

Definition einer Sprache zur Beschreibung von Prozessmustern zur Unterstützung agiler Softwareentwicklungsprozesse

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

DISSERTATION

zur Erlangung des akademischen Grades

DOCTOR RERUM NATURALIUM
(Dr. rer. nat.)

im Fachgebiet Informatik

vorgelegt von

Dipl.-Inform. Mariele Hagen

geboren am 18. Februar 1972 in Bergisch Gladbach

Die Annahme der Dissertation haben empfohlen

1. Prof. Dr. Volker Gruhn, Universität Leipzig
2. Prof. Dr. Gregor Engels, Universität Paderborn
3. Prof. Dr. Bogdan Franczyk, Universität Leipzig

Die Verleihung des akademischen Grades erfolgt auf Beschluss des Rates der Fakultät für Mathematik und Informatik vom 23. Mai 2005 mit dem Gesamtprädikat „magna cum laude“.

Bibliographische Daten

Hagen, Mariele

Definition einer Sprache zur Beschreibung von Prozessmustern zur Unterstützung agiler Softwareentwicklungsprozesse

Universität Leipzig, Diss., 272 S., 212 Lit., 143 Abb., 23 Tab.

Prozessmuster ermöglichen die modulare Modellierung und flexible Anwendung von Softwareentwicklungsprozessen. Gegenwärtige Beschreibungen von Prozessmustern weisen jedoch Mängel wie uneinheitliche und uneindeutige Beschreibungsformen und fehlende Beziehungsdefinitionen auf. Diese Mängel wirken sich nachteilig auf den effektiven Einsatz von Prozessmustern aus. In dieser Dissertation wird die Sprache PROPEL (Process Pattern Description Language) entwickelt, die Konzepte zur Beschreibung von Prozessmustern und Beziehungen zwischen Prozessmustern bereitstellt.

Mit Hilfe von PROPEL können einzelne Prozessmuster modelliert und durch Definition von Beziehungen zu komplexeren Prozessmustern zusammengesetzt werden. PROPEL basiert auf der UML und verwendet daher eine Vielzahl von erprobten und weit verbreiteten Modellierungskonzepten wie Aktivitätsdiagramme wieder. Zur Erhöhung der Ausdrucksgenauigkeit wurde PROPEL um eine formale Semantik durch Abbildung der formalen Syntax auf die Domäne der Petri-Netze ergänzt.

Für die Validierung der Nutzbarkeit und Handhabbarkeit von PROPEL wurde ein Prozessmusterkatalog basierend auf dem Rational Unified Process entwickelt. Durch die Darstellung verschiedener Sichten auf den Katalog kann ein Überblick auf alle vorhandenen Prozessmuster und deren Beziehungen gewährleistet werden. Darüber hinaus wurde gezeigt, dass die Komplexität eines Prozessmodells durch Einsatz von PROPEL reduziert wird und Prozessinkonsistenzen eliminiert werden.

Danksagung

Zunächst möchte ich Herrn Prof. Dr. Volker Gruhn für die Schaffung des kreativen Umfelds bei der adesso AG und am Lehrstuhl für angewandte Telematik/e-Business danken, in dem diese Arbeit entstehen konnte. Darüber hinaus danke ich ihm für seine Bereitschaft zur Diskussion, seine hilfreichen Anmerkungen und Beiträge und seine Aufmunterungen.

Herrn Prof. Dr. Gregor Engels danke ich für die kritische Würdigung der Arbeit und die konstruktive Diskussionsbereitschaft.

Weiterhin möchte ich mich bei allen Kollegen von der adesso AG und des Lehrstuhls für angewandte Telematik/e-Business für Diskussionen und Hilfestellungen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Insbesondere danke ich Traugott Dittmann, Andreas Luckmann und Jens Schröder für viele fruchtbare Ideen.

Besonderer Dank gilt Thomas Goesmann, der durch seine unermüdliche Unterstützung und Motivation sowohl fachlich als auch seelisch den gesamten Prozess der Erstellung dieser Arbeit begleitet hat und maßgeblichen Anteil am erfolgreichen Abschluss der Arbeit besitzt. Ferner danke ich ihm für die Korrektur der Arbeit und für noch viel mehr.

Leipzig, im Oktober 2004

Mariele Hagen

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.1.1	Prozessmuster – Chancen	2
1.1.2	... und ihre Risiken	3
1.2	Zielsetzung der Arbeit	4
1.3	Lösungsansatz	6
1.4	Aufbau der Arbeit	7
1.5	Hinweise für den Leser	7
1.5.1	Voraussetzungen	7
1.5.2	Typographische Konventionen	8
2	Grundlagen	9
2.1	Muster und Prozessmuster	9
2.1.1	Historie	9
2.1.2	Der Begriff des Musters	10
2.1.3	Workflowmuster	11
2.1.4	Der Begriff des Musterschemas	12
2.1.5	Formalisierung von Mustern	14
2.1.6	Klassifizierung von Mustern	15
2.1.7	Organisation von Mustern	18
2.1.8	Beziehungen zwischen Mustern	20
2.1.9	Pattern Management	25
2.2	Softwareprozessmodellierungssprachen	28
2.3	Semantik von UML-Aktivitätsdiagrammen	31
2.4	Vorgehensmodelle in der Softwareentwicklung – von rigide bis agil	34
2.4.1	V-Modell 97	37
2.4.2	Rational Unified Process	38
2.4.3	Extreme Programming	41
2.4.4	Crystal Family	41
2.4.5	Fazit	42

3	Konzepte der Process Pattern Description Language PROPEL...	45
3.1	Klassifizierung von Prozessmustern	45
3.2	Prozessmuster	48
3.3	Beziehungen zwischen Prozessmustern	54
3.3.1	Sequence-Beziehung	55
3.3.2	Use-Beziehung	56
3.3.3	Refinement-Beziehung	59
3.3.4	Processvariance-Beziehung	61
3.3.5	Einfluss der Objekt- und Ereignis-Komposition auf Prozessmusterbeziehungen	62
3.4	Prozessmusterschema und Problemschema	65
3.5	Zusammenfassung	67
4	Syntax von PROPEL	69
4.1	Struktur des PROPEL-Metamodells	69
4.2	Process Pattern Package	72
4.2.1	ProcessPattern	73
4.2.2	Problem	76
4.2.3	Context	77
4.2.4	Process	78
4.2.5	ActivityProblemMapping	80
4.2.6	Role	80
4.2.7	Tool	81
4.3	Activity Graphs Package	82
4.3.1	ObjectFlowState	82
4.3.2	ActionState	82
4.3.3	OFSComposition	83
4.4	State Machines Package	84
4.4.1	PseudoState	84
4.5	Relationship Package	85
4.5.1	ObjectFlowState	85
4.5.2	Problem	85
4.5.3	ProcessPattern	86
4.5.4	ProcessPatternRelationship	86
4.5.5	Sequence	86
4.5.6	Use	87
4.5.7	RefineProblem	90
4.5.8	Refinement	91
4.5.9	Processvariance	92

4.5.10	ProcessPatternCatalog	94
4.5.11	Aspect	94
4.6	Process Pattern Graph Package	95
4.6.1	ProblemGraph	95
4.6.2	ProcessPatternGraph	96
5	Semantik.....	99
5.1	Syntaktische Regeln	99
5.2	Semantische Domänen	102
5.2.1	Grundlegende Konzepte	102
5.2.2	Petri-Netze und eB/E-Systeme	104
5.2.3	Petri-Netz-Operatoren	106
5.3	Verknüpfung von Syntax und Semantik	111
5.3.1	Bezeichner für syntaktische Elemente	112
5.3.2	Prozessmuskatalog	114
5.3.3	Komposition von Objekten und Ereignissen und Kontexten	114
5.3.4	Semantik von Aktivitätsdiagrammen	115
5.3.5	Semantik von PROPEL-Prozessen	120
5.3.6	Semantik von Kontexten	121
5.3.7	Semantik von Beziehungen	122
5.3.8	Berücksichtigung der Objekt- und Ereigniskomposition bei den Prozessmusterbeziehungen	128
6	Notation.....	133
6.1	Notationselemente zur Darstellung einzelner Prozessmuster	134
6.1.1	Prozessmuster	134
6.1.2	Problem	134
6.1.3	Objekte und Ereignisse	135
6.1.4	Spezialisierung/Generalisierung von Objekten und Ereignissen	135
6.1.5	Problemdiagramm	136
6.1.6	Kontext	137
6.1.7	Action State	137
6.1.8	Rolle	138
6.1.9	Werkzeug	138
6.1.10	Prozessdiagramm	139
6.2	Notationselemente zur Darstellung von Prozessmusterbeziehungen	140
6.2.1	Sequence	140
6.2.2	Use	141
6.2.3	Problemverfeinerungsdiagramm	142
6.2.4	Refinement	142
6.2.5	Processvariance	143
6.2.6	Beziehungsdiagramm	143
6.2.7	Prozessmusterdiagramm	144
6.2.8	Katalogdiagramm	146

7	Der Prozessmusterkatalog CADs	147
7.1	Ableitung des Prozessmusterkatalogs	147
7.2	Sichten des Prozessmusterkatalogs	148
7.2.1	Sicht "Use-Beziehung"	149
7.2.2	Sicht „Sequence-Beziehung“	150
7.2.3	Sicht „Refinement-Beziehung“	151
7.2.4	Sicht „Processvariance-Beziehung“	152
7.3	Objekt- und Ereignisstruktur	153
7.4	Probleme und Prozessmuster	154
7.4.1	Problem „How to elicit the Requirements for an existing system?“	154
7.4.2	Problem „How to refine the Problem?“	157
7.4.3	Problem „How to manage the requirements?“	158
7.4.4	Problem „How can Actors and Use-Cases be found?“	159
7.4.5	Problem „How can a Glossary be produced?“	161
7.4.6	Problem „How can a Glossary be produced (detailed)?“	163
7.4.7	Problem „How to envision the system?“	165
7.4.8	Problem „ How to review Requirements?“	167
7.5	Anwendung des Prozessmusterkatalogs	169
7.6	Zusammenfassung	170
8	Die Process Pattern Workbench	173
8.1	Anforderungsspezifikation	173
8.1.1	Funktionale Anforderungen	173
8.1.2	Nicht-funktionale Anforderungen	180
8.2	Entwurf	181
8.3	Implementierung	182
8.3.1	Das Menü	183
8.3.2	Der Menüpunkt Catalog	183
8.3.3	Der Menüpunkt Problem	185
8.3.4	Der Menüpunkt Pattern	187
8.3.5	Der Menüpunkt Project	191
8.4	Erweiterungsmöglichkeiten	194
8.4.1	Benutzerverwaltung	194
8.4.2	Katalogverwaltung	195
8.4.3	Problemdefinition	195
8.4.4	Patterndefinition	196
8.4.5	Reporting	197
8.4.6	Projektverwaltung	198

8.5	Zusammenfassung	199
9	Process Pattern Management	201
9.1	Infrastruktur	201
9.2	Aufbauorganisation	202
9.3	Einführung und Etablierung	202
9.4	Einsatz	203
9.4.1	Aktivitäten des Pattern Designers	204
9.4.2	Aktivitäten des Pattern Users	205
10	Fazit und Ausblick.....	207
10.1	Zusammenfassung	207
10.2	Bewertung	208
10.3	Ausblick	210
Anhang	213
A	Ergänzungen zu den Grundlagen	215
A.1	Muster	215
A.1.1	Musterschemata	215
A.1.2	Beziehungen zwischen Mustern	219
A.1.3	Gütekriterien	221
A.2	Grundbegriffe der Softwareprozessmodellierung	222
A.3	UML	223
A.3.1	UML-Sichten, -Diagramme und -Konzepte	223
A.3.2	UML-Pakete	224
A.3.3	Syntax und Semantik des UML-Metamodells	226
A.3.4	UML 2.0	228
A.4	Vorgehensmodelle	229
A.4.1	Rational Unified Process	229
A.4.2	Crystal Methodologies	232
B	Details zur Semantik	233
B.1	Syntaktische Domänen	233

B.2 Semantische Funktionen	234
C Implementierungsdetails	237
Literatur	239
Abbildungsverzeichnis	251
Tabellenverzeichnis	255
Glossar	257
Sachindex	263
Autorenindex	267
Wissenschaftlicher Werdegang	269
Selbständigkeitserklärung	271

1 Einleitung

1.1 Motivation

In den Anfängen der Softwareentwicklung betrachtete man Software als das Produkt eines einmaligen und kreativen Prozesses. Dadurch wurde verhindert, dass man Softwarefehler auf Fehler im Softwareentwicklungsprozess zurückverfolgen konnte oder dass man den Prozess so ändern konnte, dass Fehler bei erneuter Ausführung des Prozesses nicht auftraten. Mittlerweile hat sich jedoch die Erkenntnis durchgesetzt, dass Softwareentwicklungsprozesse und Softwarequalität miteinander verknüpft sind, d.h. dass Softwareentwicklungsprozesse die Softwarequalität beeinflussen.

Softwareentwicklungsprozesse beeinflussen Softwarequalität

Das Interesse für Softwareprozesse und deren Qualität hat zu zwei verschiedenen Disziplinen, Prozessmodellierungssprachen und Vorgehensmodelle, geführt. Die eher in der akademischen Welt erforschten und genutzten Prozessmodellierungssprachen wie z.B. SPELL, SLANG und APPL/A lassen durch formale Beschreibungsmittel genaue Aussagen über Prozesse zu. Betrachtet man jedoch die Praxistauglichkeit und Handhabbarkeit dieser formalen Prozessmodellierungssprachen, kehrt sich dieser Vorteil in einen Nachteil um. Verlage konstatierte dazu in [Ver98b]: „Die verfügbaren PMLs (Process Modeling Languages) entstanden meist als Anwendungsfall von Programmiersprachenforschern oder Entwicklern von Softwareentwicklungsumgebungen und nicht als Lösung für die Entwickler von Vorgehensmodellen. Es fehlt eine umfassende Unterstützung für Autoren von Vorgehensmodellen, die ihre wesentlichen Probleme löst. Daher können die heute verfügbaren PLMs nicht als praxistauglich bezeichnet werden.“ Prozessmodellierungssprachen wie ereignisgesteuerte Prozessketten (EPK) oder die Unified Modeling Language (UML) sind auf den praktischen Einsatz fokussiert, aber weniger formal.

Prozessmodellierungssprachen: formal vs. informal

Die eher in der Unternehmenswelt eingesetzten Vorgehensmodelle wie z.B. das V-Modell des Bundes und der Rational Unified Process (RUP) oder auch individuelle Vorgehensmodelle repräsentieren eine Menge von Standardprozessen, die in einem Softwareunternehmen befolgt werden sollen und können. Diese Vorgehensmodelle benutzen eigene, eher informale Beschreibungsmittel und lassen die oben erwähnten Prozessmodellierungssprachen aufgrund ihrer mangelnden Praxistauglichkeit ungenutzt. Durch ihre informale Darstellung gehen den Vorgehensmodellen jedoch einige Informationen verloren. Auch hierzu konstatierte Verlage in [Ver98a]: „Die Verwendung einer [formalen] Sprache lenkt den Entwickler beim Schreiben und führt so zu einer systematischen Begutachtung des Vorgehensmodells. Jede Softwareprozessmodellierungssprache definiert, welche Informationsbausteine und Beziehungen zwischen diesen als wichtig erachtet werden. Entdeckt der Autor, dass diese Informationen und Beziehungen in seinem Vorgehensmodell nicht oder nur unzureichend berücksichtigt sind, beziehungsweise sich nur schwer identifizieren lassen, so scheint die Darstellung oder das Vorgehensmodell selbst verbesserungswürdig.“

Vorgehensmodelle: formal vs. informal

Die schwer wiegendste Kritik an Vorgehensmodellen betrifft jedoch deren Komplexität, Rigidität und der Hang zur Bürokratie. Wiemers bemerkte hierzu in [Wie01], S.145-146: „Einem Softwareentwickler das V-Modell des Bundes in die Hand zu drücken, kann nur zwei Gründe haben: Der Entwickler soll nichts mehr schaffen oder in eine offene Revolte geführt

Vorgehensmodelle: rigide vs. agil

werden. [...] Leider ist das [die Anwendung eines umfassenden Vorgehensmodells] bisher nur selten gelungen, da gerade die großen Vorgehensmodelle bei den Anwendern spontan zu totem Synapsenausfall führten“. Insbesondere die dynamische Anpassung eines Vorgehensmodells an die aktuelle Projektsituation erweist sich bei traditionellen Vorgehensmodellen als schwierig.

Agile Methoden als Alternative zu Vorgehensmodellen

Nur in mittleren bis größeren Unternehmen scheint der Aufwand für Einführung und Anwendung eines Vorgehensmodells den Zweck zu rechtfertigen. Ende der 90er Jahre wurden aus diesen Gründen agile Prozesse, Vorgehensweisen und Methoden propagiert, die eine Alternative zu den komplexen, standardisierten (auch „schwergewichtigen“) Vorgehensmodellen verkörpern. Anstattdessen erfolgte die Hinwendung zu einigen wenigen, einfachen Prozessen oder Methoden, die im Rahmen der Softwareentwicklung unterstützend eingesetzt werden. Agile Prozesse sollen soviel Unterstützung wie möglich und so viel Unterstützung wie nötig anbieten.

Agile Vorgehensmodelle durch Prozessmuster

In den letzten Jahren wurde das Interesse an Prozessmustern geweckt, die eine flexible Prozessunterstützung gewährleisten können. Ein Prozessmuster repräsentiert für ein bestimmtes Problem einen musterhaften Prozess. Durch Modellierung einzelner Prozessmuster und möglicher Verknüpfungen zwischen Prozessmustern kann ein Vorgehensmodell „bottom-up“ zusammengesetzt werden. Prozessmuster können aber auch dazu dienen, ein bestehendes Vorgehensmodell durch ihren modularen Charakter zu entflechten („top-down“). Je nach Erfordernis können Prozessmuster dynamisch ausgewählt und angewendet werden. D.h. zur Laufzeit eines Projekts kann ein Vorgehensmodell abgestimmt auf das Projekt erstellt werden. Auf sich ändernde Bedingungen kann durch Auswahl neuer oder anderer Prozessmuster reagiert werden. Hierdurch ergibt sich eine größtmögliche Flexibilität des Vorgehensmodells.

1.1.1 Prozessmuster – Chancen ...

Prozessmuster bieten folgende Vorteile:

Dokumentation erprobten Wissens

Prozessmuster dienen dazu, Wissen über Prozesse zu dokumentieren, welches sich bewährt hat [Stö00]. Diese Regel ist sowohl für die Autoren von Mustern als auch deren Anwender von Bedeutung. Während die Autoren nur nützliches Prozesswissen dokumentieren, können sich die Anwender darauf verlassen, dass der Prozess auch wirklich „funktioniert“ und damit wiederverwendbar ist.

Basis für Kommunikation

Prozessmuster dienen als Kommunikationsgrundlage für alle an einem Prozess beteiligten Personen [DGH02]. Prozessmuster sind hierfür besonders gut geeignet, da sie einen Prozess und das von ihm zu lösende Problem kompakt darstellen.

Abstraktion von Details

Prozessmuster lassen stets bestimmte Details unberücksichtigt. Hierdurch kann ein Prozessmuster in vielen Situationen eingesetzt („instanziiert“) werden. Einer Prozessmusterinstanz können also – wenn erforderlich – weitere Detailinformationen hinzugefügt werden.

Unterstützung sowohl rigider als auch flexibler Prozesse

Softwareprozesse müssen je nach Erfordernis sowohl rigide¹ als auch flexibel sein. Ein rigider Prozess ist dann erforderlich, wenn die Prozessverbindlichkeit, Prozessverbesserung und Prozessmessung im Vordergrund stehen. Dies wird insbesondere dann der Fall sein, wenn die Prozesse stabil sind, d.h. keine dynamischen Prozessveränderungen zu erwarten sind.

1. Rigide wird in [Lang] definiert als „unnachgiebig, streng, steif, starr“.

Gleichzeitig sollen Prozesse jedoch eine möglichst hohe Flexibilität aufweisen, d.h. Bürokratie und nicht situationsgerechte Vorgaben sollen möglichst vermieden werden. Die Prozesse traditioneller Vorgehensmodelle sind eher als rigide und ihre Anpassungsmöglichkeiten als sehr eingeschränkt einzustufen. Unkontrollierte Anpassungen zerstören zudem unter Umständen die Konsistenz der Prozesse. Wünschenswert ist dagegen die dynamische Anpassbarkeit von Prozessen.

Prozessmuster sind ein Mittel, um die Modellierung und Anwendung von sowohl rigiden als auch flexiblen Prozessen zu ermöglichen [BRS98]. Für erprobte, feststehende Prozesse können Prozessmuster, für Varianten dieser Prozesse können Prozessmustervarianten definiert werden. Je nach Erfordernis kann ein Prozessmuster ausgewählt und angewendet oder ein individueller Prozess durchgeführt werden.

1.1.2 ... und ihre Risiken

Obwohl wie oben dargestellt das Konzept der Prozessmuster sehr vielversprechend ist, ist es noch keineswegs ausgereift [Hag02]. Bisherige Arbeiten beschäftigen sich eher mit dem Auffinden neuer Prozessmuster statt mit der geeigneten Beschreibung. Dadurch ergeben sich einige Nachteile:

Es gibt mittlerweile eine Reihe von Beschreibungsschemata für Muster wie das GOF-Schema [GHJ96] oder das Alexanderian-Schema [AIS77]. Da aber jedes Beschreibungsschema von einer anderen Domäne herrührt (GOF aus der Software-Design-Domäne, Alexanderian aus der Architektur-Domäne), ist jedes Schema auch speziell auf diese Domäne abgestimmt. Trotzdem wird beispielsweise das Alexanderian-Schema für (Software-)Entwurfsprobleme verwendet oder das GOF-Schema für Prozesse. Für Autoren und Anwender von Mustern führt dies zu einigen Irritationen, da nicht klar ist, wie einige Musterelemente zu beschreiben und zu verstehen sind.

Prozessmuster werden nicht einheitlich beschrieben

Bisherige Prozessmuster vernachlässigen die Beschreibung ihrer Beziehungen zu anderen Prozessmustern. Prozessmuster können jedoch ihre volle Kraft erst im Zusammenspiel mit anderen Prozessmustern entfalten [Cop96]. Durch die Kombination einzelner Prozesse können somit komplexere Prozesse modelliert werden. Hierfür muss bekannt sein, ob zwei Prozessmuster überhaupt miteinander kombinierbar sind. Diese Information muss durch eine explizite Beschreibung der Prozessmusterbeziehungen mitgeliefert werden. Gegenwärtige Beschreibungen von Beziehungen sind jedoch nicht formal und geben keine Auskunft über ihre syntaktische oder semantische Bedeutung: Welche Schlussfolgerungen lassen sich z.B. aus den Aussagen „Pattern X uses Pattern Y“ und „Pattern A can be combined with pattern B“ ziehen? Unter welchen Bedingungen ist ein Prozessmuster die Variante eines anderen Prozessmusters? Unter welchen Bedingungen lassen sich zwei Prozessmuster nacheinander ausführen?

Prozessmuster werden nicht vollständig beschrieben

Muster – auch als literarisches Ausdrucksmittel bezeichnet [Cop96] – wurden bislang auf informale Weise durch natürliche Sprache beschrieben. Dieses Vorgehen ist einerseits ein Vorteil, da für das Verstehen eines Musters kein Wissen über Notation, Syntax und Semantik erforderlich ist. Durch die Verwendung natürlicher Sprache werden jedoch deren Ausdrücke mehrdeutig, da sie unpräzise sind (vgl. z.B. die Beziehungen „is prerequisite of“ und „may contain“ in [Stö00]). Neben der unpräzisen Beschreibung von Beziehungen sind erstaunlicherweise oftmals die Beschreibungen der Prozesse informal. Eine schrittweise Beschreibung des Prozesses fehlt oft, und es gibt sogar Prozessmuster, die gar keinen Prozess als

Prozessmuster werden nicht präzise genug beschrieben

Lösung enthalten [Cop94]. Mehrere Anwender eines Prozessmusters können also völlig unterschiedliche Schlüsse aus den Beschreibungen ziehen und gegebenenfalls falsche Entscheidungen treffen.

1.2 Zielsetzung der Arbeit

Zusammenfassend lässt sich folgendes feststellen: Vorgehensmodelle (Standard- oder Individualvorgehensmodelle) sind meistens rigide und schwer an dynamische Änderungen im Projekt anzupassen. Prozessmuster können hier Abhilfe schaffen, da sie es erlauben, Prozesse modular zu beschreiben und zu verknüpfen.

Trotz der oben genannten Nachteile von Prozessmustern formulieren wir die These, dass Prozessmuster ein geeignetes Mittel sind, um eine flexible Prozessunterstützung zu gewährleisten. Aus diesem Grund wollen wir in dieser Arbeit eine Beschreibungssprache entwickeln, die diese Nachteile beseitigt. Durch die Formalität der Beschreibungssprache können genauere Aussagen über die modellierten Prozesse getroffen werden, als es bei herkömmlichen Vorgehensmodellen der Fall ist. Trotzdem soll die Beschreibungssprache pragmatisch und verständlich bleiben, damit sie praktischen Nutzen hat.

