diesem Zusammenhang stellt sich die Frage, welche Produktmodelle sich mit Hilfe der bisherigen Schnittstelle versionieren lassen bzw., ob die Schnittstelle zu einschränkend ist. Auch sollte noch genauer untersucht werden, ob noch andere Module – außer den Delta- und Verschmelzungsmodulen – von den konkreten Produktmodellen abhängen. Weiterhin ist offen, inwieweit die enge Kopplung der monolithischen SKMS an das Produktmodell einen Performanzvorteil gegenüber einem modularen SKMS bietet. Bei der Versions-Proliferation in Dateisystem-Produktmodellen lässt sich z.B. die Objektkennung des Elternelements aus dem voll qualifizierten Elementnamen ableiten. Die Schnittstelle im Produktmodell-Kernmodul (vgl. Abschnitt 6.3.8, S. 188ff.) abstrahiert jedoch vom konkreten Produktmodell, so dass eine allgemeingültigere (aber langsamere) Abfrage bei der Versions-Proliferation durchgeführt wird (vgl. Abschnitt 6.4.22, S. 280ff.). Ein Performanzvergleich ist auf Grund der vielen unterschiedlichen Voraussetzungen (unterschiedliche Quellcode-Sprachen, Quellcode manuell implementiert bzw. generiert) sehr aufwändig und daher im Rahmen dieser Arbeit nicht möglich.

Geringe Kohäsion: *id*-Module

Bei Modulen mit geringer Kohäsion fallen lediglich die Kennungsmodule – einschließlich ihres Kernmoduls *core.id* (vgl. Abschnitt 6.3.5, S. 182ff.) – auf. Alle Module enthalten i.d.R. nur eine einzige Klasse, die zum Generieren einer Kennung als Zeichenkette verwendet wird, bzw. die eine Liste von Zeichenketten-Kennungen übersetzt. Daher bestehen nur Abhängigkeiten von generischen Datenklassen wie Zeichenketten oder Listen. Bei Einführung einer Datenklasse für Kennungen (um die Signatur der Methoden zu vereinheitlichen, vgl. Abschnitt 6.3.9, S. 191ff. und vgl. Abschnitt 6.3.4, S. 178ff.), ist diese wahrscheinlich Komponente des Kennungs-Moduls, was die Kohäsion erhöhen würde.

6.6. Zusammenfassung

Das MOD2-SKM Domänenmodell besteht aus einem Paket- und einem Fujaba-Modell, die mit Merkmalsmarkierungen aus dem Merkmalsmodell aus Kapitel 5 annotiert sind. Seiner Architektur liegen drei Anforderungen zu Grunde: Erstens die Unterscheidung zwischen gemeinsam genutzten Kernmodulen und austauschbaren Modulen einer Modulbibliothek. Zweitens die Aufteilung der Anforderungsbereiche aus Kapitel 4 auf 14 Kernmodule – gemäß Prinzip „Separation of Concerns“ [VAC+09]. Und drittens, die Trennung zwischen Arbeitsbereich-Client, Kommunikations-Middleware und Repositoriums-Server als Umsetzung der Architekturstruktur „Client-Server-Modell“ [VAC+09].

Das vollständige MOD2-SKM Domänenmodell besteht aus 14 Kernmodulen und 29 Modulen in der Modulbibliothek. Zwischen ihnen bestehen 244 Abhängigkeiten, größtenteils zwischen Kernmodulen bzw. einem austauschbaren Modul zu seinem Kernmodul. Die Austauschbarkeit wird vor allem durch Schnittstellen-Abstraktion erreicht, so dass

jedes Bibliotheksmodul von der Schnittstelle eines Kernmoduls abhängt. Es existieren kaum Kopplungen unter den Bibliotheksmodulen, somit sind diese gegen beliebige Module mit der gleichen Schnittstelle austauschbar. Anhand der Kopplungsanzahl und -stärke ließen sich auffällige Module identifizieren und genauer untersuchen. Da jedes Kernmodul einen Aufgabenbereich eines SKMS repräsentiert, lassen diese Kopplungen auch Rückschlüsse auf die Abhängigkeiten der SKM-Konzepte zu. Eine quantitative Analyse der Kopplungen liefert schnell gute Indikatoren für „kritische“ bzw. „interessante“ Pakete, lässt sich aber nur schwer zur Bewertung der Architekturqualität einsetzen. Zwischen den Modulen der Modulbibliothek existieren jedoch nur wenige Abhängigkeiten (z.B. vom Modul *server.base* aus, vgl. Abschnitt 6.4.18, S. 254ff.), so dass sich fast alle Module beliebig austauschen lassen. Die Entkopplung der SKM-Module ist also gelungen.

7. MOD2-SKM für Eclipse

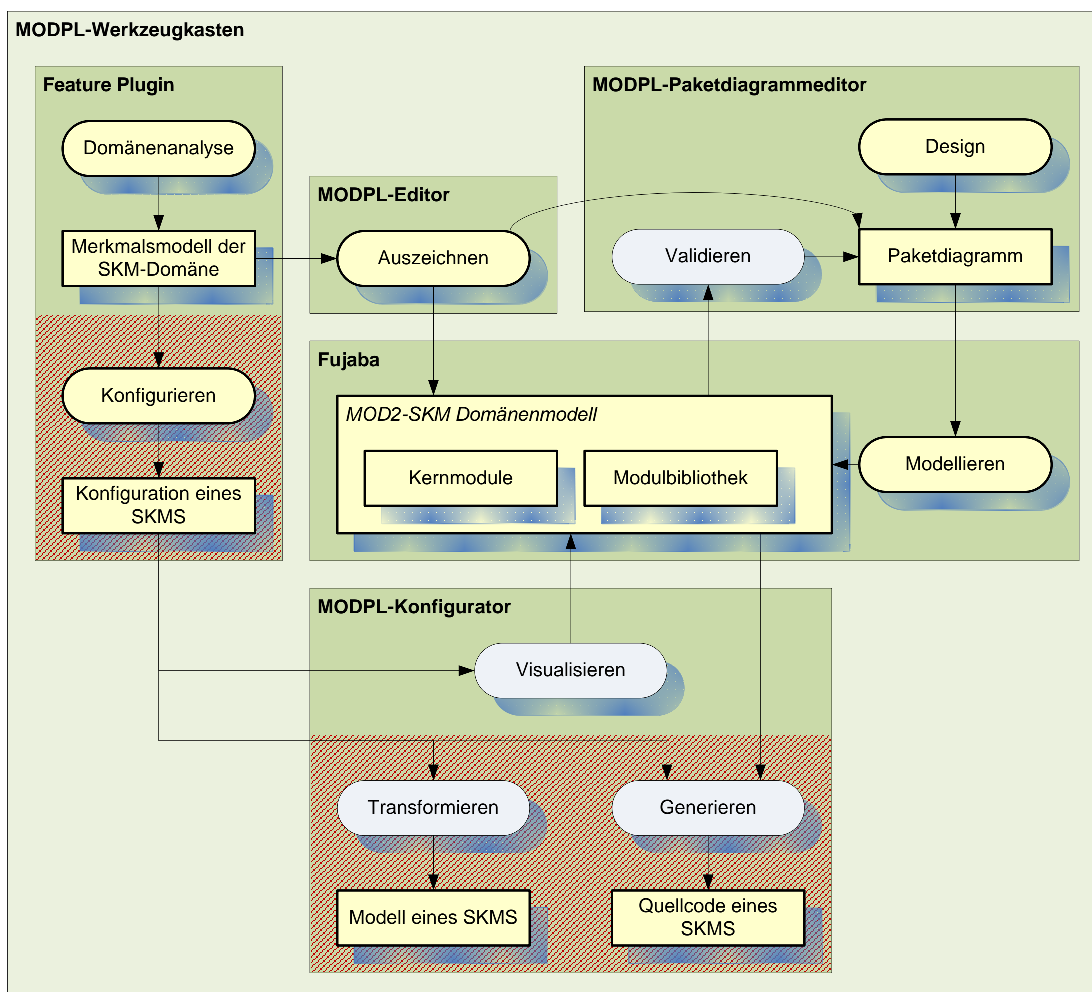


Abbildung 7.1.: Überblick über die MOD2-SKM Produktlinie

Das MODPL-Rahmenwerk reduziert die Entwicklung eines Software-Produkts aus der Produktlinie auf das Erstellen einer Konfiguration [Buc10]. Die Anwendungsentwickler verwenden zu ihrer Erstellung das Feature-Plugin [AC04], um dann mit dem MODPL-Konfigurator [Buc10] ein konfiguriertes SKMS zu generieren. Beide Werkzeuge sind in Abbildung 7.1 durch Schraffur hervorgehoben.

Trotz der Erweiterungspunkte im Merkmalsmodell (vgl. Abschnitt 5.5, S. 129ff.) und

einiger, mit minimalem Aufwand modellierter Module in der Modulbibliothek (vgl. Abschnitt 6.4, S. 211ff.) ist die MOD2-SKM eine einsatzfähige Produktlinie, die vollständig lauffähige SKMS generiert. Diese Systeme besitzen jedoch keine grafische Benutzeroberfläche sondern bieten nur eine Java-Schnittstelle für die SKMS-Operationen.

Im Rahmen dieser Arbeit wurde daher von Stefan Oehme und Matthias Kufer das **MOD2-SKM-Plugin**, eine grafische Benutzeroberfläche in Form einer Erweiterung für die Entwicklungsumgebung Eclipse [ecl10], entwickelt[1]. Diese bietet eine Benutzerschnittstelle für Arbeitsbereiche von SKMS mit Dateisystem-Produktmodell an, um ausgewählte Software-Projekte unter Versionskontrolle zu stellen.

Das MOD2-SKM-Plugin ist ein Demonstrator, der speziell für ein einzelnes konfiguriertes SKMS entwickelt wurde. Er ergänzt die modellgetriebene Entwicklung am Domänenmodell und vervollständigt das SKMS zu einem kompletten, von Entwicklern bedienbaren, SKM-Werkzeug. Die Modellierung einer Benutzeroberfläche als konfigurierbare Komponente einer Produktlinie erwies sich als zu aufwändig und war, im Rahmen dieser Arbeit, nicht umzusetzen. Allerdings lassen sich, anhand der manuellen Implementierung, die engen Kopplungen zwischen Benutzeroberfläche und dem restlichen Domänenmodell demonstrieren sowie der Lösungsansatz einer modellgetrieben entwickelten, modularen Benutzeroberfläche diskutieren.

Das Plugin ist wie folgt aufgebaut: Grundlage bildet ein konfiguriertes SKMS, dessen Arbeitsbereich als Eclipse-Plugin ausgeliefert wird (s. Abschnitt 7.1). Auf Basis des Arbeitsbereichs-Plugins und der SKM-Schnittstelle von Eclipse (s. Abschnitt 7.2) sind nun grafische Erweiterungen implementiert, die dem Benutzer Zugriff auf die Operationen und Informationen des Arbeitsbereichs gewähren (s. Abschnitt 7.3). Die Untersuchung dieser Implementierung zeigt, dass die grafische Oberfläche letztendlich als viertes Teilsystem in die Produktlinie integriert werden sollte (s. Abschnitt 7.4).

7.1. Konfigurieren des SKMS

Das MOD2-SKM-Plugin bietet eine grafische Benutzerschnittstelle für den Arbeitsbereich, die sich unmittelbar in der Entwicklungsumgebung Eclipse verwenden lässt. Auf Grund der vielen Abhängigkeiten zum proprietären Eclipse-Plugin-Rahmenwerk sind die grafische Benutzerschnittstelle und ihr Verhalten weder modelliert noch Teil der Produktlinie. Es existieren auch keine Merkmale, um die Benutzerschnittstelle zu konfigurieren oder Merkmalsmarkierungen, um sie an die Konfiguration des SKMS anzupassen. Es stellt jedoch eine Reihe von Anforderungen an die Konfiguration des SKMS, da sie nur Merkmale enthalten darf, die auch vom MOD2-SKM-Plugin unterstützt werden.

Die Konfiguration des MOD2-SKM-Plugin ist mit Hilfe des Feature-Plugin aus dem MODPL-Werkzeugkasten erstellt. Sie wird in den Abbildungen 7.2 und 7.3 gezeigt. Das MOD2-SKM-Plugin verwendet das Dateisystem als Produktmodell (*Merkmal „Dateisystem“ (B.2.2)*) und besitzt Operationen, um ausgewählte Dateisystem-Elemente zu ignorieren (*Merkmal „Ausgewählte Elemente ignorieren“ (B.1.1.2)*). Beim Einspielen von Än-

[1] Die Erweiterung ist Ergebnis eines Masterpraktikums an der Universität Bayreuth, so dass keine Publikation als Quelle existiert.

Abbildung 7.2.: Merkmalsmodell: Konfiguration des MOD2-SKM-Plugins

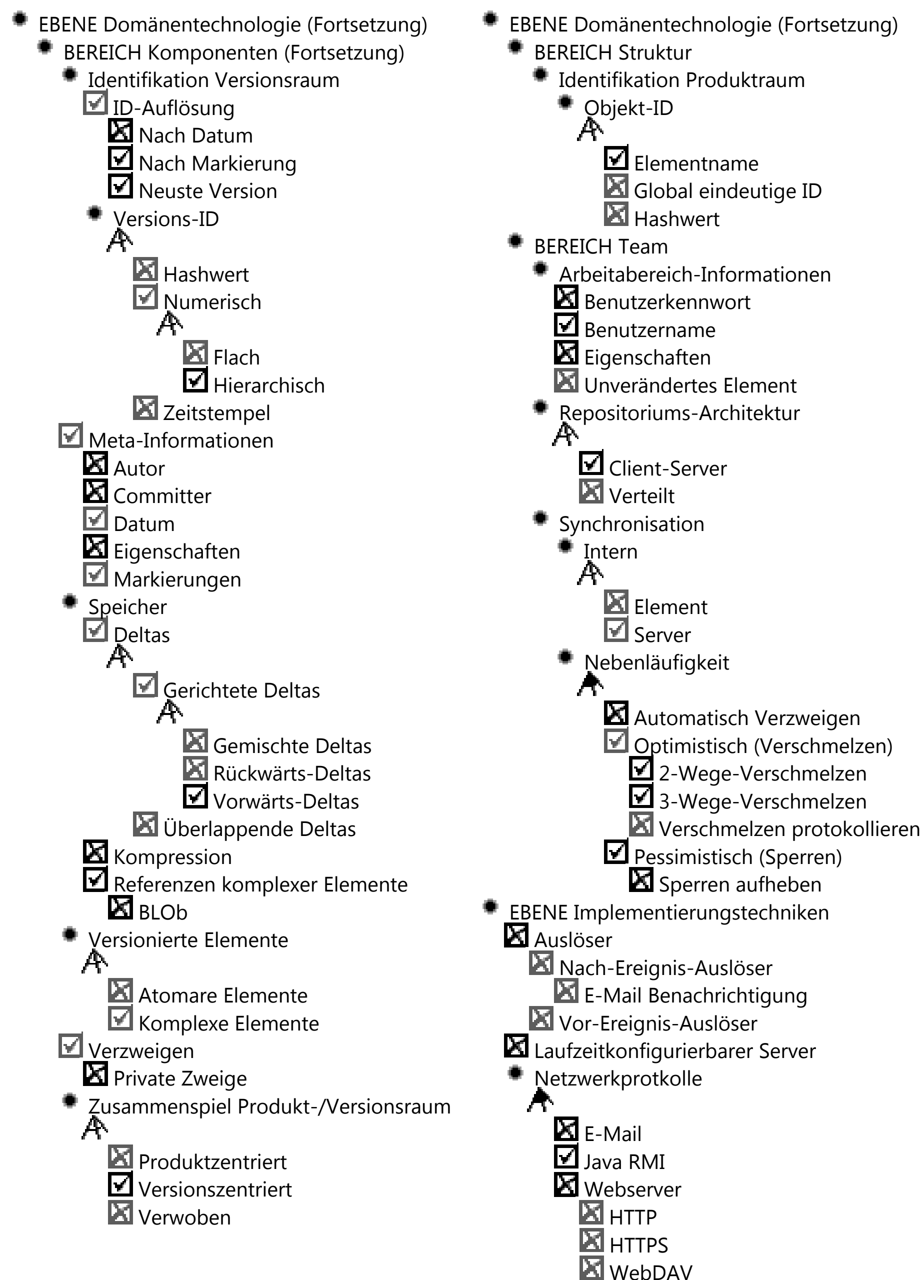


Abbildung 7.3.: Merkmalsmodell: Konfiguration des MOD2-SKM-Plugins (Fortsetzung)

derungen bzw. dem Aktualisieren des Arbeitsbereichs können sowohl mehrere Elemente als auch ganze Teilbäume ausgewählt werden (*Merkmale „Mehrere Elemente“ (B.1.1.3.2), „Vollständige Rekursion“ (B.1.1.3.4), „Mehrere Elemente“ (B.1.1.8.2), „Vollständige Rekursion“ (B.1.1.8.4)*).

Produkt- und Versionsraum sind versionszentriert integriert (*Merkmal „Versionszentriert“ (D.7.2)*). Als Historie wird ein Baum verwendet, so dass Operationen zum Verzweigen unterstützt werden (*Merkmale „Baum“ (D.1.2.1.1) und „Verzweigen“ (D.6)*). Außer über die Versionsnummern – die hierarchisch vergeben werden (*Merkmal „Hierarchisch“ (D.2.2.2.2)*) – besteht auch die Möglichkeit, auf die aktuellste Version zuzugreifen bzw. ausgewählte Versionen zu markieren (*Merkmale „Neuste Version“ (D.2.1.3) und „Nach Markierung“ (D.2.1.2)*). Unveränderte Dateisystem-Elemente werden im Speicher referenziert und Dateien zusätzlich über Vorwärts-Deltas platzsparend abgelegt (*Merkmale „Referenzen komplexer Elemente“ (D.4.3) und „Vorwärts-Deltas“ (D.4.1.1.3)*). Für den Mehrbenutzerbetrieb stehen sowohl optimistische als auch pessimistische Sperren zur Verfügung (*Merkmale „Verschmelzen“ (D.11.2.3) und „Sperren“ (D.11.2.2)*).

Auslieferung als Eclipse-Erweiterung

Das MOD2-SKM-Plugin ist ein Arbeitsbereich-Client und benötigt lediglich Komponenten des Arbeitsbereichs- und des Kommunikations-Teilsystems. Dazu ist es notwendig, beide Teilsysteme ebenfalls als Eclipse-Plugin bereitzustellen. Da der MODPL-Konfigurator das konfigurierte SKMS immer komplett ausliefert, werden die entsprechenden Komponenten über ein Stapelverarbeitungs-Skript in ein entsprechend vorbereitetes Plugin-Projekt kopiert. Dieses enthält auch die manuell erstellten Plugin-Deskriptoren, so dass hier lediglich der Quellcode ergänzt werden muss. Es erscheint möglich, die Auslieferung mit Hilfe des MODPL-Werkzeugkastens zu automatisieren. Da jedoch die Zielplattform nicht bekannt ist, wird vermutlich ein generischer, vorlagenbasierter Ansatz benötigt.

Einsatz des SKMS

Der MOD2-SKM-Server benötigt kaum Konfigurationsaufwand. Es reicht eine Java-Laufzeitumgebung aus, um den kompilierten Quellcode (z.B. als JAR-Archiv) auszuführen. Nach dem Programmstart ist der MOD2-SKM-Server auch von entfernten Systemen aus, über Java-RMI, erreichbar. Das MOD2-SKM-Plugin muss lediglich in eine bestehende Eclipse-Entwicklungsumgebung installiert werden. Dadurch integriert es seine Operationen und Fenster zur Versionsverwaltung der Projekte in den Eclipse-Arbeitsbereich.

7.2. Architektur der MOD2-SKM-Erweiterung

Abbildung 7.4 gibt einen Überblick über die Architektur des MOD2-SKM-Plugins. Die Eclipse-Entwicklungsumgebung verwendet einen Order des Dateisystems als Arbeitsbereich. Dieser abstrahiert vom Dateisystem des Betriebssystems, indem z.B. Ordner mit Binärcode versteckt oder Pakethierarchien abgeflacht werden. Eclipse verwendet daher

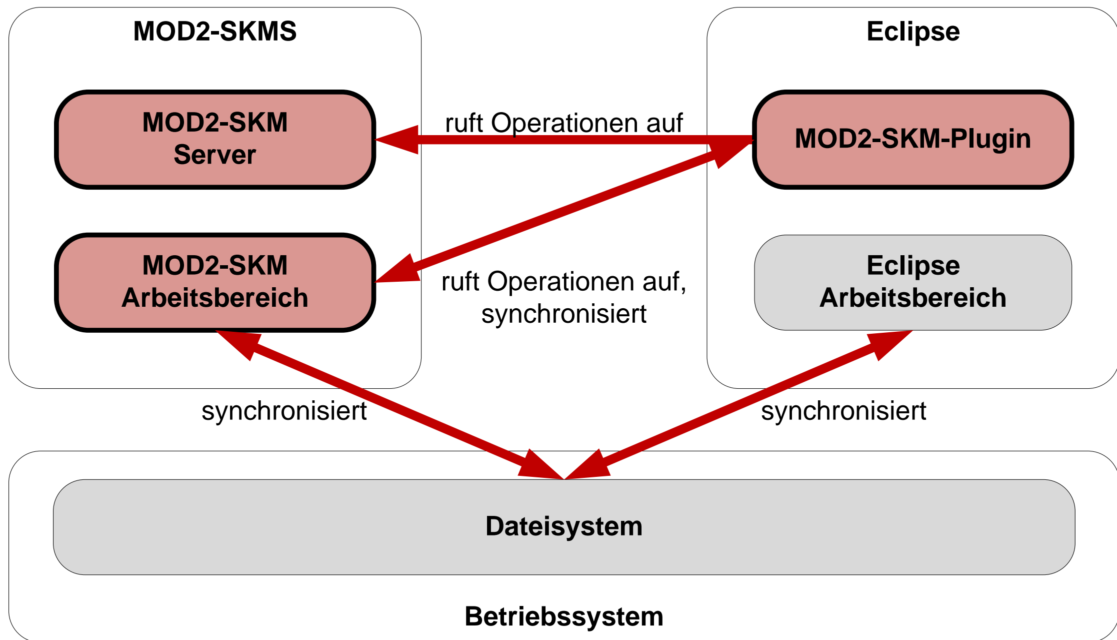


Abbildung 7.4.: Architektur des MOD2-SKM-Plugins

einen Mechanismus, um das Dateisystem mit seiner Darstellung des Arbeitsbereichs synchron zu halten. Der MOD2-SKM Arbeitsbereich mit Dateisystem-Produktmodell ist ähnlich aufgebaut und muss ebenfalls mit dem Dateisystem synchron gehalten werden (vgl. Abschnitt 6.4.14, S. 240ff.). Diese Zusammenhänge werden durch die beiden „synchronisiert"-Doppelpfeile in Abbildung 7.4 dargestellt.

Der Eclipse-Arbeitsbereich enthält – wie das Dateisystem selbst – keinerlei Versionsinformationen. Beide Teilsysteme sind daher grau dargestellt. Rötliche Teilsysteme enthalten Versionsinformationen. Dazu gehört der MOD2-SKM-Arbeitsbereich, der z.B. die Version der Dateisystem-Elemente speichert, oder ob sie sich seit der letzten Aktualisierung verändert haben (vgl. Abschnitt 6.4.27, S. 292ff.). Das MOD2-SKM-Plugin erweitert nun Eclipse, so dass Versionsinformationen aus dem MOD2-SKM Arbeitsbereich dargestellt werden („synchronisiert"-Doppelpfeil zwischen beiden Teilsystemen). Gleichzeitig bietet es auch Kommandos an, um Operationen im MOD2-SKM Arbeitsbereich aufzurufen, z.B. um veränderte Dateien in das Repositorium einzuspielen („ruft Operationen auf"). Zusätzlich existiert auch ein Server-Browser, der Operationen direkt auf dem MOD2-SKM-Server aufrufen kann. Dies ist nötig, um z.B. neue Projekte in den Arbeitsbereich auszuchecken.

Das Zusammenspiel der beiden Arbeitsbereiche lässt sich anhand der Veränderung einer Datei unter Versionskontrolle und anschließender Aktualisierung veranschaulichen:

1. Die Datei „Test.java" liegt im Betriebssystem-Dateisystem und beide Arbeitsbereiche sind mit ihm synchron, d.h., im Eclipse-Arbeitsbereich wird ihr Dateiname angezeigt. Im MOD2-SKM-Arbeitsbereich existiert ein Datei-Element in der Pro-

duktmodell-Instanz einschließlich Versionsinformationen (*WSInfo*-Objekt, vgl. Abschnitt 6.3.14, S. 206ff.), z.B. dass die Datei der Version „1.4“ entspricht und das sie nicht verändert wurde.

2. Das MOD2-SKM-Plugin erweitert die Dateinamen im Eclipse-Arbeitsbereich, indem es mit ihrer Hilfe die zugehörigen Versionsinformationen des MOD2-SKM-Arbeitsbereichs überwacht, d.h., es wird im Eclipse-Arbeitsbereich „Test.java [1.4]“ angezeigt.

3. Die Datei wird über einen Eclipse-Editor im Eclipse-Arbeitsbereich bearbeitet. Sobald der Editor die Änderungen speichert, wird die Datei im Betriebssystem-Dateisystem aktualisiert.

4. Das Dateisystem-Produktmodell (vgl. Abschnitt 6.4.14, S. 240ff.) registriert die Änderung und aktualisiert das Datei-Element (liest geänderten Inhalt aus dem Betriebssystem-Dateisystem) sowie die Versionsinformationen (die Datei wurde seit der letzten Aktualisierung verändert).

5. Das MOD2-SKM-Plugin registriert die Änderung der Versionsinformationen und aktualisiert die Anzeige des Dateinamens zu „> Test.java [1.4]“.

6. Der Entwickler selektiert den Dateinamen im Eclipse-Arbeitsbereich und wählt das Kommando „Update“ aus. Das MOD2-SKM-Plugin fragt, mit Hilfe des Dateinamens, die Versionsinformationen aus dem MOD2-SKM-Arbeitsbereich ab. Da eine Änderung vorliegt, öffnet sich ein Dialog mit der Frage, ob der Entwickler die Änderungen überschreiben möchte.

7. Die Update-Methode des MOD2-SKM-Arbeitsbereichs erhält als Parameter den Dateinamen und – in diesem Beispiel – die Aufforderung, die Änderungen zu überschreiben. Die Methode aktualisiert nun das Datei-Element und die Versionsinformationen.

8. Das Dateisystem-Produktmodell schreibt den neuen Dateiinhalt in das Betriebssystem-Dateisystem.

9. Der Eclipse-Arbeitsbereich registriert die Änderung im Betriebssystem-Dateisystem und aktualisiert z.B. den Editor mit dem neuen Dateiinhalt.

10. Das MOD2-SKM-Plugin registriert die Änderung an den Versionsinformationen und aktualisiert die Anzeige zu „Test.java [1.7]“.

Beide Arbeitsbereiche bieten unterschiedliche Sichten auf das selbe Dateisystem, d.h. beide Teilsysteme können ihre Elemente einem Dateisystem-Element zuordnen (über den voll qualifizierten Dateinamen). Für die erweiterte Darstellung ist es notwendig, die Versionsinformationen aus dem MOD2-SKM-Arbeitsbereich den entsprechenden Elementen aus dem Eclipse-Arbeitsbereich zuzuordnen. Dazu wird der voll qualifizierte Dateiname verwendet, da dieser unabhängig von beiden Teilsystemen und damit identisch ist. Auch beim Aufruf von Operationen werden die Dateinamen der selektierten Elemente als Objektkennung an den MOD2-SKM-Arbeitsbereich gesendet.

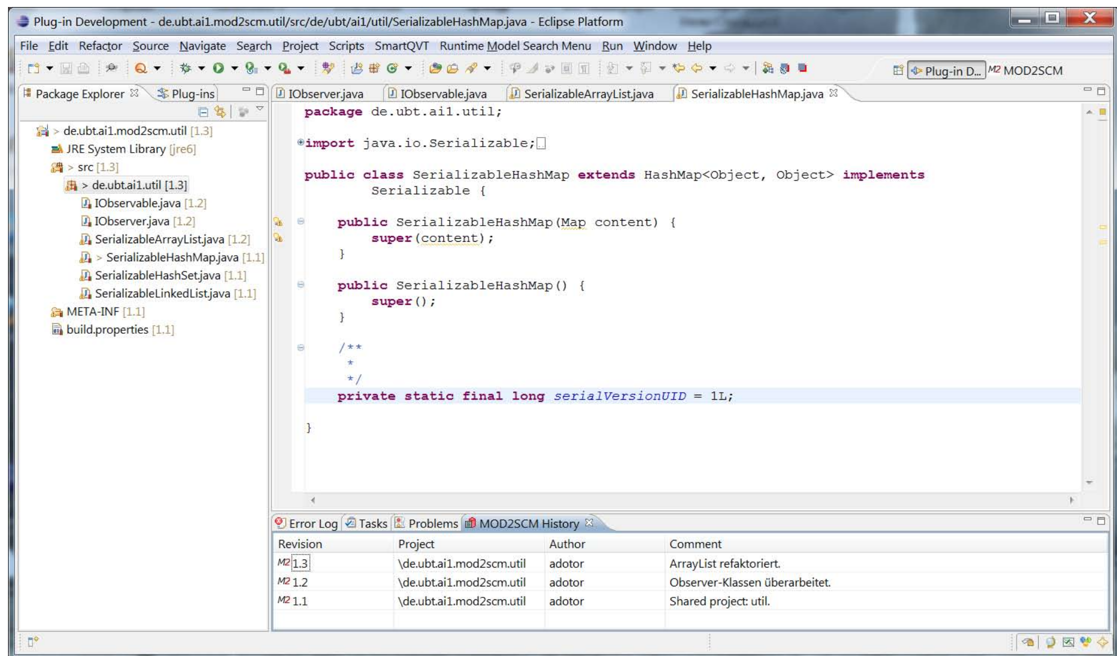


Abbildung 7.5.: Oberflächen-Erweiterung des MOD2-SKM-Plugins

7.3. Funktionalität der MOD2-SKM-Erweiterung

Abbildung 7.5 zeigt die durch das MOD2-SKM-Plugin erweiterte Benutzeroberfläche von Eclipse. Die Baumansicht des Eclipse-Arbeitsbereichs (*Fenster „Package Explorer“*) zeigt zusätzlich die Versionsinformationen auf dem MOD2-SKM-Arbeitsbereich an: Hinter dem Namen eines Elements steht, in eckigen Klammern, die aktuelle Versionsnummer (z.B. ist das Projekt „de.ubt.ai1.mod2scm.util“ in Version „1.3“ vorhanden). Ein Pfeil vor einem Element gibt an, dass es seit der letzten Aktualisierung des Arbeitsbereichs editiert wurde. Obwohl nur die Datei „SerializableHashMap“ geändert wurde (*Editor-Fenster „SerializableHashMap.java“*), ist – auf Grund der versionszentrierten Integration – die Änderungsmeldung bis zum Projekt propagiert worden. Unterhalb des Editor-Fensters zeigt die Historien-Ansicht (*Fenster „MOD2SCM History“*) die Versionshistorie des Pakets „de.ubt.ai1.util“ an (im Fenster „Package Explorer“ selektiert). Sie zeigt tabellarisch weitere Informationen über die einzelnen Revisionen, wie den Autor und den Kommentar.

Abbildung 7.6 zeigt das Kontextmenü des MOD2-SKM-Plugins, mit dessen Hilfe sich die MOD2-SKM-Operationen auf den Elementen aufrufen lassen, die im Fenster „Package Explorer“ selektiert sind. Dazu wird ihr vollständig qualifizierter Name als Objekt-Kennung an den MOD2-SKM Arbeitsbereich gesendet (vgl. Abschnitt 7.2, S. 319ff.). So lassen sich z.B. die Änderungen in das Repositorium überspielen (*Kommando „Commit“*). Stattdessen kann auch das Element im Arbeitsbereich aktualisiert werden – entweder auf eine bestimmte (*Kommando „Update to Revision / Tag“*), oder die letzte Version (*Kommando „Update“*). Dies kann u.U. zum Öffnen eines Verschmelzungsdialogs führen

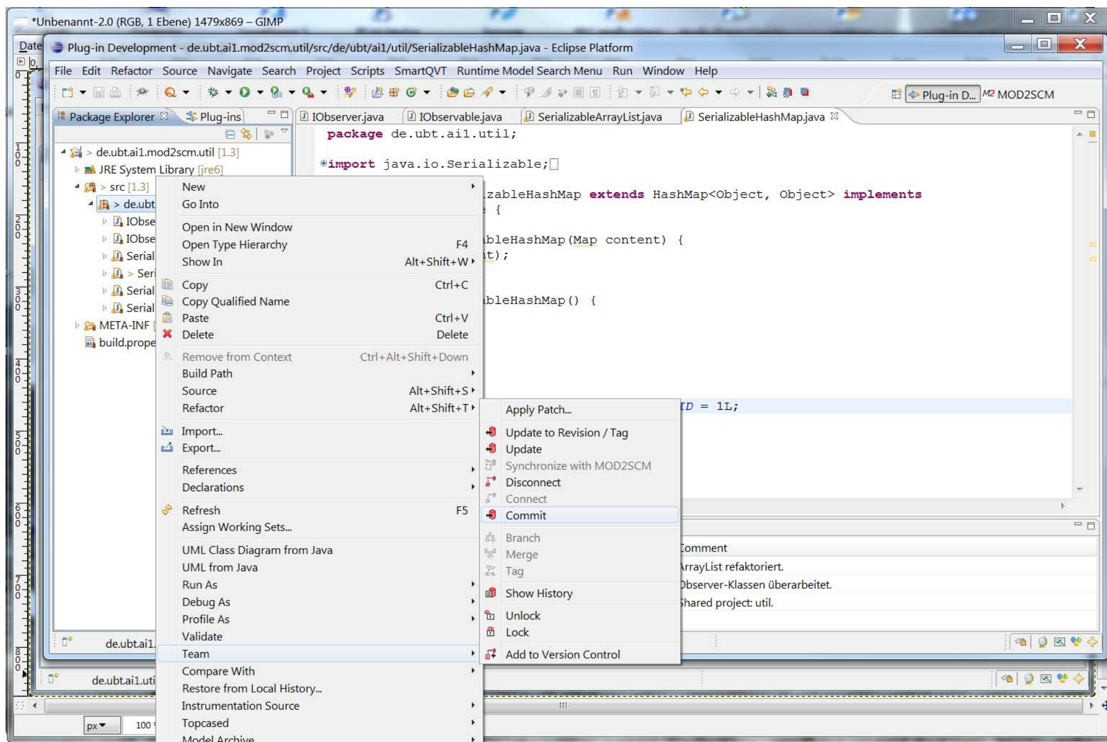


Abbildung 7.6.: Kontextmenü des MOD2-SKM-Plugins

[Bär10]. Über die Kommandos „Lock“ und „Unlock“ lässt sich dagegen das Element im Repository sperren, um konkurrierende Änderungen zu vermeiden. Die Operationen zum Verzweigen (*Kommando „Branch“*) und Markieren (*Kommando „Tag“*) stehen auf dem selektierten Element nicht zur Verfügung, da sie zur Vereinfachung des Entwicklungsaufwands nur auf unveränderten Elementen arbeiten.