Unser Forschungsziel wird folgendermaßen definiert:

Forschungsziel **Unser Ziel ist es, eine formale Beschreibungssprache zu entwickeln, mit deren Hilfe einzelne Prozessmuster und deren Beziehungen modelliert werden können. Die mit Hilfe der Beschreibungssprache modellierten Prozessmuster erlauben die Definition von Vorgehensmodellen. Diese modular beschriebenen Vorgehensmodelle können dynamisch an die jeweilige Projektsituation angepasst werden.**

Unser Forschungsziel kann durch eine Menge von Anforderungen weiter konkretisiert und überprüfbar gemacht werden. Nach dem Vorschlag von Osterweil in [SO97] unterscheiden wir zwischen Anforderungen an Prozesssprache und Prozess, d.h. zwischen Anforderungen an die Sprache, die zur Beschreibung von Prozessmustern dient, und Anforderungen an Prozessmuster. Darüber hinaus definieren wir Anforderungen an die Handhabung von Prozessmustern.

Anforderung 1 **Leichte Handhabbarkeit und Intuitivität der Beschreibungssprache**

Die Beschreibungssprache soll leicht zu erlernen und zu benutzen sein. Der Anwender soll möglichst schnell mit den Konzepten der Beschreibungssprache vertraut sein und diese schnell und ohne großen Aufwand einsetzen können. Die Beschreibungssprache soll ferner eine geeignete Notation zur Darstellung der Semantik enthalten.

Anforderung 2 **Definition einer formalen Syntax und Semantik der Beschreibungssprache**

Zur Zeit existieren nur wenige Ansätze zur Formalisierung von Prozessmustern, insbesondere der Prozessmusterbeziehungen (Abschnitt 2.1.8). Die Anforderung ist daher die Identifikation und formale Definition von Prozessmusterbeziehungen.

Definition eines Beschreibungsschemas für Prozessmuster

Anforderung 3

Existierende Beschreibungsschemata sind zum Großteil für implementierungsnahe Muster konzipiert. Eine Prozessbeschreibung unterliegt jedoch anderen Anforderungen als eine Entwurfsbeschreibung (Abschnitt 2.1.4). Prozesse müssen als eine Abfolge von Aktivitäten beschrieben werden. Dabei werden Ressourcen (menschliche oder Maschinen), Methoden und Werkzeuge eingesetzt. Aktivitäten führen zu Zuständen, produzieren Ergebnisse und werden von Rollen durchgeführt. Es muss ferner explizit definiert werden, wie der Kontext eines Prozessmusters beschaffen ist und welche Regeln erfüllt sein müssen, damit zwei Prozessmuster zueinander in Beziehung stehen können. Wir benötigen also ein Beschreibungsschema, welches speziell auf die Bedürfnisse zur Beschreibung von Prozessmustern und deren Prozesse ausgerichtet ist (Abschnitt 3.4).

Begriffsbildung

Anforderung 4

Z. Zt. existiert eine Vielzahl unterschiedlicher und unpräziser Definitionen für Begriffe wie z.B. Prozessmuster (Abschnitt 2.1.2), Pattern Language (Abschnitt 2.1.7) usw. Hierdurch werden die Begriffe unterschiedlich interpretiert und unterschiedlich genutzt. Die Anforderung besteht folglich in der Definition aller relevanten Begriffe (Kapitel 3).

Definition eines Klassifizierungsschemas für Prozessmuster

Anforderung 5

Zur Zeit existiert kein einheitlicher Klassifizierungsansatz für Prozessmuster (Abschnitt 2.1.6). Dies bedeutet, dass die Chance zur übersichtlichen Einordnung von einer Menge von Mustern nicht wahrgenommen wird. Die Anforderung besteht folglich in der Erarbeitung eines umfassenden Klassifizierungsschemas für Prozessmuster, das für die einheitliche Klassifizierung von Prozessmustern genutzt werden kann.

Entwicklung eines Werkzeugs zur Handhabung von Prozessmustern

Anforderung 6

Die Anzahl dokumentierter Muster geht mittlerweile in die Tausende. Dies führt unweigerlich zu einer großen Unübersichtlichkeit, insbesondere dann, wenn man Beziehungen zu anderen Mustern aufzeigen will. Aus diesem Grund ist eine Werkzeugunterstützung zur Dokumentation, Suche, Modifikation und Anwendung von Prozessmustern wünschenswert. Desweiteren soll das Werkzeug genutzt werden, um die Handhabbarkeit der entwickelten Sprache zu überprüfen. Die Anforderung besteht folglich in der Entwicklung eines Werkzeug zur Handhabung von Prozessmustern.

Entwicklung einer Methodik zur Handhabung von Prozessmustern

Anforderung 7

Methoden zur Handhabung von Prozessmustern und Mustern im Allgemeinen sind derzeit nur ansatzweise vorhanden (Abschnitt 2.1.9). Die meisten Ansätze beziehen sich auf die Identifikation von Mustern (sogenanntes „Pattern Mining“, s. z.B. [Ris99]). Wie Muster jedoch geeignet modelliert, dokumentiert, gesucht und angewendet werden, ist derzeit noch ungeklärt. Diese Lücke soll der Ansatz zum Process Pattern Management schließen. Die Anforderung besteht folglich in der Entwicklung eines Konzepts für ein systematisches Process Pattern Management.

1.3 Lösungsansatz

In diesem Abschnitt skizzieren wir den von uns gewählten Lösungsweg.

Kompakte Darstellung der wichtigsten Grundlagen	Zunächst stellen wir die für das Verständnis der Arbeit wichtigsten Grundlagen dar. Hierzu gehört eine Darstellung und Bewertung der relevanten Arbeiten im Muster-Bereich. Ferner geben wir einen Abriss über das Feld der Softwareprozessmodellierungssprachen und bewerten sie hinsichtlich der Einsetzbarkeit für unsere Ziele. Desweiteren betrachten und bewerten wir bisherige Ansätze zur Formalisierung von UML-Aktivitätsdiagrammen. Schließlich diskutieren wir zwei verschiedene Ausprägungen – rigide bis agil – von Vorgehensmodellen in der Softwareentwicklung und wägen deren Vor- und Nachteile gegeneinander ab.
Entwicklung einer Beschreibungssprache für Prozessmuster	Wir diskutieren und klären begrifflich die Konzepte wie z.B. Prozessmuster, die der Beschreibungssprache zugrunde liegen. Dies ist notwendig, da noch keine einheitlichen Begriffsdefinitionen vorliegen. Auf Basis dieser Konzepte definieren wir dann die Sprache PROPEL (<u>P</u> rocess <u>P</u> attern <u>D</u> escription <u>L</u> anguage). Für die Definition der Syntax, Semantik und Notation greifen wir auf eine vorhandene Sprache zurück, die Unified Modeling Language (UML) [UML1.5]. Die UML gilt als „Lingua Franca“ der Software-Engineering-Community, d.h. sie ist als einheitliches Modellierungs- und Kommunikationsmittel anerkannt. Diesen Vorteil wollen wir für unsere Beschreibungssprache nutzen. Die UML erweitern wir um syntaktische, semantische und notationale Elemente. Da die UML keine formale Semantik besitzt, ergänzen wir die Sprache PROPEL um eine formale Semantik durch Abbildung der Syntax auf die semantische Domäne der Petri-Netze.
Entwurf eines Prozessmusterkatalogs	Auf Basis von PROPEL entwickeln wir einen Prozessmusterkatalog, welcher dem Leser die Konzepte der Beschreibungssprache an einem komplexen, praxisnahen Beispiel präsentiert. Hierzu verwenden wir einen Ausschnitt aus dem RUP, den wir mit Hilfe von Prozessmustern beschreiben. Durch den Vergleich der Beschreibungselemente im RUP und der Beschreibungselemente von PROPEL wird eine Bewertung von PROPEL vorgenommen.
Entwicklung der Process Pattern Workbench	Die Process Pattern Workbench ist eine prototypische Realisierung der Konzepte von PROPEL. Mit Hilfe der Process Pattern Workbench können Prozessmuster dokumentiert, präsentiert, gesucht, modifiziert und deren Anwendung aufgezeichnet werden. Der als Beispiel erarbeitete Prozessmusterkatalog wird mit Hilfe der Process Pattern Workbench umgesetzt. Durch die Process Pattern Workbench lassen sich die an die Beschreibungssprache gesetzten Anforderungen überprüfen.
Entwicklung des Process Pattern Managements	Auch wenn eine geeignete Sprache zur Beschreibung von Prozessmustern zur Verfügung steht, ist noch unklar, wie der eigentliche Einsatz von Prozessmustern in der Praxis abläuft. Wer schon einmal mit der Einführung von Vorgehensmodellen beauftragt war, weiß um die Schwierigkeit dieser Aufgabe. Beim Process Pattern Management geht es deshalb darum zu klären, welche Maßnahmen ein Unternehmen ergreifen muss, wenn es Prozessmuster einsetzen will. Das Process Pattern Management beschreibt Aktivitäten zum Aufbau einer geeigneten Prozessmuster-Infrastruktur (z.B. ein elektronischer Prozessmusterkatalog), zur organisatorischen Einbettung, der Einführung und Etablierung von Prozessmustern und dem letztendlichen Einsatz von Prozessmustern.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist in folgende Kapitel gegliedert (Abbildung 1-1): Nach dem ersten zielsetzenden Kapitel werden in Kapitel 2 einige der Grundlagen vorgestellt, die der Leser für das Verständnis des bearbeiteten Themas benötigt. PROPEL wird in Kapitel 3 bis 6 vollständig beschrieben: In Kapitel 3 werden die logischen Konzepte der Process Pattern Description Language PROPEL vorgestellt. Anschließend werden in Kapitel 4 diese Konzepte syntaktisch formal als UML-Erweiterung definiert. Desweiteren wird die Semantik – analog zur UML-Spezifikation – informal erläutert. Die Semantik der neu eingeführten Konzepte wird im darauffolgenden Kapitel 5 formal definiert. Die zu PROPEL gehörende Notation wird in Kapitel 6 eingeführt. In Kapitel 7 geben wir einen Prozessmusterkatalog an, der die Anwendung von PROPEL durch eine Menge modellierter Prozessmuster zeigt. Dieses Kapitel dient zudem der Validierung von PROPEL an einem ausführlichen Beispiel. In Kapitel 8 stellen wir die Process Pattern Workbench vor, die eine Plattform zur Modellierung und Anwendung von Prozessmustern ist. Im Rahmen der Process Pattern Workbench wurde ein Großteil der Elemente von PROPEL implementiert. In Kapitel 9 führen wir den Begriff des Process Pattern Managements ein, einen Ansatz zur effektiven und systematischen Behandlung von Prozessmustern. In Kapitel 10 schließlich fassen wir die Konzepte und Ergebnisse dieser Arbeit zusammen, bewerten diese und geben einen Ausblick auf zukünftige Arbeiten.

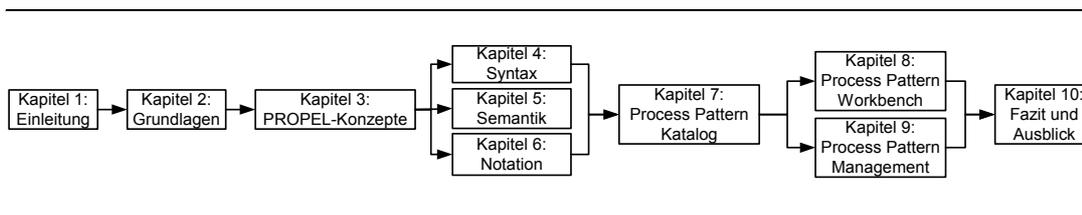


Abbildung 1-1: Aufbau der Arbeit

1.5 Hinweise für den Leser

1.5.1 Voraussetzungen

Die vorliegende Arbeit setzt ein solides Basiswissen in der Softwaretechnik voraus. Über den Bereich der Muster geben wir, da in der Literatur noch nicht vorhanden, einen breiten Überblick. Im Bereich der Softwareprozessmodellierungssprachen, der UML-Semantik und der Vorgehensmodelle geben wir nur einen kurzen Überblick, der Leser kann in den gegebenen Referenzen nachschlagen. Der Leser benötigt indes Basiswissen über das UML-Metamodel. Weitere Details sind in den Anhängen zu den Grundlagen (Anhang A), zur Semantik von PROPEL (Anhang B) und zu Implementierungsdetails (Anhang C) erläutert. Auf weiterführende Literatur verweisen wir in den entsprechenden Abschnitten in Kapitel 2.

1.5.2 Typographische Konventionen

Im Text benutzen wir fast ausschließlich die deutsche Übersetzung englischer Begriffe (z.B. Prozessmuster für Process Pattern). Für das PROPEL-Metamodell verwenden wir englischsprachige Begriffe, da diese sich so besser in das bestehende UML-Metamodell (ebenfalls in englisch) einpassen.

Von uns verwendete Abkürzungen und Akronyme werden bei ihrem ersten Auftreten erläutert. Sie werden zudem im Glossar aufgeführt und im Sachindex referenziert.

Bei der Beschreibung der UML-Erweiterungen in den Kapiteln 4, 5 und 6 verwenden wir die typographischen Konventionen, wie sie in der UML-Spezifikation [UML1.5] verwendet werden.

2 Grundlagen

In diesem Kapitel stellen wir diejenigen Grundlagen vor, die zum Verständnis der Process Pattern Description Language PROPEL von Bedeutung sind. Dazu werden die für diese Arbeit relevanten Aspekte aus den Bereichen Muster und Prozessmuster (Abschnitt 2.1), Softwareprozessmodellierungssprachen (Abschnitt 2.2), Semantik von UML-Aktivitätsdiagrammen (Abschnitt 2.3) und Vorgehensmodelle in der Softwareentwicklung – von rigide bis agil – (Abschnitt 2.4) erläutert.

2.1 Muster und Prozessmuster

Da der Begriff der Prozessmuster recht jung ist, skizzieren wir zunächst die Historie von (Prozess-) Mustern (Abschnitt 2.1.1). Hiernach folgt eine Erläuterung von den Grundbegriffen (Abschnitt 2.1.2), die Abgrenzung zu Workflowmustern (Abschnitt 2.1.3) und die Vorstellung bekannter Musterschemata (Abschnitt 2.1.4). Desweiteren erläutern wir Ansätze zur Formalisierung von Mustern (Abschnitt 2.1.5), zur Klassifizierung von Mustern (Abschnitt 2.1.6), zur Organisation von Mustern (Abschnitt 2.1.7), zur Definition von Musterbeziehungen (Abschnitt 2.1.8) und zum Management von Mustern (Abschnitt 2.1.9). In jedem dieser Abschnitte erfolgt eine Bewertung der bestehenden Ansätze und die Hinführung auf die in dieser Arbeit neu entwickelten Ansätze.

2.1.1 Historie

Der Architekt Christopher Alexander erkannte in den 70er Jahren, dass Gebäude auf wiederkehrende Architekturmuster schließen lassen. Er sammelte und veröffentlichte diese in dem Buch „A Pattern Language“ [AIS77]. Mit diesen Mustern soll jeder Mensch auch ohne Architekturkenntnisse in die Lage versetzt werden, ein Gebäude nach seinen Vorstellungen und Bedürfnissen zu entwerfen.

Alexander

Das Interesse für Entwurfsmuster für Software wurde – in Anlehnung an die Muster von Alexander – in den 80er Jahren geweckt. Zu diesem Zeitpunkt wurde deutlich, dass die objektorientierte Softwareentwicklung und insbesondere der objektorientierte Entwurf durch die Identifikation und Wiederverwendung wiederkehrender Muster unterstützt werden kann. Erste Ansätze und Ideen wurden zu dieser Zeit auf objektorientierten Konferenzen wie OOPSLA und ECOOP und in objektorientierten Büchern und Aufsätzen präsentiert (z.B. [Bec88], [Cun88], [GHJ93], [Coa92], [Cop92]). 1993 wurde in den Bergen von Colorado ein Treffen von Praktikern und Wissenschaftlern einberufen, um Erfahrungen bzgl. objekt-orientierter Muster auszutauschen. Seit diesem Treffen nennt sich diese Gruppe „The Hillside Group“ [Hillside]. 1994 fand schließlich die erste Muster-Konferenz „Pattern Languages on Program Design (PloP)“ statt [CS95].

Entwurfsmuster

Mittlerweile gibt es eine Vielzahl von regional beschränkten Muster-Konferenzen wie EuroPloP, Sugarloaf, KoalaPloP, MensorePloP, WikiPloP usw., die sich mit dem Thema Softwaremuster befassen. Als Meilenstein der Muster-Bewegung kann das Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ [GHJ96] der "Gang-of-Four"

Der Meilenstein

(GoF) aus dem Jahr 1995 betrachtet werden, das in Form eines Muster-Katalogs 23 erprobte Entwurfsmuster aufführt. Mittlerweile gibt es neben den Entwurfsmustern weitere Typen wie Prozessmuster, Organisationsmuster, Projektmanagementmuster, Philosophiemuster usw.

2.1.2 Der Begriff des Musters

Muster dienen dazu, bestimmte Probleme, die unter bestimmten Bedingungen auftreten, zu lösen. Das Ziel von Mustern ist es, das Leben von Menschen zu erleichtern, indem sie mit Musterlösungen für bestimmte Probleme ausgestattet werden. Hierdurch können Probleme schneller und richtiger gelöst werden. Zunächst präsentieren wir drei gängige Musterdefinitionen:

Musterdefinitionen

Der Architekt Christopher Alexander definiert ein Architekturmuster folgendermaßen ([Ale79], S.247): *"Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution."*

Gamma et al. betrachten Entwurfsmuster als *„Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen“* [GHJ96].

Gnatz et al. definierten Prozessmuster in [GMP01a] folgendermaßen: *„A Process Pattern defines a general solution to a certain recurring problem. The problem mentions a concrete situation that may arise during the system development. It mentions internal and external forces, namely influences of customers, competitors, component vendors, time and money constraints and requirements. A process pattern suggests an execution of a possibly temporally ordered set of activities.“* Weitere ähnliche Definitionen des Begriffs Prozessmuster finden sich in [Stö00], [FE03], [BRS98].

Alle Musterdefinitionen weisen übereinstimmend die folgenden drei Elemente auf:

- Problem
 - Das Problem eines Musters ist wiederkehrender Natur und wird von dem Muster gelöst. Probleme geben dem Anwender darüber Aufschluss, ob das Muster für seine Belange geeignet ist. Probleme sind deswegen oft der erste Anhaltspunkt bei der Suche nach Mustern. Einige Muster-Schemata wie z.B. das Coplien-Schema (Abschnitt 2.1.4) formulieren das Problem als Frage. Andere Schemata beschreiben das Problem ausführlicher, manchmal wie bei Alexander auch zusammen mit den „Forces“ (zu „Forces“ s. weiter unten bei „Kontext“). Das Ziel von Mustern ist, Wissen zu dokumentieren, das sich bewährt hat und das häufig angewendet werden kann. Dagegen lohnt es nicht, Wissen zu dokumentieren, das nur selten angewendet werden kann. In solchen Fällen sind individuelle Lösungen anzuwenden. Entsprechend formulierte Coplien in [Cop94] die „Rule of Three“, die fordert, dass ein Muster seine Nützlichkeit und Richtigkeit mindestens dreimal beweisen muss.
- Kontext
 - Ein Kontext beschreibt Vor- und Nachbedingungen eines Musters. Die Vorbedingungen bestimmen, ob ein Muster angewendet werden kann, die Nachbedingungen beschreiben den Zustand (eines Gebäudes, eines Projekts etc.) nach Anwendung des Musters. Innerhalb des Kontextes treten Bedingungen (sog. „Forces“) auf, die sich gegenseitig ausschließen und die es auszubalancieren gilt.

- Die Lösung, die das Problem löst und die „Forces“ ausbalanciert, muss sich bewährt haben. Die Lösung kann eine Bauskizze sein, ein Klassenmodell, ein Prozess usw.

Lösung

Mittlerweile gibt es eine Vielzahl von Mustertypen. Die meisten existierenden Muster sind Entwurfsmuster. Entwurfsmuster beschreiben für ein Problem im Rahmen des Softwareentwurfs eine Lösung in Gestalt eines Entwurfs. Verwandte Typen sind die Architekturmuster und Sourcecodemuster (sog. Idiome). Architekturmuster, Entwurfsmuster und Sourcecodemuster werden von einigen Autoren auch unter dem allgemeinen Begriff der Entwurfsmuster zusammengefasst, wobei Architekturmuster die abstrakteste und Idiome die detaillierteste Ausprägung von Entwurfsmustern darstellen (z.B. in [BMR96]). Weitere Mustertypen sind Philosophiemuster [Aue98], Organisationsmuster ([Cop94], [Har95]), Pädagogikmuster [Ped] und Projektmanagementmuster ([PMP], [Wel95]).

Mustertypen

Prozessmuster beschreiben in der Regel Lösungen für Probleme in der Softwareentwicklung in Gestalt von Prozessen. In der Literatur sind erst einige wenige Prozessmuster bekannt (z.B. [Amb98], [Amb99], [BRS98], [Stö00], [FE03]).

Der Typ „Prozessmuster“

Alle existierenden Definitionen enthalten als wesentliche Elemente Problem, Kontext und Lösung. Sie unterscheiden sich z.B. in der Art der Lösung (z.B. Architekturentwurf, Softwareentwurf, Codemuster, Prozess, Ratschläge) und enthalten weitere spezifische Elemente (Abbildung (Alexanderian-Schema), Klassifizierung (GoF, Störrle), Schlagwörter (Gnatz et al.) usw.). Die mangelnde Einheitlichkeit ist jedoch für die Wiederverwendung von Mustern und für die Komposition mehrerer Muster problematisch. Denn dadurch wird ein Vergleich zwischen mehreren Mustern erschwert. Desweiteren wird die Komposition gegebenenfalls unmöglich, falls bestimmte Musterelemente fehlen (z.B. das Element „Verwandte Muster“ bei Coplien), die für die Komposition notwendig sind. Allen Definitionen ist gemein, dass sie keine formale Syntax oder Semantik besitzen. Auch hierdurch wird die Komposition einzelner Muster erschwert, da beispielsweise über Relationen keine weitreichenden Aussagen getroffen werden können. Aus diesem Grund und weil die Muster-Definition das wesentliche Konzept zur Beschreibung von Mustern ist, haben wir unserer Process Pattern Description Language PROPEL eine formale Musterdefinition zugrunde gelegt (Kapitel 4).

Fazit

2.1.3 Workflowmuster

Prozessmuster werden häufig mit Workflowmustern [AHK03] von van der Aalst verwechselt. Workflowmuster dienen dazu, Workflowmanagementsysteme vergleichen und bewerten zu können. Die Workflowmuster beschreiben zu diesem Zweck Entwurfs- oder Implementierungsmechanismen (z.B. Synchronisation von Kontrollflüssen), die Workflowmanagementsysteme aufweisen (können). Zu den Workflowmustern zählen z.B. einfache Kontrollstrukturen – sogenannte „basic controls“ – wie „Sequence“, „Parallel Split“, „Synchronization“, „Exclusive Choice“ und „Simple Merge“ und komplexere Muster wie „Synchronising Merge“.

Abgrenzung zwischen Prozessmustern und Workflowmustern

Die Betrachtung von Workflowmanagementsystemen und technischen Implementierungsdetails beweist die unterschiedlichen Ziele von Workflowmustern und Prozessmustern. Prozessmuster dienen zur Beschreibung beliebiger Geschäftsprozesse unabhängig vom eingesetzten System. Workflowmuster enthalten dagegen Konzepte für den Entwurf, die Implementierung und den Vergleich von Workflowmanagementsystemen. Diese Konzepte werden zudem nicht als Prozess dargestellt.

Darüber hinaus sind die Workflowmuster sehr implementierungs- und systemnah. Im Lösungsteil der Workflowmuster wird beispielsweise erläutert, welche Workflowmanagementsysteme Muster auf welche Weise umsetzen. Ferner stehen die Formalität von und die Beziehungen zwischen Workflowmustern nicht im Vordergrund wie bei Prozessmustern.