Um ein neues Projekt aus einem bestehenden Repositorium abzuholen, ist es notwendig, das Repositorium und das Projekt zu lokalisieren. Dazu existiert ein Server-Browser, mit dessen Hilfe die Entwickler die laufenden Repositorien-Server betrachten können. Er bietet auch die Möglichkeit, einen neuen Arbeitsbereich anzulegen und mit einem dort vorhanden Repositorium zu verbinden (engl. checkout). Abbildung 7.7 zeigt diese MOD2-SKM-Perspektive. Das Fenster „MOD2SCM Repository Exploring“ stellt die Repositorien des Servers und ihren Inhalt als Baum dar. Die Elemente werden dabei in ihrer aktuellsten Version angezeigt. Eine Zugriff auf älterer Versionen ist konzeptionell möglich, aber nicht implementiert. Auf dem Server befinden sich zwei Repositorien: Erstens das bereits im Arbeitsbereich vorhandene Projekt „de.ubt.ai1,mod2scm.util“. Zweitens „de.ubt.ai1.mod2scm“, d.h. die MOD2-SKM Produktlinie selbst. Da sie im Arbeitsbereich nicht vorhanden ist, kann sie über das Kontextmenü ausgecheckt (*Kommando „Check Out“*) oder zunächst ihre Historie angezeigt werden (*Kommando „Show History“*, *Fenster „MOD2SCM History“*).

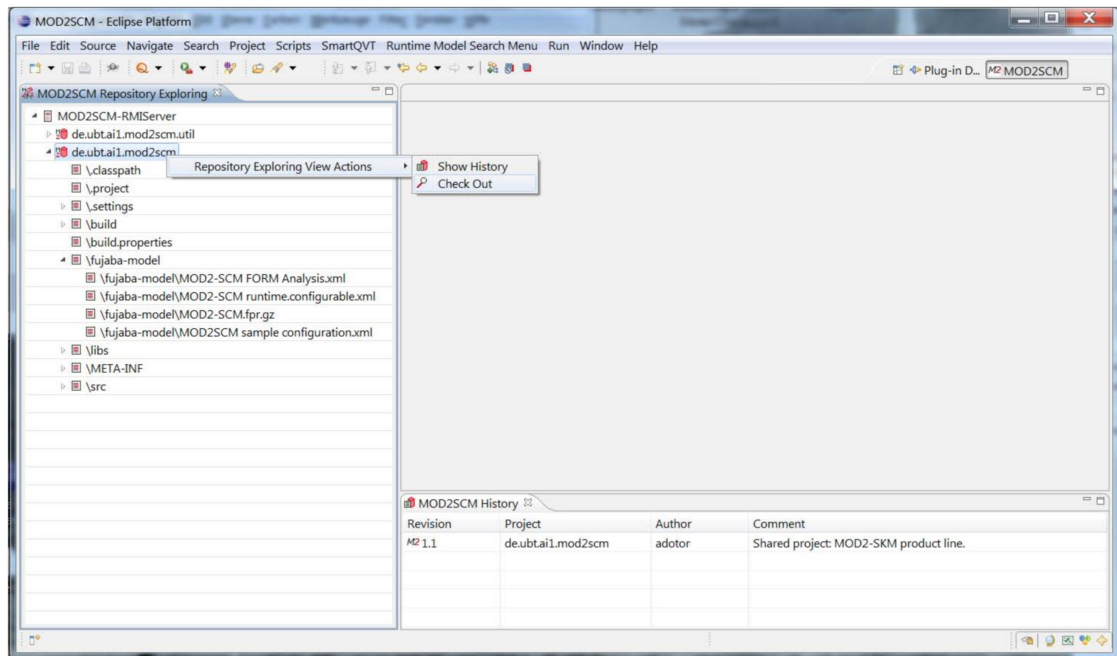


Abbildung 7.7.: Server-Ansicht im MOD2-SKM-Plugin

7.4. Modulare Benutzeroberflächen

Bereits anhand der in den Abbildungen 7.5 bis 7.7 präsentierten Bestandteile der grafischen Benutzeroberfläche, lassen sich diverse Abhängigkeiten zwischen ihr und der Konfiguration erkennen. So hängen z.B. die Spalten in der Historien-Ansicht (*Fenster „MOD2SCM History“*) von den Meta-Informationen der Historie ab. Und je nachdem, ob die Historie Zweige erlaubt oder nicht, ist die Anzeige des Zweignamens erforderlich. Dieser Sachverhalt beeinflusst zusätzlich noch weitere Elemente: Zum einen, ob überhaupt ein Kommando „Branch“ im Kontextmenü enthalten ist. Zum anderen, ob im Dialog des Kommandos „Update to Revision / Tag“ ein Element zur Auswahl des Zweignamens benötigt wird. Analog verhält es sich auch mit der Auswahl der Markierung (engl. tag).

Diese Beobachtungen führen zu dem Schluss, dass die Benutzeroberfläche in die MOD2-SKM-Produktlinie integriert werden sollte. Eine Lösung auf Ebene der Architektur bietet die Einführung eines vierten Teilsystems *gui*, das neben *server*, *communication* und *workspace* eingeführt wird. Dieses kann dann über das Model-View-Controller-Entwurfsmuster [FF04] an die bestehenden Teilsysteme angebunden werden. Das MOD2-SKM-Plugin bildet hierbei View und Controller – und die restlichen Teilsysteme das Model. Auf den ersten Blick erscheint zunächst eine Kopplung der Benutzeroberfläche an den Arbeitsbereich ausreichend. Doch eine Komponente wie der Server-Browser in Abbildung 7.7 benötigt einen direkten Zugriff auf den MOD2-SKM-Server, da vor einem Checkout noch kein Arbeitsbereich existiert.

Ungeklärt ist, wie die Komponenten einer grafischen Benutzeroberfläche im Rahmen

einer Produktlinie entwickelt werden können. Das MOD2-SKM-Plugin ist in Java implementiert und muss manuell an jede Konfiguration angepasst werden. Die Verwendung von Merkmalsmarkierungen ist nicht möglich, da das Plugin nicht als Modell vorliegt. Die Modellierung des Quellcodes für das MOD2-SKM-Plugin ist prinzipiell in Fujaba möglich. Dazu müssen die verwendeten Klassen und Schnittstellen des Eclipse-Rahmenwerks als Referenzen (vgl. Abschnitt 6.1, S. 144ff.) in das Fujaba-Modell eingefügt werden. Diese Technik kommt bereits im Verschmelzungsmodul für Eclipse zum Einsatz, das die Eclipse-Compare-Schnittstelle nutzt (vgl. Abschnitt 6.4.10, S. 233ff.) [Bär10]. Da jede Referenz manuell nachmodelliert werden muss, ist dies jedoch nur für kleine Erweiterungen möglich. Hier wäre eine Import-Funktion für Referenzen hilfreich.

Dieses Konzept halte ich für machbar, solange die Elemente der Benutzeroberfläche (1) Komponenten eines Rahmenwerks und (2) auf primitive Elemente beschränkt sind, wie z.B. Kontextmenüs, Tabellen, Dialoge oder Eingabemasken für primitive Datentypen. Die Modellierung komplexer grafischer Benutzeroberflächen mit Hilfe von Klassen- und Storydiagrammen ist jedoch zu aufwändig und abstrahiert nicht genug von der technischen Umsetzung der grafischen Benutzeroberfläche. Stattdessen könnte ein Modell eingeführt werden, das auf der Darstellung der grafischen Benutzeroberfläche basiert – ähnlich der grafischen Entwicklungsumgebungen für Benutzeroberflächen. Die grafische Darstellung würde auf die entsprechenden Elemente der Zielsprache abgebildet. Durch annotieren der grafischen Komponenten mit Merkmalsmarkierungen ließe sich dann der MODPL-Ansatz auf dieses Modell ausdehnen. Die Machbarkeit und Grenzen eines solchen Ansatzes können in zukünftigen Arbeiten untersucht werden.

Ein interessantes Konzept ist auch eine, zur Laufzeit konfigurierbare, Benutzeroberfläche. Das MOD2-SKM-Plugin erhält dazu vom MOD2-SKM-Server Informationen über dessen Konfiguration und passt seine Oberfläche entsprechend an. Damit würde es ausreichen, ein einziges MOD2-SKM-Plugin auszuliefern und zu installieren, um mit allen möglichen Server-Konfigurationen in Verbindung zu treten.

7.5. Zusammenfassung

Das MOD2-SKM-Plugin für Eclipse zeigt, dass die MOD2-SKM Produktlinie auch lauffähige und verwendbare SKMS liefert. Ein generiertes SKMS lässt sich, mit Hilfe manuell implementierter grafischer Eclipse-Erweiterungen, in die SKM-Schnittstelle integrieren und zur Versionierung von Software-Entwicklungsprojekten einsetzen (u.a. das MOD2-SKM Projekt selbst). Die Abhängigkeiten zwischen grafischer Oberfläche und der Konfiguration des SKMS zeigen deutlich, dass die Oberfläche als viertes Teilsystem in die Produktlinie integriert werden sollte. Dies wirft zwei Fragen auf: (1) Kann die grafische Benutzeroberfläche modellgetrieben entwickelt werden – und mit welchen Modellen? (2) Wie lassen sich komplexe proprietäre Rahmenwerke – für die keine Modelle existieren – effizient in das Domänenmodell einbinden? Die Untersuchung dieser komplexen Fragen ist eine Herausforderung für zukünftige Arbeiten.

8. Ergebnis und Ausblick

Diese Arbeit zeigt, dass sich SKMS aus mehreren Modulen zusammensetzen lassen, indem die Kopplungen zwischen den einzelnen Aufgabenbereichen durch ein gemeinsam genutztes Kernsystem modelliert werden (vgl. Abschnitt 6.5.3, S. 300ff.). Unmittelbares Ergebnis sind zunächst einmal das Merkmalsmodell (vgl. Abschnitt C, S. 435ff.), die Merkmalsanalyse (vgl. Abschnitt B, S. 359ff.) und das Domänenmodell aus Kapitel 6. Sie sind so weit entwickelt, dass ein lauffähiges SKMS als Erweiterung der Eclipse-Entwicklungsumgebung generiert werden konnte (vgl. Abschnitt 7, S. 315ff.). Beide Modelle dienen auch dazu, die in Abschnitt 1.7 formulierten Hypothesen zu belegen oder zu widerlegen.

Dieses Ergebnis belegt, dass es möglich ist, mit den Modellen und Werkzeugen des MODPL-Werkzeugkasten eine nicht-triviale Produktlinie modellgetrieben zu entwickeln. Dies reduziert das Erstellen eines konfigurierten SKMS auf das Erstellen einer Merkmalskonfiguration, die dann zur automatischen Generierung des SKMS genutzt wird.

Zusätzlich ist die MOD2-SKM Produktlinie sowohl ein Fallbeispiel für die Entwicklung einer modellgetriebenen Produktlinie mit dem MODPL-Werkzeugkasten, als auch für die Methode der modellgetriebenen Software-Entwicklung. Von besonderer Bedeutung ist hier die Größe des Projekts, die spezielle Techniken zum Modellieren im Großen erfordert. Aus den Anforderungen der Modelle sowie den Erfahrungen während der Entwicklung, lassen sich Rückschlüsse über die Mächtigkeit der Konzepte und Leistungsfähigkeit der Werkzeuge ziehen.

8.1. Prüfung der Hypothesen

In der Einleitung (s. Abs. 1.7, S. 26) sind folgende vier Hypothesen formuliert, die anhand dieser Arbeit belegt oder widerlegt werden können:

> **Hypothese 1:** Es ist möglich, eine MODPL für SKMS zu entwickeln, d.h. es existiert eine ausreichende Anzahl gemeinsamer Merkmale für bestehende SKMS. Gleichzeitig gibt es auch ausreichend Unterschiede, so dass eine Systemfamilie entstehen kann.

Diese Hypothese ist durch die Existenz von (1) Merkmalsanalyse (vgl. Abschnitt B, S. 359ff.), (2) Merkmalsmodell (vgl. Abschnitt C, S. 435ff.) und (3) Domänenmodell (vgl. Abschnitt 6, S. 141ff.) belegt. Die MOD2-SKM Produktlinie zeigt, dass es möglich ist, eine modellgetriebene Produktlinie für SKMS zu entwickeln. Insbesondere die Validierung des Merkmalsmodells (vgl. Abschnitt 5.4, S. 125ff.) demonstriert, dass sich mit der MOD2-SKM Produktlinie unterschiedliche SKMS wie Subversion und GIT modellieren lassen.

> **Hypothese 2:** Die Verfahren der SKM-Domäne lassen sich lose koppeln, so dass sie modularisiert werden können.

Diese Hypothese lässt sich anhand der Kopplungs-Analyse (vgl. Abschnitt 6.5, S. 296ff.) weder belegen noch widerlegen. Es ist leider nicht möglich, die Kopplung als lose zu klassifizieren, da keine Skala existiert. Auch ein Vergleich zu anderen Produktlinien ist im Rahmen dieser Arbeit nicht möglich. Es gibt jedoch Hinweise, dass eine lose Kopplung vorliegt: (1) In der ganzen Modulbibliothek existieren nur zwei Abhängigkeiten, die die Kombination der Module einschränken. (2) Die Kopplungsanzahl und -stärke ist für eine Vielzahl von Modulen relativ ähnlich und relativ niedrig (s. Quartile in Tab. 6.5, S. 307).

> **Hypothese 3:** Eine MODPL für SKMS erlaubt es, SKMS systematisch zu beschreiben und zu vergleichen, da SKMS sich anhand der Merkmale des Merkmalsmodells kategorisieren lassen.

Diese Hypothese lässt sich wiederum mit Hilfe der Validierung des Merkmalsmodells und der Merkmalsanalyse belegen. Mit ihrer Hilfe lassen sich bereits 290.304.000 unterschiedliche SKMS beschreiben. Insbesondere die Erweiterungspunkte (vgl. Abschnitt 5.5, S. 129ff.) zeigen, dass die Merkmalsanalyse noch erweitert werden kann, um die Produktfamilie zu erweitern und mehr SKMS aufzunehmen.

> **Hypothese 4:** Eine MODPL für SKMS erlaubt es, SKMS schnell an veränderte Anforderung anzupassen, da sich der Entwicklungsaufwand eines SKMS im Idealfall auf das Erstellen einer Konfiguration beschränkt. Bei neuen Merkmalen beschränkt sich der Aufwand auf die Modellierung der neuen Funktionalität, da bestehende Merkmale wiederverwendet werden können.

Die letzte Hypothese lässt sich ebenfalls anhand der Anzahl der konfigurierbaren SKMS und der geringen Abhängigkeiten der Modulbibliothek belegen. Für 290.304.080 SKMS wurde der Entwicklungsaufwand auf einen Konfigurationsschritt reduziert. Für neue Merkmale gilt: Kann ein Modul einer Schnittstelle aus den Kernmodulen zugeordnet werden, muss das neue Modul lediglich diese Schnittstelle implementieren. Um z.B. ein neues Produktmodell versionieren zu können, ist es ausreichend, ein einziges Modul auf Basis des Kernmoduls *core.product* zu implementieren (vgl. Abschnitt 6.3.8, S. 188ff.) – anschließend lässt sich dieses Modul mit einem SKM-Server aus bereits existierenden Modulen kombinieren. Gibt es für ein neues Merkmal keine Schnittstellen, so muss ein neues Kernmodul integriert werden. Auch hier erleichtern die Kernmodule die Integration bereits bestehender Module, aber der tatsächliche Aufwand lässt sich nur noch schwer abschätzen.

8.2. Modularisierung von SKMS

Die MOD2-SKM Produktlinie ist die erste nicht-triviale Produktlinie für SKMS. Sie zeigt, dass sich die Konzepte der SKM-Domäne objektorientiert modellieren und insbesondere

auch entkoppeln lassen. Sie bietet folgende Vorteile gegenüber herkömmlicher SKMS-Entwicklung: Neue SKMS lassen sich durch Konfigurieren des Merkmalsmodells und einen anschließenden automatisierten Konfigurationsschritt erzeugen (vgl. Abschnitt 3.5, S. 61ff.). So lassen sich auch SKM-Konzepte systematisch untersuchen, indem z.B. eine Serie von SKMS generiert wird, die sich nur in einem Modul unterscheiden. Auch können gezielt einzelne Konzepte verglichen werden (vgl. Abschnitt 5.4, S. 125ff.). Und sowohl neue als auch existierende SKMS lassen sich anhand ihrer Konfiguration klassifizieren. Insbesondere interne SKM-Verfahren werden explizit genannt und müssen nicht aus dem Quellcode abgeleitet werden (vgl. Abschnitt 5.4, S. 125ff.).

Neue SKM-Konzepte können mit bestehenden kombiniert werden, was ihren Entwicklungsaufwand erheblich reduziert. Einmal modelliert, werden diese Teil der Modulbibliothek und sind bei weiteren Neuentwicklungen verfügbar. Zusätzlich sind sie dadurch explizit dokumentiert und können anhand grafischer Modelle nachvollzogen werden (vgl. Abschnitt 6.4, S. 211ff.). Insbesondere ihre Abhängigkeiten zu anderen SKM-Konzepten sind explizit im Merkmalsmodell und der Architektur des Domänenmodells erfasst und unterstützen die Entwickler bei der Konfiguration (vgl. Abschnitt 5.6.2, S. 137ff.).

Aus den bestehenden Modulen und ihren Abhängigkeiten lassen sich bereits einige Erkenntnisse über SKMS gewinnen. So zeigen z.B. die Konfigurationen bestehender SKMS, dass zwischen ihnen viele Gemeinsamkeiten bestehen (vgl. Abschnitt 5.4, S. 125ff.). So unterscheiden sich z.B. die Modelle für versionszentrierte und verwobene Integration von Produkt- und Versionsraum nur in einer einzigen Aktivität, d.h. die versionszentrierte Integration kann als Spezialfall der verwobenen Integration angesehen werden (vgl. Abschnitt 6.4.16, S. 251ff.). Ein SKMS, das bei zusammengesetzten Produktmodell-Elementen auch Unterelemente als Wurzel eines Arbeitsbereichs verwenden kann, erfordert die Integration eine komplizierte Erweiterung des Speicherverfahrens (vgl. Abschnitt 6.4.16, S. 251ff.).

Auffällig ist auch, dass das Produktmodell gezielt vom konkreten SKMS entkoppelt werden kann, d.h. viele SKM-Konzepte setzen keine konkreten Produktmodell-Typen (z.B. ein Dateisystem) voraus (vgl. Abschnitt 6.3.8, S. 188ff.). Lediglich die Verfahren zum Verschmelzen (vgl. Abschnitt 6.4.10, S. 233ff.) und der Delta-Berechnung (vgl. Abschnitt 6.4.23, S. 283ff.) sind direkt vom Produktmodell abhängig.

Die Module der Persistenzmechanismen zeigen, dass SKMS auf bestehende Mechanismen aufsetzen können und sollten. Ansonsten muss aufwändig sichergestellt werden, dass die Operationen des SKMS die Eigenschaften von ACID-Transaktionen besitzen (vgl. Abschnitt 6.4.12, S. 237ff.). Dabei sollte jedoch geprüft werden, ob Verfahren zum Speicherplatz-Sparen mit dem Persistenzmechanismus harmonieren (vgl. Abschnitt 6.4.11, S. 235ff.).

8.3. Konzepte modellgetriebener Produktlinien

Die MOD2-SKM Produktlinie ist eine modellgetrieben entwickelte, komplexe Produktlinie. Sie zeigt, dass sich mit dem MODPL-Ansatz und -Werkzeugkasten von Thomas Buchmann [Buc10] Produktlinien modellgetrieben entwickeln lassen. Dieser Ansatz bie-

tet mehrere Vorteile gegenüber herkömmlicher Produktlinien-Entwicklung: Größter Vorteil ist, dass das manuelle Ableiten bzw. Zusammensetzen eines Software-Produktes aus dem Domänenmodell durch eine automatisierte Konfiguration ersetzt wird. Dabei kann sowohl sofort lauffähiger Quellcode generiert werden als auch ein konfiguriertes Modell, um es mit produktspezifischen Komponenten zu erweitern [Buc10].

Durch die explizite Verknüpfung der Modellelemente aus Merkmalsmodell und Domänenmodell sind die Abhängigkeiten von Merkmalen und Modell-Elementen zurückverfolgbar (vgl. Abschnitt 5.6, S. 133ff.). Und auch die Architektur der Produktlinie lässt sich anhand von Paketdiagrammen grafisch darstellen und ebenfalls explizit mit den Merkmalen in Verbindung bringen (vgl. Abschnitt 6.2, S. 155ff.). Ebenso lässt sich die Konfiguration eines Produktes grafisch im Domänenmodell darstellen und verwendete und nicht verwendete Modellelemente parallel anzeigen [Buc10].

8.3.1. Entwicklungsprozess

Die Entwicklung des Merkmals- und Domänenmodells erfolgt inkrementell und iterativ durch die Domänenentwickler. Es existiert keine fest definierte „Entwicklungsrichtung“ vom Merkmals- zum Domänenmodell hin, da sich aus dem Domänenmodell auch Anforderungen an das Merkmalsmodell ableiten lassen. Mit einem Merkmalsmodell und einem Domänenmodell generieren die Anwendungsentwickler dann konfigurierte Produkte [Buc10].

Domänenanalyse

Zunächst modellieren die Domänenentwickler aus einer kleinen Auswahl an Quellen ein erstes Merkmalsmodell. Sie erweitern dieses inkrementell, indem sie weitere Quellen analysieren und neue Merkmale ableiten bzw. definieren. Diese werden in das bestehende Merkmalsmodell integriert, da z.B. die Quellen die gleichen Konzepte unterschiedlich benennen oder die Domäne aus einer anderen Sicht beschreiben [KKL+98]. Die inkrementelle Analyse vergrößert das Merkmalsmodell kontinuierlich und erfordert – besonders in der Anfangsphase – häufige Umstrukturierungen.

Diese inkrementelle Entwicklung lässt sich anhand der Entwicklung des MOD2-SKM Merkmalsmodells verdeutlichen. Abbildung 8.1 zeigt die allererste Version des Merkmalsmodells mit 20 Merkmalen. Es erfasst lediglich einige Aspekte des Versionsraums (vgl. Abschnitt 4.3.1, S. 79ff.), basierend auf den Analysen und Begriffen aus [CW98]. Die Merkmale erfassen die Darstellung durch unterschiedliche Versionsgraphen („Version Graph“) mit Verzweigungen („Branch“) und Verschmelzungen („Merge“). Zusätzlich muss entschieden werden, ob das SKMS Revisionen („Revisions“), Varianten („Variants“) oder beides verwenden soll und ob die Versionierung zustandsbasiert („State based“) oder änderungsbasiert („change based“) erfolgt. Optional können auch symmetrische („symmetric“) oder gerichtete („directed“) Deltas eingesetzt werden. Die Unterscheidung zwischen extensionaler und intensionaler Versionierung ist das letzte Merkmal dieses initialen Entwurfs.

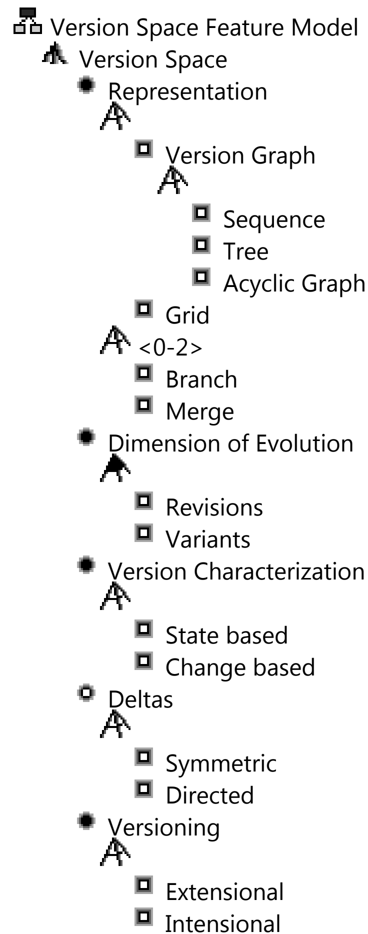


Abbildung 8.1.: Initiales Merkmalsmodell

Einfluss des Domänenmodells

Das Merkmalsmodells muss jedoch nicht nur an die Quellen der Domänenanalyse, sondern auch an das Domänenmodell angepasst werden. Denn die Verwendung der Merkmalsmarkierungen stellt zusätzliche Anforderungen an das Merkmalsmodell. Zum einen existieren Abhängigkeiten zwischen einzelnen Modulen, die nicht in der Domänenanalyse erkannt wurden oder sich aus technischen Einschränkungen ableiten. So kann z.B. das Dateisystem-Produktmodell (vgl. Abschnitt 6.4.14, S. 240ff.) – wegen einer proprietären Software-Bibliothek – nur auf einem Windows-Betriebssystem laufen, obwohl Dateisysteme konzeptionell auch in anderen Betriebssystemen existieren. Dieser Sachverhalt ist Hauptargument für die Trennung zwischen einem (idealisierten) Merkmalsmodell zur Klassifikation von SKMS und einem zur Konfiguration der MOD2-SKM Produktlinie (vgl. Abschnitt 8.3.2, S. 334ff.).

Als iterative Arbeitsschritte kommt die Anpassung bereits existierender Konfigurationen und markierter Domänenmodelle hinzu. Die Anwendungsentwickler erstellen Konfi-

gurationen für Software-Produkte, indem sie die Merkmale eines Merkmalsmodells binden [AC04, Buc10]. Eine Konfiguration ist somit die Instanz eines Merkmalsmodells. Nach jeder Erweiterung bzw. Umstrukturierung des Merkmalsmodells müssen die bereits instanzierten Konfigurationen an die Änderungen angepasst werden. Hier ist eine synchrone Übertragung der Änderungen von großem Vorteil, da so Umbenennungen und Verschiebungen korrekt übernommen werden können. Dies gilt auch für Merkmalsmarkierungen im Domänenmodell. Nach Umstrukturierungen muss erneut geprüft werden, ob z.B. gelöschte oder umbenannte Markierungen an Modellelementen existieren. Ebenso kann eine vorher gültige Merkmalskombination an einem Element auf einmal unerfüllbar geworden sein. Letztendlich müssen die Domänenentwickler bei der Überarbeitung des Merkmalsmodells auch die Auswirkungen auf Konfigurationen und Domänenmodell in ihre Überlegung mit einbeziehen.

Domänenmodellierung

Zunächst modellieren die Domänenentwickler eine erste Version des Domänenmodells, indem sie Schnittstellen für die Kernmodule definieren und eine erste (einfache) Implementierung der Schnittstellen der Modulbibliothek hinzufügen. So sind z.B. in MOD2-SKM die *base*-Module, wie z.B. *history.base* (vgl. Abschnitt 6.4.1, S. 213ff.) oder *storage.base* (vgl. Abschnitt 6.4.21, S. 279ff.) entstanden. Alternativ ist auch die Modellierung eines Standard-Kernmoduls denkbar, aus dem später die Schnittstellen abgeleitet werden. Dies war z.B. das Vorgehen bei Entwicklung der Benutzerverwaltung *user.flat* (vgl. Abschnitt 6.4.26, S. 290ff.).

Die Erweiterung von Kernmodulen und Modulbibliothek erfolgt inkrementell, so dass die Entwicklung von Merkmals- und Domänenmodell synchronisiert werden kann. In der Praxis wuchs jedoch das Merkmalsmodell von MOD2-SKM deutlich schneller als das Domänenmodell. Dies ist zwar auch auf die Anforderung der SKMS-Klassifikation zurückzuführen – die ein separates Merkmalsmodell übernehmen sollte – aber dennoch ist eine gut koordinierte inkrementelle Entwicklung von Merkmals- und Domänenmodell erforderlich. An dieser Stelle benötigen die Entwickler eine präzisere Beschreibung bzw. Anleitung.

Iterative Aufgaben sind das Generieren und Testen der gesamten Produktlinie. Nach jeder Erweiterung muss sowohl geprüft werden, ob sich die bereits konfigurierten Software-Systeme noch erzeugen lassen, ob der generierte Quellcode noch kompiliert und ob bereits funktionierende Module bzw. konfigurierte Software-Systeme noch korrekt arbeiten. Letzteres ist auf Grund der hohen Anzahl an generierbaren Systemen kaum machbar und ein mögliches Thema zukünftiger Forschung.

Konfiguration

Die Anwendungsentwickler arbeiten mit Kombinationen von Merkmals- und Domänenmodellen. An dieser Stelle wird die Frage nach der Zugehörigkeit von Merkmals- und Domänenmodell deutlich. Für die Konfiguration ist ein Merkmalsmodell erforderlich, das alle Variationspunkte des Domänenmodells enthält. Anwendungsentwickler benötigen, im

Gegensatz zu Domänenentwicklern, keine Erweiterungspunkte oder teilweise implementierte Merkmale. Sie erschweren lediglich die Konfiguration, da sie keinerlei Auswirkungen auf das konfigurierte Software-System haben. Sie brauchen lediglich ein reduziertes Merkmalsmodell (vgl. Abschnitt 5.6, S. 133ff.). Diese lässt sich ggf. automatisiert aus einem erweiterten Merkmalsmodell ableiten.

Rollen in MODPL

Domänenentwickler müssen eine Domänenanalsyse durchführen und daraus ein Merkmalsmodell erstellen. Außerdem müssen sie ein Domänenmodell modellieren und mit Merkmalsmarkierungen versehen. Sie erfüllen damit eine Vielzahl von Rollen: Sie untersuchen als Domänenanalyst die Domäne mit Hilfe von Quellen. Sie entwerfen als Software-Architekt sowohl die Architektur der Produktlinie als auch der konfigurierten Systeme. Sie modellieren als Software-Entwickler das Verhalten einzelner Module. Evtl. müssen sie auch die Rolle eines Software-Testers annehmen, da der MODPL-Prozess nicht explizit klärt, welche Rolle für die Produktlinien-Tests verantwortlich ist. Im Gegensatz dazu, müssen Anwendungsentwickler lediglich eine Konfiguration erstellen, und können dann das zugehörige konfigurierte Modell oder den Quellcode erzeugen. Die Bearbeitung der konfigurierten Systeme basiert dann vollständig auf herkömmlichen Entwicklungsmethoden. Ihnen obliegt dann auch das Testen des endgültigen Produkts.

Effektiv sind also viel mehr als zwei Rollen für die Entwicklung einer modellgetriebenen Produktlinie notwendig:

1. **Domänenanalysten** erstellen anhand von Quellen und eigenem Wissen eine Domänenanalyse. Sie benötigen gute Kenntnisse der Domäne und nutzen das Merkmalsmodell als Grundlage ihrer Diskussionen. Sie fügen dabei u.a. neue Merkmale ein und passen das Merkmalsmodell ihrer Analyse an.

2. **Software-Architekten** entwerfen die Architektur der Produktlinie und modellieren insbesondere die Kernmodule als zentrale Schnittstellen. Sie benötigen gute Kenntnisse von Architekturprinzipien und -strukturen, denn sie müssen dabei die Architektur (bzw. Architekturen!) der konfigurierten Software-Systeme in die Produktlinien-Architektur integrieren.

3. **Software-Entwickler** modellieren die einzelnen Module, in dem sie auf die bereits modellierten Kernmodule zurückgreifen. Sie benötigen gute Modellierkenntnisse, da sie Algorithmen und Datenstrukturen einzelner Module umsetzen müssen. Diese müssen sie mit Merkmalen markieren, wobei hier die Kenntnisse über einen kleinen Teilbereich der Domäne ausreichen.

4. **Software-Tester** müssen die Verfahren zum Testen einer ganzen Produktlinie entwickeln und umsetzen. Sie benötigen gute Kenntnisse im Testen einer riesigen Anzahl an Systemen, da die Anzahl konfigurierter Software-Systeme mit jedem neuen Merkmal nahezu exponentiell ansteigt.

5. **Anwendungsentwickler** sind ebenfalls Software-Entwickler, die – wie in [Buc10] beschrieben – eine Konfiguration erstellen und aus der Produktlinie generieren. Anschließend können sie noch über herkömmliche Entwicklungsmethoden das Software-Produkt verändern oder erweitern.