2.1.4 Der Begriff des Musterschemas

Schema als Strukturierungsmittel	Ein Musterschema gibt die Struktur von Mustern eines bestimmten Typs vor. D.h. es wird vorgegeben, mit welchen Elementen ein Muster zu beschreiben ist. Die Beschreibung eines Musters unterscheidet sich also je nach Typ des Musters. Deswegen sind mittlerweile verschiedene Muster-Schemata verfügbar.
Vorstellung Vergleich verschiedener Schemata	Nachfolgend werden häufig verwendete ² Musterschemata vorgestellt und anschließend in Tabelle 2-1 verglichen. Anhang A.1.1 beschreibt die einzelnen Schemata im Detail. Die meisten Schemata leiten sich von dem Alexanderian-Schema oder dem GoF-Schema ab. Sie werden deswegen hier nicht weiter behandelt. Hierzu zählen das Portland-Schema [Portland] und das Riehle-Schema [Rie96] für den Softwareentwurf, ein Schema von Cockburn für das Projektmanagement [PPP], von Beck für SmallTalk [Bec96] und von Fowler für die objektorientierte Analyse [Fow96] und Enterprise Application Architekturen [FRF02].
Alexandersches Schema	Das Alexandersche Schema ist die ursprüngliche Form aller Muster. Sie wurde von Alexander zur Beschreibung von wiederkehrenden Gebäude-Architekturen verwendet [AIS77].
Coplien-Schema	Das Coplien-Schema ist dem Alexanderschen Schema entlehnt, dient aber zur Beschreibung von Entwurfsmustern, Organisationsmustern oder Prozessmustern. Das Schema unterscheidet verschiedene Elemente eines Musters genauer als das Alexandersche Schema: Der Abschnitt „Einführung“ heißt hier „Kontext“. Um zu verdeutlichen, dass sich durch Anwendung eines Musters der Kontext ändert, wird der Abschnitt „Resultierender Kontext“ eingeführt. Coplien unterscheidet wie Alexander auch zwischen „Problemüberschrift“ und „Problembeschreibung“, nennt diese Abschnitte jedoch „Problem“ und „Pro und Contra“ (im engl. „Forces“). Der Abschnitt „Grundprinzip“ (engl. „Rationale“) ist neu hinzugefügt.
GoF-Schema	Das GoF-Schema wurde in [GHJ96] als Standard für Entwurfsmuster etabliert. Ein etwas abgewandeltes Schema findet man in [BMR96]. Das der Lösung entsprechende Element heißt hier Struktur, da die Struktur des Entwurfs, meistens in Form von Klassendiagrammen, dargestellt wird. Desweiteren gibt es entwurfsspezifische Beschreibungselemente wie Teilnehmer, Interaktion, Implementierung und Beispielcode.
Störle-Schema	Das Störle-Schema in [Stö00] ist neben dem ZEN-Schema eines der wenigen Schemata zur Beschreibung von Prozessmustern. Das Beschreibungselement „Lösung“ heißt hier „Prozess“. „Teilnehmer“ sind statt Objekte und Klassen (vgl. GoF-Schema) alle an einem Prozess teilnehmenden Entitäten. Die Elemente „Anwendbarkeit“ und „Ergebnisse“ entsprechen dem Element „Kontext“, also den Vor- und Nachbedingungen, die bei Anwendung des Musters gelten. Weder diese beiden Elemente noch das Element „Prozess“ werden formal definiert.

2. zur Häufigkeit der Verwendung vgl. [Czi01]

Das ZEN-Schema (von uns so vereinfacht genannt nach dem Forschungsprojekt ZEN, in dessen Rahmen dieses Schema entstanden ist) dient zur Beschreibung von Prozessmustern [GMP01b]. Es enthält ein zusätzliches Beschreibungselement „Realisierte Aktivität“. Eine Aktivität kann durch mehrere Prozessmuster gelöst („realisiert“) werden. Eine solche Aktivität befindet sich also auf einer höheren Abstraktionsebene als das lösende Prozessmuster.

ZEN-Schema

Entwurfsmuster			Prozessmuster	
Alexan. Schema	Coplien-Schema	GoF-Schema	Störle-Schema	ZEN-Schema
Name	Name	Mustername	Name	Name
-	-	-	-	Autor
-	-	-	-	Version
Abbildung	-	-	-	-
-	-	Klassifizierung	Klassifizierung	-
Problem	Problem/Pro und Kontra	Zweck	Zweck	Problem/Pro und Kontra
-	-	Auch bekannt als	Auch bekannt als	Auch bekannt als
-	-	-	-	Schlagwörter
Einführung	Kontext	Motivation	Motivation	Zweck
Einführung	Kontext	Anwendbarkeit	Anwendbarkeit	Initialer Kontext
Lösung	Lösung	Struktur	Prozess	Lösung
-	-	Teilnehmer	Teilnehmer	-
-	-	Interaktionen	-	-
-	-	-	-	Realisierte Aktivität
-	Resultierender Kontext	Konsequenzen	Konsequenzen	Resultierender Kontext
Diagramm	-	-	-	-
-	-	Implementierung	Implementierung	-
-	-	Beispielcode	Beispielausführung	-
-	-	Bekannte Verwendungen	Bekannte Verwendungen	-
-	Grundprinzip	-	-	Pro und Kontra
Benutzte Muster	-	Verwandte Muster	Verwandte Muster	Verwandte Muster

Tabelle 2-1: Vergleich verschiedener Musterschemata

Fazit Bekannte Beschreibungsschemata wie das GoF-Schema sind zum Großteil für implementierungsnahe Muster (d.h. Architektur-, Entwurfs- und Implementationsmuster) ausgelegt. GoF-Muster beschreiben also Struktur, Verhalten oder Erzeugung von Klassen oder Objekten. Das Beschreibungsschema weist aus diesen Gründen Elemente wie „Teilnehmer“ und „Interaktionen“ auf, um involvierte Klassen und Objekte angeben zu können. Trotzdem greifen einige Autoren (wie z.B. Störrle und die ZEN-Gruppe) auf die GoF-Definition zurück, um beispielsweise Prozessmuster zu beschreiben (vgl. [BRS98], [Stö00]). Eine Prozessbeschreibung unterliegt jedoch anderen Anforderungen als eine Entwurfsbeschreibung: Mechanismen zur Modellierung von Prozessen und deren Input und Output sind notwendig. Desweiteren sind Mechanismen notwendig, um zu beschreiben, wie Prozesse miteinander verknüpft werden können. Diesen Anforderungen muss ein Prozessmuster-Beschreibungsschema Rechnung tragen. Neben der Entwicklung einer eigenständigen, klar abgegrenzten Terminologie haben wir deshalb in Abschnitt 3.4 ein Schema zur Beschreibung von Prozessmustern definiert.

2.1.5 Formalisierung von Mustern

Nachteile der Informalität Muster – auch als literarisches Ausdrucksmittel bezeichnet [Cop96] – wurden bislang auf informale Weise durch natürliche Sprache beschrieben. Dieses Vorgehen ist einerseits ein Vorteil, da für das Verstehen eines Musters kein Wissen über Notation, Syntax und Semantik erforderlich ist. Natürliche Sprache ist andererseits ein informales Ausdrucksmittel. Durch die Verwendung natürlicher Sprache werden ihre Ausdrücke mehrdeutig, da sie unpräzise sind (vgl. z.B. die Beziehungen „is prerequisite of“ und „may contain“ in [Stö00], s. Abbildung 2-1).

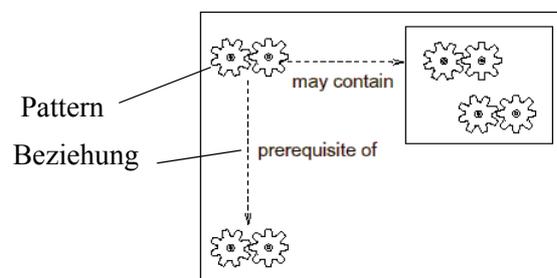


Abbildung 2-1: Prozessmuster-Beziehungen bei Störrle

Präzise Beschreibung ist notwendig Die präzise Beschreibung von Mustern ist jedoch notwendig, um ein Muster verstehen und Aussagen über Eigenschaften und Beziehungen zu anderen Mustern treffen zu können. Eden untersuchte in [Ede97] die Formalität und Eindeutigkeit der GoF-Entwurfsmuster und konstatierte folgendes:

“To summarize, coherent specifications of patterns are essential to improve their comprehension, to allow formal reasoning about their properties and relationship, and to support and automate their application.”

„Quality without a name“ verhindert Präzision Eden bewertet bisherige Musterbeschreibungen als ein vages Konzept, das schwer fassbar ist und nur schwer wörtlich ausgedrückt werden kann. Das Attribut „quality without a name“, welches für die nicht explizierbare und daher nicht analysierbare Güte eines Musters steht,

ist daher eine Glaubensfrage und kann nicht mit formalen Mitteln ausgedrückt werden. Die Qualität der Entwurfsmuster kann laut Eden allerdings durch Transformation in eine formale Spezifikationssprache verbessert werden. Mit LePUS (Language for Pattern Uniform Specification) hat Eden eine solche formale Spezifikationssprache für Entwurfsmuster geschaffen [EHL99]. Entwurfsmuster werden in LePUS-Prädikate übersetzt. Aus den LePUS-Spezifikationen kann dann der entsprechende Quellcode abgeleitet werden. Es wird allerdings nur ein Beispiel für eine PROLOG-Implementierung angegeben. Eine Validierung erfolgt dann durch Vergleich des Quellcodes mit den Spezifikationen. Z.B. ist eine Spezifikation des Begriffs „Verfeinerung“ durch die Prädikatenlogik möglich.

Neben der unzureichenden Beschreibung von Mustern kritisiert Eden auch die unzureichende Beschreibung von Musterbeziehungen:

Unzureichende Beschreibung von Musterbeziehungen

“Furthermore, in the absence of a suitable, dedicated pattern specification language, design patterns were never entirely codified, and no attempt towards putting order into their interactions has ever gone beyond ‘this pattern uses that pattern’, or ‘these two patterns often occur simultaneously’. Results of such inquiries take, at most, the form of rudimentary diagrams which shed little light on the associations between patterns.”

Neben der unpräzisen Beschreibung von Prozessmusterbeziehungen sind ferner oft die Beschreibungen der Prozesse informal. Eine schrittweise Beschreibung des Prozesses fehlt oft, und es gibt sogar Prozessmuster, die gar keinen Prozess als Lösung enthalten (z.B. bei [Cop94]). Mehrere Anwender eines Prozessmusters können also völlig unterschiedliche Schlüsse aus den Beschreibungen ziehen und gegebenenfalls falsche Entscheidungen fällen. Für die Formalisierung von Prozessmustern benötigen wir also ein Modellierungsmittel, d.h. eine Beschreibungssprache mit formaler Syntax und Semantik.

Fazit

2.1.6 Klassifizierung von Mustern

Die Klassifizierung eines Musters dient dazu, ein Muster anhand verschiedener Kriterien einordnen zu können. Hierdurch wird die Nutzbarkeit eines Musters erhöht. Fast jeder Autor benutzt einen eigenen Klassifizierungsansatz. Die am häufigsten verwendete Klasse ist der Typ eines Musters, also z.B. Entwurfsmuster, Architekturmuster, Pädagogikmuster usw. Neben dem Typ gibt es eine weitere Anzahl von Klassifizierungsansätzen. Wir führen nachfolgend einige der geläufigsten Klassifizierungsansätze auf.

Ziel

Die Mitglieder der GoF nutzten zur Unterscheidung von Entwurfsmustern die Aufgabe des Musters (Erzeugungs-, Struktur-, Verhaltensmuster) und den Gültigkeitsbereich (klassen-, objektbasiert) [GHJ96].

Klassifizierung von Entwurfsmustern

Buschmann et al. verwendeten zur Einordnung von Entwurfsmustern die Dimensionen Abstraktionsgrad (mit Ausprägungen Architekturpattern, Design Pattern und Idiome) und Problemkategorie [BMR96]. Riehle verwendete ferner als Klassifikationskriterium die Rolle von Objekten [Rie97b] und Tichy den Zweck des Musters [Tic97].

Der Nachteil dieser Klassifizierungsansätze ist, dass sie speziell auf die Eigenschaften von Entwurfsmustern abgestimmt sind. Für die Anwendung auf Prozessmuster müssen diese Klassifizierungsschemata daher noch angepasst und erweitert werden.

Klassifizierung von Prozessmustern

Störrle unterscheidet Prozessmuster und Ergebnismuster, wobei Prozessmuster den Weg zur angestrebten Lösung aufzeigen, während Ergebnismuster die Lösung selbst präsentieren [Stö01a]. Zu den Ergebnismustern zählen beispielsweise die Entwurfsmuster. Prozessmuster lassen sich anhand von vier Aspekten klassifizieren [Stö00], [Stö01a]:

- Abstraktionsgrad des Prozessmusters (Entwicklungsstil (natürlichsprachliche Beschreibung, z.B. OO, iterativ, fraktal), Prozessmuster und Techniken)
- Phase, in der das Prozessmuster eingesetzt wird (Spezifikation, Design, Realisierung, Wartung)
- Zweck, den das Prozessmuster erfüllen soll (Administration, Konstruktion, Wiederverwendung, Qualitätssicherung)
- Anwendungsbereich des Prozessmusters (Projekt, Capsule (architektonische Einheit), Komponente)

Die Abgrenzung zwischen Prozessmustern und Ergebnismustern gelingt nicht immer. Coplien führt beispielsweise in [Cop94] Organisations- und Prozessmuster auf, die zur (vorteilhaften) Formung einer Organisation und derer Prozesse dienen. Beim genaueren Hinsehen handelt es sich jedoch bei allen Mustern um Organisationsmuster mit Ausprägung als Prozess- oder Ergebnismuster. Ferner sind nicht alle Muster, die als Prozessmuster bezeichnet sind, auch solche (z.B. „Size the Schedule“).

Bergner et al. verwenden vier Abstraktionsstufen für Prozessmuster: Project Patterns, Inter-Result Patterns, Main Result Patterns und Subresult Patterns [BRS98]. Diese orientieren sich an der Struktur und Granularität der Ergebnisse (s. auch Abschnitt 2.1.8)

Klassifizierung von Mustern nach Phasen

Berten orientierte sich an den Phasen der Softwareentwicklung und schlug in [Ber97] vor, die verschiedenen Muster aus allen Bereich der Softwareentwicklung, wie Analyse, Entwurf, Codierung, Organisation etc. entsprechend des Einsatzzeitpunktes zu organisieren und zu einem Gesamtsystem zur Softwareentwicklung mit Mustern zusammenzufassen (Abbildung 2-2).

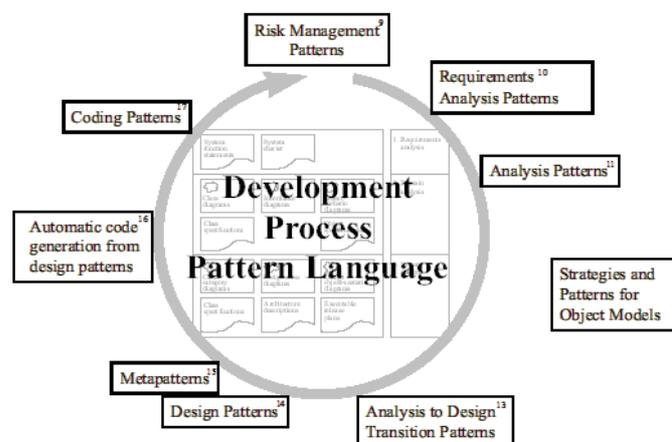


Abbildung 2-2: Klassifizierung von Mustern nach Phasen

Czichy unterschied drei verschiedene, in der Software-Entwicklung vorherrschende Muster-Klassen [Czi01]. Hierzu zählen (Abbildung 2-3):

- Die Klasse aktivitätsunabhängiger Muster mit den Unterklassen Prozess- und Organisationsmuster,
- die Klasse aktivitätsabhängiger Muster mit mehreren, den Softwareentwicklungsphasen angelehnten Unterklassen (Requirements Analysis Patterns, Maintenance Patterns usw.)
- sowie eine Klasse von Mustern, die sich nur im weiteren Sinne mit Themen der Softwareentwicklung auseinandersetzen (Pattern Writing Patterns, Educational Patterns).

Klassifizierung von aktivitätsabhängigen und -unabhängigen Mustern ...

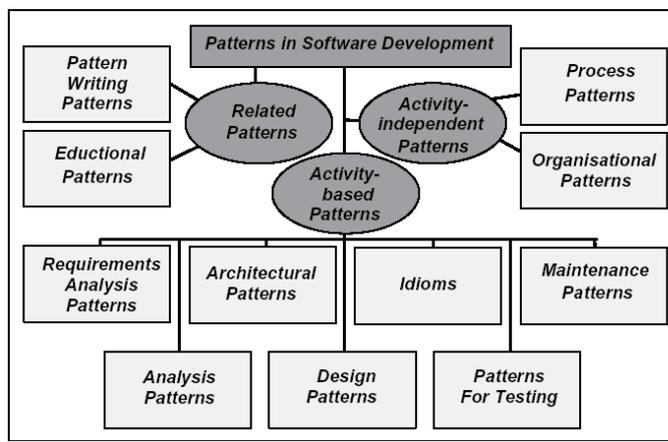


Abbildung 2-3: Klassifizierung von Mustern nach Czichy

Diese Klassifikation ist jedoch nicht nutzbar: Die Definition, was aktivitätsabhängige und aktivitätsunabhängige Muster genau unterscheidet, ist nur schwammig definiert. Worum handelt es sich z.B. bei einem Muster, das beschreibt, wie ein Testfall zu erstellen ist? Mit aktivitätsbasierten Mustern sind wohl eher nach Phasen unterschiedene Ergebnismuster gemeint. Ein weiterer Nachteil des Klassifizierungsschemas ist, dass bei der Klassifizierung der Muster nicht nach Inhalt (z.B. Analysis Patterns, System Test Patterns, Customer Interaction Patterns usw.) und Form (Ergebnis- und Prozessmuster) unterschieden wird.

... ist nicht nutzbar

Ein weiteres Beispiel für eine solche – fast überall übliche – Vermischung von Inhalt und Form ist bei Rising zu finden [Ris00]: Im „Pattern Almanac“ wurden eine Vielzahl bekannter Muster der Softwareentwicklung zusammengetragen. Die Muster wurden zum leichteren Auffinden kategorisiert. Die Autorin vermischt hierbei zweckbestimmte Kategorien von Entwurfsmustern (Behavioural, Creational, Structural) mit inhaltlich-fachlichen Kategorien (Accounting, Organization and Process, Testing usw.).

Vermischung von Inhalt und Form

Riehle und Züllighoven führten den Begriff der generativen und der deskriptiven Musterschemata ein. Sie bezeichnen z.B. das Alexandersche, das Portland- und das Coplien-Schema als generative Musterschemata, das GoF-Schema als deskriptives Musterschema [RZ96].

Klassifizierung von generativen und deskriptiven Mustern

Generative Musterschemata	Generative Musterschemata haben die Eigenschaft, weniger die Lösung (im Sinne des Ergebnisses) zu beschreiben, sondern die Schritte, die notwendig sind, um diese Lösung zu generieren. Coplien stellte beispielsweise in [Cop94] eine generative Mustersprache vor, unter deren Einsatz neue Organisationen bzw. Organisationsformen entwickelt und weiterentwickelt werden können. Die Sprache enthält Organisations- und Prozessmuster. Die Generativität der Sprache liegt darin, dass nicht konkret Organisationsformen als Lösung beschrieben werden. Beschrieben wird nur der Lösungsweg. Bei Anwendung der Sprache wird also jedesmal eine neue, individuelle Lösung (hier Organisation) generiert.
Deskriptive Musterschemata	Die GoF-Form als deskriptives Musterschema beschreibt die Struktur der Lösung (im Sinne des Ergebnisses), d.h. die Struktur der Klassen und Objekte. Beispiele für deskriptive Muster finden sich in [GHJ96]. Die Unterscheidung von generativen und deskriptiven Mustern entspricht der Betrachtung einer Lösung aus zwei verschiedenen Perspektiven. Zwischen diesen Perspektiven kann gegebenenfalls gewechselt werden. [BJ94] weisen z.B. darauf hin, dass einige der deskriptiven GoF-Muster in generative Muster transformiert werden könnten.
Fazit	Die Vielzahl der existierenden Klassifikationsausprägungen und deren unterschiedliche Verwendung macht deutlich, dass eine Uneinigkeit über die Beschreibung von Mustern besteht. Dies führt dazu, dass Begriffe und Bezeichnungen widersprüchlich oder für verschiedene Dinge benutzt werden. Die Uneinheitlichkeit der Klassifizierungsansätze führt zu Missverständnissen und falscher Nutzung von Klassifikationen und letztendlich zur falschen Nutzung der Muster selbst. Aus diesen Gründen haben wir ein umfassendes Klassifizierungsschema für Prozessmuster erarbeitet, das für die einheitliche Klassifizierung von Prozessmustern genutzt werden kann (Abschnitt 3.1).

2.1.7 Organisation von Mustern

Muster liefern die Lösung zu Einzelproblemen. Muster sind jedoch keine isolierten Entitäten, sondern haben Gemeinsamkeiten und Beziehungen zueinander (s. hierzu [Hag02]). Die Gemeinsamkeit bezieht sich meistens auf den Anwendungsbereich oder den Einsatzzweck der Muster wie z.B. das OO-Design. Solche zusammenhängenden Muster werden in Musterkatalogen (z.B. [GHJ96]), Mustersystemen (z.B. [BMR96]), Mustersprachen (z.B. [AIS77], [MR01]) oder Musterhandbüchern (z.B. [RZ96]) organisiert.

Musterkatalog (engl. Pattern Catalog)	Der bekannteste Musterkatalog ist die von der "Gang-of-Four" erstellte Sammlung von 23 Entwurfsmustern. Neben der Darstellung der Muster werden Beziehungen zu anderen Mustern aufgezeigt. Die Darstellung der Beziehungen ist jedoch nicht vollständig und zeigt nur eine Auswahl der potentiell vorhandenen Beziehungen. Im Vordergrund steht die Lösung von Entwurfsproblemen durch einzelne Muster. Generell gelten Musterkataloge deshalb als Sammlungen lose verknüpfter Muster. Sie gelten als nicht vollständig in Bezug auf das repräsentierte Wissen.
Mustersystem (engl. Pattern System)	Ein Mustersystem ist eine Menge in Beziehung stehender Muster. Es repräsentiert Wissen bezüglich eines bestimmten Themengebiets. Das Mustersystem ist organisiert in Gruppen und Untergruppen, die zueinander in Beziehung stehen und die auf verschiedenen Granularitätsstufen existieren. Ferner wird beschrieben, wie Muster kombiniert und komponiert werden. Ein Mustersystem gilt als strukturierter als ein Musterkatalog. Durch Hinzufügen einer Struktur kann ein Musterkatalog in ein Mustersystem umgewandelt werden.

Mustersprachen gelten als Mengen miteinander verknüpfter Muster [AIS77], [Cop96]. Durch die enge Verknüpfung der Muster sind mehr und komplexere Probleme lösbar als bei Musterkatalogen und Mustersystemen. Bisherige Definitionen von Mustersprachen sind informal und gehen über Aussagen wie

„A pattern language is a collection of patterns that build on each other to generate a system. We can compare a pattern language to natural language.“ [Cop96]

nicht hinaus. Deswegen ist die Unterscheidung zwischen Musterkatalogen, Mustersystemen und Mustersprachen schwierig.

Im Gegensatz zu Musterkatalogen und Mustersystemen gelten Mustersprachen als „computationally complete“, d.h. alle möglichen Kombinationen von Mustern und Mustervarianten werden dargestellt. Eine Mustersprache gilt daher als vollständig für ein Themengebiet und all seine Probleme. Ein Mustersystem kann durch Anreicherung von Beziehungen, Mustern und Problemen nach und nach in eine Mustersprache übergehen.

Der Begriff Mustersprache ist kritisch zu betrachten, da er nicht im eigentlichen linguistischen Sinne verwendet wird, wie schon Coplien in [Cop96] bemerkte:

“[A pattern language] has structure, but not the same level of formal structure that one finds in programming languages. The term pattern language has been the source of some confusion because of this, leading some authors to instead use the term pattern system.“

Eine Sprache wird stets durch eine Grammatik definiert. Betrachtet man einen durch eine Mustersprache modellierten Prozess, so müssten die Prozessmuster die Terminalsymbole, der Prozess ein Wort der Sprache, das Startsymbol das Ausgangsproblem und die Produktionsregeln sein, die angeben, in welcher Reihenfolge welche Muster angewendet werden können. Existierende Mustersprachen besitzen solche formalen Grammatiken nicht. Daher wird der Begriff Mustersprache irrtümlich verwendet. Aus diesem Grund verwenden einige Muster-Autoren wie Buschmann et al. in [BMR96] statt Mustersprache den Begriff Mustersystem.

Riehle und Züllighoven betrachten die Begriffe Musterkatalog, Mustersystem und Mustersprache aus den oben genannten Gründen als problematisch und verwenden daher anstatt dessen den Begriff Musterhandbuch [RZ96]. Ein Musterhandbuch fasst die Konzepte einer Domäne in Gestalt von Mustern zusammen. Einem Musterhandbuch liegt immer ein sogenanntes „Leitmotiv“ zugrunde, welches das generelle Prinzip der Softwareentwicklung und das Gesamtbild („big picture“) des zu entwickelnden Systems darstellt. Ferner werden Eigenschaften der durch die Muster angesprochenen Anwendungsdomäne beschrieben. Gegebenenfalls wird auch eine geeignete Entwicklungsstrategie (z.B. evolutionäre Softwareentwicklung) vorgeschlagen. Leider gehen auch hier die Autoren über eine informale Definition nicht hinaus. Musterbeziehungen werden gar nicht angesprochen.