Konsistenz Merkmals- und Domänenmodell

Merkmals- und Domänenmodell werden getrennt voneinander entwickelt und sind über die Merkmalsmarkierungen miteinander verknüpft [Buc10]. Jede der o.g. Rollen besitzt eine unterschiedliche Sicht auf das Merkmals- bzw. Domänenmodell und stellt daher unterschiedliche Anforderungen an ihre Konsistenz. Die **Domänenanalysten** nutzen das Merkmalsmodell zum Erfassen ihrer Domänenanalyse. Sie entwerfen u.a. neue Merkmale, die in die Produktlinie aufgenommen werden sollen. Ein Merkmalsmodell kann daher teilweise realisierte Merkmale oder Erweiterungspunkte enthalten, die nicht im Domänenmodell umgesetzt werden. Die ist Aufgabe der **Software-Entwickler**, welche die einzelnen Module implementieren. Ihre Aufgabe ist es, beide Modelle konsistent zu halten, in dem sie Module für neue Erweiterungspunkte implementieren. Gleichzeitig definieren sie durch Abhängigkeiten der Module evtl. zusätzliche Einschränkungen, die im Merkmalsmodell festgehalten werden müssen. **Anwendungsentwickler** und **Tester** benötigen dagegen ein Merkmalsmodell, das keine Erweiterungspunkte mehr enthält. Diese haben keine Auswirkung auf die konfigurierten Software-Systeme und erhöhen nur die Komplexität der Konfigurationen. Wie weiter oben erwähnt, benötigen sie eine, ggf. automatisierte, Reduktion des Merkmalsmodells auf für die Konfiguration relevante Merkmale.

8.3.2. Konzeption Merkmalsmodell

In MOD2-SKM dient das Merkmalsmodell (147 Merkmale, 34 aussagenlogische Einschränkungen) sowohl zur Klassifikation von SKMS als auch zur Auszeichnung des Domänenmodells. Diese Doppelfunktion führt zu unterschiedlichen Anforderungen an das Produktmodell, die sich nicht miteinander vereinbaren lassen: Zur Klassifikation sollen möglichst viele Merkmale im Merkmalsmodell vorhanden sein. Die Strukturierung der Merkmale sollte in der Domäne gebräuchlichen Kategorien und Konzepten entsprechen. Zur Konfiguration der Produktlinie sollten nur die Merkmale im Modell enthalten sein, die (1) als Merkmalsmarkierung dienen oder (2) diese Merkmale strukturieren oder (3) diese Merkmale über Einschränkungen konfigurieren. Insbesondere ist auch eine stärkere Anpassung der Merkmalsmodellstruktur an die Architektur des Domänenmodells notwendig. Daher sollte – ähnlich dem reduzierten Merkmalsmodell (vgl. Abschnitt 5.6, S. 133ff.) – für jede Aufgabe ein eigenes Merkmalsmodell existieren.

Die Identifikation der Merkmale, sowie ihre Strukturierung im Merkmalsmodell, ist auf Grund der übersichtlichen Zahl an Modellelementen einfach zu erlernen. Dadurch können Produktlinien-Entwickler ihre Aufmerksamkeit der Analyse der Dokumente und der Gliederung des Modells widmen. Gleichzeitig bietet das Merkmalsmodell genug Freiheiten, um die FORM-Ebenen und die Hilfsmerkmale einzufügen (vgl. Abschnitt 5.2, S. 100ff.). Ähnlich wie bei einer Anwendungsfall-Analyse [Bal01] steigt die Qualität des Modells

mit aussagekräftigen Bezeichnern. Weiterhin bietet sich das Merkmalsmodell, auf Grund der einfachen Syntax, als Diskussionsgrundlage bei der Konzeption der Produktlinie an.

Für ein Merkmalsmodell mit über hundert Merkmalen sind bereits zusätzliche Maßnahmen zu seiner Strukturierung nötig. Daher wurde für das MOD2-SKM Merkmalsmodell die FORM-Methode [KKL+98] eingesetzt, und nicht, wie in MODPL vorgesehen, die FODA-Methode [KCH+90, Buc10]. FORM basiert auf FODA [KKL+98] und bietet zusätzlich vier Ebenen, um unterschiedliche Sichten (z.B. von Benutzer und Entwickler) auf die Domäne zu modellieren. Insbesondere existiert auch eine „Implementierungs"-Beziehungen zwischen Merkmalen verschiedener Ebenen. So kann z.B. ein komplexer Sachverhalt auf der Entwickler-Ebene durch ein abstraktes Merkmal auf der Benutzer-Ebene ausgedrückt werden. Zusätzlich wurden noch weitere Ebenen zur Strukturierung eingesetzt, um die Merkmale gemäß einzelner Anforderungsbereiche aus der SKM-Domäne zu gliedern.

Es ist auch zu beachten, dass sich die Struktur des Merkmalsmodells auf die Verwendung der Merkmalsmarkierungen auswirkt. Anstatt ein Modellelement mit mehreren Merkmalen zu markieren, kann auch ein neues Merkmal eingeführt werden, das über aussagenlogische Einschränkung von den anderen Merkmalen abhängt. Nur dieses wird dann zur Markierung verwendet. So wird die Abhängigkeit in das Merkmalsmodell verlagert. Dieser Vorgang lässt sich auch umkehren und so die Anzahl der aussagenlogischen Formeln reduzieren und stattdessen mehrere Merkmalsmarkierungen verwenden. Ob eine Modellierungsweise der anderen vorzuziehen ist, ließ sich nicht erkennen. Die Entwickler sollten daher von Fall zu Fall entscheiden, welche Technik den betroffenen Sachverhalt präziser erfasst.

Manuelle Konfiguration des laufzeitkonfigurierbaren Servers

Der laufzeitkonfigurierbare Server (vgl. Abschnitt 6.4.20, S. 273ff.) wird nicht vollständig automatisch generiert. Stattdessen müssen einige Methoden manuell aus dem Quellcode entfernt werden, da keine Konfiguration für diesen Server existiert. Problem ist, dass letztendlich eine SKMS für eine partiell gebundene Konfiguration erzeugt werden soll, d.h., Module, die bei einer vollständig gebundenen Konfiguration als Alternativen existieren (die Merkmale schließen sich gegenseitig aus, vgl. Abschnitt 5.3.3, S. 114ff.), sollen nun gemeinsam in einem SKMS existieren. Um eine partielle Konfiguration zu realisieren, sind jedoch zwei Merkmalsmarkierungen – verbunden durch ein logisches „oder" – notwendig. Somit ließe sich ausdrücken, dass ein Element Teil des SKMS ist, wenn das entsprechende Merkmal aus der Gruppe in der Konfiguration enthalten, oder eine Laufzeitkonfiguration gewünscht ist. Im Rahmen dieser Arbeit ließ sich diese Thematik jedoch nicht ausführlich behandeln, so dass hier zukünftige Untersuchungen nötig sind.

8.3.3. Konzeption Domänenmodell

Das MOD2-SKM Domänenmodell ist in Kernmodule und eine Modulbibliothek unterteilt. Diese Aufteilung basiert u.a. auf der Trennung zwischen gemeinsamen und variablen Elementen der Produktlinie. Hierbei hat sich insbesondere Schnittstellen-Abstraktion als

nützlich erwiesen: Mehrere Module der Modulbibliothek basieren auf der Schnittstelle des gleichen Kernmoduls und sind somit austauschbar (vgl. Abschnitt 6.2.1, S. 155ff.). Dazu ist es auch nötig, die Abhängigkeiten unter den Modulen der Bibliothek so gering wie möglich zu halten.

Eine Abhängigkeit zwischen Modellelementen lässt sich jedoch nur erkennen, wenn auch eine explizite Beziehung zwischen den beiden Elementen besteht. Mit Hilfe textueller Referenzen (vgl. Abschnitt 6.5.2, S. 299ff.) lassen sich – absichtlich oder unabsichtlich – Abhängigkeiten verstecken. Bei Analyse des Modells sieht es zwar so aus, als existiere die Abhängigkeit nicht, doch stattdessen wird sie erst zur Laufzeit überprüft. Davon ist abzuraten, stattdessen sollten die Domänenentwickler Abhängigkeiten explizit modellieren und sogar in das Merkmalsmodell übertragen (vgl. Abschnitt 6.5.3, S. 300ff.).

Merkmalsmarkierungen

Auf Grund der Propagationsregeln des MODPL-Editors sind nur 57 Elemente des Domänenmodells mit Merkmalen markiert. Nach der Propagation sind es ca. 3.000 Elemente. Ohne die automatisierte Propagation wäre eine Produktlinie dieser Größe bereits nicht mehr zu handhaben. Die hohe Zahl der propagierten Markierungen ist auf die vielen markierten Pakete zurückzuführen, da die Markierung an alle Modellelemente im Paket propagiert werden. In MOD2-SKM sind folgende unterschiedliche Modellelement-Typen markiert:

- 20 Pakete
- 8 Klassen
- 7 Methoden
- 22 Aktivitäten in Story-Diagrammen

Je grobgranularer das Element, zu desto mehr Elementen wird das Merkmal i.d.R. später propagiert. Allerdings spielt beim Anbringen einer Merkmalsmarkierung nicht nur die Zahl der „erreichbaren“ Modellelemente eine Rolle. Vielmehr bietet jeder Element-Typ unterschiedliche Vor- und Nachteile bei der Modellierung von Variationspunkten.

Eine **Paketmarkierung** bedeutet, dass alle Elemente des Paketes ebenfalls von diesem Merkmal abhängen. Da die Module der MOD2-SKM Produktlinie auf Paketen basieren (vgl. Abschnitt 6.2.1, S. 155ff.), geht ihre Markierung konform mit der Architektur der Produktlinie. Abbildung 6.14 (s. S. 158) zeigt, dass es ausreicht, lediglich das oberste Paket jedes Moduls zu markieren. Mehrere nebeneinander liegende, markierte Pakete zeigen deutlich, dass es sich um einen Variationspunkt der Produktlinie handelt. Da jede Variante als Paket einen eigenen Namensraum besitzt, können sie auch – z.B. beim generieren aller Module – in einem konfigurierten SKMS parallel existieren. So lässt sich die Kompilierbarkeit des Quellcodes aller Varianten mit nur einem generierten System testen. Abhängigkeiten zu einer Variante sind direkt über die Paket-Importe erkenn- und messbar. Fehlt das Merkmal in der Konfiguration, obwohl es benötigt wird, kompiliert der Quellcode – auf Grund der fehlenden Komponenten – nicht.

Eine **Klassenmarkierung** bedeutet, dass alle Beziehungen, Eigenschaften und Methoden der Klasse ebenfalls von diesem Merkmal abhängen. Der Variationspunkt ist i.d.R. nicht mehr im Paketdiagramm erkennbar, es sei denn, ein verpflichtendes Elternmerkmal wird zur Markierung verwendet. Im Klassendiagramm sind die alternativen Klassen anhand ihrer Markierungen gut erkennbar. Da jede Klasse einen eindeutigen Namen besitzen muss, bildet jede einen eigenen Namensraum, so dass auch hier die Kompilierbarkeit aller Varianten parallel geprüft werden kann. Die Abhängigkeiten zu einer Variante lassen sich jedoch nur noch auf Ebenen der Element-Importe erkennen und messen. Fehlt eine Klasse auf Grund einer Fehlkonfiguration führt dies zu Kompilierfehlern.

Eine **Methodenmarkierung** bedeutet, dass nur die Methode und ihr Verhalten von diesem Merkmal abhängen. Der Variationspunkt ist nicht mehr im Paketdiagramm erkennbar und auch im Klassendiagramm kann eine Methodenmarkierung nur schwer ihrer Methode zugeordnet werden (vgl. Abschnitt 6.1.1, S. 149ff.). Da mehrere Methoden mit gleichem Namen und gleicher Signatur existieren dürfen, lässt sich so alternatives Verhalten ohne Veränderung der Schnittstelle modellieren. Jedoch lässt sich nun nicht mehr die Kompilierbarkeit aller Varianten parallel testen, da gleichnamige Methoden nicht in einer Klasse existieren dürfen. Die Abhängigkeiten von einer Variante lässt sich über keine Import-Beziehung mehr feststellen und Fehlkonfiguration führen entweder zu Kompilierfehlern (wenn mehrere Methoden generiert werden) oder zu einem Fehlverhalten zur Laufzeit (wenn nur genau eine Methode generiert wird).

Eine **Aktivitätsmarkierung** bedeutet, dass nur eine einzelne Aktivität von diesem Merkmal abhängt. Der Variationspunkt ist nun nicht einmal mehr im Klassendiagramm erkennbar, sondern nur noch im Storydiagramm. Außerdem lässt sich das Domänenmodell nur noch auf konfigurierten Quellcode abbilden: Ohne die markierte Aktivität existiert in einem konfigurierten Fujaba-Modell keine Transition zwischen Vorgänger- und Nachfolger-Aktivität. Das alternative Verhalten lässt sich nicht mehr parallel auf Kompilierbarkeit prüfen, da die Methode nur in genau einer Variante generiert werden kann. Die Abhängigkeit zu einer bestimmten Variante lässt sich weder an Paket- noch an Element-Importen erkennen und eine Fehlkonfiguration führt immer zu Fehlverhalten zur Laufzeit.

Die beiden letzteren Markierungsarten wurden im MOD2-SKM Domänenmodell verwendet. Sie sind jedoch für die Produktlinien-Entwickler schwer zugänglich, da sie kaum bzw. gar nicht in den Klassendiagrammen erkennbar sind. Selbst für Klassenmarkierungen sollte daher in den Paketdiagrammen ein verpflichtendes Elternmerkmal zu Dokumentationszwecken erscheinen. Weiterer Nachteil der Methoden- und Aktivitätsmarkierungen ist das Verhalten bei Fehlkonfigurationen: Insbesondere Aktivitätsmarkierungen führen oft nur zu minimalen Verhaltensänderungen, die dann bei Testfällen zu Fehlschlägen führen, aber sich nur schwer lokalisieren lassen. Beide Markierungsarten sollten daher nur nach eingehender Prüfung verwendet werden. In vielen Fällen können sie durch Klassenmarkierungen ersetzt werden, indem für jede Markierung eine eigene Subklasse modelliert wird.

Mehrere Modelle

An der MOD2-SKM Produktlinie sind mehrere Entwickler beteiligt. Hauptaufgabe ist dabei die Modellierung neuer Module für die Modulbibliothek [Mat09, Pöh09, Bär10]. Zur Unterstützung mehrerer Entwickler ist gerade hier die Aufteilung auf mehrere Modelle notwendig. Es sollte ein Modell für die Kernmodule existieren, während jedes Modul der Modulbibliothek ein eigenes Modell erhält. Dieses besitzt dann Abhängigkeiten zum Kernmodell und ggf. weiteren Modulen. So können die Entwickler separat an ihren jeweiligen Modellen arbeiten.

8.3.4. Korrektheit und Testen

Mit Hilfe der MOD2-SKM lassen sich **290.304.000** Konfigurationen, und damit ebenso viele unterschiedliche SKMS erzeugen. Die Generierung jedes SKMS dauert – mit rudimentären Tests – etwa eine Minute. Ohne Parallelisierung der Tests würde somit ein vollständiger Test der gesamten Produktlinie 552 Jahre dauern. Hier ist die Erforschung von Testverfahren für Produktlinien unbedingt notwendig.

Komplette Produktlinie kompilieren

Um wenigstens die syntaktische Korrektheit des ausführbaren Domänenmodells sicherzustellen, reicht es aus, jedes Modul einmal zu generieren und zu kompilieren. Dazu kann das Domänenmodell unkonfiguriert auf Quellcode abgebildet werden, da ohne Konfiguration für alle Elemente – unabhängig von ihren Merkmalsmarkierungen – Quellcode erzeugt wird. Dieser sollte syntaktisch korrekt sein. Nur wenn die „Lücke“ mit mehreren gleichnamigen Methoden genutzt wird (vgl. Abschnitt 6.1.1, S. 149ff.), kommt es zu Kompilierfehlern. Wird diese Markierungsart ausgeschlossen, kann ein Entwickler Kompilierfehler der Produktlinie suchen und beseitigen, wie bei einfachen Fujaba-Modellen.

Probleme beim Testen

Bei den Tests der MOD2-SKM Produktlinie handelt es sich um manuell implementierte Testfälle. In allen Fällen, in denen das Testergebnis von der Konfiguration des SKMS abhängt, ist es notwendig, einen Testfall für jedes unterschiedliche Testergebnis zu schreiben. Da ein konfiguriertes SKMS seine Konfiguration nicht explizit „kennt“, ist es schwer, die Testfälle den Konfigurationen zuzuordnen. Auch ähneln sich die Testfälle unterschiedlicher Realisierungen des gleichen Kernmoduls, so dass es notwendig ist, die Wiederverwendung der Testfälle zu planen.

Bei der manuellen Implementierung wiederverwendbarer Testfälle erweist sich die feingranulare Markierung einzelner Aktivitäten (vgl. Abschnitt 6.1.1, S. 149ff.) als schwer zu entdeckende Fehlerquelle. Ein Testfall für ein Merkmal, das einem Paket zugeordnet ist, kompiliert nur, wenn die entsprechenden Komponenten Teil des konfigurierten SKMS sind. Eine Merkmalsmarkierung an einer Aktivität verändert lediglich das Verhalten einer Methode, so dass eine fehlerhaften Zuordnung von Konfiguration und Testfall zu einem fehlgeschlagenen Test führt. Generell wird eine explizite Unterstützung der Zuordnung

zwischen Merkmalen und Testfällen benötigt, z.B. durch Modellierung der Testfälle im MODPL-Werkzeugkasten – einschließlich der Verwendung von Merkmalsmarkierungen (zwecks Wiederverwendung von Testfällen).

8.3.5. Konzeption der Auslieferung

Ein konfiguriertes SKMS muss nach Generierung auch ausgeliefert (engl. deployed) werden. Hierbei stellt die Client-Server-Architekturstruktur die Anforderung, dass zwei getrennte Software-Komponenten existieren (vgl. Abschnitt 6.2.2, S. 157ff.): Zum einen der Repositoriums-Server, der auf einem zentralen Computersystem arbeitet, und zum anderen die Arbeitsbereichs-Clienten, die an die einzelnen Entwickler ausgeliefert werden. Diese Aufteilung des konfigurierten SKMS ist nicht mehr Teil des MODPL-Werkzeugkastens und muss vom SKMS-Entwickler selbst vorgenommen werden. Im Falle von MOD2-SKM existieren Stapelverarbeitungs-Skripte, die die kompilierten Klassen auf das Server- bzw. Client-System verteilen. Dabei lassen sich die Elemente nicht eindeutig aufteilen, da z.B. die Komponenten des Kommunikations-Teilsystems in beiden Systemen vorhanden sein müssen. Eine Modellierung der Quellcode-Verteilung auf unterschiedliche Komponenten, bzw. das Erzeugen mehrerer konfigurierter Modelle, ist für komplexere Architekturen unumgänglich.

8.4. Werkzeuge und Modelle des MODPL-Werkzeugkastens

Der MODPL-Werkzeugkasten ist das hauptsächlich genutzte Entwicklungswerkzeug der MOD2-SKM Produktlinie. Abbildung 8.2 (vgl. Abb. 3.1, S. 51) zeigt, dass es sich um fünf Werkzeuge (Feature Plugin, MODPL-Editor, MODPL-Paketdiagrammeditor, Fujaba, MODPL-Konfigurator) und fünf Modelle (Merkmalsmodell, Konfigurationen, Paketdiagramm, Domänenmodell, konfiguriertes SKMS) handelt, die miteinander kombiniert wurden (vgl. Abschnitt 3, S. 51ff.) [Buc10].

8.4.1. Merkmalsmodell und Konfigurationen

Die Architektur der MOD2-SKM Produktlinie unterstützt gezielt die Erweiterbarkeit neuer Produktmodelle. Durch die Schnittstellen-Abstraktion ist sichergestellt, dass sich jedes auf *core.product* (vgl. Abschnitt 6.3.8, S. 188ff.) basierende Produktmodell versionieren lässt. MOD2-SKM erlaubt das Generieren eines SKMS ohne Produktmodell, dem anschließend ein manuell implementiertes hinzugefügt wird. Dieser Sachverhalt lässt sich mit Hilfe des Merkmalsmodells und der Konfigurationen nicht ausdrücken, denn jedes Merkmal einer Konfiguration muss auch im Merkmalsmodell vorhanden sein. In diesem Fall wäre z.B. ein Merkmal „Eigenes Produktmodell“ im Merkmalsmodell nötig, das sich während der Konfiguration mit dem „Typ“ des manuell implementierten Produktmodells „instanzieren“ lässt. Im Rahmen dieser Arbeit ließ sich kein befriedigendes Ergebnis für dieses Problem finden.

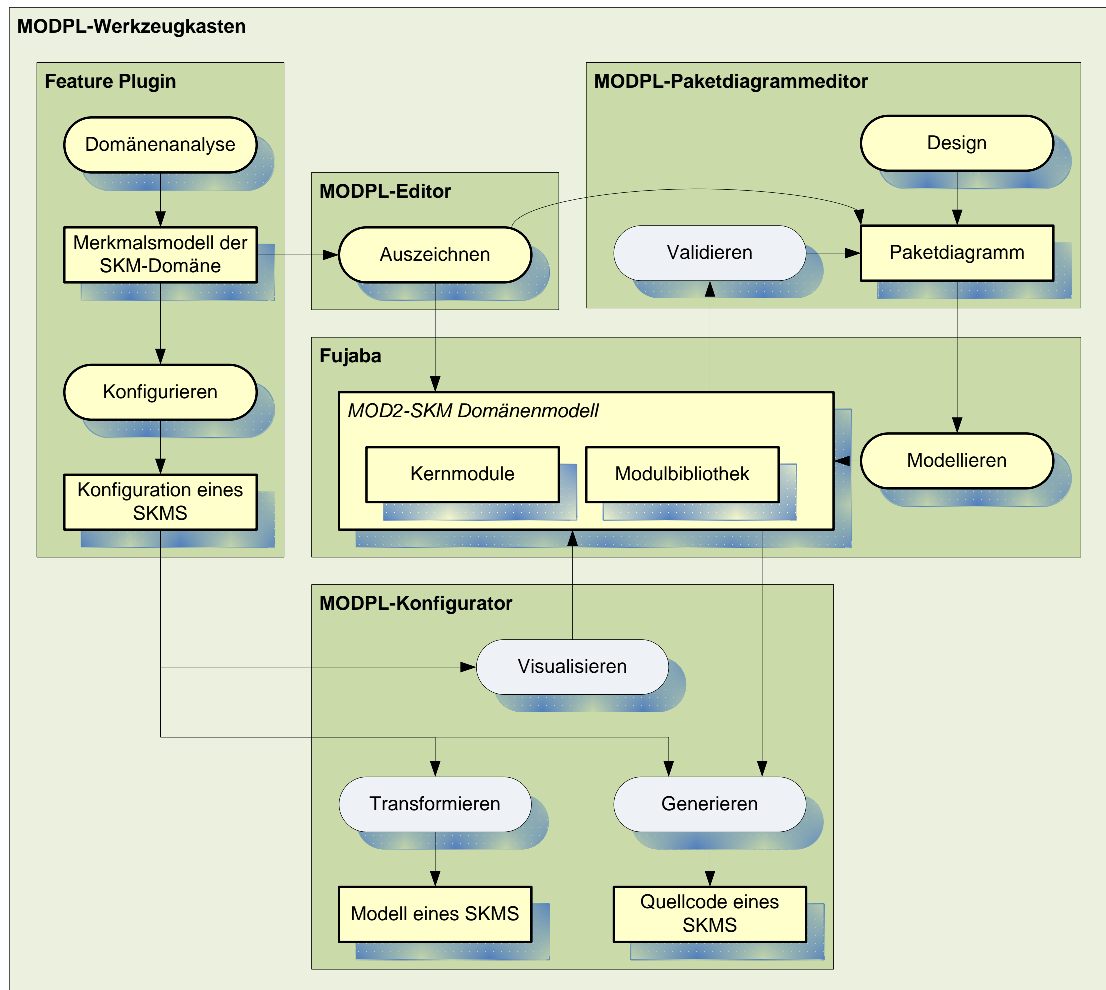


Abbildung 8.2.: Überblick über die MOD2-SKM Produktlinie

8.4.2. Feature Plugin

Das Merkmalsmodell und seine Konfigurationen werden mit Hilfe des Feature-Plugins in der Entwicklungsumgebung Eclipse erstellt und verwaltet [AC04]. Grundlage des Modells bildet die Merkmalsanalyse nach der FORM-Methode [KKL+98]. Obwohl das Feature-Plugin die FORM-Methode nicht unterstützt, ließen sich die vier FORM-Ebenen einfach als verpflichtende Merkmale integrieren. Die Strukturierung des Modells gemäß der SKM-Konzepte aus der Literatur (vgl. Abschnitt 5, S. 95ff.) verlief problemlos.

Leider besteht keine Möglichkeit, das Modell mit der textuellen Merkmalsanalyse zu verbinden, so dass die Analyse in Anhang B manuell mit dem Modell in Anhang C abgeglichen werden musste. Dieser Abgleich ist sehr fehleranfällig. Die Synchronisation des Merkmalsmodells mit den Konfigurationen erforderte oft manuelle Korrekturen. Zwar ließen sich neue Merkmale in bereits bestehende Konfiguration übernehmen, aber das Verschieben von Merkmalen ließ verwaiste Merkmale in den Konfigurationen zurück.

Alle Abhängigkeiten unter den Merkmalen ließen sich mit Hilfe der aussagenlogischen Einschränkungen beschreiben. Die Verwendung der internen Merkmalskennung macht die Ausdrücke gegenüber Umbenennungen unempfindlich, verlangt aber Kenntnis der Kennung. Kann ein Ausdruck nicht ausgewertet werden, da z.B. ein Merkmal gelöscht wurde, ist keine Konfiguration mehr möglich, und es wird eine generische Ausnahme geworfen, ohne die relevante Einschränkung zu nennen. Kann der Entwickler die Ursache nicht erahnen, bleibt nur die Möglichkeit, Haltepunkte im Quellcode des Feature-Plugins zu setzen.

8.4.3. Paketdiagramm

Das Paketdiagramm dient in MOD2-SKM zur Visualisierung der Architektur und Definition der Module. Ein MOD2-SKM Modul ist ein Paket einschließlich seiner Unterpakete und Klassen. Die Beziehungen zwischen den einzelnen Klassen können auf Paket-Importe im Klassendiagramm abgebildet werden, so dass die Abhängigkeiten zwischen Modulen grafisch darstellbar sind (vgl. Abschnitt 6.5.2, S. 297ff.). Auf diese Weise lassen sich schnell die Abhängigkeiten zwischen den Modulen erkennen – und auch minimieren. Dazu reicht es aus, fragwürdige bzw. ungewünschte Import-Beziehungen im Paketdiagramm zu löschen und anschließend das Domänenmodell gegen das Paketdiagramm zu validieren. Dadurch erhält der Entwickler Informationen, welche Elemente die Sichtbarkeiten des Paketdiagramms verletzen und kann versuchen, diese Abhängigkeiten aufzulösen. So lassen sich leicht Modellelemente identifizieren, die z.B. eine konkrete Subklasse statt einer Schnittstelle aus den Kernmodulen verwenden.

Das MOD2-SKM Domänenmodell besitzt 49 Module – zusammengesetzt aus 116 Paketen – zwischen denen 244 **private Paketimporte** bestehen. Nach Festlegung der Architekturstruktur (vgl. Abschnitt 6.2.3, S. 161ff.) lässt sich die Pakethierarchie problemlos entwickeln, und auch, beim Hinzufügen neuer Module, erweitern. Auf Grund ihrer Anzahl sind die Importbeziehung nicht mehr vollständig zu überblicken. Daher wurden einzelne Module gezielt untersucht. Hierbei ist die Hervorheben-Funktion des Editors [Buc10] eine große Hilfe, da sie importierte Pakete markiert. Die 244 Beziehungen lassen sich

wegen der vielen überlappenden Importpfeile nicht mehr erkennen. Aus diesem Grund ist eine grafische Darstellung der (noch zahlreicheren) Elementimporte nicht mehr verwendbar. Die Verwendung von Unterdiagrammen ist hier sinnvoll, wurde aber auf Grund des erneuten manuellen Layoutschritts nicht verwendet (vgl. Abschnitt 8.4.4, S. 342ff.).

Die automatische Generierung und Reduktion **öffentlicher Paketimporte** reduziert die Anzahl der Beziehungen erheblich. Ihre Anwendung erwies sich trotzdem nicht als praktikabel: Während die Ableitung eines privaten Imports unmittelbar auf eine Beziehung im Domänenmodell zurückzuführen ist, liefert die Reduktion der öffentlichen Beziehungen ein korrektes – aber nicht mehr für den Entwickler nachvollziehbares - Paketdiagramm. Der Algorithmus erzeugt lediglich ein Diagramm, das „irgendwie die Sichtbarkeitsregeln erfüllt." Im Vergleich zu den privaten Importen, müssen die öffentlichen Importe viel stärker an der Architektur-Struktur (vgl. Abschnitt 6.2.3, S. 161ff.) orientiert sein. Ein Algorithmus zur automatischen Ableitung öffentlicher Importe sollte daher z.B. mit zusätzlichen Einschränkungen versehen werden können, oder seine Anwendung auf Teildiagramme beschränken.

Ein konzeptionelles Problem des Paket-Metamodells zeigte sich beim Löschen von Paketen: Eine Import-Beziehung ist als Kindelement des Quellpakets modelliert. Wird das Zielpaket gelöscht, bleibt eine verwaiste Beziehung im Paketmodell zurück. Dies führt dazu, dass die grafische Darstellung im Paketdiagramm-Editor unbrauchbar wird und neu erstellt werden muss (vgl. Abschnitt 8.4.4, S. 342ff.). Eine Korrektur des Metamodells oder des Löschverhaltens ist unbedingt erforderlich.

Paketdiagramm und Domänenmodell sind voneinander getrennte Modelle, die – mittels des voll qualifizierten Namens der Elemente – miteinander synchronisiert werden [Buc10]. Der Abgleich von Änderungen erfolgt asynchron (auf Anfrage des Benutzers) und basiert auf dem Vergleich des Paketdiagramms mit der automatisch generierten Pakethierarchie des Domänenmodells. Die Synchronisation zwischen beiden Modellen erfolgt lediglich „additiv", d.h. es ist nur möglich, neu hinzugefügte Pakete und Importe aus dem Fujaba- in das Paketmodell zu übertragen. Umbenennen, Verschieben und Löschen führt zu verwaisten Paketen und Import-Beziehungen, die das Paketdiagramm unnötig vergrößern. Eine Analyse anhand der Import-Beziehungen ist in solch einem Paketmodell nicht mehr möglich. Die Paketmodelle von MOD2-SKM sind daher immer wieder neu erstellt worden. Ein synchroner Abgleich von Änderungen ist hier erforderlich.

8.4.4. Paketdiagramm-Editor

Der Paketdiagramm-Editor stellt das Paketdiagramm des Domänenmodells und die Abhängigkeiten zwischen den Modulen grafisch dar. So bietet er einen leicht verständlichen Zugang zur Architektur der MOD2-SKM Produktlinie. Der Editor erfüllt seine Aufgabe zur Anzeige und Bearbeitung von Paketdiagrammen gut, und er wurde sowohl zur Analyse der Abhängigkeiten als auch zur Dokumentation der Produktlinie genutzt (vgl. Abschnitt 8.4.3, S. 341ff.). Alle Paketdiagramme in dieser Arbeit wurden mit ihm erstellt.

Das Löschen von Paketen ist auf Grund der Konzeption des Paket-Metamodells sehr aufwändig (vgl. Abschnitt 8.4.3, S. 341ff.). Nach Löschen eines Pakets mit mindestens einem eingehenden Import, kann das Paketdiagramm nicht mehr gespeichert werden.

Beim erneuten Öffnen endet jede verwaiste Import-Beziehungen in der linken obere Ecke des Editors, und lässt sich nicht mehr entfernen. Vor dem Löschen muss der Entwickler daher alle eingehenden Kanten manuell löschen – oder den Löschvorgang in der Baumdarstellung des Paketmodells [Buc10] vornehmen (und anschließend das Diagramm neu erstellen).

Grafische Darstellung mittels GMF

Die Leistungsfähigkeit des Graphical Modelling Framework (GMF) [ecl10] sinkt auffällig mit steigender Anzahl an Import-Beziehungen. Bereits bei den 244 privaten Paketimporten dauert das Verschieben des Diagrammausschnitts mehrere Sekunden. Zur Analyse größerer Diagramme ist es daher nötig, die Ursache dieses Leistungsabfalls ausfindig zu machen. Ist das GMF nicht für Diagramme dieser Größe geeignet, muss auf seine Verwendung verzichtet werden. Evtl. ist jedoch lediglich die Laufzeit des Algorithmus für die Hervorhebungen ineffizient, so dass hier eine Optimierung vorgenommen werden kann.

Nach der Erzeugung eines Paketdiagramms für das Domänenmodell muss die Position der Pakete und der Verlauf der Import-Kanten noch automatisiert festgelegt werden. Diese Darstellung ist lediglich für Prüfungen der Abhängigkeiten über Hervorhebung nutzbar [Buc10] (vgl. Abschnitt 8.4.3, S. 341ff.). Die Elemente in den Diagrammen dieser Arbeit sind alle manuell positioniert. Jede Operation, die die grafische Darstellung des Paketdiagramms zurücksetzt bzw. sogar unbrauchbar macht, bedeutet einen hohen Zeitaufwand für den Entwickler. Dies betrifft folgende Operationen:

1. Löschen von Paketen mit eingehenden Kanten.
2. Umbenennen der Paketdiagramm-Datei.
3. Öffnen eines Teildiagramms.
4. Umbenennen und Verschieben eines Paketes.

(1) und (2) machen die Darstellung unbrauchbar, (3) und (4) setzen die grafische Darstellung zurück, d.h., es ist automatisierte Platzierung mit anschließender manueller Bearbeitung notwendig. Auf Grund des hohen Aufwandes der manuellen Positionierung wurden wegen (3) keine Teildiagramme verwendet. (4) zeigt die mangelnde Robustheit der grafischen Darstellung bei Veränderungen. Dies erschwert eine iterative Entwicklung von Paketdiagramm und Domänenmodell erheblich. Letztendlich existieren für den Benutzer nicht zwei, sondern drei Modelle – Domänenmodell, Paketdiagramm-Baumstruktur, Paketdiagramm-Graph – die er synchron halten muss.

Synchronisation Paketdiagramm/Domänenmodell

Der Datenabgleich findet lediglich zwischen den nicht-grafischen Komponenten von Domänenmodell und Paketdiagramm statt. Der Benutzer interagiert jedoch mit der grafischen Darstellung. Bei gelöschten Paketen im Domänenmodell sieht er nur eine Liste mit Namen, wenn er entscheiden soll, welche Pakete bei der Synchronisation entfernt werden

sollen. Und bei der Validierung werden zwar die nicht sichtbaren Elemente markiert, aber es besteht keine Möglichkeit zu den Elementen zu navigieren. Ein Modell in der Größe des MOD2-SKM Domänenmodells lässt sich ohne weitere Werkzeugunterstützung nicht mehr synchronisieren. Dies Betrifft u.a. auch die Eigenschaften der Import-Beziehung. Das Zielpaket wird nur mit dem einfachen Namen im Editor angezeigt, so dass hier über 10 Pakete mit dem Namen „server“ existieren – die Auswahl von „core.server“ ist nur durch ausprobieren möglich.

Alle Paketdiagramme dieser Arbeit sind aus dem aktuellen Stand des Domänenmodells generiert und ihre Diagrammelemente manuell positioniert. Um die Client-Server-Architektur hervorzuheben wäre es günstig, auch in den Modulen der Modulbibliothek ein Unterpaket mit dem Namen „server“ einzuführen. Außerdem ist der Name „core.lock“ für das Modul „core.synchonisation“ noch aussagekräftiger. Das korrigieren der Paketdiagramme erfordert jedoch ca. 30 Personenstunden. Die Bedeutung dieser Beobachtungen für modellgetriebene Entwicklung ist in Abschnitt 8.5 (s. S. 349) genauer erläutert.

8.4.5. Domänenmodell

Das Domänenmodell modelliert Struktur und Verhalten der MOD2-SKM Produktlinie mit Hilfe von Klassen- und Storydiagrammen in Fujaba. Die Paketstruktur, in der die Klassen enthalten sind, entspricht der des Paketdiagramms aus Abschnitt 8.4.3. Die variablen Elemente sind mit Merkmalen aus dem Merkmalsmodell in Abschnitt 8.4.1 markiert. Und der MODPL-Konfigurator (vgl. Abschnitt 8.4.9, S. 349ff.) erzeugt aus dem Domänenmodell – anhand einer Konfiguration – ein konfiguriertes SKMS. Damit ist es das zentrale Modell der MOD2-SKM Produktlinie.

Die Struktur des Domänenmodells basiert auf UML-Klassendiagrammen [OMG09a, Zün02], so dass reguläre objektorientierte Konzepte und Entwurfsmuster zur Modellierung eingesetzt werden. So basiert die Aufteilung der 14 Kernmodule u.a. auf dem Architekturprinzip „Separation of Concerns“ [VAC+09] und die Wiederverwendbarkeit der Module u.a. auf Schnittstellen-Abstraktion. Und um die Initialisierung konkreter Objekte zu kapseln, wird z.B. das Fabrik-Entwurfsmuster [FF04] eingesetzt (vgl. Abschnitt 6.2.1, S. 155ff.). Das es sich um ein reguläres Fujaba-Modell handelt, lässt sich die vollständige Modulbibliothek aus dem Domänenmodell generieren[1], und solange keine Einschränkungen des UML-Metamodells verletzt sind (s.u.) auch kompilieren. So können Produktlinien-Entwickler prüfen, ob die Module des Domänenmodells auf syntaktisch korrekten Quellcode abgebildet werden.

Beschränkung Kontrollflüsse

Es ist zwar möglich, Storyaktivitäten mit mehr als drei ausgehenden Transitionen zu modellieren, jedoch kann die Quellcode-Generierung nur mit maximal zwei ausgehenden Kanten umgehen. Daher ist u.a. die Modellierung von Ausnahmebehandlungen oder Fallunterscheidungen aufwändig und unübersichtlich. Im ersten Fall werden Anweisungsaktivitäten zur Quellcode-„Injektion“ benötigt (vgl. Abschnitt 6.1, S. 148ff.), im zweiten

[1] Ist keine Konfiguration geladen, generiert der MODPL-Konfigurator alle Module.

erschweren lange Sequenzen von Storyaktivitäten das Verständnis, da sich der Sachverhalt in einer einzigen Story-Aktivität mit vielen Transitionen ausdrücken ließe.

Wie z.B. bei der Ausnahmebehandlung, bietet Fujaba – als letzten Ausweg – die Möglichkeit, Quellcode mittels Anweisungsaktivitäten direkt in ein Modell zu „injizieren". Ein solches Modell ist damit auf diese Programmiersprache als Zielsprache für die Quellcode-Generierung festgelegt. Ihre Verwendung ist letztendlich immer als Notlösung zu betrachten und ein deutlicher Hinweis darauf, dass für diese Anweisungen noch keine Modellierungstechnik existiert.

Semantik Modellelemente

Fujaba bietet als Modellelemente sowohl negative Kanten als auch negative, optionale und Mengen-Knoten. Der Einsatz dieser Elemente erleichtert die Modellierung und führt zu kompakteren Storyaktivitäten. Leider ist ihre Semantik nicht dokumentiert und auch nicht immer eindeutig definiert, z.B. bei der Verwendung mehrerer Kanten mit einem Mengenknoten oder bei einer negativen Kante und einem optionalen Knoten. Die gezielte Verwendung dieser Elemente kann die Lesbarkeit und das Verständnis von Storyaktivitäten erhöhen, führt aber leider oft zu längeren „Modelliersitzungen", um die Semantik dieser Elemente zu „erforschen".

8.4.6. Fujaba

Fujaba wird verwendet, um ein ausführbares Domänenmodell zu modellieren. Für die Struktur kommen Klassendiagramme und für das Verhalten Storydiagramme zum Einsatz. Die Fähigkeit, Storydiagramme auf lauffähigen Quellcode abzubilden, ist eine Kernfunktionalität des MODPL-Werkzeugkastens. Fujaba ist grundsätzlich für die Entwicklung einzelner Software-Systeme vorgesehen, so dass im Rahmen der Dissertation von Thomas Buchmann der MODPL-Werkzeugkasten entwickelt wurde, um u.a. Fujaba für modellgetriebene Produktlinienentwicklung verwenden zu können [Buc10].

Kein Fujaba4Eclipse

Fujaba wurde ursprünglich als autonome modellbasierte Entwicklungsumgebung entwickelt [Zün02]. Mit Fujaba4Eclipse existiert seit einige Jahren eine in die Entwicklungsumgebung Eclipse integrierte Version. Letztere wurde zu Integration der MODPL-Werkzeuge genutzt, da sich diese als Eclipse-Plugins einbinden ließen [Buc10]. Obwohl beide Fujaba-Version die selben Kernkomponenten – und insbesondere die selbe Quellcode-Generierung – verwenden, ist Fujaba4Eclipse nicht in der Lage, Quellcode für die konfigurierten Modelle oder auch die gesamte Produktlinie zu erzeugen. Vielmehr wird sogar das Domänenmodell durch die Quellcode-Generierung verändert!

Es traten zwei Fehler während der Quellcode-Generierung auf: (1) Die Quellcode-Generierung brach ab und der Quellcode war nur teilweise erzeugt. Stattdessen war mindestens ein Modellelement aus dem Fujaba-Modell verändert oder gelöscht worden! Dieser Fehler ließ sich nicht systematisch reproduzieren, und macht letztendlich Fujaba4Eclipse für die Entwicklung von MOD2-SKM unbrauchbar. (2) Fujaba4Eclipse war

nicht in der Lage, korrekten Quellcode für Klassen zu generieren, wenn sie Abhängigkeiten zu Schnittstellen in einem anderen Fujaba-Modell besitzen. Dies betraf zwar nicht das MOD2-SKM Kernmodell, aber alle abhängigen Modelle, wie sie z.B. im Rahmen von Abschlussarbeiten verwendet wurden [Mat09, Öhm10, Bär10].

Der MODPL-Editor und der MODPL-Konfigurator können auch mit der autonomen Fujaba-Version verwendet werden, so dass für die Entwicklung der MOD2-SKM Produktlinie diese Version zum Einsatz kam. Fujaba4Eclipse wurde lediglich zur Benutzung des Paketdiagrammeditors benötigt.

Modellieren von Storydiagrammen

Das Modellieren von Objekten und ihren Verknüpfungen funktioniert reibungslos. Selbst komplexe Muster lassen sich durch die Beschränkung auf sichtbare Verknüpfungen bzw. benachbarte Objekte schnell eingeben. Auch bei der Auswahl eines Objekt-Typs werden die Entwickler durch Auto-Vervollständigung unterstützt. In einigen Teilen erfolgt die Eingabe textueller Referenzen ohne Vervollständigung, was deutlich aufwändiger – und insbesondere fehleranfälliger – ist. Dies betrifft vor allem die Eingabe von:

- Konstruktor-Aufrufen
- Methoden-Aufrufen
- Attribut-Referenzen
- Einschränkungen
- Pfadausdrücken

Gerade die Pfadausdrücke sind eine der mächtigsten Modelliertechniken in der Fujaba-Entwicklungsumgebung. Die Möglichkeit, einen transitiven Abschluss zu durchsuchen, erlaubt die Formulierung komplexer rekursiver Algorithmen in einem einzigen Muster, z.B. zum Lokalisieren von Blatt oder Wurzelelementen in Baumstrukturen. Eine grafisch unterstützte Eingabe des Pfadausdrucks dürfte den Zugang zu dieser Technik erleichtern und ihre Mächtigkeit weiter erhöhen.

Ein weiterer Nachteil der textuellen Referenzen ist auch ihre fehlende Abhängigkeit vom referenzierten Modellelement. Dessen Umbenennungen bedeutet somit eine manuelle Anpassung jeder textuellen Referenz. Im Fujaba-Metamodell existieren sogar bereits Ansätze, diese Abhängigkeiten explizit zu modellieren. Dies sollte auf jeden Fall umgesetzt werden, da eine explizite Abhängigkeit der Elemente einen großen Vorteil im Vergleich zu textuellen Programmiersprachen bietet.

Fujaba kapselt bei der Navigation von 1:n-Assoziationen effizient die notwendigen Listenoperationen. Sobald die Modellierer jedoch Listen als Methoden-Parameter oder Rückgabewerte einsetzen, muss der Zugriff auf ihre Elemente aufwändig – mit Hilfe von textuellen Methoden-Aufrufen – modelliert werden. Hier wäre eine Unterstützung für den Modellierer wünschenswert.

Multi-Projekt Unterstützung

Eine Aufteilung des Domänenmodells auf mehrere Fujaba-Modelle – die dann die entsprechenden Teile des Server- bzw. Client-Systems generieren – ist kaum möglich. Fujaba-Modelle können zwar von anderen Fujaba-Modellen abhängen, und es lässt sich auch konfigurierter Quellcode bzw. ein konfiguriertes Teilmodell generieren, doch eine solche Abhängigkeit ist weder entfernbar[2] noch lassen sich Elemente zwischen den Modellen verschieben oder kopieren. Eine einmal festgelegte Aufteilung ist damit unveränderlich. Eine Aufteilung der Produktlinie auf mehrere Modelle für Teilsysteme bzw. Kernmodule und Modulbibliothek würde eine Überarbeitung der Produktlinie erschweren, da sich z.B. keine Elemente aus der Modulbibliothek in die Kernmodule übernehmen lassen.

Auch der MODPL-Werkzeugkasten bietet nur begrenzte Multi-Projekt-Unterstützung. Zwar lassen sich mehrere Fujaba-Modelle gegen dasselbe Paketmodell validieren, und auch Elemente aus mehreren Modellen in ein Paketdiagramm übernehmen[3], doch funktioniert die Analyse der Import-Beziehungen und -Stärke nicht mehr, da der Werkzeugkasten Beziehungen zu Elementen in abhängigen Projekten mehrfach zählt. Hier sollte ein Konzept erarbeitet und die Werkzeuge angepasst werden, um die Verwendung mehrerer Projekte zu ermöglichen.

Persistenzmechanismus CoObRA2

CoObRA2 speichert ein Fujaba-Modell in Form eines Änderungsprotokolls im Dateisystem [Sch07]. Aufgrund der fehlenden Atomizität der CoObRA2-Transaktionen nahm das Protokoll mehrmals während der Entwicklung invalide Zustände an. Entweder ließen sich diese manuell reparieren, oder es mussten alle Edierschritte seit der letzten Datensicherung wiederholt werden.

Zu den protokollierten Änderungen gehört auch das Anlegen und Löschen der propagierten Merkmalsmarkierungen. Jede Validierung bzw. Generierung des Domänenmodells persistenziert das Anlegen von ca. 3.000 Merkmalsmarkierungen. Das rasche Wachstum des Änderungsprotokolls erhöht die Ladezeit und den Speicherplatzbedarf erheblich. Das Entfernen der propagierten Merkmale verschlimmerte die Situation noch, da nun auch das Entfernen der Merkmale persistenziert wird. Das „Compacting“ von CoObRA2 sollte solche invarianten Änderungen entfernen [Sch07], doch die Operation zeigt keinen Effekt.

Das Änderungsprotokoll ließ sich nur mit großem Aufwand frei von propagierten Merkmalsmarkierungen halten. Der übliche Arbeitsablauf beim Modellieren sieht wie folgt aus: Zunächst laden die Entwickler das Domänenmodell und edieren es. Anschließend validieren sie das Modell, oder erzeugen ein konfiguriertes SKMS, um es zu kompilieren. Sie müssen nun ihre Änderungen direkt nach dem Edieren speichern, da bisher noch keine Merkmale propagiert wurden. Dies geschieht beim Validieren oder Generieren, so dass sie das Modell anschließend – ohne zu speichern – schließen müssen. Validieren bzw.

[2] Der CoObRA-Persistenzmechanismus speichert die Modellabhängigkeit permanent. Wird sie entfernt, so muss dennoch während des Ladevorgangs das abhängige Modell vorhanden sein, um das Anlegen und Löschen der Abhängigkeit „abzuspielen“.

[3] Durch Laden und Übertragen der Pakete bzw. Elemente in ein einziges Paketmodell.

generieren ist vor dem Speichern ebenso wenig möglich, wie ein iterativer Arbeitsablauf. Eine Anpassung der beiden Mechanismen ist für eine iterative Modellierung eines Domänenmodells unbedingt erforderlich.

Auch bei der Verwendung mehrerer Projekte kam es zu Problemen mit der Persistenzierung: Selbst nach Entfernen einer Projektabhängigkeit wird beim Laden das referenzierte Projekt benötigt, da das Änderungsprotokoll das Anlegen und Entfernen der Beziehung „abspielt“. Zusätzlich führt die Umbenennung von Projekten und ctr-Dateien zu Fehlern beim Laden abhängiger Projekte, obwohl CoObRA2 jedem Projekt eine eindeutige interne Kennung zuordnet [Sch07]. Zum Modellieren im Großen sollte daher die Multi-Projekt-Unterstützung des Persistenzmechanismus verbessert werden.

8.4.7. MODPL-Editor

Der MODPL-Editor liest Merkmalsmodelle ein, und bietet den Domänenmodell-Entwicklern Funktionen, um mit den Merkmalen Modellelemente zu markieren. Außerdem propagiert der Editor die Markierungen entsprechend der Propagationsregeln [BD09a, Buc10]. Es ließen sich fast alle gewünschten Markierungen anbringen, und der Umgang mit dem Editor war leicht erlernbar [Bär10]. In einigen Fällen fehlte jedoch die Möglichkeit, eine Markierung mit einer Negativbedingung zu versehen. Dies ist nötig, wenn z.B. ein Variationspunkt durch ein einzelnes Merkmal beschrieben wird, und für beide Varianten (Merkmal vorhanden/nicht vorhanden) ein Modellelement existieren soll. Diese Einschränkung ließ sich nur durch Einführung von Hilfsmerkmalen (vgl. Abschnitt 5.6.1, S. 134ff.) umgehen. Um das Merkmalsmodell nicht unnötig zu vergrößern, sollte der MODPL-Editor die Angabe einer Negativ-Bedingung anbieten.

Für die Entwickler ist eine eindeutigere und explizitere Darstellung der Abhängigkeiten von Merkmalsmodell und Domänenmodell notwendig. Folgende Punkte erschweren das Verstehen der Beziehungen zwischen beiden Modellen:

1. Die Merkmale werden im Merkmalsmodell mit ihrem Namen bezeichnet, aber im Domänenmodell mit ihrer internen Merkmalskennung. Dies bedeutet eine zusätzliche Ebene der Indirektion, die Aufwand und Fehlerhäufigkeit sowohl beim Markieren als auch beim Lesen erhöht (vgl. Abschnitt 6.1.2, S. 154ff.).

2. Eine Merkmalsmarkierung lässt sich nicht visuell einer Methode zuordnen. Dies ist nur über Umwege, z.B. die Validierung, möglich (vgl. Abschnitt 6.1.1, S. 149ff.).

3. Die Trennung von Fujaba, Paketdiagramm-Editor und Feature-Plugin erschwert die parallele Entwicklung von Domänen- und Merkmals-Modell. Deren Entwicklung erfolgte iterativ und nicht einmalig im Top-Down-Verfahren (vgl. Abschnitt 8.3.1, S. 330ff.). Insbesondere der Export als XML-Datei und das anschließende Einlesen erschweren die iterative Entwicklung. Der MODPL-Werkzeugkasten bietet zwar Werkzeuge zur Analyse von Merkmalsmarkierungen an [Buc10], doch auf Grund der unterschiedlichen Identifikation (Merkmalsname vs. -kennung) lassen sich diese nur schwer in das Feature-Plugin zurückverfolgen.

Die Propagation der Merkmale steht leider in Konflikt mit CoObRA2, dem Persistenzmechanismus von Fujaba. Das Hinzufügen und Entfernen von Merkmalsmarkierungen wird durch CoObRA2 protokolliert und vergrößert das persistente Domänenmodell unnötig (vgl. Abschnitt 8.4.6, S. 347ff.). Das von CoObRA2 unterstützte „Compacting“ (das Entfernen invarianter Änderungen [Sch07]) entfernt die Markierungen nicht. Eine Anpassung der beiden Mechanismen ist für eine reibungslose Modellierung unbedingt erforderlich.

8.4.8. Konfigurierte SKMS

Aus dem Domänenmodell und einer Konfiguration lässt sich sofort der Quellcode des entsprechend konfigurierten SKMS generieren. Außerdem besteht die Möglichkeit, ein Fujaba-Modell des konfigurierten SKMS zu erstellen. So lässt sich das Modell noch nachträglich erweitern oder verändern, bzw. als abhängiges Modell für Erweiterung verwenden. Drittens besteht auch die Möglichkeit, die Konfiguration im MOD2-SKM Domänenmodell zu visualisieren. Konfigurierter Quellcode und konfiguriertes Fujaba-Modell sind dabei nicht äquivalent, da z.B. das Entfernen markierter Aktivitäten zu validem Quellcode und zu invaliden Modellen führt (vgl. Abschnitt 8.4.5, S. 344ff.).

Die Auslieferung (engl. deployment) ist nicht mehr Teil des MODPL-Werkzeugkastens. Die Aufteilung der Module auf ein Software-System für den Repositoriums-Server und für den Arbeitsbereich-Client erfolgt mit Hilfe von Stapelverarbeitungs-Skripten (vgl. Abschnitt 7, S. 315ff.). Dieser Aspekt ist jedoch Teil der Entwicklung eines Software-Systems. D.h. für einen vollständig modellgetriebenen Ansatz ist eine Modellierung und Abbildung auf die System-Architektur notwendig.

8.4.9. MODPL-Konfigurator

Der MODPL-Konfigurator erzeugt aus dem Domänenmodell und einer Konfiguration ein konfiguriertes Modell, Quellcode für ein konfiguriertes System oder visualisiert die Konfiguration im Domänenmodell [Buc10]. Die Konfiguration erfolgte fehlerlos und führte auch zu keiner messbaren Verzögerung bei der Generierung. Während der Entwicklung der MOD2-SKM Produktlinie kam aber nur der Quellcode-Konfigurator zum Einsatz, da kein konfiguriertes Modell benötigt wurde. Für die in Abschnitt 8.4.8 erwähnte Auslieferungs-Unterstützung ist evtl. eine Erweiterung des MODPL-Konfigurators bzw. die Anbindung eines Bau-Werkzeugs nötig.

8.5. Modellgetrieben Entwickeln

Die MOD2-SKM Produktlinie ist ein Fallbeispiel eines komplexen modellgetriebenen Software-Entwicklungsprojekts. Sie leistet einen Beitrag zur Analyse und Weiterentwicklung modellgetriebener Software-Entwicklung, indem sie insbesondere Anforderungen für Modellieren im Großen stellt. Anhand der Erfahrungen während der Modellierung lassen sich daher effektive Konzepte und Techniken identifizieren – aber auch Fragestellungen und ungelöste Probleme.

8.5.1. Verwendung eines Architekturmodells

Die feingranulare Struktur von MOD2-SKM ist über Klassendiagramme modelliert, doch die modulare Architektur und die Abhängigkeiten zwischen den Modulen ließen sich nicht mehr anhand von Klassen und Assoziationen verwalten (vgl. Abschnitt 6.2.3, S. 157ff.). Es ist daher eine weitere Modellart für die Architektur komplexer Software-Systeme notwendig, in die sich die feingranulare Struktur und die Verhaltensmodelle integrieren lassen (vgl. Abschnitt 6.3, S. 171ff. und vgl. Abschnitt 6.4, S. 211ff.). Diese Aufgabe übernimmt im MODPL-Werkzeugkasten das Paketdiagramm. Ihre Verwendung zur Präsentation der Modul-Abhängigkeiten in dieser Arbeit, sowie die Metrik auf Basis der Paket-Importe, zeigen (vgl. Abschnitt 6.5.2, S. 297ff.), dass (a) ein grobgranulares Modell elementar für das Verständnis einer modellgetrieben entwickelten Produktlinie ist und (b) Paketdiagramme für diese Aufgabe geeignet sind.

Anstatt der Paketdiagramme existieren in der UML noch Komponentendiagramme [OMG09a]. Da diese noch deutlicher Module – und insbesondere ihre Schnittstellen – erfassen, ist ihre Verwendung zu untersuchen. Die Spezifikation definiert eine Komponente jedoch nur sehr vage, so dass hier noch die Integration mit Klassen- und Storydiagrammen erfolgen müsste. Insbesondere das Anbieten und Verwenden von Schnittstellen für Komponenten sollte mit einer Darstellung im Klassendiagramm verbunden werden.

8.5.2. Große Modelle

Die Arbeit an einem Modell der Größe der MOD2-SKM Produktlinie zeigt, dass eine Aufteilung auf mehrere Modelldateien praktikabel ist [Bär10]. Analog zur Aufteilung eines Software-Systems auf Teilprojekte, bieten Teilmodelle den Vorteil, dass Produktlinien-Entwickler parallel arbeiten können. Die Entkopplung der Modulbibliothek träte in diesem Fall noch deutlicher zu Tage, wenn für jedes Modul der Modulbibliothek ein eigenes Modell angelegt worden wäre. Dazu sind jedoch Mechanismen zur Referenzierung von Modellelementen aus anderen Modellen notwendig.

Am MOD2-SKM Domänenmodell konnte immer nur ein Entwickler zur gleichen Zeit arbeiten da kein SKMS für Fujaba-Modelle existiert, das konkurrierende Änderungen anzeigen und mittels Benutzerinteraktion verschmelzen kann. Für den produktiven Einsatz modellgetriebener Software-Entwicklungswerkzeuge ist ein Modell-SKMS nötig. Die MOD2-SKM Produktlinie besitzt mit *product.ecore* (vgl. Abschnitt 6.4.13, S. 239ff.) und *product.usecase* (vgl. Abschnitt 6.4.15, S. 247ff.) zwei Prototypen eines Produktmodells für Modelle. Dieser Ansatz sollte in zukünftigen Arbeiten weiter verfolgt werden.

8.5.3. Mehrere Metamodelle

Im MODPL-Werkzeugkasten bedeutet der Einsatz mehrerer Metamodelle einen aufwändigen Synchronisationsprozess für den Entwickler (vgl. Abschnitt 8.4.4, S. 342ff.). Es existieren mehrere Alternativen, um diesen Aufwand zu reduzieren: (1) Die Verwendung eines einheitlichen Metamodells. Die Editoren präsentieren dann lediglich unterschiedliche Sichten auf eine Modellinstanz bzw. Teile von ihr. (2) Der Synchronisationsmechanismus wird verbessert, so dass er (a) synchron arbeitet, und Änderungen direkt in

alle verknüpften Modellinstanzen überträgt, oder (b) ein zusätzliches Abbildungsmodell einführt, z.B. durch Einsatz von Tripel-Graph-Grammatiken [Sch95, BDW08b].

8.5.4. Modellierungstechniken

Für die Modellierung kurzer, effizienter Algorithmen erweisen sich die Story-Aktivitäten als ungeeignet. Dies betrifft sowohl die Performanz als auch ihre Modellierung: Für sie wird eine hohe Anzahl an Anweisungsaktivitäten benötigt [Mat09]. Dafür bilden die Story-Aktivitäten ein mächtiges Werkzeug, um in komplexen Datenstrukturen zu navigieren. Insbesondere die Pfadausdrücke ermöglichen die Spezifikation von leicht verständlichen Mustern, die im Quellcode nur als mehrfach verschachtelte, rekursive Algorithmen implementiert werden können. Diese bieten einem Entwickler einen unmittelbaren Vorteil bei der Entwicklung, und ihre Verwendung sollte auch durch die Werkzeuge unterstützt werden. Eine weitere mächtige Technik kam in Fujaba nicht zum Einsatz: Abgeleitete Attribute. Sie erlaubt dem Entwickler die Angabe von Berechnungsvorschriften für Attributwerte [SWZ99].

Die Überarbeitung und Korrektur von Modellen gestaltet sich aufwändig, so lange textuelle Referenzen verwendet werden. Ihre Existenz führt i.d.R. zu aufwändigen manuellen Anpassungen sobald die Modellierer Elemente umbenennen. Fehlt (zusätzlich) die Auto-Vervollständigung in den Modell-Editoren, ist auch alleine schon die Eingabe der Referenz fehleranfällig. Entwicklungsumgebungen für textuelle Programmiersprachen bieten in beiden Fällen ausgefeilte Werkzeuge und Verfahren an, die den Entwickler unterstützen. Diese Arbeiten jedoch auf Basis von Syntaxanalyse und Vergleich von Zeichenketten. Modellgetriebene Werkzeuge könnten wahrscheinlich – auf Basis von expliziten Beziehungen im Metamodell – eine viel höhere Genauigkeit bei Überarbeitungsoperationen erreichen.

Idealerweise sollten die Modellierer keinen Bedarf an Quellcode-„Injektionen“ haben, denn dadurch geht die Unabhängigkeit von der Zielsprache verloren. Außerdem sind sie ebenfalls textuelle Referenzen und benötigen die o.g. manuelle Aktualisierung. Sie bieten aber gleichzeitig eine Notlösung für fehlende Modellierungstechniken und sollten daher zunächst noch Teil eines Modellierwerkzeugs bleiben.

Das Verständnis eines grafischen Modells kann – wie Quellcode – durch eine geeignete Formatierung gesteigert werden. Dazu sind jedoch auch Werkzeuge nötigt, die dem Entwickler erlauben, das Diagramm entsprechend zu gestalten. Auf Grund des hohen Entwicklungsaufwands besitzen die wenigsten modellgetriebenen Entwicklungswerkzeuge geeignete Benutzeroberflächen und Editoren. Doch ohne effiziente und arbeitserleichternde Eingabemöglichkeiten ist ein Vergleich zwischen herkömmlicher und modellgetriebener Software-Entwicklung kaum möglich. Letztendlich werden zwei Maße dieses Vergleichs die Produktivität und Fehlerrate der Entwickler sein. Bei einem Vergleich zwischen jahrzehntelang optimierten Editoren für textuelle Programmiersprachen und einer prototypischen grafischen Benutzerschnittstelle für ein Modellierungswerkzeug, lassen sich Unterschiede in der Produktivität und Fehleranfälligkeit kaum auf die Verfahren selbst zurückführen.

8.6. Zukünftige Entwicklungen

Der aktuelle Stand der MOD2-SKM Produktlinie erlaubt die Konfiguration von insgesamt 290.304.000 unterschiedlichen SKMS. Dennoch sind noch längst nicht alle Merkmale der Merkmalsanalyse umgesetzt (vgl. Abschnitt 5.5, S. 129ff.). Folgende Module können noch modelliert werden:

- Modul für gerichtete azyklische Graphen.
- Produktmodell für Merkmalsmodell.
- Synchronisation-Modul für Repositorien.

Gerade mit Hilfe des letzten Moduls ließe sich bereits eine Peer-to-Peer-Architektur realisieren, indem jedem Arbeitsbereich ein eigener (lokaler) MOD2-SKM Server zugeordnet wird. Außerdem sind noch Erweiterungen bereits existierender Module denkbar:

- Unterscheidung von Dateitypen in Dateisystem-Modul.
- Überarbeitung Ecore-Produktmodell.
- Verschmelzungsmodul für Zweige.
- Stärke Integration des Hibernate-Moduls.

Insbesondere die schnelle Entwicklung eines SKMS für neue Produktmodelle sollte genutzt werden, um eine Vielzahl von Produktmodellen zu erstellen. Kein monolithisches SKMS lässt sich (wahrscheinlich) bzgl. des Produktmodells so leicht erweitern wie die MOD2-SKM Produktlinie. Da MOD2-SKM die Integration manuell implementierter Produktmodells vorsieht, ist z.B. die Erstellung eines laufzeitkonfigurierbaren SKMS ohne Produktmodell denkbar, der als „Showcase“ für MOD2-SKM dient. Da SKMS-Anwender direkt mit dem Produktmodell interagieren, ließen sich so die Vorteile von MOD2-SKM überzeugend präsentieren.

Schnittstellen überarbeiten

In den Schnittstellen des Repositoriums (*core.repository*, vgl. Abschnitt 6.3.9, S. 191ff.), Speichers (*core.history*, vgl. Abschnitt 6.3.4, S. 178ff.) und der Kennungserzeugung (*core.id*, vgl. Abschnitt 6.3.5, S. 182ff.) bestehen bei der Kennung noch Abhängigkeiten zu Implementierungsdetails der Historie. Hier kann die Schnittstelle durch Einführung einer Schnittstelle für Kennungen weiter entkoppelt werden. In diesem Zusammenhang ist auch noch weiter zu untersuchen, welche SKM-Konzepte welche Kennungen benötigen.

Die Produktlinien-Architektur ließ sich gut mit dem Client-Server-Modell kombinieren (vgl. Abschnitt 6.2.3, S. 161ff.). Allerdings können die Server-Komponenten in den konkreten Modulen durch eigene Pakete besser hervorgehoben werden. D.h., analog zu den bereits existierenden *workspace*- und *communication*-Paketen sollte noch ein *server*-Paket angelegt werden (vgl. Abb. 6.14, S. 158).

Deutlichere Merkmalsmarkierungen

Die Merkmalsmarkierungen an Paketen und Klassen bieten Vorteile in der Handhabung und beim Testen eines konfigurierten SKMS. Daher sollten die Markierungen an Methoden und Aktivitäten entfernt und in entsprechenden Subklassen modelliert werden. Existierende Klassenmarkierungen sollten im Paketdiagramm durch evtl. vorhandene verpflichtende Elternmerkmale sichtbar gemacht werden (vgl. Abschnitt 8.3.3, S. 336ff.).

Testen und Performanz

Die vollständige MOD2-SKM Produktlinie (Modelle und Quellcode aller Module) wird nur zu Testzwecken in ein MOD2-SKM Repositorium eingespielt, denn das Testen der gesamten Produktlinie stellt Produktlinien-Entwickler vor ungelöste Probleme (vgl. Abschnitt 8.3.4, S. 338ff.). Eine Verwendung durch Endbenutzer setzt die Entwicklung geeigneter Testverfahren und -werkzeuge voraus. Bis zu deren Erforschung ist die Generierung ein oder mehrerer konfigurierter SKMS denkbar, die ohne Produktmodell generiert werden. Diese lassen sich herkömmlich testen und später durch selbst implementierte Produktmodelle erweitern (s.o.).

Die MOD2-SKM-Server-Tests mit dem MOD2-SKM Projekt zeigen, dass eine Optimierung der Performanz notwendig ist. Dies bedeutet insbesondere die Einführung von mehr qualifizierten Assoziationen, die eine Laufzeit von $O(1)$ besitzen (vgl. Abschnitt 6.1, S. 144ff.). Weiterhin ist hier auch der Vergleich mit monolithischen SKMS notwendig. Die enge Kopplung bietet u.U. einen Performanzvorteil gegenüber modularen SKMS. Dabei müssen sowohl Produktmodell-Instanzen mit besonders großen als auch besonders vielen Elementen untersucht werden.

8.7. Zusammenfassung

Drei der vier Hypothesen aus Abschnitt 1.7 ließen sich belegen: (1) Es ist möglich eine MODPL für SKMS zu entwickeln. (2) Mit Merkmalsmodellen lassen sich SKMS klassifizieren. (3) Eine MODPL beschränkt die Entwicklung eines SKMS auf Konfigurieren. Neue Funktionalität beschränkt sich auf die Entwicklung eines neuen Moduls. Die Hypothese, dass sich SKMS-Verfahren lose koppeln lassen, wurde weder belegt noch widerlegt, da keine Skala für lose Kopplung in Modellen existiert. Zwei einschränkende Abhängigkeiten zwischen den 29 Modulen der Modulbibliothek deuten jedoch auf eine lose Kopplung hin.