Wie schon weiter oben angeführt sind Eigenschaften und Struktur der verschiedenen Organisationsformen eher schwammig definiert. Deswegen werden die Begriffe auch oft verschiedentlich benutzt. Auch wenn der Begriff Mustersprache dies vermuten lässt, besitzen existierende Mustersprachen weder eine formale Syntax noch eine formale Semantik. Ferner sind existierende Muster, -Kataloge und -Sprachen aufgrund mangelnder Definitionen nicht miteinander integrierbar. Stattdessen werden kontinuierlich neue Muster ohne Integration

Mustersprache (engl. Pattern Language)

Musterhandbuch (engl. Pattern Handbook)

Fazit

mit existierenden Mustern dokumentiert. Entsprechend fiel das Ergebnis einer Studie von Czichy aus [Czi01]. Sie belegte, dass 84% aller Befragten es als schwierig empfinden, zu einem Problem ein passendes Muster finden.

Aus diesen Gründen haben wir die Begriffe Mustersprache, Mustersystem und Musterhandbuch nicht weiter berücksichtigt und den Begriff des Musterkatalogs definatorisch neu besetzt (s. auch Kapitel 3). Denn der Begriff Musterkatalog scheint uns am ehesten geeignet, eine strukturierte Menge von Mustern für eine bestimmte Domäne zu repräsentieren.

2.1.8 Beziehungen zwischen Mustern

Muster können in Beziehung zu anderen Mustern stehen. Im linguistischen Sinne sind Muster die Terminalsymbole einer Sprache für eine bestimmte Domäne. Wörter dieser Sprache sind Aneinanderreihungen von Mustern. Mit welchen Grammatikregeln die Wörter abgeleitet werden, hängt von den Beziehungen zwischen Mustern ab. Nachfolgend stellen wir einige bekannte Musterbeziehungen vor:

"Gang-of-Four"

Bislang haben sich nur sehr wenige Autoren mit Musterbeziehungen thematisch beschäftigt. In dem populären Muster-Katalog „Design Patterns“ der "Gang-of-Four" wird zwar für jedes Muster auf Beziehungen zu anderen Mustern im Abschnitt „Verwandte Muster“ eingegangen [GHJ96]. Die Beziehungen werden jedoch weder syntaktisch noch semantisch definiert. Was bedeutet z.B. die Beziehung „zusammengesetzt aus“ zwischen den Mustern „Befehl“ und „Kompositum“ und die Beziehung „definiert Grammatik“ zwischen den Mustern „Interpreter“ und „Kompositum“? (Abbildung 2-4). Beide Beziehungen stellen eine Nutzungsbeziehung dar. Da die beiden Beziehungen jedoch informal beschrieben werden, ist unklar, ob es sich hier um zwei unterschiedliche Beziehungen oder um eine Beziehung mit zwei unterschiedlichen Namen handelt. Nur durch genaue Lektüre des Textes ist klar, ob das Muster „Kompositum“ unter den gleichen Bedingungen von den Mustern „Befehl“ und „Interpreter“ angewendet werden kann.

Ferner werden einige Beziehungen gar nicht erst genannt: Beispielsweise werden im Abschnitt „Verwandte Muster“ des Musters „Adapter“ drei verwandte Muster – „Brücken“-Muster, „Dekorierer“-Muster und „Proxy“-Muster – aufgezählt, ohne den Beziehungstyp anzugeben. Weiterhin werden zwei Adapter-Varianten – nämlich Objekt- und Klassenadapter – in der Musterbeschreibung nicht explizit genannt.

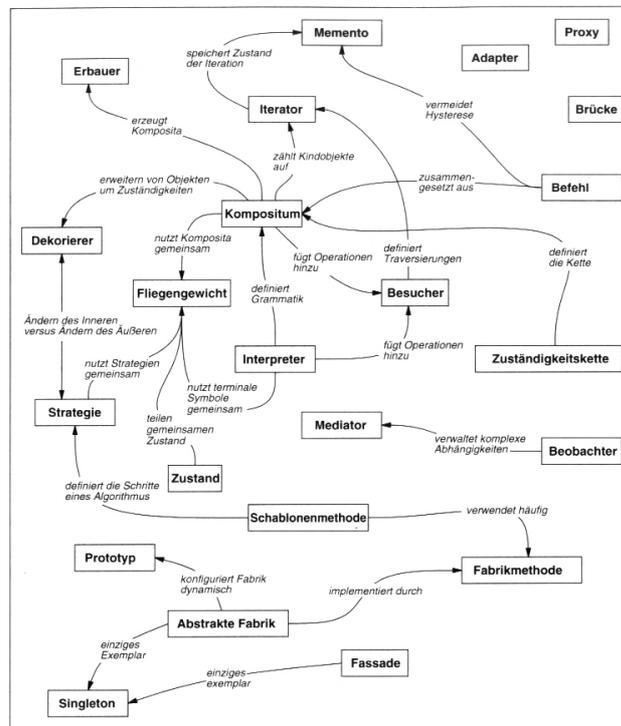


Abbildung 2-4: Beziehungen zwischen GoF-Entwurfsmustern

Um die Vielzahl von Beziehungen zu vereinheitlichen, hat Zimmer die Beziehungen zwischen den GoF-Mustern analysiert und kategorisiert [Zim95]. Die GoF-Beziehungen wurden auf die drei Beziehungen „X uses Y in its solution“, „X is similar to Y“ und „X can be combined with Y“ reduziert (Abbildung 2-5, weitere Details in Anhang A.1.2).

Zimmer

Trotz der Vereinheitlichung der Beziehungen lassen auch die Formulierungen von Zimmer konkrete und formale Definitionen vermissen. Was bedeutet z.B. die Beziehung „X uses Y in its solution“ für die Kontexte der beiden in Beziehung zueinander stehenden Mustern? Auch in anderen Arbeiten werden stets informale Definitionen für Beziehungen verwendet (z.B. [ABW88], [BMR96], [MD98], [Lor97], [Tic97]).

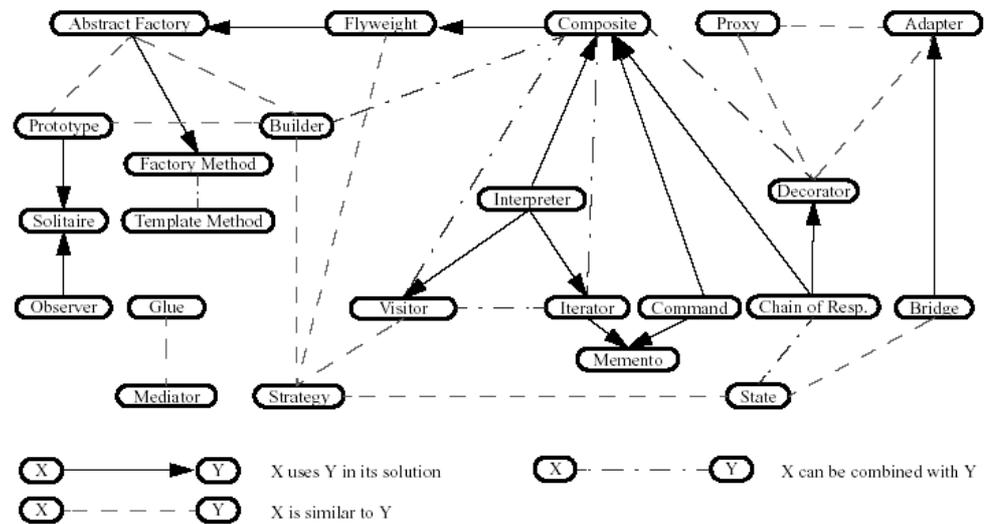


Abbildung 2-5: Klassifikation von Entwurfsmusterbeziehungen nach Zimmer

Noble Noble hat die Ergebnisse von Zimmer konsolidiert und weitergeführt [Nob98a]. Er klassifiziert Beziehungen zwischen objektorientierten Entwurfsmustern als primäre und sekundäre Beziehungen (Details in Anhang A.1.2). Als primäre Beziehungen definiert er „Uses“ (Abbildung 2-6), „Refines“, und „Conflicts“. Als sekundäre Beziehungen definiert er „Used by“, „Refined by“, „Variants“, „Variant Uses“, „Similar“, „Combines“, „Require“, „Tiling“, „Sequence of Elaboration“. Sekundäre Beziehungen zeichnen sich dadurch aus, dass sie durch die primären Beziehungen konstruiert werden können. Z.B. ist die Used-By-Beziehung die inverse Beziehung der Beziehung Uses.

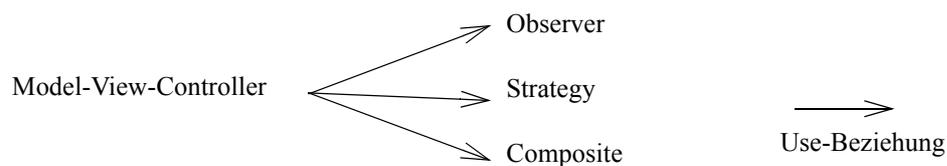


Abbildung 2-6: Use-Beziehung von Noble

Mit Nobles Schema für primäre und sekundäre Beziehungen sind zwar schon eine Vielzahl durchaus komplexer Beziehungen identifiziert worden, allerdings lassen diese Beziehungen Präzision und Formalität vermissen.

Ambler Bei Ambler werden Prozessmuster in einer Hierarchie eingeordnet ([Amb98], [Amb99]). Prozessmuster sind genau einer der drei Ebenen „Phase“, „Stage“ und „Task“ zugeordnet. Prozessmuster der „Phase“-Ebene setzen sich aus Prozessmustern der „Stage“-Ebene zusammen, diese wiederum setzen sich aus Prozessmustern der „Task“-Ebene zusammen (Abbildung 2-7). Hierbei handelt es sich also um eine Nutzungsbeziehung. Es werden jedoch keine

Definitionen oder Bedingungen für die Erfüllung dieser Beziehung angegeben. Ferner wird auch nicht angegeben, ob und wie Prozessmuster sequentiell angewendet werden können. Es ist z.B. unklar, was die Pfeile zwischen den einzelnen Prozessmustern bedeuten (z.B. zwischen „Justify“ und „Define Infrastructure“). Findet dort ein Datenaustausch statt, oder können die Prozessmuster wechselseitig angewendet werden?

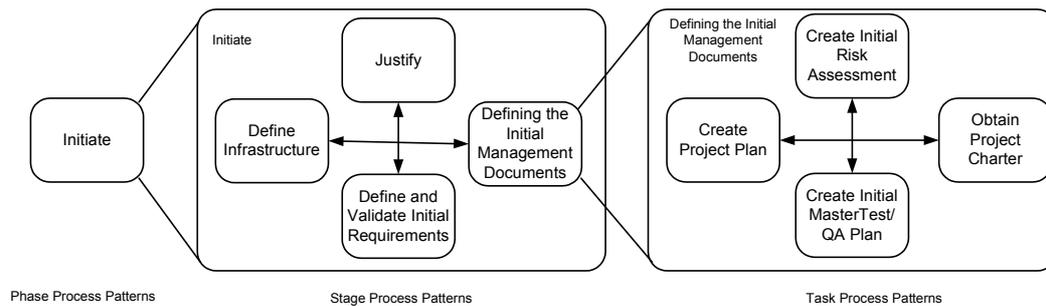


Abbildung 2-7: Nutzungsbeziehung bei Ambler

Störrle entwickelte in [Stö00] u.a. eine Muster-Sprache zur Modellierung von komponentenbasierten Software-Architekturen. Dort verwendet er die zwei Beziehungen „may contain“ und „prerequisite of“ (Abbildung 2-1). „May contain“ impliziert eine Nutzungsbeziehung, „prerequisite of“ eine sequentielle Abfolge von Prozessmustern. Ein besonderes Merkmal dieser Muster-Sprache ist die Verwendung der Beziehung „prerequisite of“ als rekursive Beziehung. Diese Besonderheit resultiert aus der Erkenntnis, dass eine komponentenbasierte Software-Architektur einen ebenso strukturierten Entwicklungsprozess benötigt. Komponentenbasierte Software-Architekturen sind rekursiv (auch fraktal genannt), da sie sich aus Komponenten zusammensetzen, die sich wiederum aus Komponenten zusammensetzen usw. Auch in der Arbeit von Störrle wird keine formale Definition dieser Beziehungen vorgenommen.

Störrle

Gnatz et al. identifizierten in [GMP01b] zwei Prozessmusterbeziehungen, nämlich „realized by“ und „execute“ (Abbildung 2-8). Die Beziehung „realized by“ wird für zwei oder mehr Prozessmuster verwendet, die das gleiche Problem lösen, d.h. Varianten sind. Die Beziehung „execute“ stellt die Kompositionsbeziehung dar. Auch in der Arbeit von Gnatz et al. wird keine formale Definition der Beziehungen vorgenommen.

Gnatz et al.

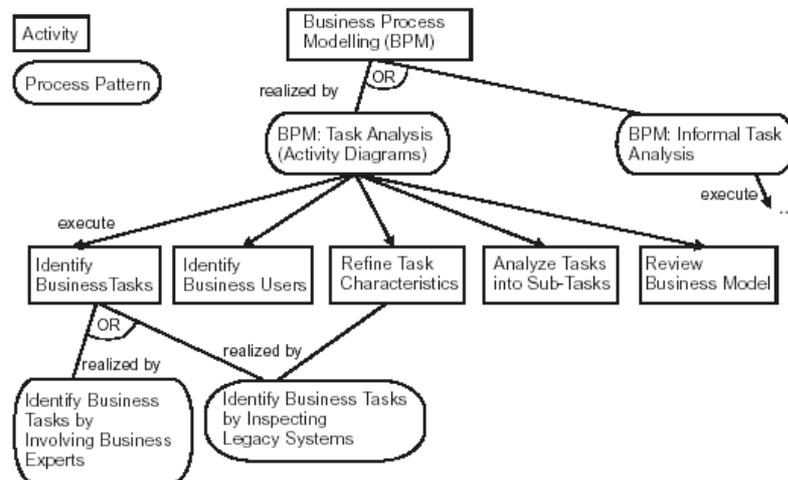


Abbildung 2-8: Prozessmusterbeziehungen bei Gnatz et al.

Bergner et al.

Bergner et al. stellen in [BRS98] Prozessmuster für die komponentenbasierte Softwareentwicklung vor. Statt Prozessmusterbeziehungen werden Beziehungen von Erzeugnissen in einer vorgegebenen Struktur angegeben (Abbildung 2-9). Über Erzeugnisse und deren Beziehungen sind dann entsprechende erzeugende Prozessmuster identifizierbar. Die Erzeugnisrelationen werden nicht formal definiert, sondern es wird nur erwähnt, dass es beispielsweise Beziehungen wie „Refinement“ geben kann. Wie Prozessmuster zu den Erzeugnissen identifiziert werden, bleibt unklar.

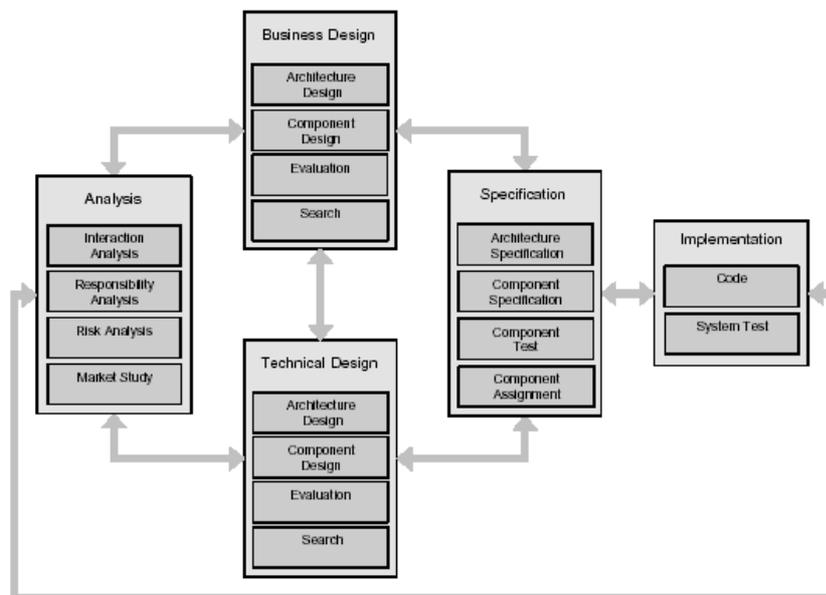


Abbildung 2-9: Erzeugnisstruktur bei Bergner et al.

Förster verwendet Prozessmuster zur Modellierung von Geschäftsprozessen [FE03]. Er verwendet implizit eine Prozessmusterbeziehung, nämlich die Verfeinerung eines Prozessmusters. Die Verfeinerung eines Prozessmusters erfolgt durch die Verfeinerung einer oder mehrerer Aktivitäten des Musterprozesses und resultiert in einem neuen Prozessmuster. Welche Regeln für die Verfeinerung gelten, wird jedoch nicht angegeben.

Förster

Erst durch die Definition von Musterbeziehungen können komplexere Konstrukte – die Wörter einer Sprache – erzeugt werden. In den bislang erschienenen Veröffentlichungen werden jedoch Beziehungen unterschiedlich und nur informal definiert. Diagramme, die zur Darstellung der Musterbeziehungen dienen, lassen an Aussagefähigkeit zu wünschen übrig. Diese Ungenauigkeit hat zur Folge, dass jeder Muster-Anwender ein anderes Verständnis bzgl. verschiedener Musterbeziehungen hat³. Hierbei droht die Gefahr, dass gegebenenfalls ein falsches Muster angewendet oder ein geeignetes Muster gar nicht erst gefunden wird. Die mangelnde Formalität verhindert auch, dass die Definitionen von Muster-Beziehungen nicht organisationsübergreifend genutzt werden können. Als Beispiel betrachte man den GoF-Katalog. Dort werden verschiedene Beziehungen verwendet. Will man nun diesen Katalog erweitern, ist fraglich, ob die vorhanden Beziehungen auf neue Muster angewendet werden können, oder wie neue Beziehungen, die zwischen den neu hinzukommenden Mustern bestehen, mit den existierenden Beziehungen integriert werden können. Ohne eine formale Basis ist eine Beantwortung dieser Fragen nicht möglich.

Fazit: Mangelnde Einheitlichkeit und Informalität erlaubt beliebig viele Interpretationen ...

Damit ist das erklärte Ziel der Muster, nämlich die Wiederverwendung von Wissen durch Explizierung in Form von Mustern, ad absurdum geführt. Denn die Wiederverwendung von Mustern innerhalb einer Muster-Organisationsform und über die Grenzen einer Muster-Organisationsform hinweg werden durch die inkonsistenten Begrifflichkeiten fast unmöglich.

... und verhindert auf diese Weise die Wiederverwendung von Mustern, ...

Für Prozessmuster wurden bislang sehr viel weniger Beziehungen identifiziert als dies bei Entwurfsmustern der Fall ist. Dies liegt sicherlich auch daran, dass die Anzahl der Prozessmuster und die Erfahrung mit ihrem Einsatz noch sehr viel geringer ist als bei Entwurfsmustern. Wie auch bei Entwurfsmustern fehlt bei den Prozessmusterbeziehungen die Formalisierung. Bei der Modellierung von Prozessen fällt dieses Manko jedoch sehr viel mehr ins Gewicht als bei Entwurfsmustern, da die korrekte Verknüpfung und Komposition von Prozessen genaue Regeln erfordert, wie diese Verknüpfung bzw. Komposition vollzogen werden kann. Aus diesen Gründen haben wir im Rahmen dieser Arbeit bereits identifizierte Musterbeziehungen verwendet, diese aber für unsere Zwecke angepasst und formal definiert (Kapitel 3).

... was ein besonderes Problem für Prozesse bzw. Prozessmuster darstellt.

2.1.9 Pattern Management

Unter Pattern Management verstehen wir alle mit Mustern verknüpften Tätigkeiten wie Muster-Dokumentation, Problem-Identifikation und -Spezifikation, Mustersuche, Musterauswahl, Musteranwendung und Mustereinführung und -einsatz.

Definition

Damit neue Muster entstehen können, muss das vorhandene Wissen von Experten über wiederkehrende Probleme und deren Lösung dokumentiert werden. Dieses Wissen ist oft nur implizit (sog. implizites Wissen) in den Köpfen der Experten vorhanden und muss deshalb

3. Ein Beispiel für die Verwirrung, die durch solche Ungenauigkeiten gestiftet werden, sieht man in der Diskussion um die Muster „Multicast“ und „Observer“, in der es darum ging, die Unterschiede zwischen den beiden Entwurfsmustern herauszuarbeiten ([Vli97a], [Vli97b]).

zunächst externalisiert (sog. explizites Wissen) werden [Non91]. Dieses explizite Wissen (in Form von Mustern) kann dann von anderen Individuen wieder zu impliziten Wissen verinnerlicht, d.h. internalisiert werden. Czichy hat für diesen Wissenstransfer von Mustern einen Prozess (siehe Abbildung 2-10) definiert, den sogenannten „Microprocess for Patterns“ [Czi01]. Der Microprocess wird als Ergänzung zum Macroprocess betrachtet, der den generellen Software-Entwicklungsprozess beschreibt.

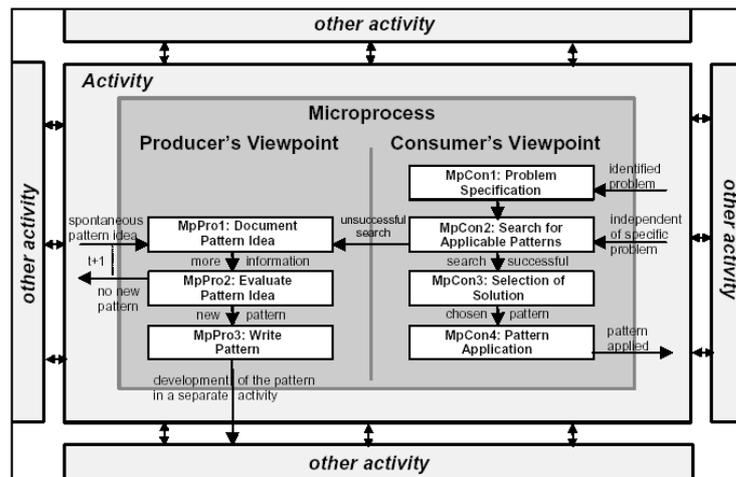


Abbildung 2-10: Microprocess for Patterns

Der Microprocess wird von zwei Rollen, dem Produzenten und dem Konsumenten, beherrscht. Der Produzent externalisiert⁴ sein oder das Wissen anderer und erzeugt neue Muster. Der Konsument stellt den Anwender von Mustern dar, der beim Auftreten bestimmter Probleme entsprechende Muster sucht und anwendet oder gegebenenfalls die Erzeugung neuer Muster anstößt.

Pattern-Erzeugung

Der Prozess zur Identifikation und Dokumentation von Wissen in Form von Mustern wird auch „Pattern Mining“ genannt. Rising hat sieben Techniken für das Pattern Mining entwickelt, die die Identifikation und Dokumentation neuer Muster erleichtern [Ris99]:

- Mining by Interviewing
Interview mit Fachleuten und Dokumentation der Informationen in Form von Mustern
- Mining by Borrowing
Adaption von bereits vorhandenen Mustern der gleichen Domäne
- Mining by Teaching Patterns Writing
Identifikation neuer Muster im Rahmen von Writers Workshops
- Mining in Workshops
Identifikation neuer Muster im Rahmen geleiteter, themenbezogener Workshops
- Mining Your Own Experience
Identifikation neuer Muster durch Dokumentation eigener Erfahrungen, geht immer einher mit allen anderen Techniken

4. Externalisierung bedeutet die Äußerung des Wissens durch Dokumentation.

- Mining in Meetings
Identifikation neuer Muster durch Begleitung bestimmter Meetings (Postmortem, Präsentationen usw.) durch den Pattern-Beauftragten
- Mining in Classes
Ableiten neuer Muster aus professionellen Seminaren und Schulungen

Neben den oben vorgestellten Methoden zur manuellen, interaktionsreichen Identifikation von Mustern gibt es auch bereits einige Ansätze zur automatischen, d.h. computergestützten Suche von Mustern. Diese Ansätze beschränken sich allerdings hauptsächlich auf Idiome und Entwurfsmuster. Voraussetzung hierfür ist die Verwendung von Mark-Up-Sprachen wie XML oder SGML zur Beschreibung der Musterstruktur. Auf Basis dieser Beschreibungen wird der Sourcecode durch statische oder dynamische Analyse zur Laufzeit auf Übereinstimmungen mit einem vordefinierten Muster-Repository, in dem die Beschreibungen liegen, untersucht (Beispiele in [Bro96], [KSR99]). Hierdurch kann das Reverse Engineering unterstützt werden. Auch die Identifikation neuer Muster ist möglich.