Der MODPL-Werkzeugkasten erlaubt die Entwicklung einer modellgetrieben entwickelten, modularen Produktlinie für SKMS. Er reduziert das Entwickeln eines Produktes auf das Ableiten einer Konfiguration aus dem Merkmalsmodell. Anschließend kann das konfigurierte SKMS aus dem Domänenmodell generiert werden. Die MOD2-SKM Produktlinie (29 austauschbare Module, 290.304.000 unterschiedliche SKMS) zeigt, dass der Werkzeugkasten auch für große Projekte geeignet ist. Auf Grund der Integration von fünf Werkzeugen (Feature Plugin, MODPL-Editor, MODPL-Paketdiagrammeditor, Fujaba, MODPL-Konfigurator) und fünf Modellen (Merkmalsmodell, Konfigurationen,

Paketdiagramm, Domänenmodell, konfiguriertes SKMS) sind insbesondere noch Anpassungen der Modell-Synchronisation nötig.

Ergebnis dieser Arbeit sind:

1. Eine Merkmalsanalyse der SKMS-Domäne (vgl. Abschnitt 5, S. 95ff.), einschl. Beispiel-Klassifikation von vier SKMS (CVS, GIT, Mercurial, Subversion, vgl. Abschnitt 5.4, S. 125ff.).
2. Ein Domänenmodell einer SKMS-Produktlinie (vgl. Abschnitt 6, S. 141ff.) mit 29 austauschbaren Modulen und einem in die Eclipse-IDE integriertem Client.
3. Ein Nachweis der Machbarkeit des MODPL-Ansatzes [Buc10] sowie der Verwendbarkeit des MODPL-Werkzeugkastens.
4. Eine Untersuchung der MODPL-Konzepte, sowie Formulierung von Verfahren zur Domänenmodell-Markierung und von Verbesserungsvorschlägen für den MODPL-Werkzeugkasten.
5. Eine Demonstration der Anwendungs- und Ausdrucksmöglichkeiten der modellgetriebene Entwicklung und insbesondere der Storydiagramme der Entwicklungsumgebung Fujaba [Zün02].

MOD2-SKM ist eine komplexe modellgetrieben entwickelte Produktlinie. Diese Arbeit leistet damit einen Beitrag zum Verständnis von Produktlinienentwicklung und auch modellgetriebener Entwicklung generell. Die Modulbibliothek bietet eine einfach zu erweiternde Sammlung von SKM-Verfahren. Diese sind explizit, objektorientierte und auch grafisch modelliert. Die Darstellungsform und die modulare Architektur ermöglichen die systematische Analyse von Abhängigkeiten zwischen einzelnen SKM-Verfahren. Diese Arbeit präsentiert damit ein Konzept zur wissenschaftlichen Untersuchung von SKMS. Und die MOD2-SKM Produktlinie bietet – auf Grund ihrer Erweiterbarkeit und Modularität – ein Rahmenwerk für die systematische Untersuchung von SKM-Verfahren. Sie leistet somit einen Beitrag zur Untersuchung und Verbesserung von SKMS. Außerdem unterstützt sie als komplexe Fallstudie die Erforschung modellgetriebener Entwicklung, und liefert so die Möglichkeit zur Verbesserung existierender Entwicklungsmethoden und -werkzeuge.

A. Fachbegriffe: Deutsch – Englisch

Deutsch	**Englisch**
Änderungsanfragen-Verwaltung	change request management
Änderungskennung	change identifier
Änderungsmenge	change set
aktualisieren	update
Aktionsmodul	action module
Aktivität	activity
Anforderung	requirement
Anfrageverfolgungssystem	issue tracking system
Arbeitsbereich	workspace
auschecken	check out
Auslieferung	deployment
Auslöser	trigger
Ausnahme	exception
Ausnahmebehandlung	exception handling
Bau-Werkzeug	build tool
Befähigungs-Ebene	capability layer
Beschränkungsmodul	constraint module
Betriebsumgebungs-Ebene	operating environment layer
Beziehung	relation, link
„C“-Knoten	composite node
Domäne	domain
Domänentechnologie-Ebene	domain technology layer
Doppel-Wartungs-Problem	double maintenance problem
Eigenschaft	property
einchecken	check in
einspielen	commit
Enthaltenseins-Modell	containment model
Entität	entity
Entitäts-/Beziehungsdiagramm	Entity/Relationship-Diagram
Entwicklungsprozess	development process
Gemeinsame-Daten-Problem	shared data problem
Gemeinsamkeiten-Unterschiede-Analyse	Commonality-Variability-Analysis
Gerichteter azyklischer Graph	directed acyclic graph
Gleichzeitige-Aktualisierungs-Problem	simultaneous update problem

Deutsch	**Englisch**
Grundfassung	baseline
Hauptzweig	main branch, trunk
Implementierungstechniken-Ebene	implementation technique layer
Inhaltsverwaltungs-System	content management system
Instrumentierbare Versionierungsmaschine	instrumentable version engine
Kandidatenversionen	candidate version collection
Klasse	class
Konfiguration	configuration
Konfigurationsmanagement	configuration management
Kopf	head
„L“-Knoten	link node
Manifest	manifest
Markierung	tag
Merkmal	feature
Merkmalslogik	feature logic
Merkmalsmarkierung	feature tag
Methode	method
„N“-Knoten	atomic node
Objektkennung	object identifier
Optionales Merkmal	optional feature
privat	private
Produkt	product
produktzentriert	product first
öffentlich	public
quelloffen	open source
Repositorium	repository
„Schiebe/Ziehe“-Verfahren	push/pull
Schlüssel	key
Schnittstelle	interface
Sichtbarkeit	visibility
Software-Anwendungssystem	software application system
Software-Bibliothek	software library
Software-Element	configuration item
Software-Komponente	software component
Software-Lebenszyklus	software life cycle
Speicher-Beschränkungsmodul	storage constraint module
Speicherkennung	storage id
Sperre	lock
Spitzenmerkmal	top-level feature
Stereotyp	stereotype
Teilnehmer	peer
Variante	variant, alternative

Deutsch	**Englisch**
Verbindliches Merkmal	mandatory feature
Verfahren	procedure
verschränkt, verwoben	intertwined
Version	revision, version
Versionierte Objektbasis	versioned object base
Versionskennung	version identifier
versionszentriert	version first
Vorlage	template
Vorgehensmodell	process model
Technik	technique
Werkzeug	tool
Werkzeugkette	tool chain
Zurücksetzen-Operation	revert operation
Zweig	branch

B. MOD2-SKM Merkmals-Analyse der SKM-Domäne

Merkmale der Befähigungs-Ebene

Merkmale auf der Befähigungs-Ebene beschreiben die Produktlinie mit Hilfe von Diensten, Operationen und Qualitäten, welche die Produktlinie anbieten soll. Sie ist von allen vier Ebenen am weitesten von den technischen Umsetzung entfernt und beschreibt die Produktlinie aus der Perspektiv eines Anwenders. Als Quelle dienen:

- Das initiale Übersichtspapier [CW98].
- Zwei Vergleiche zwischen diversen VKS, die Anwender bei der Wahl eines geeigneten VKS unterstützen sollten ([Ehr06], [Fis09]).
- Die Handbücher der VKS *CVS* [Ced05], *Subversion* [CSFP08], *GIT* [Ca09] und *Mercurial* [O'S09].
- Der IEEE Standard *828* [IEE05].
- Die wissenschaftlichen Arbeiten über *RCS* [Tic82] und *Adele* [EC94].

Abbildung B.1 zeigt eine Übersicht der Merkmale auf der Befähigungs-Ebene.

Überblick Spitzenmerkmale

B.1 Versionskontrolle *engl.* Version control

Ebene : Komponenten (Befähigung)

Beschreibung: Mit Hilfe der Versionskontrolle wird die Entwicklung eines Software-Produkts verwaltet. [CW98]. Dazu gehört zum einen die Möglichkeit frühere Versionen eines Software-Produkts wiederherstellen zu können, aber auch die Verwaltung von Varianten. Welche Artefakte und Beziehungen verwaltet werden und welche Operationen auf diesen Daten möglich sind, legt das Versionsmodell fest. Abbildung B.2 (siehe S. 364) zeigt die detaillierten Merkmale der Versionskontrolle.

Umsetzung : Teilweise implementiert

Kinder : „Commit/Update“ (B.1.1), „Versionskontroll-Operationen“ (B.1.2)

Setzt voraus : –

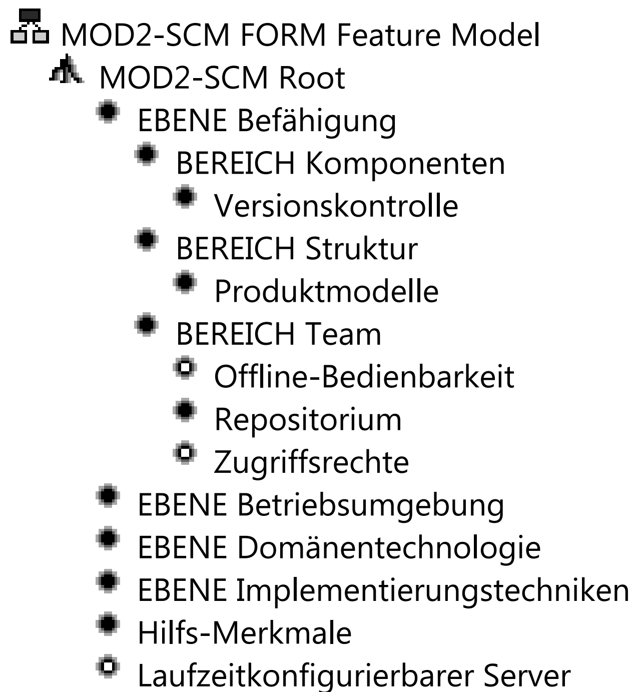


Abbildung B.1.: Merkmale auf der Befähigungs-Ebene

Benötigt von : –
Quellen : [CW98]

B.2 Produktmodelle *engl.* Product models

Ebene : Struktur (Befähigung)

Beschreibung: Das Produktmodell beschreibt die Struktur eine Software-Produkts [CW98]. Unterschiedliche Versionen eines Software-Produkts lassen sich als unterschiedliche Instanzen eines Produktmodells auffassen. Die Granularität und Semantik der Unterschiede steht im direkten Zusammenhang zu den Elementen des Produktmodells: Das gebräuchlichste Produktmodell basiert auf Dateien und Verzeichnissen, d.h. Unterschiede zwischen zwei Versionen eines Software-Produkts werden mit Hilfe von Änderungen am Dateisystem beschrieben. MOD2-SCM unterstützt jedoch noch weitere Produktmodelle (siehe Abbildung B.3, S.374).

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Anwendungsfall-Modell“ (B.2.1), „Dateisystem“ (B.2.2), „EMF-Modell“ (B.2.3), „Merkmals-Modell“ (B.2.4)}

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

B.3 Offline-Bedienbarkeit *engl.* Offline operability

Ebene : Team (Befähigung)

Beschreibung: Sobald das Repositorium und die Arbeitsumgebung auf unterschiedlichen Systemen laufen, muss eine Verbindung zwischen den beiden Systemen bestehen, um Daten auszutauschen. Eine solche Verbindung ist jedoch nicht jederzeit vorhanden. Durch das Zwischenspeichern von Repositoriums-Daten in der Arbeitsumgebung ist es möglich, einige Operationen auch ohne eine Verbindung zum Repositorium auszuführen. Abbildung B.4 (siehe S. 376) gibt einen Überblick über die detaillierten Merkmale der Offline-Bedienbarkeit.

Umsetzung : Erweiterungspunkt

Kinder : „Vollständiges Repositorium" (B.3.1), „Zurücksetzen-Operation" (B.3.2)

Setzt voraus : –

Benötigt von : –

Quellen : [Ehr06], [Ca09]

B.4 Repositorium *engl.* Repository

Ebene : Komponenten (Befähigung)

Beschreibung: Das Repositorium dient als Speicher für versionierte Objekte (engl. versioned object base) [CW98]. In ihm werden die Versionen und Varianten verwaltet, so dass jeder Entwickler Zugriff auf das Repositorium benötigt. Für eine Zusammenarbeit mehrerer Entwickler muss daher das Repositorium entweder zentral verfügbar sein, oder die Synchronisation mehrerer Repositorien untereinander erlauben. Bei sehr großen Projekten sind auch Hierarchien von Repositorien möglich, wenn jeder Standort sein eigenes Repositorium besitzt. Abbildung B.5 (siehe S. 377) gibt einen Überblick über die detaillierten Merkmale für Repositorien.

Umsetzung : Teilweise implementiert

Kinder : „Hierarchie-Unterstützung" (B.4.1), „Peer-to-Peer-Unterstützung" (B.4.2), „Replikations-Unterstützung" (B.4.3)

Setzt voraus : –

Benötigt von : –

Quellen : [Fis09], [Ca09]

B.5 Zugriffsrechte *engl.* Permissions

Ebene : Team (Befähigung)

Beschreibung: Unterschiedliche Positionen in einem Projekt sind mit unterschiedlichen Aufgaben und Verantwortungsbereiche verbunden [Bal98]. Durch Zugriffsrechte kann das SKMS helfen diese Verantwortungsbereiche zu wahren, indem eine Person nur Operationen ausführen kann, die zur Erledigung ihrer Aufgaben notwendig sind. Je nach Projektgröße und Projektumfeld sind Zugriffsrechte unterschiedlicher Komplexität nötig. Abbildung B.6 gibt einen Überblick über die detaillierten Merkmale der Zugriffsrechte.

Umsetzung : Teilweise implementiert

Kinder : „Benutzerverwaltung“ (B.5.1), „Granularität“ (B.5.2)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

Merkmale im Bereich: Komponenten

Detaillierte Merkmale: Versionskontrolle

- Versionskontrolle
  - Commit/Update
    - Atomare Commits
    - Ausgewählte Elemente ignorieren
    - Commitauswahl
      - Einzelnes Element
      - Mehrere Elemente
      - Rekursion mit Tiefenbeschränkung
      - Vollständige Rekursion
    - Commits sammeln
    - Commits löschen
    - Kommentare
      - Kommentare erzwingen
    - Strikte Trennung von Lesen/Schreiben
    - Updateauswahl
      - Einzelnes Element
      - Mehrere Elemente
      - Rekursion mit Tiefenbeschränkung
      - Vollständige Rekursion
  - Versionskontroll-Operationen
    - Element kopieren
    - Element umbenennen
    - Elemente verschmelzen
    - Element verschieben

Abbildung B.2.: Detaillierte Merkmale für die Versionskontrolle

B.1.1 Versionskontrolle ::Commit/Update

engl. Commit/Update

Ebene : Komponenten (Befähigung)

Beschreibung: Ein SKMS mit Repositoriums-Arbeitsbereich-Architektur synchronisiert ein Repositorium mit einem Arbeitsbereich, indem entweder Elemente aus dem Repositorium gelesen oder in das Repositorium geschrieben werden. Der Lesevorgang wird auch als *Update* und der Schreibvorgang als *Commit* bezeichnet [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : „Atomare commits“ (B.1.1.1), „Ausgewählte Elemente ignorieren“ (B.1.1.2), „Commitauswahl“ (B.1.1.3), „Commits sammeln“ (B.1.1.4), „Commits löschen“ (B.1.1.5), „Kommentare“ (B.1.1.6), „Strikte Trennung von Lesen/Schreiben“ (B.1.1.7), „Updateauswahl“ (B.1.1.8)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.1 Versionskontrolle ::Commit/Update ::Atomare commits

engl. Atomic commits

Ebene : Komponenten (Befähigung)

Beschreibung: Bei atomaren Transaktionen werden entweder alle oder keine Änderungen in das Repositorium übertragen [HR83].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : „Hibernate ORM“ (O.4.2)

Benötigt von : –

Quellen : [Ehr06]

B.1.1.2 Versionskontrolle ::Commit/Update ::Ausgewählte Elemente ignorieren

engl. Ignore selected items

Ebene : Komponenten (Befähigung)

Beschreibung: Im Arbeitsbereich existieren Elemente, die unter Versionskontrolle gestellt werden sollen. Diese Elemente entstehen dadurch, dass Werkzeuge, die zum Bearbeiten der Elemente im Arbeitsbereich verwandt werden, den Arbeitsbereich ohne Rücksicht auf das VKS nutzen und z.B. temporäre Elemente anlegen [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08] [Ca09]

B.1.1.3 Versionskontrolle ::Commit/Update ::Commitauswahl

engl. Commit selection

Ebene : Komponenten (Befähigung)

Beschreibung: Um Änderungen in das Repositorium zu übertragen (Commit), müssen zunächst die Elemente ausgewählt werden, die übertragen werden sollen. Dazu bieten sich verschiedene Auswahlstrategien an, um bei großen Mengen von Elementen nicht jedes Element einzeln auswählen zu müssen [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : „Einzelnes Element“ (B.1.1.3.1), „Mehrere Elemente“ (B.1.1.3.2), „Rekursion mit Tiefenbeschränkung“ (B.1.1.3.3), „Vollständige Rekursion“ (B.1.1.3.4)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.3.1 Versionskontrolle ::Commit/Update ::Commitauswahl ::Einzelnes Element

engl. Single item

Ebene : Komponenten (Befähigung)

Beschreibung: Überträgt die Änderungen eines einzigen Elements aus dem Arbeitsbereich in das Repositorium [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.3.2 Versionskontrolle ::Commit/Update ::Commitauswahl ::Mehrere Elemente

engl. Multiple items

Ebene : Komponenten (Befähigung)

Beschreibung: Überträgt die Änderungen mehrerer Elemente aus dem Arbeistbereich in das Repositorium [CSFP08].

Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.1.3.3 Versionskontrolle ::Commit/Update ::Commitauswahl ::Rekursion mit Tiefenbeschränkung

engl. Fixed depth

Ebene : Komponenten (Befähigung)
Beschreibung: Um bei großen Projekten nur Teile in das Repositorium zu schreiben, kann eine Beschränkung der Rekursionstiefe vorgenommen werden [CSFP08].
Umsetzung : Erweiterungspunkt
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.1.3.4 Versionskontrolle ::Commit/Update ::Commitauswahl ::Vollständige Rekursion

engl. Fully recursive

Ebene : Komponenten (Befähigung)
Beschreibung: Ein Commit eines Elements führt dazu, dass rekursiv alle darin enthaltenen Elemente zum Commit hinzugefügt werden [CSFP08].
Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.1.4 Versionskontrolle ::Commit/Update ::Commits sammeln

engl. Collect commits

Ebene : Komponenten (Befähigung)

Beschreibung: Mit Hilfe eines Index lassen sich geänderte Elemente für den nächsten Commit sammeln. Bei einem Commit wird dann der Index in das Repositorium geschrieben [Ca09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Ca09]

B.1.1.5 Versionskontrolle ::Commit/Update ::Commits löschen

engl. Delete commits

Ebene : Komponenten (Befähigung)

Beschreibung: Schreibvorgänge in ein Repositorium sind additiv, d.h. auch ein Löschvorgang wird dem Repositorium hinzugefügt, damit er wieder Rückgängig gemacht werden kann. In einigen Fällen ist es jedoch wünschenswert, Elemente vollständig aus dem Repositorium zu löschen [O'S09].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Ca09] [O'S09]

B.1.1.6 Versionskontrolle ::Commit/Update ::Kommentare

engl. Log messages

Ebene : Komponenten (Befähigung)

Beschreibung: Ein Kommentar beschreibt einen Commit, der in das Repositorium geschrieben wird [CSFP08].

Umsetzung : In Arbeit

Kinder : „Kommentare erzwingen“ (B.1.1.6.1)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08] [Ca09] [O'S09]

B.1.1.6.1 Versionskontrolle ::Commit/Update ::Kommentare ::Kommentare erzwingen

engl. Enforce log messages

Ebene : Komponenten (Befähigung)

Beschreibung: Commits ohne Kommentar werden abgelehnt, so dass der Nutzer einen Kommentar eingeben muss [KOL09].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [KOL09]

B.1.1.7 Versionskontrolle ::Commit/Update ::Strikte Trennung von Lesen/Schreiben

engl. Strict Pull/Push seperation

Ebene : Komponenten (Befähigung)

Beschreibung: Ein Schreibvorgang in das Repositorium löst keinen Lesevorgang aus. Dadurch werden nur die betroffenen Elemente bei einem Commit auf die aktuellste Version gesetzt, während alle anderen Elemente ihren Stand beibehalten [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.8 Versionskontrolle ::Commit/Update ::Updateauswahl

engl. Update selection

Ebene : Komponenten (Befähigung)

Beschreibung: Um Änderungen aus dem Repositorium in den Arbeitsbereich zu übertragen (Update), müssen zunächst die Elemente ausgewählt werden, die übertragen werden sollen. Dazu bieten sich verschiedene Auswahlstrategien an, um bei großen Mengen von Elementen nicht jedes Element einzeln auswählen zu müssen [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : „Einzelnes Element“ (B.1.1.8.1), „Mehrere Elemente“ (B.1.1.8.2), „Rekursion mit Tiefenbeschränkung“ (B.1.1.8.3), „Vollständige Rekursion“ (B.1.1.8.4)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.8.1 Versionskontrolle ::Commit/Update ::Updateauswahl ::Einzelnes Element

engl. Single item

Ebene : Komponenten (Befähigung)

Beschreibung: Überträgt die Änderungen eines einzigen Elements aus dem Repositorium in den Arbeitsbereich [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.8.2 Versionskontrolle ::Commit/Update ::Updateauswahl ::Mehrere Elemente

engl. Multiple items

Ebene : Komponenten (Befähigung)

Beschreibung: Überträgt die Änderungen mehrerer Elemente aus dem Repositorium in den Arbeitsbereich [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.8.3 Versionskontrolle ::Commit/Update ::Updateauswahl ::Rekursion mit Tiefenbeschränkung

engl. Fixed depth

Ebene : Komponenten (Befähigung)

Beschreibung: Um bei großen Projekten nur Teile des Arbeitsbereichs zu aktualisieren, kann eine Beschränkung der Rekursionstiefe vorgenommen werden [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.1.8.4 Versionskontrolle ::Commit/Update ::Updateauswahl ::Vollständige Rekursion

engl. Fully recursive

Ebene : Komponenten (Befähigung)

Beschreibung: Ein Update eines Elements führt dazu, dass rekursiv alle darin enthaltenen aktualisierten Elemente in den Arbeitsbereich übertragen werden [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.1.2 Versionskontrolle ::Versionskontroll-Operationen

engl. Version control operations

Ebene : Komponenten (Befähigung)

Beschreibung: VKS können – außer Commit und Update – weitere Operationen auf dem Repositorium anbieten. Dadurch können komplexe Interaktionen mit dem Repositorium mit Hilfe spezieller Operationen (teilweise) automatisiert werden [CSFP08] [Ca09] [O'S09].

Umsetzung : Teilweise implementiert

Kinder : „Element kopieren“ (B.1.2.1), „Element umbenennen“ (B.1.2.2), „Elemente verschmelzen“ (B.1.2.3), „Element verschieben“ (B.1.2.4)
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.2.1 Versionskontrolle ::Versionskontroll-Operationen ::Element kopieren

engl. Copy element

Ebene : Komponenten (Befähigung)
Beschreibung: Kopiert ein Element und verzeichnet dies in der Historie [CSFP08].
Umsetzung : Erweiterungspunkt
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.2.2 Versionskontrolle ::Versionskontroll-Operationen ::Element umbenennen

engl. Rename element

Ebene : Komponenten (Befähigung)
Beschreibung: Benennt ein Element um und verzeichnet dies in der Historie [CSFP08].
Umsetzung : Erweiterungspunkt
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.1.2.3 Versionskontrolle ::Versionskontroll-Operationen ::Elemente verschmelzen

engl. Merge items

Ebene : Komponenten (Befähigung)

Beschreibung: Beim kollaborativen Arbeiten kann es zu gleichzeitigen Änderungen an Elementen kommen, die miteinander verschmolzen werden sollen. Und auch für die Rückführung von Zweigen auf den Hauptast wird die Verschmelzen-Operation benötigt [CSFP08]. Es existieren verschiedene Methoden zum Verschmelzen [CW98].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : „Verschmelzen protokollieren“ (D.11.2.3.3)

Benötigt von : –

Quellen : [CW98] [CSFP08]

B.1.2.4 Versionskontrolle ::Versionskontroll-Operationen ::Element verschieben

engl. Move element

Ebene : Komponenten (Befähigung)

Beschreibung: Verschiebt ein Element und verzeichnet dies in der Historie [CSFP08].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

Merkmale im Bereich: Struktur

Detaillierte Merkmale: Produktmodelle

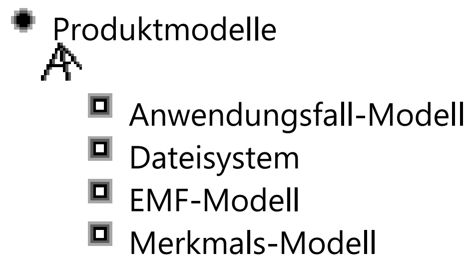


Abbildung B.3.: Detaillierte Merkmale für die Produktmodelle

B.2.1 Produktmodelle ::Anwendungsfall-Modell

engl. Feature model

Ebene : Struktur (Befähigung)

Beschreibung: Um Modelle zu versionieren, ist ein VKS auf Basis von Dateisystem-Elementen zu grobgranular. Die Granularität sollte vielmehr einzelne Modell-Elemente versionieren.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : ohne „Client für Arbeitsbereich“ (O.2)

Benötigt von : –

Quellen : Demo

B.2.2 Produktmodelle ::Dateisystem

engl. Filesystem

Ebene : Struktur (Befähigung)

Beschreibung: Die Elemente vieler VKS entsprechen Dateien und Verzeichnissen eines Dateisystems. Während das Repositorium ein versioniertes Dateisystem abbildet, entspricht der Arbeitsbereich einem unversionierten Dateisystem [CW98] [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : „MS Windows“ (O.1.2), ohne „Browser-Schnittstelle“ (O.2.1), ohne „Grafischer Client“ (O.2.2), ohne „Kommandozeile“ (O.2.4)

Benötigt von : –

Quellen : [Tic82] [CW98] [CSFP08] [Ca09]

B.2.3 Produktmodelle ::EMF-Modell

engl. EMF-model

Ebene : Struktur (Befähigung)

Beschreibung: Um Modelle zu versionieren, ist ein VKS auf Basis von Dateisystem-Elementen zu grobgranular. Die Granularität sollte vielmehr einzelne Modell-Elemente versionieren [Pöh09].

Umsetzung : Teilweise implementiert

Kinder : –

Setzt voraus : ohne „Browser-Schnittstelle“ (O.2.1), ohne „Grafischer Client“ (O.2.2), ohne „Kommandozeile“ (O.2.4)

Benötigt von : –

Quellen : [Pöh09]

B.2.4 Produktmodelle ::Merkmals-Modell

engl. Feature model

Ebene : Struktur (Befähigung)

Beschreibung: Um Modelle zu versionieren ist ein VKS auf Basis von Dateisystem-Elementen zu grobgranular. Die Granularität sollte vielmehr einzelne Modell-Elemente versionieren. Dies wäre ein erster Schritt in die Richtung *Versionierung von Produktlinien*.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : ohne „Browser-Schnittstelle“ (O.2.1), ohne „Grafischer Client“ (O.2.2), ohne „Kommandozeile“ (O.2.4)

Benötigt von : –

Quellen : –

Merkmale im Bereich: Team

Detaillierte Merkmale: Offline-Bedienbarkeit

- Offline-Bedienbarkeit
  - Vollständiges Repositorium
  - Zurücksetzen-Kommando

Abbildung B.4.: Detaillierte Merkmale für die Offline-Bedienbarkeit

B.3.1 Offline-Bedienbarkeit ::Vollständiges Repositorium

engl. Full repository

Ebene : Team (Befähigung)

Beschreibung: Wird das vollständige Repositorium zwischengespeichert [CSFP08], entspricht dies dem Arbeiten mit dezentralen Repositorien. Weiterhin entspricht der Abgleich zwischen einem Arbeitsbereich und einem zentralen Repositorium der Synchronisierung zweier (verteilter) Repositorien [Ca09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : „Peer-to-Peer-Unterstützung“ (B.4.2)

Quellen : [Ca09]

B.3.2 Offline-Bedienbarkeit ::Zurücksetzen-Operation

engl. Revert operation

Ebene : Team (Befähigung)

Beschreibung: Ein zentrales Repositorium mit verteilten Arbeitsbereichen erfordert einen Kommunikationskanal zwischen Repositorium und Arbeitsbereich. Fehlt dieser Kanal, so ist ein Arbeitsbereich „offline“ [CSFP08]. Alle Operationen, die Zugriff auf das Repositorium benötigen, sind nicht ausführbar. Das Zwischenspeichern von Daten aus dem Repositorium erlaubt die Verwendung der betroffenen Operationen.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : „Unverändertes Element“ (D.9.4)

Benötigt von : –

Quellen : [CSFP08]

Detaillierte Merkmale: Repositorium

- Repositorium
  - Hierarchie-Unterstützung
  - Peer-to-Peer-Unterstützung
  - Replikations-Unterstützung

Abbildung B.5.: Detaillierte Merkmale für das Repositorium

B.4.1 Repositorium ::Hierarchie-Unterstützung

engl. Hierarchy support

Ebene	: Team (Befähigung)
Beschreibung:	Bei umfangreichen Projekten oder mehreren Standorten erleichtert eine Hierarchisierung der Repositorien den Überblick über die versonierten Elemente [EC94] [Ca09].
Umsetzung	: Erweiterungspunkt
Kinder	: –
Setzt voraus	: –
Benötigt von	: –
Quellen	: [EC94] [Ca09] [Fis09]

B.4.2 Repositorium ::Peer-to-Peer-Unterstützung

engl. Peer-to-peer support

Ebene	: Team (Befähigung)
Beschreibung:	Peer-to-Peer-Repositorien erlauben den direkten Abgleich zwischen zwei Repositorien. Dabei wird pro Arbeitsbereich ein Repositorium verwendet [O'S09].
Umsetzung	: Erweiterungspunkt
Kinder	: –
Setzt voraus	: „Vollständiges Repositorium“ (B.3.1), „Replikations-Unterstützung“ (B.4.3), „Verschmelzen protokollieren“ (D.11.2.3.3), „Verteilt“ (D.10.2)
Benötigt von	: –
Quellen	: [O'S09] [Ca09]

B.4.3 Repositorium ::Replikations-Unterstützung

engl. Replication support

Ebene	: Team (Befähigung)
Beschreibung	: Wartungsarbeiten und Lastverteilung erfordern die Möglichkeit ein Repositorium zu replizieren [CSFP08].
Umsetzung	: Teilweise implementiert
Kinder	: –
Setzt voraus	: –
Benötigt von	: „Peer-to-Peer-Unterstützung“ (B.4.2)
Quellen	: [CSFP08] [Fis09]

Detaillierte Merkmale: Zugriffsrechte

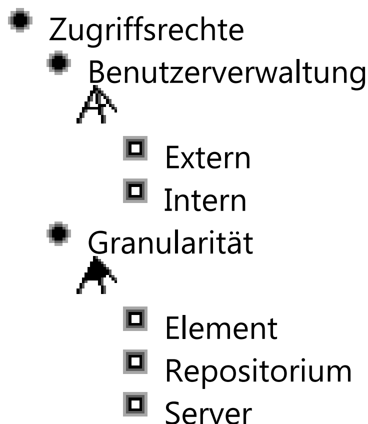


Abbildung B.6.: Detaillierte Merkmale für die Zugriffsrechte

B.5.1 Zugriffsrechte ::Benutzerverwaltung

engl. User management

Ebene : Team (Befähigung)

Beschreibung: Die Benutzerverwaltung kann entweder direkt über das SKM-System erfolgen oder auch über eine externe Software, zur zentralen Veraltung von Nutzerdaten (wie z.B. LDAP) [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Extern“ (B.5.1.1), „Intern“ (B.5.1.2)}

Setzt voraus : –

Benötigt von: „Autor“ (D.3.1), „Committer“ (D.3.2), „Benutzerkennwort“ (D.9.1), „Benutzername“ (D.9.2), „Sperren“ (D.11.2.2)

Quellen : [CSFP08]

B.5.1.1 Zugriffsrechte ::Benutzerverwaltung ::Extern

engl. External user management

Ebene : Team (Befähigung)

Beschreibung: Für die Prüfung der Zugriffsrechte kann auch eine externe Benutzerverwaltung, wie z.B. LDAP oder htaccess, verwendet werden [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von: –

Quellen : [CSFP08]

B.5.1.2 Zugriffsrechte ::Benutzerverwaltung ::Intern
engl. Internal user management

Ebene : Team (Befähigung)

Beschreibung: Für die Prüfung von Zugriffsrechten wird eine Benutzerverwaltung benötigt. Falls keine externe Verwaltung existiert, kann eine einfach interne Benutzerverwaltung verwendet werden [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.5.2 Zugriffsrechte ::Granularität
engl. Granularity

Ebene : Team (Befähigung)

Beschreibung: Feingranulare Zugriffsrechte bedeuten einen Mehraufwand an Administration, so dass die Granularität an die Anforderungen des SKM angepasst werden sollte [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : 1 bis 3 aus der Menge {„Element“ (B.5.2.1), „Repositorium“ (B.5.2.2), „Server“ (B.5.2.3)}

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

B.5.2.1 Zugriffsrechte ::Granularität ::Element
engl. Item based

Ebene : Team (Befähigung)

Beschreibung: Erhöht den Administrationsaufwand und erlaubt eine fein-granulare Vergabe der Zugriffsrechte auf Element-Niveau. So können z.B. Teilbereiche des Repositoriums für bestimmt Nutzer gesperrt oder freigschaltet werden [CSFP08].