Nicht nur für das Reverse Engineering ist es wünschenswert, Muster in einer Programmiersprache beschrieben zu haben, sondern auch für das Forward Engineering. Beispielsweise wurde die Programmiersprache Eiffel zu einer musterorientierten Programmiersprache (Pattern oriented Language, PaL) erweitert, die zusätzliche Konzepte wie Komposition und Verfeinerung beinhaltet ([Bün99], [Mei96]).

Die Dokumentation und Bewertung neuer Muster findet in sogenannten Pattern Workshops oder Writers Workshops statt. In diesen Workshops tragen Pattern-Autoren ihr Pattern dem Publikum vor, welches anschließend das Muster diskutiert und Verbesserungsvorschläge macht. Zur Vorbereitung auf die Workshops wird jedem Muster-Autor als Betreuer ein „Pattern Sheperd“ zugewiesen, der das Muster schon vorab auf dessen Praktikabilität prüft und dem Autor Hinweise zur Verbesserung gibt.

Der Schwerpunkt in der Pattern Community liegt in der Entdeckung und Beschreibung neuer Muster. Für die Verinnerlichung von Mustern und deren Anwendung sind vergleichsweise wenige methodische Ansätze bekannt. Pattern-Anwender recherchieren meistens im Internet nach geeigneten Mustern. Dort gibt es einige umfangreiche Portale wie z.B. das Wiki-Web. Diese Portale sind jedoch nicht vollständig. Auch werden die Portale nicht miteinander integriert. Allerdings ist gerade das Finden eines geeigneten Musters zur Lösung eines Problems ausschlaggebend für Erfolg und Akzeptanz der Muster. Ist einem Anwender das passende Muster nicht bekannt, oder sind vorhandene Muster-Kataloge zu unübersichtlich oder nicht systematisch aufgebaut, nützt das beste Muster nichts. Noble nannte dies auch das „pattern selection problem“ [Nob98b]. Die Suche nach geeigneten Mustern kann durch Anordnung der Muster in Musterkatalogen, Mustersystemen oder Mustersprachen erleichtert werden (Kapitel 2.1.7).

Pattern-Anwendung

Für die Einführung und Schulung von Mustern gibt es ebenfalls nur wenige Arbeiten. Zu diesen zählt die Muster-Sprache von Manns und Rising, die Muster zur Einführung von neuen Konzepten in einer Organisation beinhaltet [MR01].

Für die automatisierte, d.h. werkzeuggestützte Anwendung von Mustern gibt es bereits Ansätze zur automatischen Generierung von Design Mustern bzw. Idiomen in Designmodellen bzw. Sourcecode [Bos01].

Fazit Bislang sind nur sehr wenige Arbeiten über Pattern Management verfasst worden. Folgende Fragestellungen wurden noch nicht beantwortet:

- Wie erreicht man, dass Konsument und Produzent von Mustern das gleiche Verständnis/Vokabular benutzen, damit keine Zweideutigkeiten/Unklarheiten entstehen?
- Wie erreicht man, dass der Konsument erfährt, dass es zu einem von ihm erkannten Problem ein oder mehrere Muster zu dessen Lösung bereitstehen?
- Wie erreicht man, dass der Konsument einen vollständigen Überblick über alle vorhandenen Muster hat (das Durchblättern eines Handbuchs ist als unzureichende Alternative anzusehen, vgl. hierzu, dass der Pattern Almanac von Rising mehrere hundert Muster enthält)
- Wie erreicht man, dass die Muster angewendet und deren Anwendung protokolliert werden? Für die Anwendung von Entwurfsmustern bzw. Idiomen gibt es bereits Werkzeugunterstützung, wie sieht es aber mit Prozessmustern aus? Wie funktioniert die Anwendung von Prozessmustern in einem Projekt mit mehreren Mitarbeitern? Wie erreicht man, dass alle Projektmitarbeiter wissen, welche Muster angewendet werden?

Wir sind der Überzeugung, dass für eine effektive und produktive Anwendung von Mustern dem Pattern Management mehr Aufmerksamkeit geschenkt werden muss. Obwohl immer mehr Muster dokumentiert werden, werden nur wenige Anstrengungen unternommen, wie diese Muster integriert werden können, um diese einer breiten Öffentlichkeit bereitzustellen. D.h. der Wiederverwendungszweck tritt hier zunächst in den Hintergrund. In Kapitel 9 haben wir daher ein Konzept für ein systematisches Pattern Management entwickelt.

2.2 Softwareprozessmodellierungssprachen

Prozessmodellierungssprache zur Beschreibung von Prozessmustern

Für die Beschreibung von Prozessmustern benötigen wir eine Modellierungssprache, die alle relevanten Aspekte eines Prozessmusters ausdrücken kann (darunter Prozess, Aktivität, Rolle, Werkzeug, Prozessmusterrelationen, Prozessmusterkatalog usw., s. Kapitel 3). Um einen möglichst hohen Praxisnutzen zu erzielen, sollte die Modellierungssprache graphische Beschreibungselemente, d.h. eine Notation besitzen. Denn wie schon Verlage in [Ver98b] betonte, hat die Modellrepräsentation Einfluss auf die Verständlichkeit der Modelle. Ferner sollte zur Erhöhung der Ausdrucksgenauigkeit die Sprache eine präzise, vorzugsweise formale Semantik besitzen.

Da der Prozess das zentrale Element eines Prozessmusters ist, beurteilen wir nachfolgend eine Auswahl von Softwareprozessmodellierungssprachen (PML, Process Modeling Language) anhand der oben genannten Kriterien. Eine Softwareprozessmodellierungssprache dient dazu, Softwareentwicklungsprozesse in Gestalt eines Prozessmodells zu beschreiben⁵. In den vergangenen fünfzehn Jahren wurde eine Vielzahl von PML entwickelt, die auf unterschiedlichen Paradigmen (regelbasiert, zustandsbasiert usw., Erläuterung und Vergleiche z.B. in [Xie01], [DKW99], [ABG93], [ARF97] und [Ver98b]) basieren. Für jedes Paradigma stellen wir ein oder zwei Repräsentanten vor.

5. Eine Definition weiterer Begriffe aus dem Bereich Softwareprozessmodellierung findet sich in Anhang A.2.

Regelbasierte PML nutzen Mengen von Regeln, um die Verarbeitung von Informationen auszudrücken. Eine Regel besteht dabei aus der Aktivität und Vor- und Nachbedingungen, die vor und nach der Ausführung der Aktivität gelten. Hierarchien sind durch die flache Regelstruktur schwer darstellbar.

Regelbasierte PML

SPELL [JLC92] ist eine in der prozesssensitiven Softwareentwicklungsumgebung EPOS [CHL94] eingebettete PML. Sie basiert auf der Programmiersprache Prolog erweitert um Konzepte der Objektorientierung und Metatypisierung (sog. Reflexivität). Eine visuelle Darstellung ist nicht vorgesehen.

Zu den regelbasierten PML gehören ferner Oikos [MA94], LATIN [CDG95], MARVEL Strategy Language [KFP88], GOLOG [Ple95] und GRAPPLE [HL89].

Zustandsbasierte PML können komplexe Konzepte wie Nichtdeterminismus, Nebenläufigkeit und Synchronisation von Aktivitäten ausdrücken. Hervorzuheben sind ferner die Möglichkeit zur graphischen Darstellung der Prozesse und die Möglichkeit zur hierarchischen Strukturierung.

Zustandsbasierte PML

SLANG [BFG94] ist eine PML der Softwareentwicklungsumgebung SPADE [BFG94]. Sie basiert auf höheren Petri-Netzen [Rei86].

FUNSOFT [DG94] basiert auf den höheren P/T-Netzen. Sie unterstützt die Analyse der Eigenschaften des modellierten Prozesses, die Prozesssimulation, Modellverifikation, Prozessbewertung usw.

Zu den zustandsbasierten PML gehören ferner Entity Model [HK89] und DesignNets [LH89].

Funktionale PML drücken Aktivitäten als Funktionen aus, die bei gegebenen Inputobjekten bestimmte Outputobjekte erzeugen. Sie eignen sich gut zur hierarchischen Dekomposition von Prozessen.

Funktionale PML

HFSP [Kat89] basiert auf der funktionalen Sprache AG und der Umgebung SAGE [SK88] und verwendet den Ansatz attributierter Grammatiken. Aktivitäten werden durch Regeln der Grammatik dekomponiert. Attributregeln beschreiben Ein- und Ausgabe der Aktivitäten und die Beziehung zu der Ein- und Ausgabe der Subaktivitäten.

Zu den funktionalen PML gehört ferner PDL [IST89].

Prozedurale PML beschreiben den Kontrollfluss zwischen Aktivitäten ähnlich wie Programmiersprachen.

Prozedurale PML

APPL/A [SHO95] basiert auf Ada und dient zur Codierung von Prozessen. Sie enthält keine Konzepte wie Aktivitäten, Rollen oder Werkzeuge, sondern dient eher zur Modellierung von Kontroll- und Datenfluss.

JIL [SO97] und Little-JIL [CLS00] sind Weiterentwicklungen von APPL/A.

Objektorientierte PML konzentrieren sich auf die Beschreibung von Prozessartefakten. Als Artefakt verstehen wir ein durch einen Prozess konsumiertes oder erzeugtes Element. Hierdurch lässt sich die komplexe Struktur von Artefakten und deren Beziehungen darstellen.

Objektorientierte PML

SOCCA (Specification Of Coordinated and Cooperative Activities) [EG94] beschreibt einen Softwareprozess in drei verschiedenen Perspektiven, nämlich die Daten-, die Verhaltens- und die Prozessperspektive. Die Datenperspektive beschreibt die statische Struktur eines Prozesses anhand erweiterter Entity-Relationship-Modelle. Die Verhaltensperspektive beschreibt den dynamischen Teil eines Prozesses anhand von Zustandsübergangsdiagrammen in Kombination mit PARADIGM, einem Formalismus zur Modellierung von Parallelität. Die Prozessperspektive wird durch Objektflussdiagramme modelliert. SOCCA besitzt eine formale Syntax und graphische Notationslemente, jedoch keine formale Semantik. Allerdings gibt es Arbeiten zur Formalisierung der (statischen) Semantik mit Z [EDG99].

Die Unified Modeling Language (UML) [UML1.5], wir beziehen uns in dieser Arbeit stets auf die Version 1.5, ist eine durch die OMG standardisierte graphische Sprache zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Artefakten software-intensiver Systeme. Neben der Beschreibung der Artefakte wird die UML mittlerweile auch als PML zur Darstellung der Abläufe innerhalb eines Objekts und der Interaktion zwischen Objekten eingesetzt [JSW99]. Hierzu dienen Interaktions- und Aktivitätsdiagramme. Ziel der UML ist – neben der Bereitstellung einer sofort einsetzbaren, ausdrucksstarken Modellierungssprache – die Erweiterbarkeit der vorhandenen Konzepte. Eine kurze Einführung in den kommenden Standard UML 2.0 findet sich in Anhang A.3.4.

Die ereignisgesteuerten Prozessketten (EPK) beschreiben, welche Ereignistypen welche Funktionstypen auslösen und welche Ereignistypen von welchen Funktionstypen erzeugt werden [Sch01]. Dadurch dass ein Ereignistyp, der von einem Funktionstyp erzeugt wird, auch Auslöser für einen folgenden Funktionstyp ist, entsteht eine zusammenhängende Kette. Es können Verknüpfungsoperatoren zwischen Ereignistypen oder Funktionstypen angegeben werden.

Zu den objektorientierten PML gehören ferner E3 [JLP98] und ESCAPE+ [RSS97].

Fazit Tabelle 2-2 fasst die vorgestellten Prozessmodellierungssprachen überblicksartig zusammen.

PML	Graphische Notation	Formale Syntax	Formale Semantik	Erweiterbarkeit
SPELL	nein	ja	ja	nicht vorgesehen
SLANG	ja	ja	ja	nicht vorgesehen
FUNSOFT	ja	ja	ja	nicht vorgesehen
HFSP	nein	ja	nein	nicht vorgesehen
APPL/A	nein	ja	nein	nicht vorgesehen
SOCCA	ja	ja	nein	nicht vorgesehen
UML	ja	ja	nein	ja
EPK	ja	nein	nein	nicht vorgesehen

Tabelle 2-2: Prozessmodellierungssprachen im Vergleich

Visuelle Notationen besitzen lediglich die Sprachen Marvel, Slang, SOCCA, Oikos und UML. Die visuelle Darstellung der UML ist ihre besondere Stärke. Während die ersteren Sprachen eher im akademischen Umfeld eingesetzt und erforscht werden, hat sich UML – die „lingua franca“ der Software-Engineering-Community [SKE00] – in der Praxis mittlerweile weitgehend als Modellierungssprache durchgesetzt.

Alle Sprachen bis auf die EPKs besitzen eine formale Syntax. Eine formale Semantik besitzen lediglich die petrinetz-basierten Sprachen Slang und FUNSOFT. Für die UML existieren einige Ansätze zur Formalisierung der Semantik von Teilbereichen der UML (Abschnitt 2.3).

Die UML bietet als einzige Sprache den Vorteil, dass sie einen „eingebauten“ Mechanismus zu ihrer Erweiterung durch Profile anbietet. Auf diese Weise können Konzepte wie Prozessmuster und Prozessmusterbeziehungen dem UML-Metamodell hinzugefügt werden.

Nicht alle PML besitzen Konzepte zur Beschreibung von Rollen (z.B. Merlin, Oikos) und zur Beschreibung von Werkzeugen (z.B. Merlin). Keine der PML besitzt ein sprachliches Konstrukt zur Darstellung von Prozessmustern. Konzepte zur Darstellung von Prozessmusterbeziehungen und Prozessmusterkatalogen fehlen verständlicherweise ebenso.

Für die Entwicklung der Process Pattern Description Language (PROPEL) verwenden wir die UML als Basis. Die Wahl erfolgte aus folgenden Gründen:

Durch Wiederverwendung der UML kann PROPEL leichter einem größeren Kreis potentieller Nutzer zugeführt werden, da die meisten Konzepte wie Aktivitätsdiagramme schon bekannt und anerkannt sind. Dies hat zum einen den Vorteil, dass wir die Syntax der UML nur für die neu eingeführten Konzepte erweitern müssen und auf der vorhandenen Syntax aufbauen können. Das Verständnis und die Akzeptanz von PROPEL wird um so höher sein, je akzeptierter die darunterliegende Basis ist. Ferner besitzt die UML einen „eingebauten“ Mechanismus zu ihrer eigenen Erweiterung. Es gibt also klare Regeln zur Erweiterung von UML. Dadurch kann der Leser leicht erkennen, welche Konzepte der UML und welche PROPEL angehören.

Nutzung eines Quasi-Standards und Wiederverwendung der UML-Konzepte

Zur Darstellung von Konzepten wie Prozessmuster und Prozessmusterrelationen müssen neue Notationselemente eingeführt werden. Diese können mit den bestehenden Notationselementen der UML leicht zusammengeführt werden.

Einführung neuer graphischer Elemente

Das Fehlen einer formalen Semantik der UML ist ein häufig geäußelter Kritikpunkt (Abschnitt 2.3). Da jedoch die Vorteile der Nutzung der UML ihre Nachteile überwiegen, entwickeln wir die formale Semantik für die von uns relevanten Konzepte wie Prozess, Prozessmuster und Prozessmusterrelation (Kapitel 5).

2.3 Semantik von UML-Aktivitätsdiagrammen

Die Syntax und informale Semantik der UML wird durch ein Metamodell in [UML1.5] spezifiziert (Abbildung 2-11). Es enthält Konzepte in Form von Metaklassen und Metaassoziationen zur Modellierung von Systemen. Das UML-Metamodell basiert auf dem Metametamodell der Meta Object Facility (MOF) [MOF1.4], einer Sprache zur Spezifikation von technologieunabhängigen Metamodellen. Beispiele solcher Metamodelle sind die

UML-Metamodell

UML und das Common Warehouse Metamodel (CWM) [CWM03], eine Spezifikation zum Austausch von Metadaten in Data-Warehouse-Systemen. Die Entitäten des UML-Metamodells sind also Instanzen der Elemente des MOF-Metametamodells. Auf Basis des UML-Metamodells können beliebige Modelle von Systemen erstellt werden. Die Entitäten eines UML-Modells sind also Instanzen der Elemente des UML-Metamodells. Sie werden zur Laufzeit eines Systems zu Objekten und System- oder Anwendungszuständen instanziiert.

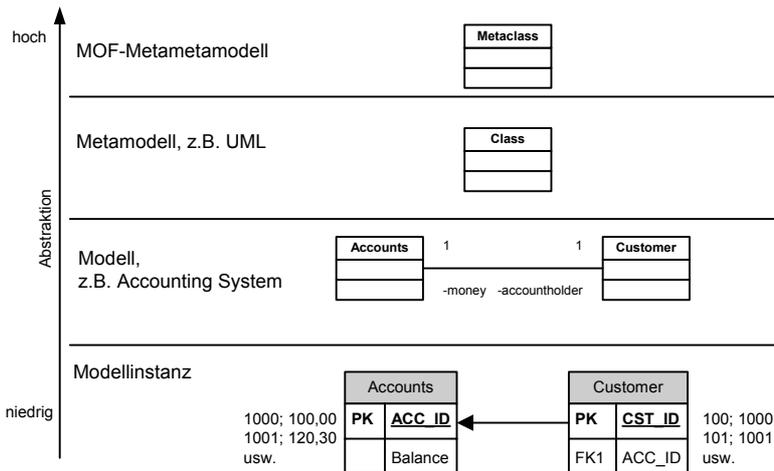


Abbildung 2-11: Das UML-Metamodell und seine Einbettung in die Modellhierarchie

Syntax des UML-Metamodells

Die abstrakte Syntax des UML-Metamodells wird hauptsächlich in Form von Klassendiagrammen definiert. Obwohl Klassendiagramme ein intuitives Mittel sind, um die Syntax der UML zu beschreiben, sind sie weniger ausdrucksfähig als z.B. Graphgrammatiken. Aus diesen Gründen werden zusätzliche Regeln in Form von OCL-Constraints modelliert. Zusätzliche, nicht in Form von OCL-Constraints definierbare Regeln, werden natürlichsprachig modelliert (s. Anhang A.3 für ein Beispiel).

Semantik des UML-Metamodells

Im Unterschied zur Syntax ist die Semantik der UML nur informal, d.h. natürlichsprachig, definiert. Obwohl die UML-Spezifikation Unterabschnitte mit der Überschrift „Detailed Semantics“ enthält, wird dort hauptsächlich noch einmal die Syntax der UML natürlichsprachig erläutert. Aus diesen Gründen wurde für die verschiedenen Teilbereiche der UML eine Formalisierung der Semantik erarbeitet (z.B. für Pakete [SW97], Klassendiagramme [BLM97], [FEL97] und State Charts [BCM98], [WB98]). Unter pUML (preciseUML) firmiert eine Gruppe von Wissenschaftlern und Praktikern, die sich mit der Präzisierung der Semantik der UML beschäftigen [pUML].

Nachfolgend geben wir eine Übersicht über die Formalisierungsarbeiten bzgl. der Semantik von Aktivitätsdiagrammen. Ein wesentliches Element von PROPEL ist der „Prozess“, der auf dem UML-Konzept „Aktivitätsdiagramm“ basiert. Aus diesem Grund interessieren uns nur solche Arbeiten, die sich mit Aktivitätsdiagrammen beschäftigen. Für Konzepte wie Prozessmuster oder Prozessmusterrelation bestehen derzeit noch keine entsprechenden Arbeiten.

Gehrke et al. [GGW98] definieren die Semantik von Aktivitätsdiagrammen auf informale Weise durch Abbildung auf Petri-Netze. Die Abbildung wird anhand von Beispielen erläutert. Zustände werden auf Stellen abgebildet und UML-Transitionen auf Petri-Netz-Transitionen. Der Name eines Zustands bleibt durch Etikettierung der entsprechenden Stelle erhalten. Zustände des Typs „initial“ und „final“ werden auf Stellen abgebildet, die das Etikett „initial“ oder „final“ tragen. Die Anfangsmarkierung eines Netzes wird bestimmt, indem jede Stelle mit Etikett „initial“ eine Marke erhält.

Gehrke et al.

Börger et al. [BCR00] definieren die Semantik von Aktivitätsdiagrammen auf formale Weise durch Abbildung auf sogenannte „abstrakte Zustandsautomaten“ (Abstract State Machine (ASM)), ein Formalismus zur Modellierung von Algorithmen.

Börger et al.

Pinheiro da Silva [Pin01] spezifiziert die Semantik von Aktivitätsdiagrammen auf formale Weise durch LOTOS, eine von der International Standardization Organisation (ISO) entwickelte Sprache zur Spezifikation von Struktur- und Verhaltensaspekten eines Softwaresystems.

Pinheiro da Silva

Eshuis [Esh02] redefiniert die Syntax von Aktivitätsdiagrammen mit Hilfe sogenannter Aktivitäts-Hypergraphen. Ein Aktivitäts-Hypergraph ist ein Graph, der als Knoten Aktivitäten und als Kanten Hyperkanten besitzt. Eine Hyperkante entspricht einer zusammengesetzten UML-Transition (compound transition, [UML1.5], S. 2-158). Eshuis definiert direkt auf dieser Syntax die Semantik. D.h. es findet keine Abbildung von der Syntax einer Sprache in die einer anderen statt. Es werden gleich zwei Semantiken auf Basis von getakteten Transitionssystemen (Clocked Transition System (CTS)) definiert: Einmal unter der Annahme, dass Systeme so schnell reagieren, dass das Verarbeiten eines Ereignisses keine Zeit verbraucht (perfect Technology), und unter der Annahme, dass das System eine gewisse Zeit zur Reaktion braucht. Bei der ersten Annahme kann man sich auf die Anforderungen des Systems konzentrieren, die zweite Annahme konzentriert sich auf die Performanz des Systems.

Eshuis

Zwei weitere Ansätze, die wir hier nicht weiter detailliert betrachten, sind der Ansatz von Apvrille (Abbildung von Aktivitätsdiagrammen auf LOTOS (Language of Temporal Ordering Specifications), eine Spezifikationstechnik zur Beschreibung nebenläufiger und verteilter Prozesse) [ASL01], und der Ansatz von Bolton und Davies (Abbildung von Aktivitätsdiagrammen auf die Prozessalgebra Communicating Sequential Processes (CSP)) [BD00].

Tabelle 2-3 zeigt die verschiedenen Ansätze in einer tabellarischen Übersicht.

Ansatz	Syntaktische Domäne	Semantische Domäne	Formalität
Gehrke et al.	Konkrete Syntax	Petri-Netze	informal
Börger et al.	Konkrete Syntax	Abstract State Machines	formal
Pinheiro da Silva	Konkrete Syntax	LOTOS	formal
Eshuis	Aktivitätshypergraphen	Clocked Transition System, Z	formal

Tabelle 2-3: Ansätze zur Formalisierung von UML-Aktivitätsdiagrammen

Fazit Für die Formalisierung der Semantik von UML gibt es derzeit eine Vielzahl von Ansätzen. Diese beschäftigen sich stets nur mit einem Teilbereich der UML. Dies hat für unsere Arbeit keinen Einfluss, da wir uns ohnehin nur auf das Konzept der Aktivitätsdiagramme konzentrieren.

In der Regel wird im Rahmen der Semantikformalisierung die Syntax der Ausgangssprache (mit der informalen Semantik) auf die Syntax einer Zielsprache (mit formaler Semantik) abgebildet. Durch diese Syntaxabbildung kann man die formale Semantik der Zielsprache auf die Ausgangssprache übertragen. Die Ansätze zur Formalisierung von Aktivitätsdiagrammen verwenden als syntaktische Domäne meistens die konkrete Syntax, d.h. die Notation der UML. Dies ist insofern problematisch, da Eigenschaften der abstrakten Syntax, die nicht durch die konkrete Syntax erfasst werden, unberücksichtigt bleiben. Diese Vorgehensweise verwenden fast alle Autoren bis auf Eshuis. Gehrke et al. benutzen nur Beispiele, um die Abbildung von UML auf Petri-Netze zu erläutern. In dem von uns gewählten Ansatz verwenden wir die relevanten UML-Metamodellelemente als syntaktische Domäne, d.h. wir leiten die grammatischen Regeln direkt vom UML-Metamodell ab.

Für die Definition der formalen Semantik werden verschiedenartige formale Konzepte wie Petri-Netze, Abstract State Machines, Z usw. eingesetzt. Als semantische Domäne verwenden wir Petri-Netze, da uns diese am besten geeignet scheinen, die Semantik von Aktivitätsdiagrammen auszudrücken (s. hierzu auch Abschnitt 5.3.4).

2.4 Vorgehensmodelle in der Softwareentwicklung – von rigide bis agil

Rigidität und Agilität In den folgenden beiden Abschnitten betrachten wir verschiedene Ausprägungen von Vorgehensmodellen, von schwergewichtigen Vorgehensmodellen bis hin zu leichtgewichtigen Vorgehensmodellen. Die aus der Komplexität und Rigidität resultierenden negativen Erfahrungen mit schwergewichtigen Vorgehensmodellen führten zur Hinwendung zu den schlankeren, leichtgewichtigeren Vorgehensmodellen, Methoden und Verfahren. Diese sogenannten agilen Methoden⁶ enthalten sehr viel weniger Vorschriften als schwergewichtige Vorgehensmodelle, bieten im Gegenzug aber mehr Flexibilität. Die Befürwortung von agilen Prozessen liegt darin begründet, dass nicht wenige Kritiker schwergewichtige Vorgehensmodelle in vielen Fällen als kontraproduktiv empfinden (s. [MS99] für eine tiefgehendere Betrachtung), weil sie die Flexibilität eines Unternehmens durch eine übermäßige Bürokratie hemmen.

Schwergewichtige Vorgehensmodelle Die bekanntesten Vertreter der schwergewichtigen Vorgehensmodelle sind das im deutschsprachigen Raum verbreitete V-Modell 97 und der international bekannte Rational Unified Process (RUP). Weitere schwergewichtige Vorgehensmodelle sind das mit dem V-Modell 97 vergleichbare schweizerische Vorgehensmodell HERMES der öffentlichen Hand für Softwareprojekte [Hermes], das britische Vorgehensmodell Prince2 der öffentlichen Hand für Projektmanagement [Prince2], das Vorgehensmodell CMII für den Konfigurationsmanagementprozesse [CMII] und der im Auftrage der britischen Regierung entwickelte Leitfaden für Servicemanagement ITIL (IT Infrastructure Library) [ITIL].

6. Wir verzichten hier auf die Nutzung des Begriffs „Methodologie“, wie sie manchmal (z.B. in [CH02a]) in Zusammenhang mit Agilität verwendet wird, und sprechen von agilen Vorgehensmodellen bzw. Methoden.

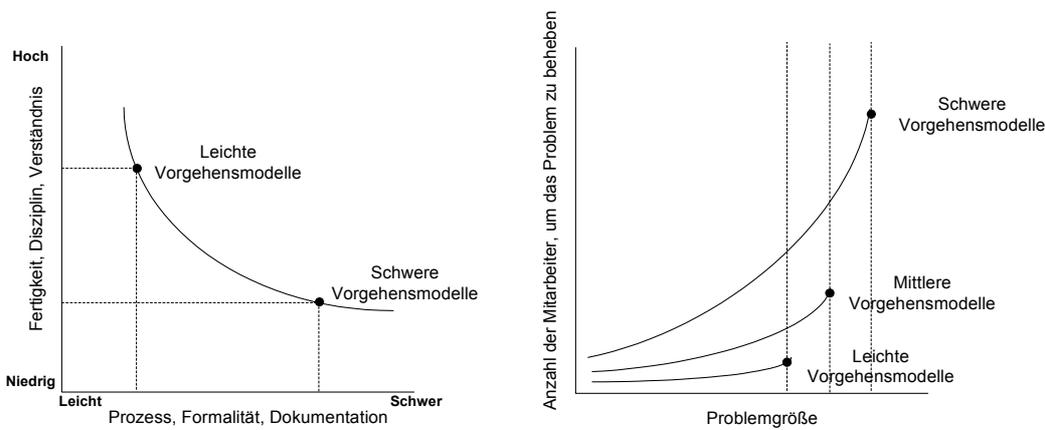


Abbildung 2-12: Gegenüberstellung von leichten und schweren Vorgehensmodellen (aus [CH02a])

Das Extreme Programming zählt eher zu den leichten Vorgehensmodellen bzw. Methoden. Leichte Vorgehensmodelle erfordern mehr Fertigkeiten, Disziplin und Verständnis der Projektmitarbeiter, während bei schweren Vorgehensmodellen Prozesse, Formalität und die Dokumentation von Bedeutung sind (Abbildung 2-12, links). Leichte Vorgehensmodelle sollten zur Lösung kleinerer Problemstellungen eingesetzt werden. Hierbei sind weniger Mitarbeiter nötig als bei schweren Vorgehensmodellen, die komplexere Problemstellungen lösen können, dafür aber mehr Mitarbeiter benötigen (Abbildung 2-12, rechts).

Leichtgewichtige Vorgehensmodelle bzw. Methoden

Sowohl leichte als auch schwere Vorgehensmodelle können agil sein. Die Bedeutung des Begriffs Agilität wird gerne mit dem Motto „light but sufficient“ erläutert: Das Vorgehensmodell soll so wenig wie möglich vorgeben (light), dabei aber das Projekt noch ausreichend unterstützen (sufficient). Was genau „so wenig wie möglich“ und „ausreichend“ bedeutet, hängt von der jeweiligen Projektsituation ab. Schwere Vorgehensmodelle wie z.B. der RUP können also unter Umständen agil sein. Genau dieser Sachverhalt charakterisiert Vor- und Nachteile des Konzepts der Agilität. Denn was genau „light but sufficient“ ist, kann nur mit viel Erfahrung und Fingerspitzengefühl herausgefunden werden.

Agilität

Die Verbreitung und Weiterentwicklung der agilen Methoden wird durch die Agile Alliance, gebildet von Gründern und Anwendern agiler Methoden, vorangetrieben. Die Agile Alliance verfasste das Agile Software Development Manifesto, das Forderungen an die agile Softwareentwicklung in Gestalt eines Wertesystems (Tabelle 2-4) und einer Menge von Prinzipien, die dieses Wertesystem stützen, beschreibt [AA02]:

Agile Alliance

Individuen und deren Interaktion	wichtiger als	Prozesse und Werkzeuge
Erarbeitung der Software	wichtiger als	Erarbeitung umfassender Dokumentation
Zusammenarbeit mit dem Kunden	wichtiger als	Vertragsverhandlungen
Fähigkeit, auf Änderungen zu reagieren	wichtiger als	Einhaltung von Plänen

Tabelle 2-4: Wertesystem der Agile Alliance

Prinzipien der Agilität

Zusätzlich zu dem Wertesystem einigte sich die Agile Alliance auf eine Menge von Prinzipien, die die Ziele und Eigenschaften agiler Methoden kennzeichnen [AA02]:

- Zufriedenstellung des Kunden durch frühe und häufige Auslieferung hochwertiger Software
- Auslieferung der Software in kurzen Intervallen von ein paar Wochen bis hin zu drei bis vier Monaten
- Erstellung von Software als Maß für den Projektfortschritt
- Bereitschaft und Fähigkeit, geänderte Anforderungen jederzeit – auch noch zu Ende des Projekts – aufzunehmen
- Enge, tägliche Zusammenarbeit zwischen Fachbereichsmitarbeitern und Entwicklern
- Durchführung von Projekten mit motivierten Mitarbeitern in einer geeignet ausgestatteten Arbeitsumgebung. Autonomie der Projekte ohne Störung durch die Unternehmensleitung
- Vorrangig Nutzung persönlicher Face-to-Face-Kommunikation
- Selbstorganisierte Teams
- Fokussierung auf ein gutes – wenn auch inkrementell erstelltes – Design
- Agile Prozesse als Grundstein für eine eine konstante Arbeitsbelastung
- Einfachheit⁷ als das wesentliche Prinzip der agilen Softwareentwicklung.
- Teams reflektieren regelmäßig ihre Arbeitsweise und passen sie entsprechend der Erkenntnisse an.

Die bekanntesten Vertreter der agilen Methoden und Vorgehensmodelle sind das Extreme Programming (XP) und die Crystal Methodologies. Weitere agile Methoden sind das Adaptive Software Development (ASD) [Hig99] bzw. [CH02a], SCRUM [CC02], das Feature Driven Development (FDD) [Coa99] und das Dynamic System Development (DSDM) [Sta97].

Bewertungskriterien

Nachfolgend stellen wir einige Vertreter schwer- und leichtgewichtiger Vorgehensmodelle bzw. Methoden vor. Wir beurteilen sie anhand ihrer Möglichkeiten, die Komplexität von Softwareentwicklungsprozessen zu beherrschen und dem Anwender begreifbar zu machen. Komplexität lässt sich durch Aufbrechen des Ganzen in eine modulare, sinnvolle Struktur

7. Einfachheit bedeutet in diesem Kontext Zweckmäßigkeit und Klarheit. Ein zweckmäßiges Design ist zwar schwieriger zu erstellen als ein umständliches Design, aber einfacher zu ändern und damit agiler.

vereinfachen. Von unserem Interesse sind daher die Möglichkeiten, Prozesssequenzen, Prozesshierarchien, Prozessvarianten und Prozessverfeinerungen darstellen zu können. Denn diese Prozessbeziehungen sind wichtig, um ein Vorgehensmodell übersichtlich, offen und flexibel zu gestalten. Übersichtlich bedeutet, dass Abhängigkeiten zwischen Prozessen aufgezeigt werden können, offen, dass durch die definierten Relationen neue Prozesse hinzugefügt werden können und flexibel, dass die Anwender während eines Projekts einen passenden Prozess auswählen können.

2.4.1 V-Modell 97

Das V-Modell 97 [VM97] wurde ursprünglich für den Bereich der deutschen Bundeswehr und der Bundesverwaltung geschaffen, wird mittlerweile aber auch von vielen Unternehmen als Standard eingesetzt. Die Einführung des V-Modell 97 dient oft auch als vorbereitende Maßnahme zur ISO-Zertifizierung eines Unternehmens, da das V-Modell 97 die Mehrheit der ISO 9001 Anforderungen erfüllt. Zur Zeit wird an einer neuen Version des V-Modells, V-Modell 200x, gearbeitet (f. aktuelle Informationen s. [WEIT], Fertigstellung voraussichtlich Mitte 2005).

Ursprung und Ziele

Der frei verfügbare „Softwareentwicklungsstandard der Bundeswehr“ beschreibt Aktivitäten und Produkte, die während der Entwicklung von Software durchzuführen bzw. zu erstellen sind. Zusätzlich werden Methoden angegeben und Anforderungen an Werkzeuge definiert, mit denen die Aktivitäten ausgeführt werden können. Bzgl. der Entwicklungsstrategie ist das V-Modell 97 generisch. In der zugehörigen Handbuchsammlung werden jedoch verschiedene Entwicklungsstrategien wie z.B. inkrementelle Entwicklung, Grand Design, Einsatz von Fertigprodukten, Objektorientierte Entwicklung, Entwicklung wissensbasierter Systeme, Software-Pflege und -Änderung betrachtet. Die inkrementelle Entwicklung gilt als Regelfall.

Gegenstand der Beschreibung

Das V-Modell 97 gliedert sich in die vier thematischen Bereiche („Submodelle“) Projektmanagement, Qualitätssicherung, Systemerstellung und Konfigurationsmanagement, die miteinander verknüpft sind. Jedem dieser Submodelle wird eine Menge von Aktivitäten und Produkten zugeordnet. Die Aktivitäten werden in einem einfachen Kontrollflussdiagramm miteinander verknüpft. Über den sogenannten „Produktfluss“ wird für jede Aktivität angegeben, welche Produkte in welchem Zustand von welchen anderen Aktivitäten geliefert werden und welche Produkte in welchem Zustand erzeugt und für welche Aktivitäten geliefert werden (Abbildung 2-13).

Struktur

von		Produkt	nach	
Aktivität	Zustand		Aktivität	Zustand
SE 1	akzeptiert	Anwenderforderungen	—	—
SE 2.1	in Bearb.	Systemarchitektur	—	—
—	—	Systemarchitektur. Anforderungszuordnung	SE 1, SE 2.5, SE 2.6, SE 3, SE 4-SW, PM 4, PM 5	vorgelegt

Abbildung 2-13: Produktflusses im V-Modell 97

Das V-Modell besitzt eine implizite und sehr flache, ein- bis zweistufige Prozesshierarchie. Für jedes Submodell wird ein Kontrollfluss der Aktivitäten angegeben, es gibt jedoch keine Darstellung von Aktivitätsabfolgen über mehrere Submodelle hinweg. In jeder Aktivität wird allerdings für jedes Produkt angegeben, von welcher Aktivität dieses erzeugt bzw. modifiziert wurde und welche Aktivität das Produkt weiterverarbeitet. Die Darstellung der Aktivität-/Produktabhängigkeit kommt unserer Auffassung von Prozesssequenzen recht nahe. Weder Prozessverfeinerungen noch -varianten werden vom V-Modell berücksichtigt.

2.4.2 Rational Unified Process

Fokus: Objektorientierte Entwicklung

Der Rational Unified Process (RUP) ist ein käuflich zu erwerbendes Produkt von Rational [RUP]. Der RUP beschreibt wie das V-Modell 97 Aktivitäten und Artefakte (d.h. Ergebnisse) von Softwareentwicklungsprozessen. Der RUP sieht als Entwicklungsschema die iterative inkrementelle Entwicklung vor. Die vier Phasen des RUPs werden jeweils in mehreren Iterationen, die Aktivitäten aus allen Disziplinen umfassen, durchlaufen. Der RUP ist speziell auf die objektorientierte Softwareentwicklung und die Modellierungssprache UML ausgerichtet. Hierdurch ist der RUP weniger generisch als das V-Modell 97. Trotzdem oder gerade deswegen wird der RUP international bei vielen Unternehmen eingesetzt.

Beschreibung von Disziplinen

Abbildung 2-14 zeigt, wie Disziplinen im RUP graphisch beschrieben werden. Hierzu verwendet der RUP ein leicht angepasstes UML-Aktivitätsdiagramm. Auf die Darstellung von Objekten wird verzichtet. Die Grafik wird durch eine natürlichsprachige Beschreibung ergänzt. Desweiteren wird natürlichsprachig angegeben, welche Objekte von der Disziplin produziert werden. Hierbei werden auch Zwischenergebnisse angegeben. Allerdings gibt es in dieser Art von Diagramm keine Angabe, welche Artefakte von einem Workflowdetail konsumiert und produziert werden. Hierzu muss sich der Anwender die Beschreibung des Workflowdetails ansehen (Abbildung 2-15).

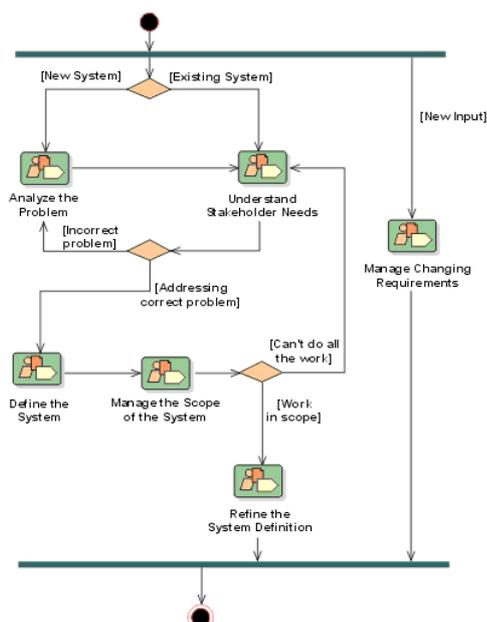


Abbildung 2-14: Disziplin „Requirements“ (aus [RUP])

Abbildung 2-15 zeigt, wie Workflowdetails im RUP am Beispiel des Workflowdetails „Analyze the Problem“ graphisch beschrieben werden. Auffällig ist die fehlende Angabe von Aktivitätsreihenfolgen. Z.B. ist unklar, ob die Aktivität „Capture a Common Vocabulary“ vor der Aktivität „Develop Requirements Management Plan“ ausgeführt werden soll. Ferner ist unklar, wie Input- und Outputartefakte des Workflowdetails mit den Input- und Outputartefakten der Aktivitäten und der übergeordneten Disziplin korrespondieren. Z.B. sind in der Beschreibung der Aktivität „Capture a Common Vocabulary“ die Inputartefakte „Use-Case-Model“, „Vision“, „Business Case“ usw. angegeben (Abbildung 2-16). In der Ansicht des Workflowdetails erscheint jedoch das Artefakt „Business Case“ gar nicht. Auch ist hier nicht erkennbar, dass das „Use-Case Model“ Input von der Aktivität „Capture a Common Vocabulary“ ist. Die Angabe der Input- und Outputartefakte ist in dieser Sicht also nicht vollständig.

Beschreibung von
Workflowdetails

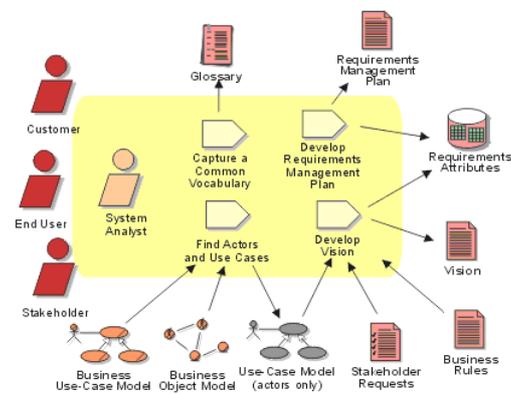


Abbildung 2-15: Workflow Detail „Analyze the Problem“ (aus [RUP])

Abbildung 2-16 zeigt, wie Aktivitäten im RUP am Beispiel der Aktivitäten „Capture a common Vocabulary“ und „Develop Vision“ beschrieben werden. Diese Beschreibung ist recht detailliert und umfasst den Zweck, Aktivitätsschritte, Input- und Outputartefakte, die ausführende Rolle, Hinweise für einen möglichen Werkzeugeinsatz und die Zuordnung zu Workflowdetails.

Beschreibung von
Aktivitäten

Die Abschnitte „Input- und Outputartefakte“ enthalten auch optionale Artefakte (z.B. „Business Use-Case Model“ und „Business Object Model“, s. Abbildung 2-16). Dies ist für den Anwender nur durch Lektüre der natürlichsprachigen Beschreibung der Aktivität erkennbar. Diese Art der Modellierung birgt die Gefahr, dass Anwender alle Artefakte einer Aktivität als „Muss“-Artefakte betrachten (z.B. „Business Use-Case Model“).

Ferner enthalten die Abschnitte „Input- und Outputartefakte“ Zwischenergebnisse. Dies ist meistens dann der Fall, wenn ein Artefakt erstellt wird (z.B. Use-Case Model), welches sich aus anderen Artefakten zusammensetzt (z.B. Use Case, Actor). Da die Komposition von Artefakten im RUP nicht explizit angegeben wird, kann dies zur Irritationen des Anwenders führen (der z.B. denkt, er hätte drei verschiedenen Artefakte erzeugt).

Die Angabe der Aktivitätsschritte ist nicht zwingend sequentiell, d.h. hier wird keine Reihenfolge angegeben. Diese Information ist im RUP nur implizit vorhanden. Hierzu müsste der Anwender sich die Input- oder Outputartefakte einer Aktivität merken und dann alle Aktivi-

täten suchen, die auf diese Input- oder Outputartefakte passen (entsprechend der Sequence-Beziehungsdefinition, Definition 3-16). Hierdurch bleibt dem Anwender der wichtige Hinweis vorenthalten, was er als nächstes tun kann bzw. was er zunächst tun muss, um die gewünschte Aktivität durchführen zu können. Beispielsweise kann ein Anwender nach Durchführung des Prozesses „Develop Vision“ den Prozess „Develop Requirements Management Plan“ durchführen. Eine solche Information ist im RUP nicht vorhanden.

Gerade weil im RUP die Aktivitäten, Workflowdetails und Disziplinen eng miteinander verzahnt sind, ist eine solche informative Darstellung von Abfolgen hilfreich. Beispielsweise wird im Workflow der Disziplin „Requirements“ suggeriert, dass nach dem Workflowdetail „Analyze the Problem“ das Workflowdetail „Understand Stakeholder Needs“ ausgeführt werden kann (Abbildung 2-14). Dies ist jedoch nicht der Fall. Stattdessen wirken eine Vielzahl anderer Workflowdetails bzw. deren Aktivitäten an der Erstellung von Artefakten mit, die für die Ausführung des Workflowdetails „Understand Stakeholder Needs“ notwendig sind.

Purpose	
<ul style="list-style-type: none"> ■ To define a common vocabulary that can be used in all textual descriptions of the system, especially in use-case descriptions. 	
Steps	
<ul style="list-style-type: none"> ■ Find Common Terms ■ Evaluate Your Results 	
Input Artifacts:	Resulting Artifacts:
<ul style="list-style-type: none"> ■ Use-Case Model ■ Vision ■ Business Case ■ Business Use-Case Model ■ Business Object Model ■ Stakeholder Requests ■ Use Case ■ Business Rules 	<ul style="list-style-type: none"> ■ Glossary
Role: System Analyst	
Tool Mentors:	
<ul style="list-style-type: none"> ■ Capturing a Common Vocabulary Using Rational RequisitePro 	
Workflow Details:	
<ul style="list-style-type: none"> ■ Requirements <ul style="list-style-type: none"> ■ Analyze the Problem ■ Define the System ■ Understand Stakeholder Needs 	

Abbildung 2-16: RUP-Aktivität „Capture a Common Vocabulary“ (aus [RUP])

Keine Angabe von
Prozess-
verfeinerungen und
-varianten

Der RUP bietet keine Möglichkeiten zur Beschreibung von Prozessverfeinerungen oder Prozessvarianten an. Dies ist insofern ein Nachteil, da der Anwender auf ein bestimmtes Vorgehen festgelegt ist. Es gibt für ihn keine Möglichkeit, Prozessverfeinerungen oder Prozessvarianten als Prozessmodelle in den RUP zu integrieren.

2.4.3 Extreme Programming

Extreme Programming (XP) wurde in den 90er Jahren von Kent Beck entwickelt und ist der bekannteste Vertreter der agilen Methoden ([Bec00], [JHA01], [XP]). XP versucht mit wenigen, aber verpflichtenden Regeln und Dokumenten auszukommen. Vom Mitarbeiter wird dabei ein hohes Maß an Disziplin erwartet. Der Feedbackkreislauf ist bei XP besonders kurz und den Testern bzw. den Kunden werden früher als sonst üblich halbfertige Produkte präsentiert.

Ziele

Folgende Paradigmen kennzeichnen das Extreme Programming:

Paradigmen

- Kleine, räumlich nahe Teams (3-10 Entwickler)
- Intensive, informale Kommunikation innerhalb des Teams und zwischen Team und Kunde
- Inhaltliche Reduktion der zu erstellenden Zwischenprodukte auf das notwendigste
- Iterative Entwicklung (Richtlinie: dreiwöchige Iterationen)
- Entwicklung und Verwendung von vom Kunden spezifizierter „User Stories“, die die im Rahmen einer Iteration zu realisierenden Funktionalität beschreiben
- Pairprogramming (Zwei Entwickler arbeiten zusammen an einem Rechner)
- Unit-Tests bei jedem Checkin
- Integration mehrmals am Tag
- Kontinuierliche Verbesserung des Codes (sog. Refactoring)
- Tägliches „Stand-up“-Statusmeeting (Durch das Stehen sollen Meetings möglichst kurz geraten.)

Da das XP nur einige wenige Aktivitäten besitzt, gibt es keine Prozesshierarchien. Rudimentäre Prozesssequenzen sind durch Angabe der sogenannten XP-Maps festgelegt. Es gibt weder Prozessverfeinerungen noch -varianten.

Bewertung

2.4.4 Crystal Family

Crystal Family ist eine Familie von Methoden, deren Mitglieder mit verschiedenen Farben bezeichnet werden: Crystal Clear, Crystal Yellow, Crystal Orange (Web), Crystal Red, Crystal Magenta, Crystal Blue und Crystal Violet [Crystal]. Die Methoden werden anhand der Anzahl der Mitarbeiter, der Systemkritikalität und der Projektprioritäten eingeordnet. Die Farbe dient als Kennzeichen für die jeweilige Leichtigkeit oder Schwere der Methode. D.h. Crystal Clear ist z.B. eine leichte Methode, geeignet für Projekte von 1- 6 Mitarbeitern, mit eher geringem Risiko. Crystal Yellow ist „schwerer“ als Crystal Clear, da sie für bis zu 20 Mitarbeitern ausgelegt ist und auch Projekte beherrschen können soll, bei denen es um große Geldbeträge geht, deren Verlust für das Unternehmen existenzbedrohend ist. Eine Einordnung der verschiedenen Methoden findet sich in Anhang A.4.2.