Umsetzung : Erweiterungspunkt
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.5.2.2 Zugriffsrechte ::Granularität ::Repositorium

engl. Repository based

Ebene : Team (Befähigung)
Beschreibung: Für die meisten Systeme ist eine Steuerung der Zugriffsrechte pro Repositorium ausreichend [CSFP08].
Umsetzung : Teilweise implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

B.5.2.3 Zugriffsrechte ::Granularität ::Server

engl. Server based

Ebene : Team (Befähigung)
Beschreibung: Vereinfacht die Administration, indem die Zugriffsrechte serverweit vergeben werden [CSFP08].
Umsetzung : Teilweise implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

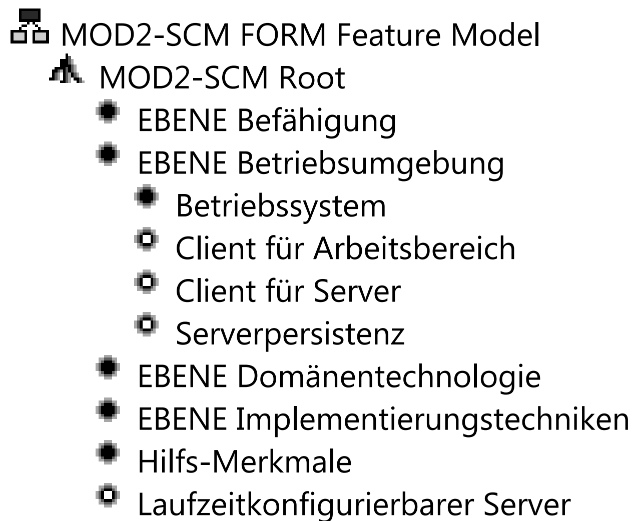


Abbildung B.7.: Merkmale auf der Betriebsumgebungs-Ebene

Merkmale der Betriebsumgebungs-Ebene

Merkmale auf der Betriebsumgebungs-Ebene beschreiben die Produktlinie mit Hilfe von Eigenschaften der Hard- und Software-Umgebung, in denen die konkrete Anwendung eingesetzt wird. Typische Merkmale beziehen sich auf die Hardware, das Betriebssystem, Datenspeicher oder auch das Netzwerk [KKL+98]. Diese Sicht lässt sich mit der eines System-Administrators vergleichen, der die Anwendung in Betrieb nehmen möchte. Als Quelle dienen:

- Die Handbücher der VKS *CVS* [Ced05], *Subversion* [CSFP08], *GIT* [Ca09] und *Mercurial* [O'S09], sowie des Subversion-Clients *TortoiseSVN* [KOL09].
- Ein Handbuch zu *Hibernate* [BHRS07] und eins zu *Java* [Ull09].
- Die *Eclipse*-Dokumentation [ecl10]
- Eine wissenschaftliche Arbeit über *CoObRA* [Sch07].

Abbildung B.7 zeigt eine Übersicht der Merkmale auf der Betriebsumgebungs-Ebene.

O.1 Betriebssystem *engl.* Operating system

Ebene : Betriebsumgebung

Beschreibung: Sowohl das Repositorium, als auch die Clients für die Verwaltung von Arbeitsbereich und Server benötigen ein Betriebssystem, in dem sie ausgeführt werden. Ist das erforderliche System nicht vorhanden, ist es unmöglich, das SKMS auszuführen. Abbildung B.8 (siehe S. 385) gibt einen Überblick über die detaillierten Merkmale des Betriebssystems.

Umsetzung : Vollständig implementiert

Kinder : „Java VM“ (O.1.1), „MS Windows“ (O.1.2)

Setzt voraus : –

Benötigt von: –

Quellen : –

O.2 Client für Arbeitsbereich *engl.* Workspace client

Ebene : Betriebsumgebung

Beschreibung: Der Arbeitsbereichs-Client ist die Benutzerschittstelle für den Arbeitsbereich eines SKMS. Sie bietet dem Nutzer Zugriff auf die Operationen des Arbeitsbereichs, erfragt notwendige Parameterwerte und stellt zurückgelieferte Informationen dar. Primär bietet ein Client Funktionen, um den lokalen Arbeitsbereich zu verwalten, aber es ist auch ein kombinierter Client für Arbeitsbereich und Server denkbar. Abbildung B.9 (siehe S. 386) gibt einen Überblick über die detaillierten Merkmale des Clients für den Arbeitsbereich.

Umsetzung : Teilweise implementiert

Kinder : „Browser-Schnittstelle“ (O.2.1), „Grafischer Client“ (O.2.2), „Integration in Eclipse“ (O.2.3), „Kommandozeile“ (O.2.4)

Setzt voraus : ohne „Anwendungsfall-Modell“ (B.2.1)

Benötigt von: –

Quellen : [CSFP08] [Ca09]

O.3 Client für Server *engl.* Server client

Ebene : Betriebsumgebung

Beschreibung: Der Server-Client ist die Benutzerschittstelle für den Server eines SKMS. Sie bietet dem Nutzer Zugriff auf die Operationen des Servers, erfragt notwendige Parameterwerte und stellt zurückgelieferte Informationen dar. Der Client bietet ausschließlich Funktionen, um den Server zu verwalten. Abbildung B.10 (siehe S. 388) gibt einen Überblick über die detaillierten Merkmale des Clients für den Server.

Umsetzung : Teilweise implementiert

Kinder : „Browser-Schnittstelle“ (O.3.1), „Grafischer Client“ (O.3.2), „Kommandozeile“ (O.3.3)

Setzt voraus : –

Benötigt von: –

Quellen : [CSFP08] [Ca09]

O.4 Serverpersistenz *engl.* Server persistency

Ebene : Betriebsumgebung

Beschreibung: Die Daten eines Repositoriums werden, mit Hilfe eines Persistenzmechanismus, dauerhaft gespeichert. Abbildung B.11 (siehe S. 390) gibt einen Überblick über die detaillierten Merkmale der Serverpersistenz.

Umsetzung : Teilweise implementiert

Kinder : „CoObRA Protokoll“ (O.4.1), „Hibernate ORM“ (O.4.2), „Java Objekt-Serialisierung“ (O.4.3)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

Detaillierte Merkmale: Betriebssystem

- Betriebssystem
  - Java VM
  - MS Windows

Abbildung B.8.: Detaillierte Merkmale für das Betriebssystem

O.1.1 Betriebssystem ::Java VM — *engl.* Java VM

Ebene : Betriebsumgebung

Beschreibung: In MODPL lassen sich Produkte nur auf Java-Quellcode abbilden [Buc10].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : „Java RMI“ (I.3.2)

Quellen : [Buc10]

O.1.2 Betriebssystem ::MS Windows — *engl.* MS Windows

Ebene : Betriebsumgebung

Beschreibung: Das Dateisystem-Produktmodell benötigt Zugriff auf das Dateisystem des Arbeitsbereichs.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : „Dateisystem“ (B.2.2)

Quellen : –

Detaillierte Merkmale: Client für Arbeitsbereich

- Client für Arbeitsbereich
  - Browser-Schnittstelle
  - Grafischer Client
  - Integration in Eclipse
  - Kommandozeile

Abbildung B.9.: Detaillierte Merkmale für den Arbeitsbereich-Client

O.2.1 Client für Arbeitsbereich ::Browser-Schnittstelle — *engl.* Webrowser interface

Ebene : Betriebsumgebung

Beschreibung: *Subversion* bietet einen Webbrowser-basierten Client, um den Arbeitsbereich mit dem Repositorium zu synchronisieren [CSFP08]

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : ohne „Anwendungsfall-Modell“ (B.2.1), ohne „Dateisystem“ (B.2.2), ohne „EMF-Modell“ (B.2.3), ohne „Merkmals-Modell“ (B.2.4)

Benötigt von : –

Quellen : [CSFP08]

O.2.2 Client für Arbeitsbereich ::Grafischer Client — *engl.* Graphical user interface

Ebene : Betriebsumgebung

Beschreibung: Bei SKMS mit Dateisystem-Produktmodell werden – zusätzlich zum Kommandozeilen-Client – Clients mit grafischer Benutzerschnittstelle verwendet, um den Arbeitsbereich mit dem Repositorium zu synchronisieren [CSFP08] [Ca09] [O'S09]

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : ohne „Anwendungsfall-Modell“ (B.2.1), ohne „Dateisystem“ (B.2.2), ohne „EMF-Modell“ (B.2.3), ohne „Merkmals-Modell“ (B.2.4)

Benötigt von : –

Quellen : [CSFP08] [Ca09] [O'S09]

O.2.3 Client für Arbeitsbereich ::Integration in Eclipse

engl. Eclipse intergation

Ebene : Betriebsumgebung

Beschreibung: Ein SKMS für Ecore- bzw. Feature-Produktmodelle sollte direkt in Eclipse integriert werden, um Eclipse für die Synchronisation mit dem Repositorium einsetzen zu können. Eclipse bietet auch eine Schnittstelle für die Synchronisation des Dateisystem-Produktmodells an.

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : ohne „Anwendungsfall-Modell“ (B.2.1)

Benötigt von : –

Quellen : Masterpraktikum

O.2.4 Client für Arbeitsbereich ::Kommandozeile

engl. Command line

Ebene : Betriebsumgebung

Beschreibung: Bei SKMS mit Dateisystem-Produktmodell werden häufig Kommandozeilen-Clients verwendet, um den Arbeitsbereich mit dem Repositorium zu synchronisieren [CSFP08] [Ca09] [O'S09]

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : ohne „Anwendungsfall-Modell“ (B.2.1), ohne „Dateisystem“ (B.2.2), ohne „EMF-Modell“ (B.2.3), ohne „Merkmals-Modell“ (B.2.4)

Benötigt von : –

Quellen : [CSFP08] [Ca09] [O'S09]

Detaillierte Merkmale: Client für Server

- Client für Server
  - Browser-Schnittstelle
  - Grafischer Client
  - Kommandozeile

Abbildung B.10.: Detaillierte Merkmale für den Server-Client

O.3.1 Client für Server ::Browser-Schnittstelle — *engl.* Webrowser interface

Ebene : Betriebsumgebung

Beschreibung: Statt über Kommandozeile kann ein Server auch über eine Applikation in einem Webbrowser verwaltet werden.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : Masterpraktikum

O.3.2 Client für Server ::Grafischer Client — *engl.* Graphical user interface

Ebene : Betriebsumgebung

Beschreibung: Clients mit grafischer Benutzerschnittstelle bieten – außer den Arbeitsbereich mit dem Repositorium zu synchronisieren – ebenfalls die Möglichkeit, den Server zu verwalten [KOL09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [KOL09]

O.3.3 Client für Server ::Kommandozeile — *engl.* Command line

Ebene : Betriebsumgebung

Beschreibung: Bei SKMS mit Dateisystem-Produktmodell werden häufig Kommandozeilen-Clients verwendet, um den Server zu verwalten [CSFP08] [Ca09] [O'S09]

Umsetzung	: Erweiterungspunkt
Kinder	: –
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CSFP08] [Ca09] [O'S09]

Detaillierte Merkmale: Serverpersistenz

- EBENE Betriebsumgebung
  - Serverpersistenz
    - CoObRA Protokoll
    - Hibernate ORM
    - Java Objekt-Serialisierung

Abbildung B.11.: Detaillierte Merkmale für die Serverpersistenz

O.4.1 Serverpersistenz ::CoObRA Protokoll *engl.* CoObRA protocol

---

Ebene : Betriebsumgebung

Beschreibung: MOD2-SCM unterstützt die Verwendung des CoObRA Potokolls [Sch07]. Die Verwendung führt jedoch zu zwei Einschränkungen: (1) Es wird nur ein CoObRA-Repositorium pro Server unterstützt. (2) Um den aktuellen Zustand zu laden, führt CoObRA alle bisher ausgeführten Operationen mit den zugehörigen Daten erneut aus, so dass z.B. bei Rückwärtsdeltas alle Deltas zunächst als Baseline angelegt und wieder entfernt werden.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : ohne „Gemischte Deltas“ (D.4.1.1.1), ohne „Rückwärts-Deltas“ (D.4.1.1.2)

Benötigt von : –

Quellen : [Sch07]

O.4.2 Serverpersistenz ::Hibernate ORM *engl.* Hibernate ORM

Ebene : Betriebsumgebung

Beschreibung: Hibernate ermöglicht eine objektrelationale Abbildung von Laufzeitobjekten in eine relationale Datenbank [Öhm10].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : „Atomare commits“ (B.1.1.1)

Quellen : [Öhm10]

O.4.3 Serverpersistenz ::Java Objekt-Serialisierung *engl.* Java object serialization

Ebene : Betriebsumgebung

Beschreibung: Java bietet die Möglichkeit, mittels Java Objekt-Serialisierung (JOS) die Objektstruktur und Zustände einer virtuellen Maschine zu serialisieren [Ull09].

Umsetzung : Teilweise implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Ull09]

Merkmale der Domänentechnologie-Ebene

Merkmale auf der Domänentechnologie-Ebene beschreiben die Produktlinie mit Hilfe von Techniken die speziell in der SKM-Domäne verwendet werden. Typische Merkmale behandeln die Realisierung von Elementen aus dem Produktraum und Versionsraum [KKL+98]. Diese Sicht lässt sich mit der eines Software-Entwicklers vergleichen, der ein Versionskontrollsystem für einen Entwicklungsprozess oder sogar ein Software-Projekt entwerfen und implementieren soll. Als Quelle dienen:

- Das initiale Übersichtspapier [CW98].
- Ein Vergleich zwischen diversen VKS, die Anwender bei der Wahl eines geeigneten VKS unterstützen soll [Ehr06].
- Die Handbücher der VKS *CVS* [Ced05], *Subversion* [CSFP08], *GIT* [Ca09] und *Mercurial* [O'S09], sowie des Subversion-Clients *TortoiseSVN* [KOL09].
- Die wissenschaftlichen Arbeiten über *SCCS* [Roc75] und *RCS* [Tic82].
- Wissenschaftliche Arbeiten über: Datenbank-Transaktionen [HR83], Delta-Algorithmen [HVT98] und zeitabhängige Entwurfsmuster [CEF99].

Abbildung B.12 zeigt eine Übersicht der Merkmale auf der Domänentechnologie-Ebene.

Überblick über Top-Level-Merkmale

D.1 Historie *engl.* History

Ebene	: Komponenten (Domänentechnologie)
Beschreibung:	Die Beziehungen zwischen unterschiedlichen Versionen eines Elements lassen sich mit Hilfe einer Entwicklungshistorie beschreiben [CW98]. Es existieren mehrere Typen von Historien, die sich in Komplexität und Ausdrucksstärke unterscheiden. Abbildung B.13 (siehe S. 398) gibt einen Überblick über die detaillierten Merkmale der Historie.
Umsetzung	: Teilweise implementiert
Kinder	: 1 aus der Menge {„Änderungsbasiert“ (D.1.1), „Zustandsbasiert“ (D.1.2)}
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CW98]

- MOD2-SCM FORM Feature Model
  - MOD2-SCM Root
    - EBENE Befähigung
    - EBENE Betriebsumgebung
    - EBENE Domänentechnologie
      - BEREICH Komponenten
        - Historie
        - Identifikation Versionsraum
        - Meta-Informationen
        - Speicher
        - Versionierte Elemente
        - Verzweigen
        - Zusammenspiel Produkt-/Versionsraum
      - BEREICH Struktur
        - Identifikation Produktraum
      - BEREICH Team
        - Arbeistbereich-Informationen
        - Repositoriums-Architektur
        - Synchronisation
    - EBENE Implementierungstechniken
    - Hilfs-Merkmale
    - Laufzeitkonfigurierbarer Server

Abbildung B.12.: Merkmale auf der Domänentechnologie-Ebene

D.2 Identifikation Versionsraum *engl.* Identification

Ebene	: Komponenten (Domänentechnologie)
Beschreibung:	Versionierte Elemente können über ihre ID im Produkt- bzw. Versionsraum eindeutig identifiziert werden. Um eine Version anhand von Meta-Informationen zu identifizieren, ist eine Auflösung dieser Informationen auf die entsprechenden IDs notwendig [CW98]. Abbildung B.14 (siehe S. 402) gibt einen Überblick über die detaillierten Merkmale der Identifikation.
Umsetzung	: Vollständig implementiert
Kinder	: „ID-Auflösung“ (D.2.1), „Versions-ID“ (D.2.2)
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CW98]

D.3 Meta-Informationen *engl.* Meta-information

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Versionierbare Elemente können mit beliebigen Meta-Informationen ausgezeichnet werden [CSFP08]. Diese Meta-Informationen strukturieren den Produkt- oder Versionsraum und können zur Auswahl spezifischer Versionen verwendet werden [CW98]. Abbildung B.15 (siehe S. 407) gibt einen Überblick über die detaillierten Merkmale der Meta-Informationen.

Umsetzung : Teilweise implementiert

Kinder : „Autor“ (D.3.1), „Committer“ (D.3.2), „Datum“ (D.3.3), „Eigenschaften“ (D.3.4), „Markierungen“ (D.3.5)

Setzt voraus : –

Benötigt von : –

Quellen : [CW98] [CSFP08]

D.4 Speicher *engl.* Storage

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Da in einem SKMS viele verschiedene Versionen eines Elements gespeichert werden müssen, existieren verschiedene Methoden, um den Speicherpatzbedarf zu verringern [HVT98]. Abbildung B.16 (siehe S. 409) gibt einen Überblick über die detaillierten Merkmale des Speichers.

Umsetzung : Vollständig implementiert

Kinder : „Deltas“ (D.4.1), „Kompression“ (D.4.2), „Referenzen komplexer Elemente“ (D.4.3)

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.5 Versionierte Elemente *engl.* Versioned items

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Zwischen Elementen im Produktraum können Kompositionsbeziehungen bestehen [CW98]. Abbildung B.17 (siehe S. 413) gibt einen Überblick über die detaillierten Merkmale der versionierten Elemente.

Umsetzung : Vollständig implementiert

Kinder : 1 aus der Menge {„Atomare Elemente“ (D.5.1), „Komplexe Elemente“ (D.5.2)}

Setzt voraus : –

Benötigt von: –

Quellen : [CW98]

D.6 Verzweigen *engl.* Branching

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Graph-basierte Historien unterstützen das Versionieren von Varianten durch Verzweigen [CW98]. Abbildung B.18 (siehe S. 414) gibt einen Überblick über die detaillierten Merkmale des Verzweigens.

Umsetzung : Vollständig implementiert

Kinder : „Private Zweige“ (D.6.1)

Setzt voraus : ohne „Einfache Menge“ (D.1.2.1.2), ohne „Sequenz“ (D.1.2.1.4)

Benötigt von: „Verschmelzen protokollieren“ (D.11.2.3.3)

Quellen : [CW98]

D.7 Zusammenspiel Produkt-/Versionsraum *engl.* PS-VS Interplay

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Sowohl zwischen versionierten Elementen bestehen Beziehungen im Produktraum, als auch zwischen den Versionen jedes Elements im Versionsraum [CW98]. Wird eine Version eines Elements im Versionsraum ausgewählt, wirkt sich das auch auf andere versionierte Elemente aus, mit denen es im Produktraum in Beziehung steht. Abbildung B.19 (siehe S. 415) gibt einen Überblick über die detaillierten Merkmale des Zusammenspiels.

Umsetzung : Vollständig implementiert

Kinder : 1 aus der Menge {„Produktzentriert“ (D.7.1), „Versionszentriert“ (D.7.2), „Verwoben“ (D.7.3)}

Setzt voraus : –

Benötigt von: –

Quellen : [CW98]

D.8 Identifikation Produktraum *engl.* Identification

Ebene : Struktur (Domänentechnologie)

Beschreibung: Versionierte Elemente können über ihre ID im Produkt- bzw. Versionsraum eindeutig identifiziert werden. Um eine Version anhand von Meta-Informationen zu identifizieren, ist eine Auflösung dieser Informationen auf die entsprechenden IDs notwendig [CW98]. Abbildung B.14 (siehe S. 402) gibt einen Überblick über die detaillierten Merkmale der Identifikation.

Umsetzung : Teilweise implementiert

Kinder : „Objekt-ID“ (D.8.1)

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.9 Arbeitsbereich-Informationen *engl.* Workspace information

Ebene : Team (Domänentechnologie)

Beschreibung: Die Elemente in einem Arbeitsbereich sind nur für den entsprechenden Entwickler sichtbar [Fei91]. Daher werden Informationen, die nur für den Entwickler relevant sind, im Arbeitsbereich gespeichert und nicht in das Repositorium übertragen. Während sich ein Repositorium auch auf einem entfernten System befinden kann, sind die Informationen im Arbeitsbereichs immer verfügbar. Daher können auch Informationen für den Offline-Betrieb darin gespeichert werden. Abbildung B.21 (siehe S. 419) gibt einen Überblick über die detaillierten Merkmale der Arbeitsbereich-Informationen.

Umsetzung : Erweiterungspunkt

Kinder : „Benutzerkennwort“ (D.9.1), „Benutzername“ (D.9.2), „Eigenschaften“ (D.9.3), „Unverändertes Element“ (D.9.4)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.10 Repositoriums-Architektur *engl.* Repository-Architecture

Ebene : Team (Domänentechnologie)

Beschreibung: Die Synchronisation der Arbeitsbereiche einzelner Benutzer kann entweder über einen zentralen Server oder direkt untereinander erfolgen [Ca09]. Abbildung B.22 (siehe S. 421) gibt einen Überblick über die detaillierten Merkmale des Repositoriums.

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Client-Server“ (D.10.1), „Verteilt“ (D.10.2)}

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08] [Ca09]

D.11 Synchronisation *engl.* Synchronization

Ebene : Team (Domänentechnologie)

Beschreibung: Sobald mehrere Benutzer ein Repositorium verwenden, tritt das Problem von nebenläufiger Modifikation von versionierten Elementen auf [CSFP08]. Dies kann durch optimistische oder pessimistisches Synchronisation gelöst werden, wobei optimistische noch das Verschmelzen von versionierten Elementen unterstützen sollte. Zusätzlich ist es notwendig, die auf dem Repositorium ausgeführten Operationen zu synchronisieren. Abbildung B.23 (siehe S. 422) gibt einen Überblick über die detaillierten Merkmale der Synchronisation.

Umsetzung : Teilweise implementiert

Kinder : „Intern“ (D.11.1), „Nebenläufigkeit“ (D.11.2)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

Merkmale im Bereich: Komponenten

Detaillierte Merkmale: Historie

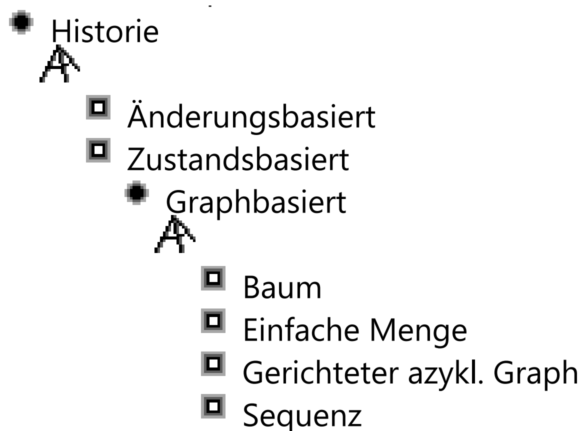


Abbildung B.13.: Detaillierte Merkmale für die Historie

D.1.1 Historie ::Änderungsbasiert

engl. Change based

Ebene : Komponenten (Domänentechnologie)

Beschreibung: In änderungsbasierten Versionsmodellen, wird eine Version in Form von Änderungen gegenüber einer Grundfassung beschrieben. So können Änderungsanfragen direkt auf die Änderungen abgebildet werden [CW98].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.1.2 Historie ::Zustandsbasiert

engl. State based

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Eine Version ist der Zustand eines sich verändernden Elements [CW98]. Zwischen Zuständen können Beziehungen definiert werden (z.B. Vorgänger-Nachfolger-Beziehungen). Zustände sind auch miteinander vergleichbar, so dass sich gemeinsame und unterschiedliche Eigenschaften identifizieren lassen.

Umsetzung : Teilweise implementiert

Kinder : „Graphbasiert“ (D.1.2.1)

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.1.2.1 Historie ::Zustandsbasiert ::Graphbasiert

engl. Graph based

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Versionsgraphen setzen Zustände miteinander in Beziehung. Zustände werden auf Knoten und Beziehungen auf Kanten eines Graphen abgebildet [CW98]. Die gebräuchlichste Beziehung zwischen zwei Zuständen in die Vorgänger-Nachfolger-Beziehung.

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Baum“ (D.1.2.1.1), „Einfache Menge“ (D.1.2.1.2), „Gerichteter azyklischer Graph“ (D.1.2.1.3), „Sequenz“ (D.1.2.1.4)}

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.1.2.1.1 Historie ::Zustandsbasiert ::Graphbasiert ::Baum

engl. Tree

Ebene : Komponenten (Domänentechnologie)

Beschreibung: In einem Baum kann ein Zustand mehrere Nachfolger haben [CW98]. Beim „Verzweigen“ (D.6) entsteht ebenfalls ein Baum mit zwei Kantentypen.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : kein „Verschmelzen protokollieren“ (D.11.2.3.3)

Benötigt von : –

Quellen : [CW98]

D.1.2.1.2 Historie ::Zustandsbasiert ::Graphbasiert ::Einfache Menge *engl.* Simple set

Ebene : Komponenten (Domänentechnologie)

Beschreibung: In einer einfachen Menge gibt es keine Beziehungen unter den Zuständen.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : ohne „Verzweigen“ (D.6)

Benötigt von : –

Quellen : [CW98]

D.1.2.1.3 Historie ::Zustandsbasiert ::Graphbasiert ::Gerichteter azyklischer Graph *engl.* Directed acyclic graph

Ebene : Komponenten (Domänentechnologie)

Beschreibung: In einem gerichteten azyklischen Graphen kann ein Zustand mehrere Vorgänger besitzen [CW98]. Bei „Verschmelzen protokollieren“ (D.11.2.3.3) entsteht mit dem Kantentyp des Merkmals „Verzweigen“ (D.6) ein gerichteter azyklischer Graph mit drei Kantentypen.

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : „Verschmelzen protokollieren“ (D.11.2.3.3)

Quellen : [CW98]

D.1.2.1.4 Historie ::Zustandsbasiert ::Graphbasiert ::Sequenz *engl.* Sequence

Ebene : Komponenten (Domänentechnologie)

Beschreibung: In einer Sequenz besitzt ein Zustand genau einen Vorgänger und genau einen Nachfolger [CW98].

Umsetzung : Vollständig implementiert

Kinder : –
Setzt voraus : ohne „Verzweigen“ (D.6)
Benötigt von : –
Quellen : [CW98]

Detaillierte Merkmale: Identifikation Versionsraum

- Identifikation Versionsraum
  - ID-Auflösung
    - Nach Datum
    - Nach Markierung
    - Neuste Version
  - Versions-ID
    - Hashwert
    - Numerisch
      - Flach
      - Hierarchisch
    - Zeitstempel

Abbildung B.14.: Detaillierte Merkmale für die Identifikation im Versionsraum

D.2.1 Identifikation Versionsraum ::ID-Auflösung — *engl.* ID resolution

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Statt einer Versionsnummer können auch andere Metadaten zur Identifikation einer Version verwendet werden, z.B. Schlüsselwörter oder ein Datum [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : „Nach Datum" (D.2.1.1), „Nach Markierung" (D.2.1.2), „Neuste Version" (D.2.1.3)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.2.1.1 Identifikation Versionsraum ::ID-Auflösung ::Nach Datum — *engl.* By date

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Durch Angabe eines Datums wird die letzte Version ausgewählt, die vor diesem Datum in das Repositorium übertragen wurde [CSFP08].

Umsetzung : Erweiterungspunkt
Kinder : –
Setzt voraus : „Datum“ (D.3.3)
Benötigt von : –
Quellen : [CSFP08]

D.2.1.2 Identifikation Versionsraum ::ID-Auflösung ::Nach Markierung

engl. By tag

Ebene : Komponenten (Domänentechnologie)
Beschreibung: Beim Markieren wird der aktuelle Stand des Repositoriums gekennzeichnet, so dass durch Angabe der Markierung dieser Zustand wiederhergestellt wird.
Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : „Markierungen“ (D.3.5)
Benötigt von : –
Quellen : [CSFP08]

D.2.1.3 Identifikation Versionsraum ::ID-Auflösung ::Neuste Version

engl. Latest

Ebene : Komponenten (Domänentechnologie)
Beschreibung: Durch Angabe des LATEST-Schlüsselworts wird die zuletzt übertragene Version ausgewählt [CSFP08].
Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : „Datum“ (D.3.3)
Benötigt von : –
Quellen : [CSFP08]

D.2.2 Identifikation Versionsraum ::Versions-ID *engl.* Version ID

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Jede Version eines Produktmodell-Elements wird durch eine Versions-ID genau identifiziert. Durch ein Tupel $(Objekt-ID, Versions-ID)$ wird somit eine Version eineindeutig identifiziert [CW98].

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Hashwert“ (D.2.2.1), „Numerisch“ (D.2.2.2), „Zeitstempel“ (D.2.2.3)}

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.2.2.1 Identifikation Versionsraum ::Versions-ID ::Hashwert *engl.* Hash-Value

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Als Versions-ID wird der Funktionswert einer Hashfunktion verwendet, die auf die Objekt-ID und alle weiteren Metadaten angewendet wird [Ca09]. In Abhängigkeit von der Länge des Funktionswertes besteht eine minimale Wahrscheinlichkeit, dass zwei unterschiedliche Elemente fälschlicherweise die gleiche Objekt-ID erhalten [Cha09].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Ca09]

D.2.2.2 Identifikation Versionsraum ::Versions-ID ::Numerisch *engl.* Numerical

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Als Versions-ID werden Ganzzahlen bzw. mehrere, durch Punkte getrennte, Ganzzahlen verwendet [Tic82]. Es existieren verschiedene Methoden, wann die Ganzzahlen inkrementiert werden.

Umsetzung : Vollständig implementiert

Kinder : 1 aus der Menge {„Flach“ (D.2.2.2.1), „Hierarchisch“ (D.2.2.2.2)}
Setzt voraus : –
Benötigt von : –
Quellen : [Tic82]

D.2.2.2.1 Identifikation Versionsraum ::Versions-ID ::Numerisch ::Flach *engl.* Flat

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die Versions-ID wird bei jeder neuen Version inkrementiert – unabhängig davon, ob es sich um eine Variante oder eine Revision handelt [Roc75].

Umsetzung : Vollständig implementiert

Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [Roc75]

D.2.2.2.2 Identifikation Versionsraum ::Versions-ID ::Numerisch ::Hierarchisch *engl.* Hierarchical

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die letzte Stelle der Versions-ID wird bei einer neuen Revision inkrementiert. Bei einer Variante wird eine neue Ganzzahl – durch einen Punkt getrennt – angehängt [Tic82].

Umsetzung : Vollständig implementiert

Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [Tic82]

D.2.2.3 Identifikation Versionsraum ::Versions-ID ::Zeitstempel

engl. Timestamp

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die Versions-ID ist der Zeitstempel der Übertragung ins Repositorium. Diese ID ist solange eindeutig, wie die Uhrzeit der Plattform nicht verändert wird [CEF99].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CEF99]

Detaillierte Merkmale: Meta-Informationen

- Meta-Informationen
  - Autor
  - Committer
  - Datum
  - Eigenschaften
  - Markierungen

Abbildung B.15.: Detaillierte Merkmale für die Meta-Informationen

D.3.1 Meta-Informationen ::Autor

engl. Author

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Der Autor ist der Urheber der Änderungen und daher für sie verantwortlich [Ca09]. Ohne Merkmal „Committer" (D.3.2) ist der Autor auch der Comitter.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : „Benutzerverwaltung" (B.5.1)

Benötigt von : –

Quellen : [CSFP08] [Ca09] [O'S09]

D.3.2 Meta-Informationen ::Committer

engl. Commiter

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Der Comitter ist verantwortlich für die Übertragung der Änderungen in das Repositorium [Ca09]. Ohne Merkmal „Autor" (D.3.1) ist der Comitter auch gleichzeitig der Autor.

Umsetzung : Teilweise implementiert

Kinder : –

Setzt voraus : „Benutzerverwaltung" (B.5.1)

Benötigt von : –

Quellen : [Ca09]

D.3.3 Meta-Informationen ::Datum — *engl.* Date

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Das Datum entspricht dem Zeitpunkt der Übertragung der Änderung in das Repositorium [Ca09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : „Nach Datum“ (D.2.1.1), „Neuste Version“ (D.2.1.3)

Quellen : [CSFP08] [Ca09] [O’S09]

D.3.4 Meta-Informationen ::Eigenschaften — *engl.* Properties

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Außer einem Element selbst können auch zugehörige Metadaten versioniert werden [CSFP08]. Werden Metadaten als Schlüssel-Werte-Paar einem Element zugeordnet, ist es möglich, eigene Metadaten hinzuzufügen.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.3.5 Meta-Informationen ::Markierungen — *engl.* Tags

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Mit Hilfe von Markierungen kann ein Abbild eines bestimmten Zustands des Repositoriums markiert werden [CSFP08]. Durch Angabe dieser Markierung kann dieser Zustand nun explizit wiederhergestellt werden.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : „Nach Markierung“ (D.2.1.2)

Quellen : [CSFP08]

Detaillierte Merkmale: Speicher

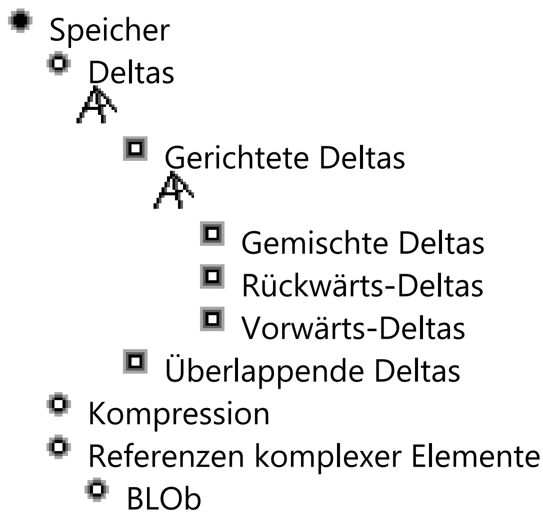


Abbildung B.16.: Detaillierte Merkmale für den Speicher

D.4.1 Speicher ::Deltas *engl.* Deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Das Speichern einer Version mit Hilfe von Deltas spart Speicherplatz [HVT98]. Es existieren mehrere Methoden der Delta-Speicherung, die sich in Ausdrucksmöglichkeiten und Laufzeiten bei der Wiederherstellung unterscheiden.