Ziele

Die Beschreibung der einzelnen Methoden ist recht vage und geschieht meistens anhand von Projektbeispielen. Eine systematische Darstellung lässt der Autor Alistair Cockburn hier leider vermissen. „Crystal Clear“ wird anhand einer Menge von Rollen, einer Menge einzuhaltender Regeln, einer Menge von Ergebnissen und einer Menge von Werkzeugen spezifiziert. Prozesse im Sinne von schrittweisen Abfolgen werden nicht beschrieben. Die Methodologien Crystal Red, Crystal Magenta, Crystal Blue und Crystal Violet werden gar nicht näher

Bewertung

erläutert. Auch fehlt für diese Methoden die Zuordnung zu den Projekten. Für lebenskritische Systeme können noch keine Aussagen über geeignete Methoden getroffen werden. Crystal Family kennt ferner weder Prozesshierarchien, -verfeinerungen noch -varianten.

2.4.5 Fazit

Schwergewichtige Vorgehensmodelle sind komplex

Komplexe Vorgehensmodelle wie das V-Modell 97 und der RUP wurden in den letzten zehn Jahren vermehrt zur Organisation der Softwareentwicklungsprozesse eingesetzt. Die ersten Erfahrungen mit diesen Vorgehensmodellen führten zu einer Ernüchterung, da die Vorgehensmodelle wegen ihrer Abstraktheit und Unübersichtlichkeit oft schwer umzusetzen waren und die Bürokratie förderten. Gerade für kleine Software-Unternehmen erscheinen diese Vorgehensmodell als überdimensioniert. Ein Grund hierfür ist, dass die oben genannten Modelle für einen erfolgreichen Einsatz eine gewisse Unternehmensgröße erfordern, die Raum bietet für Einrichtungen wie z. B. das Change Control Board im Änderungsmanagement. Ferner droht durch die Vielzahl und die Komplexität der zu erfüllenden Forderungen die Einschränkung der Flexibilität, der Agilität und der Kommunikationsfähigkeit von Unternehmen. Mechanismen zur Auflösung der Komplexität wie die Entflechtung der Prozesse durch Identifikation von Prozesssequenzen, -varianten, -hierarchien und -verfeinerungen existieren nur teilweise (Tabelle 2-5).

Komplexität verhindert Agilität

Die Erhaltung der Flexibilität, der Agilität und der Kommunikationsfähigkeit ist für viele Software-Unternehmen sehr wichtig. Wegen des zunehmenden Wettbewerbs- und Zeitdrucks ist schnelles Handeln wirtschaftlich betrachtet lebensnotwendig. Trotzdem will der Kunde ein qualitativ hochwertiges Software-Produkt erhalten, was ein systematisches Vorgehen voraussetzt. Die meisten Unternehmen praktizieren jedoch einen intuitiven Softwareentwicklungsprozess (d.h. die Prozesse sind nicht definiert, dokumentiert, analysiert, bewertet und verbessert). Hierdurch sind Entscheidungen nicht nachvollziehbar; der langfristige Unternehmenserfolg ist in Frage gestellt [Mer99a].

Ohne Agilität kein Unternehmenserfolg

Software-Unternehmen sehen sich also heute mit zwei konkurrierenden Forderungen konfrontiert: Zum einen sind Produkte in schnelleren Zyklen zu entwickeln, was automatisch zu einer Vernachlässigung von Qualitätsaspekten führt. Die Herstellung der Kundenzufriedenheit wird somit dem Zufall überlassen. Zum anderen wollen Unternehmen langfristig die Kundenzufriedenheit und den Unternehmenserfolg sichern. Wenden solche Unternehmen traditionelle Vorgehensmodelle an, kann sich dies unter Umständen unproduktiv auf die Unternehmensleistung auswirken (vgl. mit [MS99]). Die Kritik an traditionellen Vorgehensmodellen drückt sich auch in der Verwendung negativer Adjektive wie „schwer“ oder „schwergebig“ aus.

Leichtgewichtige Vorgehensmodelle als Alternative

Für Unternehmen, deren Prozesse sich häufig ändern, die auf sich ändernde Anforderungen schnell reagieren können müssen und die immer wieder neue Produkte entwickeln, sind die leichtgewichtigen Methoden eine Alternative zu den schwergewichtigen Vorgehensmodellen. Insgesamt ist die definitorische Basis der leichtgewichtigen Vorgehensmodelle jedoch eher als schwach einzustufen. Ob das Wertesystem und die Prinzipien ausreichen, um die Eigenschaften leichtgewichtiger Vorgehensmodelle zu kennzeichnen, ist im Moment noch in Frage zu stellen. Offen ist zur Zeit auch noch, ob die leichtgewichtigen Vorgehensmodelle auch den gewünschten Erfolg bringen. Denn da die leichtgewichtigen Vorgehensmodelle noch jung sind, fehlt noch eine breite Erfahrungsbasis. Über die Unterstützung von lebenskritischen oder sicherheitskritischen Systemen fehlt ferner noch jegliche Aussage.

Bei den leichtgewichtigen Vorgehensmodellen wurde die Komplexität durch Konzentration auf das Wesentliche bereits reduziert. Bei ihnen stellt sich eher die Frage, ob der Anwender durch die Angabe von Regeln ausreichend unterstützt ist. Wie in den vorhergehenden Abschnitten gezeigt wurde, ist die Beschreibung der leichtgewichtigen Vorgehensmodelle eher vage. Hier droht eher die Gefahr, dass der Anwender durch unpräzise Regeln alleine gelassen wird. Durch die Beschreibung der wenigen Prozesse und Regeln durch Prozessmuster könnte diese Vagheit vermieden werden.

Schwammige Beschreibung verhindert Agilität

Prozessmuster erlauben die flexible Durchführung von Prozessen je nach Erfordernis der Projektsituation. Sie können daher den notwendigen Unterbau für die Anwendung agiler, aber auch rigider Vorgehensmodelle bzw. Methoden bilden. Rigide Vorgehensmodelle werden durch die Beschreibung mit Prozessmustern entflochten, d.h. die Einsetzbarkeit eines Prozesses kann gegebenenfalls leichter beurteilt werden. Prozessmuster können ferner dazu dienen, agile Vorgehensmodelle mit einer geeigneten Beschreibungsweise auszustatten. Agile Methodologien fordern, flexibel auf Änderungen zu reagieren. Hierfür müssen auch die Arbeitsprozesse flexibel sein. Prozessmuster helfen, diese Arbeitsprozesse zu flexibilisieren.

Agilität durch Prozessmuster

Vorgehensmodell/ Methode	Prozesshierarchie	Prozessesequenzen darstellbar	Prozessverfeinerungen	Prozessvarianten	Komplexität
V-Modell 97	nur implizit, vorgegeben, flach (Submodell, Aktivitäten)	eingeschränkt, pro Produkt per Produktfluss ersichtlich und im Kontrollfluss des Submodells	nein	nein	hoch
RUP	nur implizit, vorgegeben, 3-stufig (Discipline, Workflow Detail, Activity)	eingeschränkt, pro Artefakt konsumierende und produzierende Aktivitäten, pro Aktivität In- und Outputartefakte	nein	nein	hoch
Extreme Programming	nein	per XP-Map wird Abfolge der XP-Prozesse angezeigt	nein	nein	niedrig, abhängig von Projektkomplexität
Crystal Family	nein	keine Prozesse vorgegeben	nein	nein	niedrig, abhängig von Projektkomplexität

Tabelle 2-5: Vergleich verschiedener Vorgehensmodelle bzw. Methoden

3 Konzepte der Process Pattern Description Language PROPEL

In diesem Kapitel erläutern wir die grundlegenden Begriffe und Konzepte der Process Pattern Description Language PROPEL. Das Kapitel dient dazu, den Leser mit den Konzepten von PROPEL vertraut zu machen. Die Formalisierung der Begriffe und Konzepte findet in den nachfolgenden Kapiteln 4 und 5 statt. Dort wird die Syntax und Semantik von PROPEL definiert.

In den folgenden Abschnitten geben wir ein Klassifizierungsschema für Prozessmuster an (Abschnitt 3.1). Dabei grenzen wir Prozessmuster von Ergebnismustern ab. Anschließend definieren wir den Begriff des Prozessmusters (Abschnitt 3.2). Für die Kombination von Prozessmustern benötigt man die Definition von Prozessmusterbeziehungen (Abschnitt 3.3). Die auf diese Weise definierten Konzepte fassen wir in Gestalt des Prozessmusterschemas zusammen, das zur einheitlichen Beschreibung von Prozessmustern dient (Abschnitt 3.4).

3.1 Klassifizierung von Prozessmustern

Die Klassifizierung dient dazu, ein Prozessmuster anhand verschiedener Kriterien einordnen zu können. Die Suche und Auswahl von Prozessmustern wird hierdurch vereinfacht. Eine einheitliche Klassifizierung von Prozessmustern und Ergebnismustern steht im wissenschaftlichen Diskurs derzeit noch aus (Abschnitt 2.1.6).

Ziel: Leichtere Handhabung

Zunächst teilen wir (analog zu Störrle [Stö01a]) Muster in zwei Klassen ein, nämlich in die Klasse der Ergebnismuster und in die Klasse der Prozessmuster. In dieser Arbeit beschäftigen wir uns ausschließlich mit Prozessmustern. In Abschnitt 3.2 untersuchen wir im Detail, wie ein Prozessmuster aufgebaut ist.

Prozessmuster und Ergebnismuster

Definition 3-1: Prozessmuster

Ein Prozessmuster beschreibt einen bewährten Prozess, um ein Problem, das in einem bestimmten Kontext wiederholt aufgetreten ist, zu lösen.

□

Prozessmuster präsentieren den Weg zur Lösung des Problems. Ein Beispiel für Prozessmuster sind die „Architectural Modeling Patterns“ von Störrle [Stö00], welche Prozesse für die komponentenbasierte Architekturmodellierung beschreiben.

Definition 3-2: Ergebnismuster

Ein Ergebnismuster beschreibt in seiner Lösung ein Prozessresultat, welches ein bestimmtes Problem in einem bestimmten Kontext löst. Der Prozess selbst bleibt bei Ergebnismustern stets unberücksichtigt, d.h. im Vordergrund steht das Ergebnis.

□

Ergebnismuster

Ergebnismuster präsentieren die gesuchte Lösung z.B. in Form eines Softwareentwurfs oder einer Gebäudeskizze. Ergebnismuster sind beispielsweise die GoF-Patterns [GHJ96], welche Softwareentwürfe beschreiben. Ergebnismuster beschreiben also eher das „Was“ (z.B. Project Plan Pattern), während Prozessmuster das „Wie“ (z.B. Project Planning Pattern) beschreiben. Jedes Muster einer Domäne (z.B. Project Management) gehört also entweder zur Klasse der Prozessmuster oder der Ergebnismuster. Da wir uns in dieser Arbeit nur mit Prozessmustern beschäftigen, vernachlässigen wir die Klassifizierung von Ergebnismustern.

Für Prozessmuster definieren wir drei weitere Klassifikationsmerkmale, nämlich Domäne, Phase und Aspekt:

Definition 3-3: Domäne

Die Domäne bezeichnet den fachlichen Einsatzbereich von Prozessmustern.

□

Fokus auf Domäne
Softwareent-
wicklung

Beispiele für Domänen sind Architektur, Softwareentwicklung, Lehren und Lernen, Organisationslehre (z.B. [OP]), Pädagogik (z.B. [PPP]), Requirements Management (z.B. [Whi95]), Project Management (z.B. [PMP]) und Workflow Management (z.B. [ABT00]). In dieser Arbeit beschäftigen wir uns ausschließlich mit Prozessmustern der Domäne Softwareentwicklung.

Definition 3-4: Phase

Die Phase beschreibt, in welchem Projekt- oder Arbeitsabschnitt ein Prozessmuster eingesetzt werden kann.

□

Beispiele

Beispiele für Phasen sind die RUP-Phasen Konzeption, Entwurf, Realisierung und Nutzungsübergang [RUP].

Definition 3-5: Aspekt

Der Aspekt eines Prozessmusters gibt Aufschluss über die thematische Ausrichtung des Prozessmusters.

□

Beispiele

In jeder Phase eines Projekts können Prozessmuster mit verschiedenen thematischen Aspekten zum Einsatz kommen. Beispiele für Aspekte sind: Projektanbahnung, Projektmanagement, Risikomanagement, Konfigurationsmanagement, Releasemanagement,

Changemanagement, Qualitätssicherung, Errormanagement, Anforderungsmanagement, Design, Softwareentwicklung, Reuse Management, System Management, Integration Engineering, Usability Engineering, Content Management, Inbetriebnahme und Administration. Desweiteren kann ein Prozessmuster einem Prozessmusterkatalog zugeordnet werden.

Definition 3-6: Prozessmusterkatalog

Ein Prozessmusterkatalog umfasst das explizierte prozessuale Wissen über ein bestimmtes Themengebiet. Er repräsentiert eine Menge von Prozessmustern und eine Menge von Prozessmusterbeziehungen, die die Prozessmuster miteinander verknüpfen. Mathematisch betrachtet handelt es sich bei einem Prozessmusterkatalog also um einen gerichteten Graphen. Die Knoten des Graphen sind Prozessmuster, während die Kanten Beziehungen zwischen Prozessmustern repräsentieren.

□

Abbildung 3-1 zeigt die Skizze eines Prozessmusterkatalogs, wobei es hier noch unerheblich ist, um welche Prozessmusterbeziehungen es sich handelt.

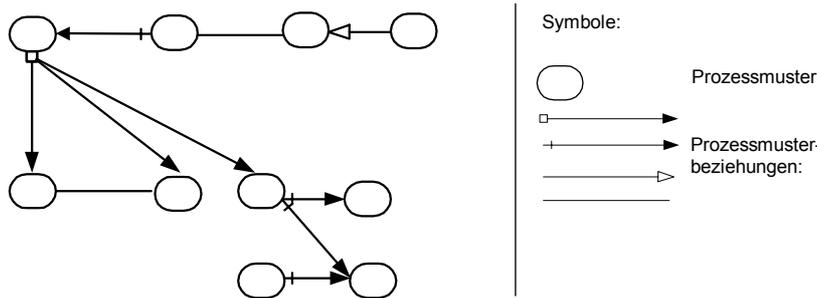


Abbildung 3-1: Skizzenhafte Darstellung eines Prozessmusterkatalogs

Nachdem wir nun verschiedene Eigenschaften zur Klassifizierung von Prozessmustern definiert haben, können wir diese zu einem Klassifizierungsschema integrieren. Tabelle 3-1 zeigt einige Klassifizierungsbeispiele:

Klassifizierungsschema

Prozessmuster	Domäne	Phase	Aspekt	Katalog (bzw. Sprache)
„Construct Capsule“ [Stö00]	Software Development	Konstruktion	Design	Architecture Modeling Language
„Architecture driven“ [BRS98]	Software Development	alle Phasen	Design	Component Based Development Language
„Review“ [Cop94]	Software Development	alle Phasen	Qualitätssicherung	Development Process Generative Pattern Language

Tabelle 3-1: Klassifizierungsbeispiele

Vernachlässigung
des Abstraktions-
grades

Abschließend sei noch darauf hingewiesen, dass wir das Klassifikationsmerkmal Abstraktionsgrad nicht verwenden (wie z.B. von Störrle in [Stö00] vorgeschlagen), da es schwierig ist, ein Prozessmuster nach seinem Abstraktionsgrad einzuordnen. Sicherlich kann angegeben werden, ob ein Prozessmuster abstrakter als ein anderes Prozessmuster ist. Diese Angabe ist jedoch eine relative; eine objektive Angabe, auf welchem Abstraktionsgrad sich ein einzelnes Prozessmuster befindet, ist kaum möglich. Will man außerdem rekursive Beziehungen verwenden (wie wir dies später in Abschnitt 3.3.2 tun), kann kein Abstraktionsgrad mehr angegeben werden, da dann ja ein Prozessmuster in mehreren Abstraktionsgraden existieren müsste.

Durch die Einordnung eines Prozessmusters in den Prozessmusterkatalog ist es unserer Meinung nach eher möglich, zu erkennen, welche Prozessmuster sich auf ähnlichen Abstraktionsstufen befinden (und zwar dadurch, dass sie durch eine Sequence-Beziehung (Abschnitt 3.3.1) oder Processvariance-Beziehung (Abschnitt 3.3.4) verknüpft sind) oder ob sie sich auf untergeordneten Abstraktionsstufen befinden (und zwar dadurch, dass sie durch eine Use-Beziehung (Abschnitt 3.3.2) oder Refinement-Beziehung (Abschnitt 3.3.3) verknüpft sind).

3.2 Prozessmuster

Im vorhergehenden Abschnitt haben wir bereits den Begriff des Prozessmusters definiert. Abbildung 3-2 verdeutlicht, wie sich ein Prozessmuster zusammensetzt. Zu einem gegebenen Problem, das einen bestimmten Kontext vorgibt, gibt es einen Prozess, der auf musterhafte Weise vorgibt, wie das Problem gelöst werden kann. Der Kontext eines Prozessmusters wird durch das Problem bestimmt. Dies bedeutet, dass der Prozess die Objekte und Ereignisse des initialen Kontexts konsumieren und die Objekte und Ereignisse des resultierenden Kontexts produzieren muss.

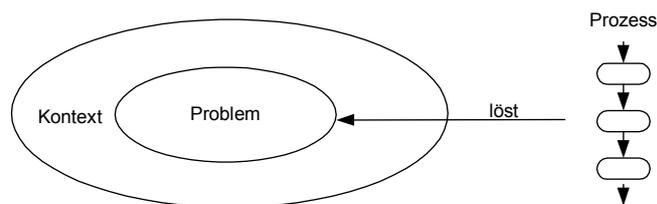


Abbildung 3-2: Elemente eines Prozessmusters

Um den Begriff des Prozessmusters verstehen zu können, müssen wir noch weitere Begriffe definieren:

Definition 3-7: Objekt

Ein Objekt repräsentiert ein bestimmtes Artefakt (z.B. Anforderungsspezifikation), das während eines Prozesses konsumiert, erstellt oder modifiziert wird. Objekte sind Teil des Kontexts von Prozessmustern.

□

Definition 3-8: Ereignis

Ein Ereignis bezeichnet ein Geschehen, das einen Prozess auslöst oder während eines Prozesses ausgelöst wird.^a Ereignisse sind Teil des Prozessmusterkontexts. Ein Ereignis repräsentiert das Eintreten eines bestimmten Zustands (z.B. Kundenanfrage liegt vor).

□

a. Vgl. mit [Wikipedia]: Ein Ereignis bezeichnet ein Geschehen, also eine prozessuale Entität im Gegensatz zu einem Ding oder einer Substanz.

Definition 3-9: Komposition von Objekten und Ereignissen

Objekte und Ereignisse können atomar oder zusammengesetzt sein. Dabei setzt sich ein zusammengesetztes Objekt stets aus Objekten zusammen, ein zusammengesetztes Ereignis setzt sich stets aus Ereignissen zusammen. Durch die Komposition von komplexeren Objekten und Ereignissen aus einer Menge von Objekten bzw. Ereignissen kann ein Prozess übersichtlicher modelliert werden.

Dies bedeutet ferner, dass in initialen oder resultierenden Kontexten komponierte Objekte und Ereignisse durch ihre Komponenten ersetzt werden können. Auf diese Weise werden die Kontexte in ihre Einzelbestandteile zerlegt.

□

Abbildung 3-3 zeigt, wie das Objekt „Business Models“ durch die beiden Objekte „Business Use-Case Models“ und „Business Object Model“ komponiert wird.

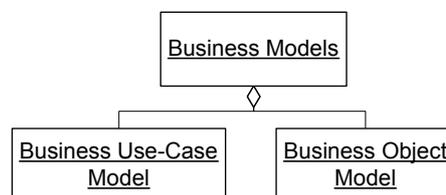


Abbildung 3-3: (De-)komposition eines Objekts

Definition 3-10: Problem

Ein Prozessmuster löst ein häufig auftretendes Problem. Probleme sind stets eingebettet in einen Rahmen von Ereignissen und Objekten, die dieses Problem bedingen bzw. die durch Lösung des Problems ausgelöst werden. D.h. Probleme haben einen Kontext. Üblicherweise^a wird das Problem eines Musters mit einem Fragesatz (z.B. „How long should the project take?“), Muster „Size the Schedule“ aus [Cop94]), einem Aussagesatz oder einer kurzen Problembeschreibung dargestellt. Die Beschreibung eines Problems muss jedoch präziser erfolgen, um Aussagen darüber treffen zu können, ob ein Prozessmuster dieses Problem löst oder nicht. Diese präzisere Beschreibung erfolgt durch Angabe des initialen Kontexts (z.B. „product is understood“, „project size estimated“) und des resultierenden Kontexts (z.B. „project with flexible target date“) als Mengen von Objekten und Ereignissen.

Ein Problem beschreibt eine Ausgangs- und eine Zielsituation. Durch Einsatz eines Prozessmusters wird die Ausgangssituation (initialer Kontext) in die Zielsituation (resultierender Kontext) transformiert. Der initiale Kontext eines Problems repräsentiert eine Menge von Objekten und Ereignissen, die die Problemstellung konkretisieren (d.h. durch welche Objekte und Ereignisse wird ein Problem bedungen). Der resultierende Kontext beschreibt eine Menge von Objekten und Ereignissen, die als Ergebnis der Problemlösung produziert wurden (d.h. durch welche Objekte und Ereignisse wird ein Problem gelöst). Zu dem Problem gehört ferner eine natürlichsprachige Beschreibung der Forces, die durch das Problem bestimmt werden.

□

a. S. hierzu Abschnitt 2.1.4 Der Begriff des Musterschemas.

Separate Definition
von Problem und
Prozessmuster

Ein Problem wird stets unabhängig von Prozessmustern definiert. Dies bedeutet, dass ein Problem existieren kann, für das kein lösendes Prozessmuster existiert. Es bedeutet ferner, dass für ein Problem mehrere Prozessmuster existieren können. Dies ist dann der Fall, wenn mehrere Prozessmuster das gleiche Problem lösen. Solche Prozessmuster nennt man Prozessvarianten (Abschnitt 3.3.4). Die Suche nach Prozessmustern wird also zweistufig durchgeführt: In der ersten Stufe wird eine Übereinstimmung mit spezifizierten Problemen durchgeführt, und erst in der zweiten Stufe wird ermittelt, welche Muster dieses Problem lösen. Das Suchverfahren wird dadurch effizient, dass durch die genauere Problembeschreibung das passende Problem schnell gefunden wird und dass zu diesem Problem dann alle lösenden Muster mitgeliefert werden. Der Anwender muss sich dann für eine der varianten Prozessmuster entscheiden. Dies bedeutet schließlich, dass ein Prozessmusterkatalog durch Probleme und Prozessmuster (und durch Prozessmusterbeziehungen, s. Abschnitt 3.3) strukturiert wird und damit die Übersichtlichkeit erhöht.

Beispiel

In Abbildung 3-4 sehen wir ein Beispiel für den Kontext des Problems „How to perform Reviews?“. Die Problemstellung besteht aus einer Reviewanfrage und dem zu reviewenden Objekt. Die Zielstellung besteht aus dem Objekt, das nach dem Review freigegeben wurde.

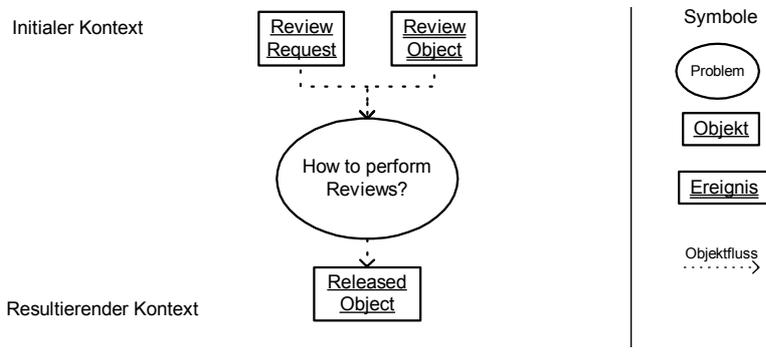


Abbildung 3-4: Beispiel: Kontext des Problems „How to perform Reviews?“

Definition 3-11: Kontext

Der Kontext eines Prozessmusters definiert die Bedingungen, die vor und nach Anwendung des Prozessmusters erfüllt sein müssen.

Die Bedingungen, die vor Anwendung des Prozessmusters erfüllt sein müssen, nennt man den initialen Kontext des Prozessmusters. Die Bedingungen, die nach Anwendung des Prozessmusters erfüllt sein müssen, nennt man den resultierenden Kontext des Prozessmusters. Initialer und resultierender Kontext sind jeweils Mengen von Objekten und Ereignissen. Der Kontext setzt sich also aus zwei Mengen zusammen, dem initialen Kontext und dem resultierenden Kontext.

□

In Abbildung 3-5 sehen wir das Prozessmuster „Review“. Das Prozessmuster „Review“ beschreibt, wie ein Review-Prozess durchzuführen ist. Es verlangt als Voraussetzung für die Durchführung die Anfrage für ein Review und das zu reviewende Objekt. Ergebnis der Anwendung des Prozessmusters „Review“ ist die Freigabe des Review-Objekts.