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Gerichtete Deltas“ (D.4.1.1), „Überlappende Deltas“ (D.4.1.2)}

Setzt voraus : –

Benötigt von : –

Quellen : [Tic82, CW98, HVT98]

D.4.1.1 Speicher ::Deltas ::Gerichtete Deltas *engl.* Directed deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Bei gerichteten Deltas wird eine Sequenz von Änderungen auf eine Grundversion angewendet, bis die gewünscht Version wiederhergestellt wurde [CW98, Mat09].

Umsetzung : Vollständig implementiert

Kinder : 1 aus der Menge {„Gemischte Deltas" (D.4.1.1.1), „Rückwärts-Deltas" (D.4.1.1.2), „Vorwärts-Deltas" (D.4.1.1.3)}

Setzt voraus : –

Benötigt von : –

Quellen : [Tic82, CW98, Mat09]

D.4.1.1.1 Speicher ::Deltas ::Gerichtete Deltas ::Gemischte Deltas

engl. Mixed deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Nur eine einzige Version – die aktuellste Version – wird als Grundfassung herangezogen. Alle weiteren Versionen, einschließlich Varianten, werden durch Anwendung von Deltas wiederhergestellt [Tic82].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : kein „CoObRA Protokoll" (O.4.1)

Benötigt von : –

Quellen : [Tic82]

D.4.1.1.2 Speicher ::Deltas ::Gerichtete Deltas ::Rückwärts-Deltas

engl. Backward deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die aktuellste Version jeder Variante wird als Grundfassung gespeichert, während alle Vorgänger mittels Deltas wiederhergestellt werden [CW98].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : kein „CoObRA Protokoll" (O.4.1)

Benötigt von : –

Quellen : [CW98]

D.4.1.1.3 Speicher ::Deltas ::Gerichtete Deltas ::Vorwärts-Deltas

engl. Forward deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die erste Version wird als Grundfassung gespeichert, während alle nachfolgenden Revisionen und Varianten als Delta gespeichert werden [CW98].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.4.1.2 Speicher ::Deltas ::Überlappende Deltas

engl. Interleaved deltas

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Alle Versionen eines Elements werden überlappend gespeichert, d.h. die einzelnen Teile aller Elemente werden markiert und so den Versionen zugeordnet, in denen sie enthalten sind [CW98].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Roc75] [CW98]

D.4.2 Speicher ::Kompression

engl. Compression

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Durch die Verwendung von Byte-Kompressions-Algorithmen werden die Daten im Repositorium komprimiert [Mat09]

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –
Benötigt von : –
Quellen : [Mat09]

D.4.3 Speicher ::Referenzen komplexer Elemente

engl. Complex item references

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Unterschiedliche Versionen eines komplexen Elements können gleiche Unterelemente besitzen. Durch einmaliges Speichern des Elements und anschließendes Referenzieren wird Speicherplatz eingespart. Bei gleichzeitiger Verwendung von „BLOb“ (D.4.3.1) kann der Speicherverbrauch weiter optimiert werden, indem dieses Konzept auf alle Elemente im gesamten Repositorium erweitert wird [Ca09].

Umsetzung : Vollständig implementiert

Kinder : „BLOb“ (D.4.3.1)

Setzt voraus : „Komplexe Elemente“ (D.5.2)

Benötigt von : „Komplexe Elemente“ (D.5.2)

Quellen : [Ca09]

D.4.3.1 Speicher ::Referenzen komplexer Elemente ::BLOb

engl. BLOb

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Blob steht für **Binary Large Object** (engl. für „großes binäres Objekt“) und beschreibt ein Element als Binärsequenz – ohne weitere Metadaten. So kann ein mehrfach vorkommendes Unterelement von allen beinhaltenden komplexen Elementen referenziert werden [Ca09].

Umsetzung : In Arbeit
Kinder : –
Setzt voraus : „Hashwert“ (D.8.1.3)

Benötigt von : –
Quellen : [Ca09]

Detaillierte Merkmale: Versionierte Elemente

Abbildung B.17.: Detaillierte Merkmale für die Versionierten Elemente

D.5.1 Versionierte Elemente ::Atomare Elemente *engl.* Atomic items

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Behandelt alle Elemente als atomar, d.h. es existieren keine Teil-Ganze-Beziehungen zwischen den Elementen.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CW98] [Tic82]

D.5.2 Versionierte Elemente ::Komplexe Elemente *engl.* Complex item

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Komplexe Elemente können Teil-Ganze-Beziehungen zu weiteren versionierten Elementen besitzen [CW98]. So lassen sich generische Methoden zum Sparen von Speicherplatz nutzen.

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : „Referenzen komplexer Elemente“ (D.4.3), ohne „Produktzentriert“ (D.7.1)

Benötigt von : „Referenzen komplexer Elemente“ (D.4.3), „Verwoben“ (D.7.3)

Quellen : [CW98]

Detaillierte Merkmale: Verzweigen

- Verzweigen
  - Private Zweige

Abbildung B.18.: Detaillierte Merkmale für das Verzweigen

D.6.1 Verzweigen ::Private Zweige *engl.* Private branches

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Die Änderungen eines Benutzer werden zunächst in einem nur für ihn sichtbaren Zweig versioniert und erst nach Abschluss aller Änderungen in den Hauptzweig übertragen [Ehr06].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Ehr06] [CSFP08]

Detaillierte Merkmale: Zusammenspiel Versions-/Produktraum

- Versionierte Elemente
  - Atomare Elemente
  - Komplexe Elemente

Abbildung B.19.: Detaillierte Merkmale für das Zusammenspiel Versions-/Produktraum

D.7.1 Zusammenspiel Produkt-/Versionsraum ::Produktzentriert — *engl.* Product first

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Bei produktzentriertem Zusammenspiel müssen zunächst alle Elemente des Produktmodells bestimmt werden, bevor die Versionen der Elemente festgelegt werden. Dadurch können komplexe Elemente nicht speziell versioniert werden [CW98].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : ohne „Komplexe Elemente“ (D.5.2)

Benötigt von : –

Quellen : [CW98]

D.7.2 Zusammenspiel Produkt-/Versionsraum ::Versionszentriert — *engl.* Version first

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Durch Bestimmen einer Version werden alle Elemente des Produktmodells und ihre Version festgelegt [CW98].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : ohne „Element“ (D.11.1.1)

Benötigt von : –

Quellen : [CW98]

D.7.3 Zusammenspiel Produkt-/Versionsraum ::Verwoben *engl.* Intertwined

Ebene : Komponenten (Domänentechnologie)

Beschreibung: Durch Bestimmen einer Version eines Produktmodells wird bestimmt, welche Elemente des Produktmodells darin enthalten sind. Anschließend wird für jedes dieser Produktmodelle die Version bestimmt. Zwischen beiden Schritten wird alterniert, bis keine neuen Elemente mehr hinzukommen [CW98].

Umsetzung : Teilweise implementiert

Kinder : –

Setzt voraus : „Komplexe Elemente“ (D.5.2), ohne „Element“ (D.11.1.1)

Benötigt von : –

Quellen : [CW98]

Merkmale im Bereich: Struktur

Detaillierte Merkmale: Identifikation Produktraum

- Identifikation Produktraum
  - Objekt-ID
    - Elementname
    - Global eindeutige ID
    - Hashwert

Abbildung B.20.: Detaillierte Merkmale für die Identifikation im Produktraum

D.8.1 Identifikation Produktraum ::Objekt-ID — *engl.* Object ID

Ebene	: Struktur (Domänentechnologie)
Beschreibung	: Die Elemente des Produktraumes werden durch eine eineindeutige Objekt-ID identifiziert. Besitzen zwei Elemente die gleiche Objekt-ID, so beziehen sie sich auf das gleiche Element (*Gleichheitskriterium*) [CW98]. Als Objekt-ID können verschiedene Daten verwendet werden, z.B. Elementnamen [Ced05] oder Hashwerte [Ca09].
Umsetzung	: Teilweise implementiert
Kinder	: 1 aus der Menge {„Elementname“ (D.8.1.1), „Global eindeutige ID“ (D.8.1.2), „Hashwert“ (D.8.1.3)}
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CW98] [Ca09] [O'S09]

D.8.1.1 Identifikation Produktraum ::Objekt-ID ::Elementname — *engl.* Item name

Ebene	: Struktur (Domänentechnologie)
Beschreibung	: Der Name eines Elements wird als Objekt-ID verwendet [Ced05]. Dies ist nur möglich, wenn die Namen der Elemente eindeutig sind. Änderungen des Elementinhalts sind invariant in Bezug auf das Gleichheitskriterium, Umbenennungen dagegen nicht.
Umsetzung	: Vollständig implementiert
Kinder	: –
Setzt voraus	: –
Benötigt von	: –
Quellen	: [Ced05]

D.8.1.2 Identifikation Produktraum ::Objekt-ID ::Global eindeutige ID

engl. Global unique identifier

Ebene : Struktur (Domänentechnologie)

Beschreibung: Als Objekt-ID wird eine im gesamten Produktraum eindeutige ID verwendet [CW98]. Sowohl Änderungen des Elementinhalts und Umbenennungen sind invariant im Bezug auf das Gleichheitskriterium. Es kann jedoch nicht sichergestellt werden, dass beim Erzeugen zweier gleicher Objekte in unterschiedlichen Arbeitsbereichen die gleiche Objekt-ID vergeben wird, so dass das Gleichheitskriterium – entgegen den Erwartungen des Benutzers – nicht gilt.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.8.1.3 Identifikation Produktraum ::Objekt-ID ::Hashwert

engl. Hash-Value

Ebene : Struktur (Domänentechnologie)

Beschreibung: Als Objekt-ID wird der Funktionswert einer Hashfunktion verwendet, die auf das entsprechende Element angewendet wurde [Ca09].Umbenennung sind invariant im Bezug auf das Gleichheitskriterium, Änderungen des Elementinhalts dagegen nicht. In Abhängigkeit von der Länge des Funktionswertes besteht eine minimale Wahrscheinlichkeit, dass zwei unterschiedliche Elemente fälschlicherweise die gleiche Objekt-ID erhalten [Cha09].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : „BLOb“ (D.4.3.1)

Quellen : [Ca09]

Merkmale im Bereich: Team

Detaillierte Merkmale: Arbeitsbereich-Informationen

- Arbeitsbereich-Informationen
  - Benutzerkennwort
  - Benutzername
  - Eigenschaften
  - Unverändertes Element

Abbildung B.21.: Detaillierte Merkmale für die Arbeitsbereich-Informationen

D.9.1 Arbeitsbereich-Informationen ::Benutzerkennwort — *engl.* User password

Ebene : Team (Domänentechnologie)

Beschreibung: Ein Benutzer muss sich per Benutzername und Kennwort authentifizieren, um auf das Repositorium zugreifen zu können [CSFP08]. Das Kennwort kann im Arbeitsbereich des Benutzers zwischengespeichert werden, um es nicht jedes Mal erneut einzugeben [KOL09]

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : „Benutzerverwaltung“ (B.5.1)

Benötigt von: –

Quellen : [KOL09]

D.9.2 Arbeitsbereich-Informationen ::Benutzername — *engl.* Username

Ebene : Team (Domänentechnologie)

Beschreibung: Ein Benutzer muss sich per Benutzername und Kennwort authentifizieren, um auf das Repositorium zugreifen zu können [CSFP08]. Der Benutzername kann im Arbeitsbereich des Benutzers zwischengespeichert werden, um ihn nicht jedes Mal erneut einzugeben [KOL09]

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : „Benutzerverwaltung“ (B.5.1)

Benötigt von: –

Quellen : [KOL09]

D.9.3 Arbeitsbereich-Informationen ::Eigenschaften

engl. Properties

Ebene : Team (Domänentechnologie)

Beschreibung: Elemente im Arbeitsbereich können mit selbst definierten Meta-Informationen verknüpft werden [CSFP08]. Bei Bedarf können diese Meta-Informationen auch nur im Arbeitsbereich gespeichert werden – und nicht ins Repositorium übertragen werden.

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.9.4 Arbeitsbereich-Informationen ::Unverändertes Element

engl. Unchanged item

Ebene : Team (Domänentechnologie)

Beschreibung: Um Änderungen an einem versionierten Element rückgängig zu machen, ohne dass Zugriff auf das Repositorium besteht, ist eine lokale Kopie des unveränderten Elements nötig [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : „Zurücksetzen-Operation“ (B.3.2)

Quellen : [CSFP08]

Detaillierte Merkmale: Repositoriums-Architektur

- Repositoriums-Architektur

  - Client-Server
  - Verteilt

Abbildung B.22.: Detaillierte Merkmale für das Repositorium

D.10.1 Repositoriums-Architektur ::Client-Server

engl. Client-Server

Ebene : Team (Domänentechnologie)

Beschreibung: Ein Repositorium ist ein zentrales Computersystem, das zum Speichern der versionierten Daten verwendet wird. Jeder Arbeitsbereich wird von einem Client verwaltet, der ihn mit genau einem Repositorium synchronisiert [CSFP08].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.10.2 Repositoriums-Architektur ::Verteilt

engl. Distributed

Ebene : Team (Domänentechnologie)

Beschreibung: Zu jedem Arbeitsbereich gehört ein lokales Repositorium. Dies kann mit beliebigen anderen Repositorien synchronisiert werden [O'S09]. Dadurch kann ebenfalls eine Client-Server-Architektur nachgebildet werden, doch können einzelne Arbeitsbereiche einzeln miteinander synchronisiert werden [Ca09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : „Peer-to-Peer-Unterstützung“ (B.4.2)

Quellen : [Ca09] [O'S09]

Detaillierte Merkmale: Synchronisation

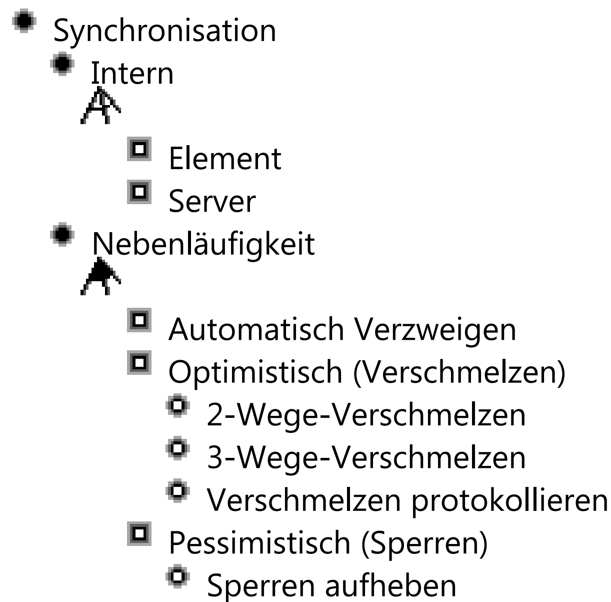


Abbildung B.23.: Detaillierte Merkmale für die Synchronisation

D.11.1 Synchronisation ::Intern — *engl.* Internal

Ebene : Team (Domänentechnologie)

Beschreibung: Synchronisation behandelt zum einen das Problem konkurrierender Änderungen im Mehrbenutzerbetrieb, und zum anderen die Isoliertheit der Transaktionen [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : 1 aus der Menge {„Element“ (D.11.1.1), „Server“ (D.11.1.2)}

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.11.1.1 Synchronisation ::Intern ::Element — *engl.* Item

Ebene : Team (Domänentechnologie)

Beschreibung: Stellt die Isoliertheit einer Transaktion sicher, indem die erforderlichen Elemente gesperrt werden.

Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : ohne „Versionszentriert“ (D.7.2), ohne „Verwoben“ (D.7.3)
Benötigt von : –
Quellen : [HR83]

D.11.1.2 Synchronisation ::Intern ::Server

engl. Server

Ebene : Team (Domänentechnologie)
Beschreibung: Stellt die Isoliertheit einer Transaktion sicher, indem der ganze Server gesperrt wird.
Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [HR83]

D.11.2 Synchronisation ::Nebenläufigkeit

engl. Concurrency

Ebene : Team (Domänentechnologie)
Beschreibung: Nebenläufigkeit behandelt das Problem konkurrierender Änderungen – entweder durch Sperren oder durch Verschmelzen [CSFP08].
Umsetzung : Teilweise implementiert
Kinder : Mindestens 1 aus der Menge {„Automatisch Verzweigen“ (D.11.2.1), „Sperren“ (D.11.2.2), „Verschmelzen“ (D.11.2.3)}
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08]

D.11.2.1 Synchronisation ::Nebenläufigkeit ::Automatisch Verzweigen *engl.* Branch automatically

Ebene : Team (Domänentechnologie)

Beschreibung: Wird eine neue Revision ins Repositorium eingespielt, obwohl bereits ein Nachfolger existiert, wird die neue Revision automatisch als Variante angelegt [Tic82].

Umsetzung : Vollständig implementiert

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [Tic82]

D.11.2.2 Synchronisation ::Nebenläufigkeit ::Sperren *engl.* Lock

Ebene : Team (Domänentechnologie)

Beschreibung: Durch Sperren eines oder mehrerer Elemente werden konkurrierende Änderungen an diesen Elementen verboten, bis die Sperre aufgehoben wird [CSFP08].

Umsetzung : Teilweise implementiert

Kinder : „Sperren aufheben" (D.11.2.2.1)

Setzt voraus : „Benutzerverwaltung" (B.5.1)

Benötigt von : –

Quellen : [CSFP08]

D.11.2.2.1 Synchronisation ::Nebenläufigkeit ::Sperren ::Sperren aufheben *engl.* Break locks

Ebene : Team (Domänentechnologie)

Beschreibung: Gestattet das Aufheben von Sperren durch privilegierte Benutzer [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

D.11.2.3 Synchronisation ::Nebenläufigkeit ::Verschmelzen — *engl.* Merge

---

Ebene : Team (Domänentechnologie)

Beschreibung: Verschmelzen vereinigt zwei unterschiedliche Elemente zu einer neuen Version [CW98]. So können z.B. Verzweigungen auf den Hauptzweig zurückgeführt oder konkurrierende Änderungen kombiniert werden.

Umsetzung : In Arbeit

Kinder : „2-Wege Verschmelzen“ (D.11.2.3.1), „3-Wege Verschmelzen“ (D.11.2.3.2), „Verschmelzen protokollieren“ (D.11.2.3.3)

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.11.2.3.1 Synchronisation ::Nebenläufigkeit ::Verschmelzen ::2-Wege Verschmelzen — *engl.* 2-way merge

---

Ebene : Team (Domänentechnologie)

Beschreibung: Unterstützt den Benutzer beim Verschmelzen zweier Elemente, indem Unterschiede und Gemeinsamkeiten angezeigt werden [CW98].

Umsetzung : In Arbeit

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CW98]

D.11.2.3.2 Synchronisation ::Nebenläufigkeit ::Verschmelzen ::3-Wege Verschmelzen — *engl.* 3-way merge

---

Ebene : Team (Domänentechnologie)

Beschreibung: Unterstützt den Benutzer beim Verschmelzen zweier Elemente, indem durch Einbeziehen des letzten gemeinsamen Vorgängers der Verschmelzungsvorgang teilweise automatisiert wird [CW98].

Umsetzung : In Arbeit
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : [CW98]

D.11.2.3.3 Synchronisation ::Nebenläufigkeit ::Verschmelzen ::Verschmelzen protokollieren

engl. Merge tracking

Ebene : Team (Domänentechnologie)

Beschreibung: Verschmelzen kann als zusätzliche Abhängigkeit in die Historie aufgenommen werden [CW98]. So können bei weiteren Verschmelzen-Operationen zusätzliche Abhängigkeiten verwendet werden.

Umsetzung : In Arbeit
Kinder : –
Setzt voraus : „Verzweigen“ (D.6), „Gerichteter azyklischer Graph“ (D.1.2.1.3)
Benötigt von : „Elemente verschmelzen“ (B.1.2.3), „Peer-to-Peer-Unterstützung“ (B.4.2)
Quellen : [CW98]

- EBENE Implementierungstechniken
  - Auslöser
  - Laufzeitkonfigurierbarer Server
  - Netzwerkprotkolle

Abbildung B.24.: Merkmale auf der Implementierungstechniken-Ebene

Merkmale der Implementierungstechniken-Ebene

Merkmale auf der Implementierungstechnik-Ebene beschreiben die Produktlinie mit Hilfe der verwendeten grundlegenden Techniken. Diese sind allerdings nicht domänenspezifisch sondern grundlegende Techniken der Datenübertragung und -verarbeitung. Methoden [KKL+98]. Diese Sicht lässt sich mit der eines Software-Entwicklers vergleichen, der die grundlegenden Datenstrukturen und Protokolle zur Datenübertragung festlegen soll. Als Quelle dienen:

- Die Handbücher der VKS *Subversion* [CSFP08], *GIT* [Ca09] und *Mercurial* [O'S09].
- Die Spezifikation des *WebDAV*-Protokolls [CSA+02].
- Ein Handbuch für Java [Ull09].

Abbildung B.24 zeigt eine Übersicht der Merkmale auf der Implementierungstechniken-Ebene.

I.1 Auslöser *engl.* Trigger

Ebene	: Implementierungstechniken
Beschreibung:	Für vordefinierte Ereignissen im Repositorium werden Benachrichtigungen gesendet, so dass die Funktionalität des VKS erweitert werden kann [CSFP08].
Umsetzung	: Erweiterungspunkt
Kinder	: „Nach-Ereignis-Auslöser“ (I.2.1), „Vor-Ereignis-Auslöser“ (I.2.2)
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CSFP08] [O'S09]

I.2 Laufzeitkonfigurierbarer Server *engl.* runtime configurable server

Ebene	: Implementierungstechniken
Beschreibung:	Mit Hilfe von Netzwerkprotokollen können Repositorien lokalisiert und Daten übertragen werden [CSFP08].

Umsetzung : Vollständig implementiert
Kinder : –
Setzt voraus : –
Benötigt von : –
Quellen : –

I.3 Netzwerkprotokolle *engl.* Network protocols

Ebene : Implementierungstechniken
Beschreibung: Mit Hilfe von Netzwerkprotokollen können Repositorien lokalisiert und Daten übertragen werden [CSFP08].
Umsetzung : Teilweise implementiert
Kinder : „E-Mail“ (I.3.1), „Java RMI“ (I.3.2), „Webserver“ (I.3.3)
Setzt voraus : –
Benötigt von : –
Quellen : [CSFP08] [Ca09] [O'S09]

Detailliert Merkmale: Auslöser

- Auslöser
  - Nach-Ereignis-Auslöser
    - E-Mail Benachrichtigung
  - Vor-Ereignis-Auslöser

Abbildung B.25.: Detaillierte Merkmale für die Auslöser

I.2.1 Auslöser ::Nach-Ereignis-Auslöser — *engl.* Post-Event trigger

Ebene : Implementierungstechniken

Beschreibung: Notifikationen werden nach einem Ereignis ausgelöst [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : „E-Mail Benachrichtigung“ (I.2.1.1)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

I.2.1.1 Auslöser ::Nach-Ereignis-Auslöser ::E-Mail Benachrichtigung — *engl.* E-Mail notification

Ebene : Implementierungstechniken

Beschreibung: Benutzer werden über Ereignisse im Repositorium per E-Mail benachrichtigt [O'S09].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [O'S09]

I.2.2 Auslöser ::Vor-Ereignis-Auslöser — *engl.* Pre-event trigger

Ebene : Implementierungstechniken

Beschreibung: Notifikationen werden vor einem Ereignis ausgelöst [CSFP08].

Umsetzung	: Erweiterungspunkt
Kinder	: –
Setzt voraus	: –
Benötigt von	: –
Quellen	: [CSFP08]

Detaillierte Merkmale: Netzwerkprotokolle

Abbildung B.26.: Detaillierte Merkmale für die Netzwerkprotokolle

I.3.1 Netzwerkprotokolle ::E-Mail

engl. E-Mail

Ebene	: Implementierungstechniken
Beschreibung	: Patches können per E-Mail versandt und direkt aus dem Postfach in das VKS übertragen werden [Ca09].
Umsetzung	: Erweiterungspunkt
Kinder	: –
Setzt voraus	: –
Benötigt von	: –
Quellen	: [Ca09]

I.3.2 Netzwerkprotokolle ::Java RMI

engl. Java RMI

Ebene	: Implementierungstechniken
Beschreibung	: Java ermöglicht die Übertragung von Daten und Funktionsaufrufen innerhalb verteilter Systeme [Ull09].
Umsetzung	: Vollständig implementiert
Kinder	: –
Setzt voraus	: „Java VM“ (O.1.1)
Benötigt von	: –
Quellen	: [Ull09]

I.3.3 Netzwerkprotokolle ::Webserver — *engl.* Webserver

Ebene : Implementierungstechniken

Beschreibung: Das Lokalisieren von Repositorien in verteilten Systemen sowie die Übertragung von Daten kann mittels Webserver geschehen [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : „HTTP“ (I.3.3.1), „HTTPS“ (I.3.3.2), „WebDAV“ (I.3.3.3)

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08] [Ca09] [O’S09]

I.3.3.1 Netzwerkprotokolle ::Webserver ::HTTP — *engl.* HTTP

Ebene : Implementierungstechniken

Beschreibung: Ermöglicht das Lokalisieren eines Repositoriums per URL und die Übertragung von Daten über das HTTP-Protokoll [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

I.3.3.2 Netzwerkprotokolle ::Webserver ::HTTPS — *engl.* HTTPS

Ebene : Implementierungstechniken

Beschreibung: Ermöglicht das lokalisieren eines Repositoriums per URL und die verschlüsselte Übertragung von Daten mittels HTTPS-Protokoll [CSFP08].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSFP08]

I.3.3.3 Netzwerkprotokolle ::Webserver ::WebDAV

engl. WebDAV

Ebene : Implementierungstechniken

Beschreibung: Das WebDAV Protokoll ermöglicht das verteilte Bearbeiten von Dateien und besitzt auch eine Protokollerweiterung für VKS [CSA+02].

Umsetzung : Erweiterungspunkt

Kinder : –

Setzt voraus : –

Benötigt von : –

Quellen : [CSA+02] [CSFP08]

C. MOD2-SKM Merkmalsmodell der SKM-Domäne

Ebene: Befähigung

Abbildung C.1.: Merkmalsmodell: Befähigungs-Ebene

Ebene: Betriebsumgebung

- EBENE Betriebsumgebung
  - Betriebssystem
    - Java VM
    - MS Windows
  - Client für Arbeitsbereich
    - Browser-Schnittstelle
    - Grafischer Client
    - Integration in Eclipse
    - Kommandozeile
  - Client für Server
    - Browser-Schnittstelle
    - Grafischer Client
    - Kommandozeile
  - Serverpersistenz
    - CoObRA Protokoll
    - Hibernate ORM
    - Java Objekt-Serialisierung

Abbildung C.2.: Merkmalsmodell: Betriebsumgebungs-Ebene

Ebene: Implementierungstechniken

- EBENE Implementierungstechniken
  - Auslöser
    - Nach-Ereignis-Auslöser
      - E-Mail Benachrichtigung
    - Vor-Ereignis-Auslöser
  - Netzwerkprotkolle
      - E-Mail
      - Java RMI
      - Webserver
        - HTTP
        - HTTPS
        - WebDAV

Abbildung C.3.: Merkmalsmodell: Implementierungstechniken-Ebene

Ebene: Domänentechnologie

Abbildung C.4.: Merkmalsmodell: Domänentechnologie-Ebene

D. Konfigurationen des MOD2-SKM Merkmalsmodells

Konfiguration: CVS

Abbildung D.1.: Merkmalsmodell: Konfiguration für CVS

Abbildung D.2.: Merkmalsmodell: Konfiguration für CVS (Fortsetzung)

Konfiguration: GIT

Abbildung D.3.: Merkmalsmodell: Konfiguration für GIT

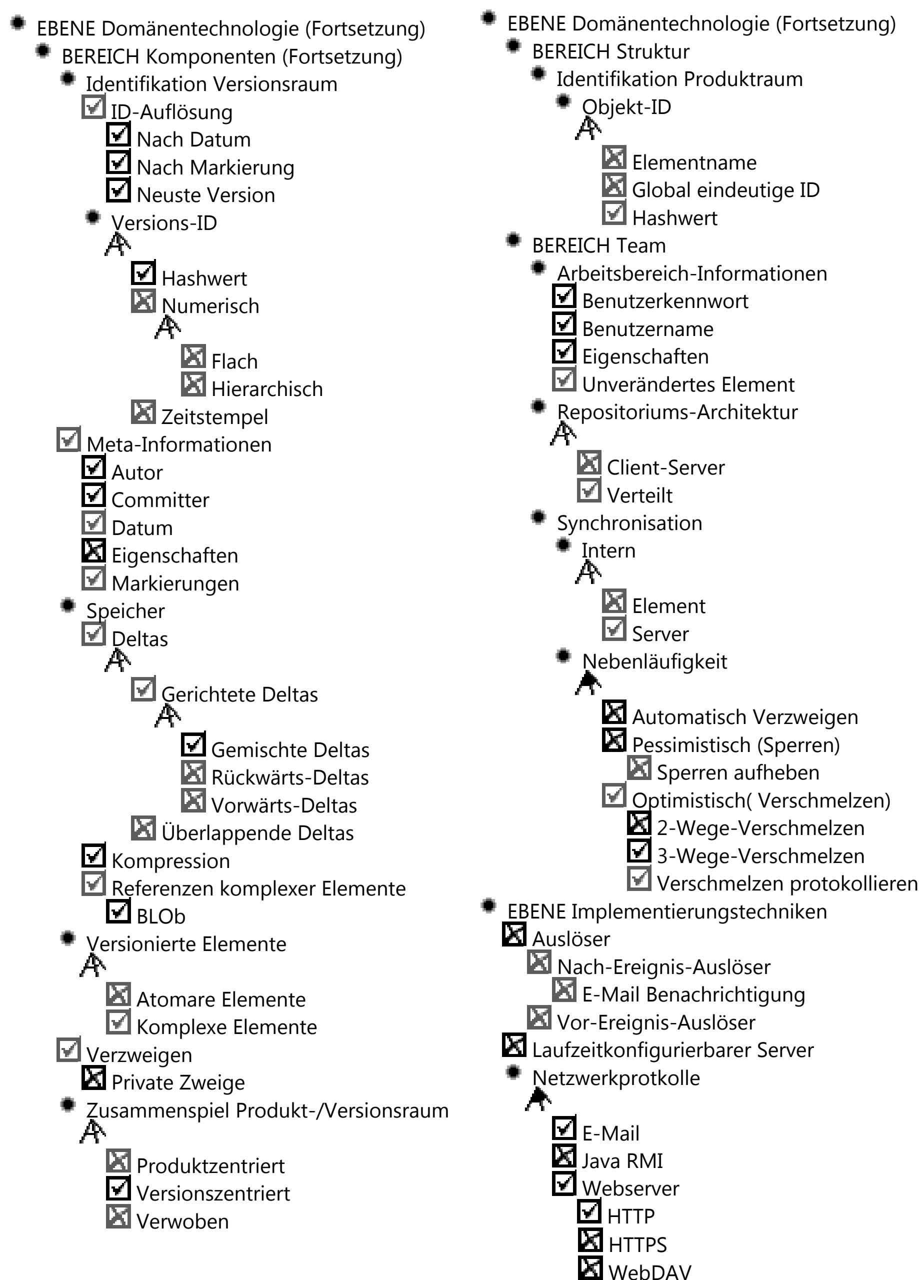


Abbildung D.4.: Merkmalsmodell: Konfiguration für GIT (Fortsetzung)

Konfiguration: Mercurial

Abbildung D.5.: Merkmalsmodell: Konfiguration für Mercurial

Abbildung D.6.: Merkmalsmodell: Konfiguration für Mercurial (Fortsetzung)

Konfiguration: Subversion

- EBENE Befähigung
  - BEREICH Komponenten
    - Versionskontrolle
      - Commit/Update
        - Atomare Commits
        - Ausgewählte Elemente ignorieren
        - Commitauswahl
          - Einzelnes Element
          - Mehrere Elemente
          - Rekursion mit Tiefenbeschränkung
          - Vollständige Rekursion
        - Commits sammeln
        - Commits löschen
        - Kommentare
          - Kommentare erzwingen
        - Strikte Trennung von Lesen/Schreiben
        - Updateauswahl
          - Einzelnes Element
          - Mehrere Elemente
          - Rekursion mit Tiefenbeschränkung
          - Vollständige Rekursion
      - Versionskontroll-Operationen
        - Element kopieren
        - Element umbenennen
        - Elemente verschmelzen
        - Element verschieben
  - BEREICH Struktur
    - Produktmodelle
      - Anwendungsfall-Modell
      - Dateisystem
      - EMF-Modell
      - Merkmals-Modell
  - BEREICH Team
    - Offline-Bedienbarkeit
      - Vollständiges Repositorium
      - Zurücksetzen-Kommando
    - Repositorium
      - Hierarchie-Unterstützung
      - Peer-to-Peer-Unterstützung
      - Replikations-Unterstützung

- EBENE Befähigung (Fortsetzung)
  - BEREICH Team (Fortsetzung)
    - Zugriffsrechte
      - Benutzerverwaltung
        - Extern
        - Intern
      - Granularität
        - Element
        - Repositorium
        - Server
- EBENE Betriebsumgebung
  - Betriebssystem
    - Java VM
    - MS Windows
  - Client für Arbeitsbereich
    - Browser-Schnittstelle
    - Grafischer Client
    - Integration in Eclipse
    - Kommandozeile
  - Client für Server
    - Browser-Schnittstelle
    - Grafischer Client
    - Kommandozeile
  - Serverpersistenz
    - CoObRA Protokoll
    - Hibernate ORM
    - Java Objekt-Serialisierung
- EBENE Domänentechnologie
  - BEREICH Komponenten
    - Historie
      - Änderungsbasiert
      - Zustandsbasiert
        - Graphbasiert
          - Baum
          - Einfache Menge
          - Gerichteter azykl. Graph
          - Sequenz

Abbildung D.7.: Merkmalsmodell: Konfiguration für Subversion

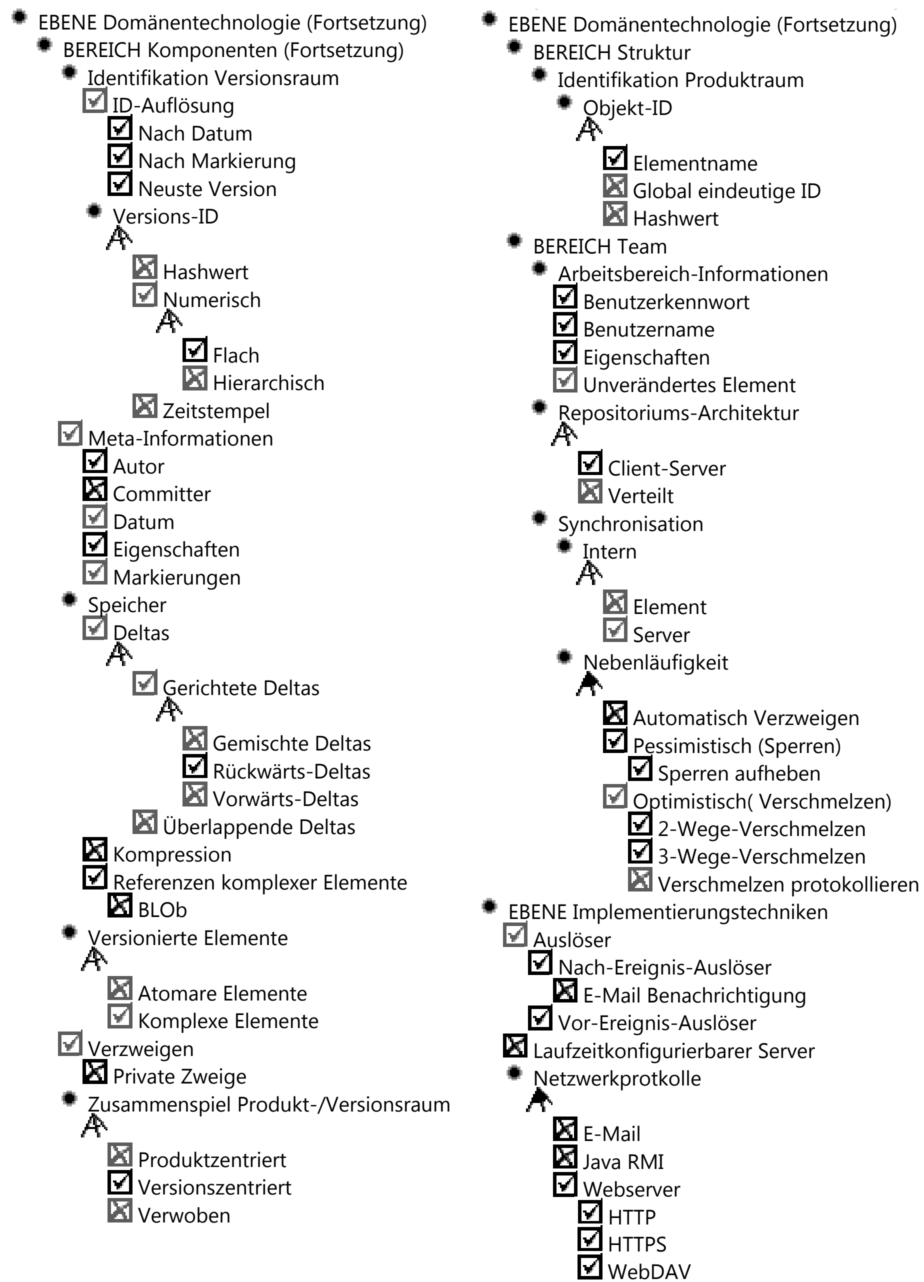


Abbildung D.8.: Merkmalsmodell: Konfiguration für Subversion (Fortsetzung)

E. Stand der Umsetzung

Liste der vollständig implementierten Merkmale	
„Anwendungsfall-Modell“ (B.2.1)	„Markierungen“ (D.3.5)
„Atomare Elemente“ (D.5.1)	„Mehrere Elemente“ (B.1.1.3.2)
„Ausgewählte Elemente ignorieren“ (B.1.1.2)	„Mehrere Elemente“ (B.1.1.8.2)
„Automatisch Verzweigen“ (D.11.2.1)	„Merkmals-Modell“ (B.2.4)
„Baum“ (D.1.2.1.1)	„MS Windows“ (O.1.2)
„Betriebssystem“ (O.1)	„Nach Markierung“ (D.2.1.2)
„Browser-Schnittstelle“ (O.3.1)	„Neuste Version“ (D.2.1.3)
„Client-Server“ (D.10.1)	„Numerisch“ (D.2.2.2)
„CoObRA Protokoll“ (O.4.1)	„Produktzentriert“ (D.7.1)
„Dateisystem“ (B.2.2)	„Referenzen komplexer Elemente“ (D.4.3)
„Einfache Menge“ (D.1.2.1.2)	„Rückwärts-Deltas“ (D.4.1.1.2)
„Einzelnes Element“ (B.1.1.3.1)	„Sequenz“ (D.1.2.1.4)
„Einzelnes Element“ (B.1.1.8.1)	„Server“ (D.11.1.2)
„Element“ (D.11.1.1)	„Speicher“ (D.4)
„Elementname“ (D.8.1.1)	„Strikte Trennung von Lesen/Schreiben“ (B.1.1.7)
„Flach“ (D.2.2.2.1)	„Versionierte Elemente“ (D.5)
„Gemischte Deltas“ (D.4.1.1.1)	„Versionszentriert“ (D.7.2)
„Gerichtete Deltas“ (D.4.1.1)	„Verwoben“ (D.7.3)
„Graphbasiert“ (D.1.2.1)	„Verzweigen“ (D.6)
„Hibernate ORM“ (O.4.2)	„Verwoben“ (D.7.3)
„Hierarchisch“ (D.2.2.2.2)	„Vollständige Rekursion“ (B.1.1.3.4)
„Identifikation Versionsraum“ (D.2)	„Vollständige Rekursion“ (B.1.1.8.4)
„Java RMI“ (I.3.2)	„Vorwärts-Deltas“ (D.4.1.1.3)
„Java VM“ (O.1.1)	„Zusammenspiel Produkt-/Versionsraum“ (D.7)
„Komplexe Elemente“ (D.5.2)	„Zustandsbasiert“ (D.1.2)
„Kompression“ (D.4.2)	
„Laufzeitkonfigurierbarer Server“ (I.2)	

Tabelle E.1.: Vollständig implementierte Merkmale

Liste der teilweise implementierten Merkmale

„Benutzerverwaltung“ (B.5.1)
„Client für Arbeitsbereich“ (O.2)
„Client für Server“ (O.3)
„Commit/Update“ (B.1.1)
„Commitauswahl“ (B.1.1.3)
„Committer“ (D.3.2)
„Deltas“ (D.4.1)
„EMF-Modell“ (B.2.3)
„Granularität“ (B.5.2)
„Historie“ (D.1)
„ID-Auflösung“ (D.2.1)
„Identifikation Produktraum“ (D.8)
„Intern“ (B.5.1.2)
„Intern“ (D.11.1)
„Java Objekt-Serialisierung“ (O.4.3)
„Meta-Informationen“ (D.3)
„Nebenläufigkeit“ (D.11.2)
„Netzwerkprotokolle“ (I.3)
„Objekt-ID“ (D.8.1)
„Versions-ID“ (D.2.2)
„Produktmodelle“ (B.2)
„Replikations-Unterstützung“ (B.4.3)
„Repositorium“ (B.5.2.2)
„Repositorium“ (B.4)
„Repositoriums-Architektur“ (D.10)
„Server“ (B.5.2.3)
„Serverpersistenz“ (O.4)
„Sperren“ (D.11.2.2)
„Synchronisation“ (D.11)
„Updateauswahl“ (B.1.1.8)
„Versionskontrolle“ (B.1)
„Versionskontroll-Operationen“ (B.1.2)
„Zugriffsrechte“ (B.5)

Tabelle E.2.: Teilweise implementierte Merkmale

Liste der noch bearbeiteten Merkmale

„2-Wege Verschmelzen“ (D.11.2.3.1)
„3-Wege Verschmelzen“ (D.11.2.3.2)
„Atomare commits“ (B.1.1.1)
„BLOb“ (D.4.3.1)
„Commits löschen“ (B.1.1.5)
„Elemente verschmelzen“ (B.1.2.3)
„Element verschieben“ (B.1.2.4)
„Gerichteter azyklischer Graph“ (D.1.2.1.3)
„Hashwert“ (D.2.2.1)
„Hashwert“ (D.8.1.3)
„Integration in Eclipse“ (O.2.3)
„Kommentare“ (B.1.1.6)
„Kommentare erzwingen“ (B.1.1.6.1)
„Verschmelzen“ (D.11.2.3)
„Verschmelzen protokollieren“ (D.11.2.3.3)

Tabelle E.3.: Noch bearbeitete Merkmale

Liste der Erweiterungspunkte

„Änderungsbasiert“ (D.1.1)
„HTTP“ (I.3.3.1)
„Arbeitsbereich-Informationen“ (D.9)
„Auslöser“ (I.1)
„Autor“ (D.3.1)
„Benutzerkennwort“ (D.9.1)
„Benutzername“ (D.9.2)
„Browser-Schnittstelle“ (O.2.1)
„Commits sammeln“ (B.1.1.4)
„Datum“ (D.3.3)
„Eigenschaften“ (D.3.4)
„Eigenschaften“ (D.9.3)
„Element kopieren“ (B.1.2.1)
„Element umbenennen“ (B.1.2.2)
„Element“ (B.5.2.1)
„E-Mail“ (I.3.1)
„E-Mail Benachrichtigung“ (I.2.1.1)
„Extern“ (B.5.1.1)
„Global eindeutige ID“ (D.8.1.2)
„Grafischer Client“ (O.2.2)
„Grafischer Client“ (O.3.2)
„Hierarchie-Unterstützung“ (B.4.1)
„HTTP“ (I.3.3.1)
„HTTPS“ (I.3.3.2)
„Kommandozeile“ (O.2.4)
„Kommandozeile“ (O.2.4)
„Nach Datum“ (D.2.1.1)
„Nach-Ereignis-Auslöser“ (I.2.1)
„Offline-Bedienbarkeit“ (B.3)
„Peer-to-Peer-Unterstützung“ (B.4.2)
„Private Zweige“ (D.6.1)
„Rekursion mit Tiefenbeschränkung“ (B.1.1.3.3)
„Rekursion mit Tiefenbeschränkung“ (B.1.1.8.3)
„Sperren aufheben“ (D.11.2.2.1)
„Überlappende Deltas“ (D.4.1.2)
„Unverändertes Element“ (D.9.4)
„Verteilt“ (D.10.2)
„Vollständiges Repositorium“ (B.3.1)
„Vor-Ereignis-Auslöser“ (I.2.2)
„WebDAV“ (I.3.3.3)
„Webserver“ (I.3.3)
„Zeitstempel“ (D.2.2.3)
„Zurücksetzen-Operation“ (B.3.2)

Tabelle E.4.: Erweiterungspunkte

Liste der Hilfs-Merkmale

ohne „Automatisch Verzweigen“ (D.11.2.1)
ohne „Java RMI“ (I.3.2)
ohne „Komplexe Elemente“ (D.5.2)
ohne „Kompression“ (D.4.2)
ohne „Laufzeitkonfigurierbarer Server“ (I.2)

Tabelle E.5.: Hilfs-Merkmale

Danksagungen

Ich bedanke mich bei meinem Betreuer *Prof. Dr. Bernhard Westfechtel* für die vielen hilfreichen Kommentare und die gute Unterstützung.

Dank gebührt auch meinem Kollegen *Thomas Buchmann*, für die vielen Gespräche und die Erstellung des MODPL-Werkzeugkastens, sowie *Dennis Wiebusch*, *Martin Fischbach* und *Anke Schubert* für ihre Kommentare. Und ohne die Arbeiten von *Christopher Bär*, *Matthias Kufer*, *Stefan Matthaei* und *Stefan Oehme* wäre die MOD2-SKM-Modulbibliothek nicht so groß geworden.

Nicht zu vergessen die Korrekturleser – allen voran *Brigitte Eisenmann*, sowie *Mathias Grimm*, *Felix Schwägerl* und *Laura Zimmermann* – die mir bei dieser umfangreichen Arbeit geholfen haben. Wenn noch ein Fehler in dieser Arbeit steckt, dann habt ihr das Kapitel nicht gelesen.

Besonderer Dank geht an *Albert Zündorf*, für Fujaba[1], und an *Leif Geiger*, für seine Hilfe beim „Wiederbeleben“ von beschädigten Fujaba-Modellen.

[1]other tools play,...

Literaturverzeichnis

[ABCM99] ASKLUND, Ulf ; BENDIX, Lars ; CHRISTENSEN, Henrik B. ; MAGNUSSON, Boris: The Unified Extensional Versioning Model. In: *Proc. 9th International Symposium on System Configuration Management (SCM-9)*. London, UK : Springer-Verlag, 1999, S. 100–122

[AC04] ANTKIEWICZ, Michal ; CZARNECKI, Krzysztof: FeaturePlugin: feature modeling plug-in for Eclipse. In: *Proc. 2004 OOPSLA workshop on eclipse technology eXchange (eclipse '04)*. New York, NY, USA : ACM Press, 2004, S. 67–72

[AKK+08] ALTMANNINGER, K. ; KAPPEL, G. ; KUSEL, A. ; RETSCHITZEGGER, W. ; SCHWINGER, W. ; SEIDL, M. ; WIMMER, M.: AMOR - Towards Adaptable Model Versioning. In: [DGPS08], S. 130–136

[Bab86] BABICH, Wayne A.: *Software Configuration Management*. Reading, MA, USA : Addison-Wesley, 1986

[Bal98] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Management*. Heidelberg : Spektrum, 1998

[Bal01] BALZERT, Helmut: *Lehrbuch der Software-Technik I: Software-Entwicklung*. 2. Aufl. Heidelberg : Spektrum, 2001

[BCK03] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software Architecture in Practice*. 2. Aufl. Boston, MA, USA : Addison-Wesley, 2003

[BD09a] BUCHMANN, Thomas ; DOTOR, Alexander: Constraints for a fine-grained mapping of feature models and executable domain models. Twente, NL : Centre for Telematics and Information Technology, 2009 (CTIT Workshop Proceedings 1), S. 9–17

[BD09b] BUCHMANN, Thomas ; DOTOR, Alexander: Mapping Features to Domain Models in Fujaba. In: *Proc. 7th International Fujaba Days*, 2009, S. 20–24

[BDW98] BRIAND, Lionel C. ; DALY, John W. ; WÜST, Jürgen: A Unified Framework for Cohesion Measurement in Object-OrientedSystems. In: *Empirical Software Engineering* Bd. 3 (1998), Nr. 1, S. 65–117

[BDW08a] BUCHMANN, Thomas ; DOTOR, Alexander ; WESTFECHTEL, Bernhard: MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In: [DGPS08], S. 50–65

[BDW08b] BUCHMANN, Thomas ; DOTOR, Alexander ; WESTFECHTEL, Bernhard: Triple Graph Grammars or Triple Graph Transformation Systems? In: [DGPS08], S. 34–49

[Ber90] BERLINER, Brian: *CVS II: Parallelizing Software Development.* Colorado Springs, CO, USA, 1990

[BGK91] BARGHOUTI, Naser S. ; GAIL ; KAISER, E.: Concurrency control in advanced database applications. In: *ACM Computing Surveys* Bd. 23 (1991), S. 269–317

[BHRS07] BEEGER, Robert F. ; HAASE, Arno ; ROOCK, Stefan ; SANITZ, Sebastian: *Hibernate.* 2. Aufl. Heidelberg : dpunkt, 2007

[BHS78] BERSOFF, Edward H. ; HENDERSON, Vilas D. ; SIEGEL, Stan G.: Software Configuration Management. In: *Proceedings of the software quality assurance workshop on Functional and performance issues.* New York, NY, USA : ACM Press, 1978, S. 9–17

[BHS80] BERSOFF, Edward H. ; HENDERSON, Vilas D. ; SIEGEL, Stanley G.: *Software Configuration Management: An Investment in Product Integrity.* Englewood Cliffs, NJ, USA : Prentice-Hall, 1980

[BM05] BELLAGIO, David E. ; MILLIGAN, Tom J.: *Software Configuration Management Strategies and IBM Rational ClearCase.* 2. Aufl. Upper Saddle River, NJ, USA : IBM Press, 2005

[Bär10] BÄR, Christopher: *Modellierung und Integration eines Verschmelzungsmoduls in das MOD2-SKM Rahmenwerk.* Bayreuth, Universität Bayreuth, Diplomarbeit, 2010

[Buc10] BUCHMANN, Thomas: *Modelle und Werkzeuge für modellgetriebene Softwareproduktlinien am Beispiel von Softwarekonfigurationsverwaltungssystemen.* Bayreuth, Universität Bayreuth, Doktorarbeit, 2010

[Ca09] CHACON, Scott ; AL. et: *The Git Community Book*, 2009. – `http://book.git-scm.com/` (Stand: 15. Okt. 2009)

[Ced05] CEDERQVIST, Per ; FREE SOFTWARE FOUNDATION, INC. (Hrsg.): *Version Management with CVS.* Boston, MA, USA: Free Software Foundation, Inc., 2005

[CEF99] *Kapitel* 13 – Temporal Patterns. In: [FRH99], S. 241–262

[Cha09] CHACON, Scott: *Pro Git.* Berkeley, CA, USA : Apress, 2009

[CHE05] CZARNECKI, Krzysztof ; HELSEN, Simon ; EISENECKER, Ulrich W.: Formalizing cardinality-based feature models and their specialization. In: *Software Process: Improvement and Practice* Bd. 10 (2005), Nr. 1, S. 7–29

[CN01] Clements, Paul ; Northrop, Linda: *Software product lines: practices and patterns*. Bd. 0201703327. Boston, MA, USA : Addison-Wesley, 2001

[CSA+02] Clemm, G. ; Software, Rational ; Amsden, J. ; Ellison, T. ; IBM ; Kaler, C. ; Microsoft ; Whitehead, J. ; The Internet Society (Hrsg.): *Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning)*. Santa Cruz, CA, USA: The Internet Society, 2002. – `http://webdav.org/specs/rfc3253.pdf` (Stand: 05. Feb. 2010)

[CSFP08] Collins-Sussman, Ben ; Fitzpatrick, Brian W. ; Pilato, C. M.: *Version Control with Subversion*, 2008

[CTZ01] Chien, Shu yao ; Tsotras, Vassilis J. ; Zaniolo, Carlo: Efficient Management of Multiversion Documents by Object Referencing. In: *Proc. 27th International Conference on Very Large Data Bases (VLDB 2001)*, 2001, S. 291–300

[CW97] Conradi, Reidar ; Westfechtel, Bernhard: Towards a Uniform Version Model for Software Configuration Management. In: Conradi, Reidar (Hrsg.): *Software Configuration Management*. Boston, MA, USA : Springer-Verlag, 1997 (Lecture Notes of Computer Science), S. 1–17

[CW98] Conradi, Reidar ; Westfechtel, Bernhard: Version models for software configuration management. In: *ACM Computing Surveys* Bd. 30 (1998), Nr. 2, S. 232–282

[Dar90] Dart, Susan: Concepts in Configuration Management Systems. In: Feiler, Peter H. (Hrsg.): *Proceedings of the 3rd International Workshop on Software Configuration Management*. Trondheim, N : ACM Press, 1990, S. 1–18

[DGPS08] Deridder, Dirk (Hrsg.) ; Gray, Jeff (Hrsg.) ; Pierantonio, Alfonso (Hrsg.) ; Schobbens, Pierre-Yves (Hrsg.): *Proc. 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM08)*. Bd. *1*. 2008

[EC94] Estublier, Jacky ; Casallas, Rubby: The Adele Configuration Manager. In: Tichy, Walter F. (Hrsg.): *Configuration Management* Bd. 2. New York, NY, USA : John Wiley & Sons, 1994, S. 99–134

[ecl10] eclipse.org ; Eclipse Foundation (Hrsg.): *The Eclipse Documentation*. k.A.: Eclipse Foundation, 2010. – `http://help.eclipse.org/galileo/` – (Stand: 11. Jan. 2010)

[Ehr06] Ehrlich, Debbie: *Version Control and Configuration Management Features Breakdown*. Website, 2006. – `http://www.relisoft.com/co_op/vcs_breakdown.html` (Stand: 14. Jan. 2011)

[Est95] ESTUBLIER, Jacky: Process Session. In: ESTUBLIER, Jacky (Hrsg.): *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers.* Seattle, WA, USA : Springer-Verlag, 1995 (Lecture Notes of Computer Science), S. 136–137

[Fei91] FEILER, Peter H.: Configuration Management Models in Commercial Environments / Carnegie Mellon University. 1991. – Forschungsbericht

[Fel79] FELDMAN, Stuart I.: Make — A Program for Maintaining Computer Programs. In: *Software Practice and Experience* Bd. 9 (1979), Nr. 4, S. 255–265

[FF04] FREEMAN, Eric ; FREEMAN, Elisabeth: *Head First Design Patterns.* Beijing, CH : O'Reilly, 2004

[Fis09] FISH, Shlomi: *Better SCM Initiative : Comparison.* Website, 2009. – `http://better-scm.berlios.de/comparison/scm-comparison.xml` (Stand: 6. Nov. 2010)

[FRH99] FOOTE, Brian ; ROHNERT, Hans ; HARRISON, Neil: *Pattern Languages of Program Design 4.* Boston, MA, USA : Addison-Wesley, 1999 (Pattern Languages of Program Design)

[Ge05] GE, Guozheng: *ICSE'05 Doctoral Symposium Presentation.* Presentation, 2005. – `http://users.soe.ucsc.edu/\~guozheng/doc/ICSE05_DocSymp_guozheng.ppt` (Stand: 14. Jan. 2011)

[GJSO91] GIFFORD, David K. ; JOUVELOT, Pierre ; SHELDON, Mark A. ; O'TOOLE, James W.: Semantic File Systems. In: *13th ACM Symposium on Operating Systems Principles*, 1991

[GS10] GUIFFY SOFTWARE, Inc.: *Guiffy.* Webseite., 2010. – `http://www.guiffy.com/` (Stand: 17. Jan. 2011)

[Guo08] GUOZHENG, Ge: *RHIZOME: A Feature Modeling and Generation Platform for Software Product Lines.* Santa Cruz, CA, USA, University of California, Diss., 2008

[HHW96] HOEK, André van d. ; HEIMBIGNER, Dennis ; WOLF, Alexander L.: A generic, peer-to-peer repository for distributed configuration management. In: *Proc. 18th international conference on Software engineering (ICSE '96).* Washington, DC, USA : IEEE Computer Society Press, 1996, S. 308–317

[HK99] HITZ, Martin ; KAPPEL, Gerti: *UML@Work: Von der Analyse zur Realisierung.* dpunkt, 1999

[Öhm10] ÖHME, Stefan: *Auswahl und Integration eines Persistenzrahmenwerks für Fujaba-Modellinstanzen.* Bayreuth, Universität Bayreuth, Diplomarbeit, 2010

[HN86] Habermann, A N. ; Notkin, D: Gandalf: software development environments. In: *IEEE Transactions on Software Engineering* Bd. 12 (1986), Nr. 12, S. 1117–1127

[Hoe00] Hoek, Adriaan van d.: *A Reusable, Distributed Repository for Configuration Management Policy Programming.* Boulder, CO, USA, University of Colorado, Diss., 2000

[Hol05] Holzner, Steve: *Ant: The Definitive Guide.* Sebastopol, CA, USA : O'Reilly, 2005

[HR83] Haerder, Theo ; Reuter, Andreas: Principles of transaction-oriented database recovery. In: *ACM Computing Surveys* Bd. 15 (1983), Nr. 4, S. 287–317

[HVT98] Hunt, James J. ; Vo, Kiem-Phong ; Tichy, Walter F.: Delta algorithms: an empirical analysis. In: *ACM Transactions on Software Engineering and Methodology* Bd. 7 (1998), Nr. 2, S. 192–214

[Ie10] Informatik e.V., GI G.: *Informatik-Begriffsnetz.* Webseite., 2010. – `http://public.beuth-hochschule.de/~giak/` (Stand: 14. Jan. 2011)

[IEE05] IEEE: IEEE Standard for Software Configuration Management Plans (Revision of IEEE Std 828-1998) / IEEE Computer Society. 2005. – Forschungsbericht. – 0_1–19 S.

[Jäg03] Jäger, Dirk: *Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen.* Aachen, Technischen Hochschule Aachen, Diplomarbeit, 2003

[Kat90] Katz, Randy H.: Toward a unified framework for version modeling in engineering databases. In: *ACM Computing Surveys* Bd. 22 (1990), Nr. 4, S. 375–409

[KCH+90] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. 1990. – Forschungsbericht

[KG04] Kovse, Jernej ; Gebauer, Christian: VS-Gen: A case study of a product line for versioning systems. In: *Proceedings on Generative Programming and Component Engineering 2004*, 2004, S. 396–415

[KKL+98] Kang, Kyo C. ; Kim, Sajoong ; Lee, Jaejoon ; Kim, Kijoo ; Kim, Gerard J. ; Shin, Euiseob: FORM: A feature-oriented reuse method with domain-specific reference architectures. In: *Annals of Software Engineering* Bd. 5 (1998), S. 143–168

[KLD02] Kang, Kyo C. ; Lee, Jaejoon ; Donohoe, Patrick: Feature-Oriented Product Line Engineering. In: *IEEE Software* Bd. 19 (2002), Nr. 4, S. 58–65

[KMLO98] Kneuper, Ralf (Hrsg.) ; Müller-Luschnat, Günther (Hrsg.) ; Oberweis, Andreas (Hrsg.): *Vorgehensmodelle für die betriebliche Anwendungsentwicklung*. Stuttgart : B.G. Teubner, 1998

[KOL09] Küng, Stefan ; Onken, Lübbe ; Large, Simon: *TortoiseSVN - A Subversion client for Windows*. 1.6.6, 2009

[Kov05] Kovése, J.: *Model-driven development of versioning systems*. Kaiserslautern, Technische Universität Kaiserslautern, Diss., 2005

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. Boston, MA, USA : Addison-Wesley, 2003

[Leb95] Leblang, D.B.: The CM Challenge: Configuration Management that Works. In: Tichy, W.F. (Hrsg.): *Configuration Management*. 1. Aufl. West Sussex, UK : Wiley, 1995 (Trends in Software), Kapitel 1

[LH04] Lingen, Ronald van d. ; Hoek, Andre van d.: An Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition. In: *Proc. 26th International Conference on Software Engineering (ICSE '04)*. Washington, DC, USA : IEEE Computer Society Press, 2004, S. 573–582

[Mac00] MacDonald, Joshua P.: File System Support for Delta Compression / University of California. Berkeley, CA, USA, 2000. – Forschungsbericht

[Mat09] Matthaei, Stefan: *Modellierung des GNU-DIFF-Algorithmus in FUJABA*. Bayreuth, Universität Bayreuth, Diplomarbeit, 2009

[MMM+93] Magnusson, Boris ; Magnusson, Boris ; Minör, Sten ; Asklund, Ulf ; Asklund, Ulf: Fine-Grained Revision Control for Collaborative Software Development. In: *Proc. 1993 ACM SIGSOFT Conference on Foundations of Software Engineering*. Los Angeles, CA, USA, 1993, S. 7–10

[MN05] Meyer, M. ; Niere, J.: Calculation and Visualization of Software Product Metrics. In: *Proc. 3rd International Fujaba Days 2005*. Paderborn, 2005

[Ngu06] Nguyen, Tien N.: Model-oriented Configuration Management for Relational Database Applications. In: *Proc. 6th IEEE International Conference on Computer and Information Technology (CIT '06)*, 2006, S. 194–194

[NNMB05] Nguyen, T.N. ; Nguyen, T.N. ; Munson, E.V. ; Boyland, J.T.: An infrastructure for development of object-oriented, multi-level configuration management services. In: Munson, E.V. (Hrsg.): *Proc. 27th International Conference on Software Engineering (ICSE '05)*, 2005, S. 215–224

[OMG09a] OMG ; OMG (Hrsg.): *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2.* Needham, MA, USA: OMG, 2009. – Version 2.2

[OMG09b] OMG ; OMG (Hrsg.): *OMG Unified Modeling Language (OMG UML), Superstructure V2.2.* Needham, MA, USA: OMG, 2009. – Version 2.2

[O'S09] O'Sullivan, Bryan: *Mercurial: The Definitive Guide.* O'Reilly, 2009

[Par72] Parnas, D. L.: On the Criteria To Be Used in Decomposing Systems into Modules. In: *Communications of the ACM* Bd. 15 (1972), S. 1053–1058

[PBL05] Pohl, Klaus ; Böckle, Günter ; Linden, Frank van d.: *Software Product Line Engineering: Foundations, Principles and Techniques.* Berlin : Springer-Verlag, 2005

[Pöh09] Pöhner, Udo: *Entwurf und Implementierung eines versionierbaren Ecore-Metamodells.* Bayreuth, Universität Bayreuth, Diplomarbeit, 2009

[Pop09] Popp, Gunther: *Konfigurationsmanagement mit Subversion, Maven und Redmine.* 3. Aufl. dpunkt, 2009

[Rev10a] *List of Revision Control Software.* Webseite, 2010. – `http://en.wikipedia.org/w/index.php?title=List_of_revision_control_software\&oldid=340205844` (Stand: 8. Feb. 2010)

[Rev10b] *Top: Computers: Software: Configuration Management: Tools.* Webseite, 2010. – `http://www.dmoz.org/Computers/Software/Configuration_Management/Tools/` (Stand: 8. Feb. 2010)

[Roc75] Rochkind, Marc J.: The Source Code Control System. In: *IEEE Transactions on Software Engineering* Bd. 1 IEEE Computer Society Press, 1975, S. 364–370

[SBPM09] Steinberg, Dave ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *EMF Eclipse Modeling Framework.* 2. Aufl. Boston, MA, USA : Addison-Wesley, 2009 (The Eclipse Series)

[Sch95] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: *Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science.* London, UK : Springer-Verlag, 1995, S. 151–163

[Sch07] Schneider, Christian: *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten.* Kassel, Universität Kassel, Diss., 2007

[SFH+99] Santry, Douglas S. ; Feeley, Michael J. ; Hutchinson, Norman C. ; Veitch, Alistair C. ; Carton, Ross W. ; Ofir, Jacob: Deciding when to forget in the Elephant file system. In: *Proc. 17th ACM symposium on Operating systems principles* Bd. 33 (1999), Nr. 5, S. 110–123

[SMC74] Stevens, W. P. ; Myers, G. J. ; Constantine, L. L.: Structured design. In: *IBM Systems Journal* Bd. 13 (1974), Nr. 2, S. 115–139

[Sno86] Snodgrass, Richard T.: Temporal databases. In: *IEEE Computer* Bd. 19 (1986), S. 35–42

[SWZ99] Schürr, Andy ; Winter, Andreas ; Zündorf, Albert: The PROGRES Approach: Language and Environment. In: *Handbook of graph grammars and computing by graph transformation* Bd. 2. River Edge, NJ, USA : World Scientific Publishing, 1999, S. 487–550

[Tic82] Tichy, Walter F.: Design, implementation, and evaluation of a Revision Control System. In: *Proc. 6th international conference on Software engineering (ICSE '82)*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1982, S. 58–67

[Tic85] Tichy, Walter F.: RCS – A system for version control. In: *Software: Practice and Experience* Bd. 15 (1985), Nr. 7, S. 637–654

[Tic88] Tichy, Walter F.: Tools for Software Configuration Management. In: Winkler, J. F. H. (Hrsg.): *Proc. International Workshop on Software Version and Configuration Control*. Grassau : Teubner, 1988, S. 1–20

[Uhr08] Uhrig, Sabrina: Matching class diagrams: with estimated costs towards the exact solution? In: *Proc. 2008 international workshop on Comparison and versioning of software models (CVSM '08)*. New York, NY, USA : ACM Press, 2008, S. 7–12

[Ull09] Ullenboom, Christian: *Java ist auch eine Insel*. 8. Aufl. Galileo Computing, 2009

[VAC+09] Vogel, Oliver ; Arnold, Ingo ; Chughtai, Arif ; Ihler, Edmund ; Kehrer, Timo ; Mehlig, Uwe ; Zdun, Uwe: *Software-Architektur: Grundlagen – Konzepte – Praxis*. 2. Aufl. Heidelberg : Spektrum, 2009

[WGP04] Whitehead, James E. ; Ge, GuoTheng ; Pan, Kai: Automatic Generation of Version Control Systems / University of California. Santa Cruz, CA, USA, 2004. – Forschungsbericht

[WJ95] Wieringa, Roel ; Jonge, Wiebren de: Object identifiers, keys, and surrogates: object identifiers revisited. In: *Theory and Practice of Object Systems* Bd. 1 (1995), Nr. 2, S. 101–114

[WL99] Weiss, David M. ; Lai, Chi Tau R.: *Software product-line engineering: a family-based software development process*. Boston, MA, USA : Addison-Wesley, 1999

[WMC01] WESTFECHTEL, Bernhard ; MUNCH, Björn P. ; CONRADI, Reidar: A Layered Architecture for Unfiorm Version Management. In: *IEEE Transactions on Software Engineering* Bd. 27 IEEE Computer Society Press, 2001, S. 1111–1133

[Zel95] ZELLER, Andreas: A Unified Version Model for Configuration Management. In: KAISER, Gail (Hrsg.): *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* Bd. 20 (4), ACM Press, 1995 (ACM Software Engineering Notes), S. 151–160

[Zün02] ZÜNDORF, Albert: *Rigorous Object Oriented Software Development.* Kassel, 2002

[ZS97] ZELLER, Andreas ; SNELTING, Gregor: Unified Versioning through Feature Logic. In: *ACM Transactions on Software Engineering and Methodology* Bd. 6 (1997), Nr. 4, S. 397–440