Beispiel

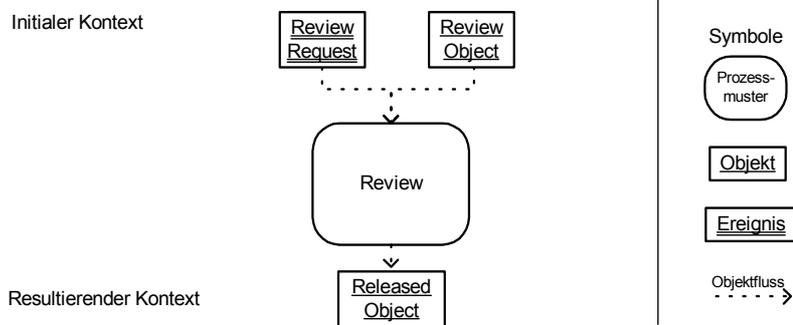


Abbildung 3-5: Beispiel: Kontext des Prozessmusters „Review“

Das Prozessmuster „Review“ repräsentiert eine Lösung für das Problem „How to perform Reviews“ von Abbildung 3-4. Man beachte, dass der initiale Kontext des Prozessmusters mit dem initialen Kontext des Problems übereinstimmt. Dies gilt analog für den resultierenden Kontext von Prozessmuster und Problem.

In dieser Arbeit benutzen wir die Begriffe Prozess und Softwareentwicklungsprozess synonym. Die Konzeption und Veranschaulichung von PROPEL erfolgt anhand von Softwareentwicklungsprozessen, könnte aber auch für jeden anderen beliebigen Prozess (solange er nicht weitere Attribute außer Objekte, Ereignisse, Rollen und Werkzeuge benötigt) erfolgen.

Definition 3-12: Prozess

Unsere Definition von Prozess basiert auf der UML-Spezifikation von Aktivitätsdiagrammen. Dort werden Aktivitätsdiagramme als Spezialfall von Zustandsautomaten definiert, die einen Kontroll- und Objektfluss zwischen Aktivitäten spezifizieren ([UML1.5], S. 2-172).

UML-Aktivitätsdiagramme kennen das Konzept eines initialen und resultierenden Kontextes nicht. Dort kann lediglich für jede Aktivität eine Menge von Objekten angegeben werden, die im Rahmen eines Objektflusses verknüpft sind. Wir definieren daher einen Prozess als Aktivitätsdiagramm, das zusätzlich einen initialen und resultierenden Kontext besitzt.

Der Prozess repräsentiert eine Lösung für das durch das Prozessmuster zugeordnete Problem. Dabei wird die Ausgangssituation (d.h. der initiale Kontext des Prozessmusters) in die Zielsituation (d.h. der resultierende Kontext des Prozessmusters) überführt. Der Prozess kann nur dann durchgeführt werden, wenn die aktuellen Bedingungen (d.h. existierende Objekte und Ereignisse) dem initialen Kontext des Prozessmusters entsprechen. Nach Durchführung des Prozesses entspricht die aktuelle Situation (d.h. welche Objekte und Ereignisse liegen vor) dem resultierenden Kontext des Prozessmusters, d.h. der resultierende Kontext wurde durch den Prozess hergestellt.

□

Beispiel Abbildung 3-6 zeigt den Prozess des Prozessmusters „Review“. Der Objekte und Ereignisse des initialen Kontexts sowie des resultierenden Kontexts sind grau eingefärbt.

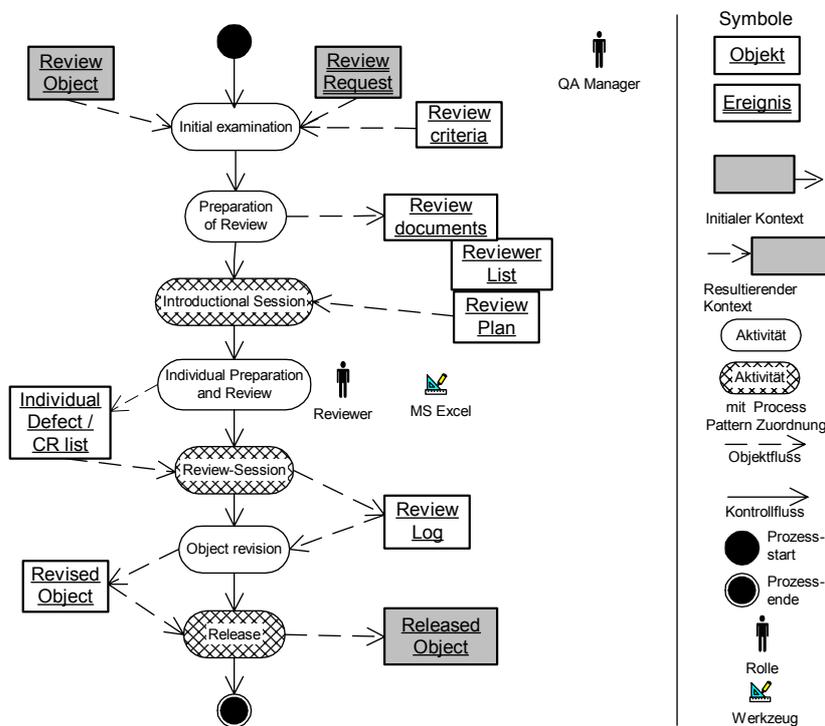


Abbildung 3-6: Beispiel: Prozess des Prozessmusters „Review“

Für die Definition von Aktivitäten gilt der gleiche Hinweis wie für Prozesse. D.h. in der UML-Spezifikation existiert bereits eine Definition von Syntax und Semantik von Aktivitäten. Wir nehmen lediglich Erweiterungen daran vor:

Definition 3-13: Aktivität

In der UML-Spezifikation ([UML1.5], S. 2-171) repräsentiert eine Aktivität (UML: Action-State) die Ausführung einer atomaren Aktion (z.B. eine Klassenoperation). Wir definieren darüber hinaus für eine Aktivität einen initialen und einen resultierenden Kontext. Der initiale Kontext einer Aktivität enthält alle Objekte und Ereignisse, die von der Aktivität konsumiert werden. Der resultierende Kontext einer Aktivität enthält alle Objekte und Ereignisse, die von der Aktivität produziert werden.

Jedes Element des initialen Kontexts eines Prozessmusters muss Element mindestens eines initialen Kontexts einer Aktivität sein, die Element des Prozesses des Prozessmusters ist. Dies gilt analog für alle Elemente des resultierenden Kontexts eines Prozessmusters. Dies bedeutet, dass die Elemente des Kontexts eines Prozessmusters von einer Aktivität des Prozesses konsumiert oder produziert werden müssen. Im Beispiel von Abbildung 3-6 konsumiert beispielsweise die Aktivität „Initial examination“ die Elemente des initialen Kontexts „Review Object“ und „Review Request“. Die Aktivität „Release“ produziert das Objekt „Released Object“, welches Element des resultierenden Kontexts ist.

□

Prozessen und Aktivitäten wird eine verantwortliche Rolle zugeordnet. Durch die Zuordnung wird festgelegt, welche Prozesse bzw. Aktivitäten durch welche Rollen auszuführen sind.

Definition 3-14: Rolle

Eine Rolle beschreibt Aufgaben, Verantwortlichkeiten und Rechte einer Gruppe von Individuen zur Erreichung eines Ziels. Jeder Aktivität und jedem Prozess wird mindestens eine Rolle zugeordnet.

□

Im Beispiel von Abbildung 3-6 ist dem Prozess „Review“ die Rolle „QA Manager“ zugeordnet. Der Aktivität „Individual Preparation and Review“ ist die Rolle „Reviewer“ zugeordnet. Die Rollen der anderen Aktivitäten wurden in diesem Beispiel aus Platzgründen ausgeblendet.

Einer Aktivität können beliebige viele Werkzeuge zugeordnet werden.

Definition 3-15: Werkzeug

Ein Werkzeug unterstützt die Ausführung einer Aktivität. Bei Werkzeugen handelt es sich meistens um Softwaresysteme, aber auch Hilfsmittel wie Checklisten können als Werkzeuge betrachtet werden. Einer Aktivität können beliebig viele Werkzeuge zugeordnet werden.

□

Im Beispiel von Abbildung 3-6 wird beispielsweise der Aktivität „Individual Preparation and Review“ das Werkzeug „MS Excel“ zugeordnet, mit dem die Reviewer ihre identifizierten Mängel auflisten.

3.3 Beziehungen zwischen Prozessmustern

Beziehungen ermöglichen Organisation

Eine der von uns formulierten Anforderungen postuliert die formale Definition von Prozessmusterbeziehungen. Mit der formalen Definition von Beziehungen zwischen Mustern verfolgen wir folgende Ziele: Aufgrund der syntaktischen und semantischen Definition von Beziehungen zwischen Mustern können bestehende Muster analysiert und in einem Organisationsschema wie z.B. einem Musterkatalog angeordnet werden. Durch die Analyse können auch Inkonsistenzen zwischen Prozessmustern identifiziert und aus den Mustern entfernt werden. Gegebenenfalls können aus der Analyse auch neue Prozessmuster entstehen.

Besondere Anforderungen an Beziehungen bei Prozessen

Prozessmuster bzw. deren Prozesse müssen Informationen miteinander austauschen können. Hierfür ist die Beschreibung von Kontexten und die Möglichkeit der Kopplung von Kontexten notwendig. Der Betrachtung der Kontexte ist also essentiell für die Definition von Beziehungen zwischen Prozessmustern. Durch Kontexte können erst Prozessabfolgen, Prozessverfeinerungen oder Prozessvarianten erklärt und definiert werden. Die Voraussetzung für eine Musterbeziehung ist die Erfüllung bestimmter Regeln, die sich auf die Kontexte der an der Beziehung teilnehmenden Prozessmuster beziehen. Diese Regeln erläutern wir in den nachfolgenden Abschnitten.

Vorab sei darauf hingewiesen, dass die Menge der Instanzen der Prozessmusterbeziehungen eine echte Teilmenge der Menge aller möglichen Instanzen ist. Dies bedeutet, dass nicht jede Beziehungsinstanz, die möglich ist, auch sinnvoll ist. Ob zwei oder mehrere Prozessmuster miteinander in Beziehung stehen, entscheidet letztendlich immer der Modellierer des Prozessmusters. D.h. aus der Entscheidung, dass Prozessmuster miteinander in einer Beziehung stehen, folgen eine Menge von Regeln, die erfüllt sein müssen. Umgekehrt bedeutet dies, dass aus der Erfüllung der Regeln nicht die Beziehungsinstanziierung folgt (d.h. keine logische Äquivalenz). Das Vorhandensein einer Beziehung zwischen Prozessmustern bedeutet also nicht, dass diese Beziehung auch vom Anwender des Prozessmusters in Anspruch genommen wird. Die Entscheidung, wie Prozessmuster miteinander kombiniert werden, liegt letztendlich beim Anwender. Das Aufzeigen von Beziehungen ist lediglich eine Hilfestellung für den Anwender, mögliche Kombinationen schneller erkennen zu können.

Modellierer entscheidet über Beziehungen

3.3.1 Sequence-Beziehung

Definition 3-16: Sequence-Beziehung

Bei der Sequence-Beziehung werden Prozessmuster sequentiell miteinander verknüpft. Die Beziehung bedeutet, dass ein oder mehrere Prozessmuster (sogenannte Vorgängermuster) gemeinsam die Voraussetzungen für die Anwendung eines nachfolgenden Prozessmusters (sogenanntes Nachfolgermuster) erfüllen. Nach Anwendung der Vorgängermuster kann also anschließend das Nachfolgermuster angewendet werden. Betrachten wir die Kontexte der Muster, bedeutet dies, dass jedes Element des initialen Kontexts des Nachfolgermusters Element der Vereinigung der resultierenden Kontexte aller Vorgängermuster sein muss.

□

Man beachte, dass es keine einschränkende Bedingung gibt, die fordert, dass die Vorgängermuster und das Nachfolgermuster verschiedene Prozessmuster sein müssen. Es ist also durchaus möglich, dass ein Prozessmuster sein eigener Nachfolger bzw. sein eigener Vorgänger sein kann. In der Praxis wird dieses Phänomen jedoch sehr selten auftreten, bedeutet es doch, dass die Objekte und Ereignisse, die konsumiert werden, auch wieder produziert werden.

Vorgängermuster gleich Nachfolgermuster

Die Definition einer Beziehung, die erlaubt, mehrere Nachfolgermuster mit einem Vorgängermuster zu verknüpfen, ist im übrigen nicht notwendig. Denn diese Verknüpfung kann bereits durch die Sequence-Beziehung abgebildet werden. Als Beispiel betrachte man ein Vorgängermuster pp , welches drei Objekte o_1, o_2 und o_3 erzeugt. o_1 ist Inputobjekt vom Nachfolgermuster pp'_1 , die Objekte o_2 und o_3 Inputobjekte vom Nachfolgermuster pp'_2 . pp steht also jeweils mit pp'_1 und pp'_2 in einer Sequence-Beziehung.

Beziehung 1:n nicht notwendig

Im ersten Beispiel in Abbildung 3-7 produziert das Prozessmuster „Review Session“ das Objekt „Review Log“, d.h. ein Protokoll der Review-Sitzung. Dieses Protokoll ist Voraussetzung, um das Prozessmuster „Object Revision“ ausführen zu können, welches den Prozess zur Überarbeitung eines Objekts beschreibt.

Beispiel

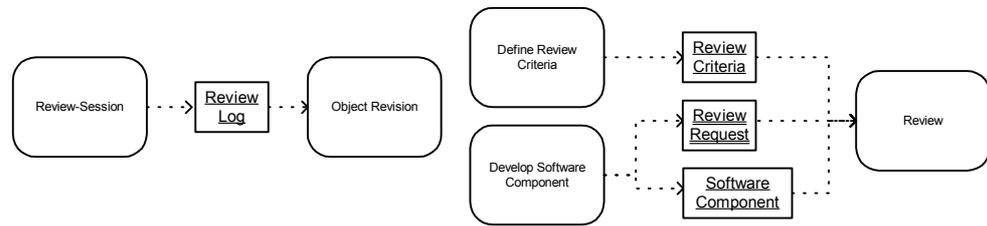


Abbildung 3-7: Beispiele für die Sequence-Beziehung

Im zweiten Beispiel produzieren die beiden Vorgängermuster „Define Review Criteria“ und „Develop Software Component“ die Objekte „Review Criteria“, „Software Component“ und das Ereignis „Review Request“. Diese beiden Prozessmuster bilden also zusammen die notwendigen Objekte und Ereignisse, um das Prozessmuster „Review“ ausführen zu können.

3.3.2 Use-Beziehung

Prozessmuster können nicht nur in einer sequentiellen Abfolge wie in der Sequence-Beziehung, sondern auch in einer wechselseitigen Abfolge ausgeführt werden. Z.B. wird mit der Anwendung eines Prozessmusters pp begonnen, dieses Prozessmuster benötigt aber zu seiner Ausführung ein anderes Prozessmuster pp' . D.h. beide Prozessmuster pp und pp' stehen in einer Nutzungs-Beziehung. Erst wenn das genutzte Prozessmuster pp' angewendet wurde, kann mit der Anwendung des Prozessmusters pp fortgefahren werden. Diese Nutzungs-Beziehung nennen wir Use-Beziehung.

Definition 3-17: Use-Beziehung

Die Use-Beziehung bedeutet, dass ein Prozessmuster (sogenanntes Kompositmuster) ein anderes Prozessmuster (sogenanntes Komponentenmuster) nutzt. Die Use-Beziehung entspricht einer Komposition von Komponentenmustern zu einem Kompositmuster. Eine Aktivität des Kompositmusters adressiert ein Problem, welches ein Teilproblem des Prozessmusters ist. Das Prozessmuster, das dieses Teilproblem löst, ist dann Komponentenmuster des Kompositmusters. Ein Kompositmuster kann beliebig viele Komponentenmuster nutzen.

Die Use-Beziehung fordert, dass der initiale Kontext einer Aktivität des Kompositmusters mit dem initialen Kontext des Komponentenmusters übereinstimmt und dass der resultierende Kontext der gleichen Aktivität des Kompositmusters mit dem resultierenden Kontext des Komponentenmusters übereinstimmt. Dies bedeutet, dass alle Objekte und Ereignisse des Kontexts der Aktivität im Kontext des genutzten Komponentenmusters vorkommen müssen. Auf diese Weise wird sichergestellt, dass alle notwendigen Objekte und Ereignisse vorhanden sind, um das Komponentenmuster auszuführen und dass das Komponentenmuster alle Objekte und Ereignisse produziert, um mit der Anwendung des Kompositmusters fortfahren zu können.

□

Die Uses-Beziehung kann auch für die Beschreibung rekursiver⁸ Prozessmuster eingesetzt werden. Eine Use-Beziehung heißt dann rekursiv, wenn es ein Prozessmuster mit sich selbst verknüpft. Dies bedeutet, dass das Prozessmuster sich selbst benutzt.

Bei einer Use-Beziehung ist das nutzende Prozessmuster eher grobgranularer, übergeordneter Natur und das genutzte Prozessmuster eher feingranularer, untergeordneter Natur.

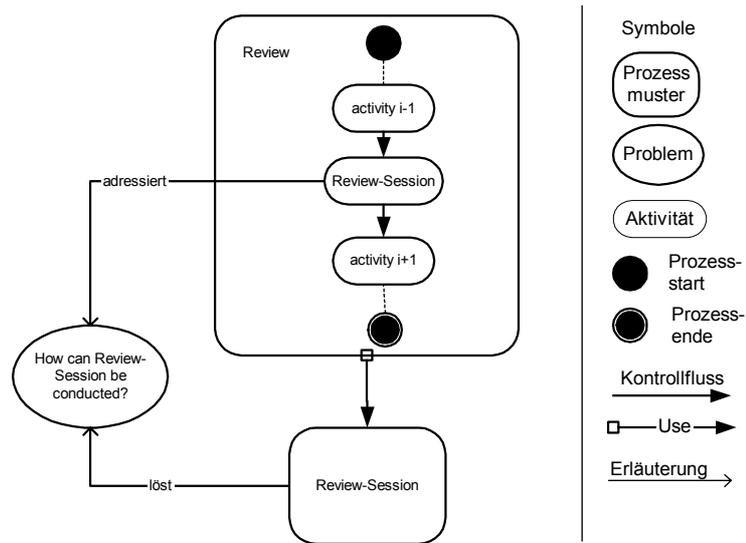


Abbildung 3-8: Veranschaulichung der Use-Beziehung

Abbildung 3-8 veranschaulicht die Use-Beziehung. Das Prozessmuster „Review“ enthält die Aktivität „Review-Session“. Diese Aktivität adressiert das Problem „How can Review-Session be conducted?“. Dieses Problem wird durch das Prozessmuster „Review-Session“ gelöst. Das Prozessmuster „Review-Session“ steht also in einer Use-Beziehung mit dem Prozessmuster „Review“.

Beispiel

Der Prozess des Prozessmusters „Review“ besteht aus sieben Aktivitäten (Abbildung 3-6). Von diesen sieben Aktivitäten werden drei (gekennzeichnet durch eine Schraffur) detaillierter durch weitere Prozessmuster beschrieben. Das Kompositmuster „Review“ nutzt also zur näheren Beschreibung der Aktivitäten „Introductory Session“, „Review Session“ und „Release“ drei Komponentenmuster, nämlich „Introductory Session“, „Review-Session“ und „Release“ (Abbildung 3-9).

8. Auch fraktal genannt. Die rekursive Komposition von Prozessmustern wurde von Störrle erstmals erwähnt, der für die komponentenbasierte Entwicklung einen wie Komponenten beschaffenen Prozess fordert, d.h. einen Prozess, der selbständig und fraktal ist [Stö00].

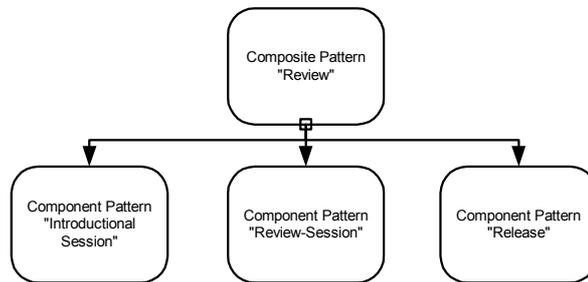


Abbildung 3-9: Kompositmuster und Komponentenmuster

Wir zeigen die Use-Beziehung am Musterpaar „Review“, „Review-Session“: Die Aktivität „Review-Session“ des Kompositmusters „Review“ (Abbildung 3-10, links) adressiert das Problem „How can Review-Session be conducted?“ (Abbildung 3-10, Mitte). Dieses Problem wird durch das Prozessmuster „Review-Session“ (Abbildung 3-10, rechts) gelöst.

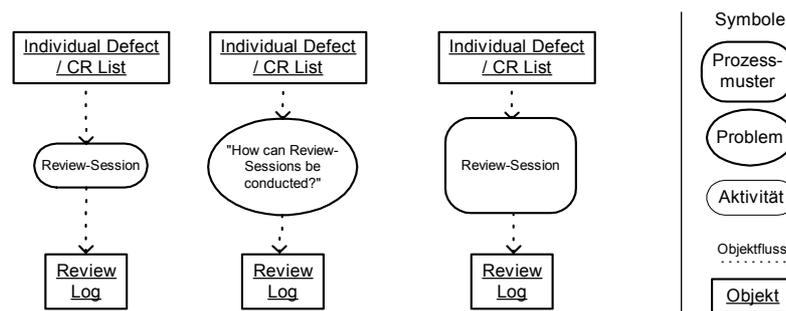


Abbildung 3-10: Zusammenhang zwischen Aktivität, Problem und Prozessmuster

Als Beispiel für eine rekursive Use-Beziehung betrachte man das Muster „Review-Session“ (Abbildung 3-10). Im Rahmen der Review-Session werden die von den Reviewern gefundenen Defects oder Change Requests (CR) bzgl. bestimmter Kriterien (Richtigkeit, Relevanz, Priorität etc.) bewertet. Sind keine Defects/CRs vorhanden, kann das Review-Objekt für die Freigabe vorbereitet werden. Sind nur wenige und geringfügige Defects/CRs vorhanden, kann das Review-Objekt unter Berücksichtigung bestimmter Auflagen (Beseitigung der Defects/CRs) für die Freigabe vorbereitet werden. Sind die Defects/CRs zu schwerwiegend, muss das Review-Objekt erneut in die Entwicklungsphase gehen und kann nicht freigegeben werden. In den beiden ersten Fällen schließt sich eine erneute Review-Session an, d.h. an dieser Stelle wird erneut das Prozessmuster „Review-Session“ benutzt.

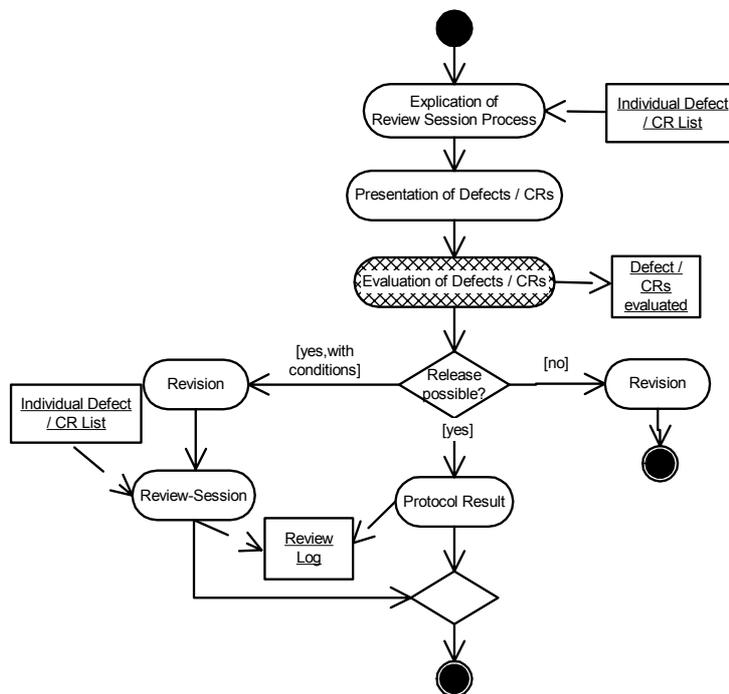


Abbildung 3-11: Beispiel: Prozess des Prozessmusters „Review-Session“

3.3.3 Refinement-Beziehung

Prozessmuster können auf verschiedenen Abstraktionsstufen existieren. Prozessmuster hoher Abstraktionsstufen besitzen weniger Details als Prozessmuster niedriger Abstraktionsstufen. Stellt ein Prozessmuster eine Abstraktion eines anderen Prozessmusters dar, so nennen wir diese Beziehung Refinement-Beziehung.

Definition 3-18: Refinement-Beziehung zwischen Prozessmustern

Die Refinement-Beziehung verknüpft ein abstrakteres Prozessmuster (sogenanntes Supermuster) und ein detaillierteres Prozessmuster (sogenanntes Submuster). Die Refinement-Beziehung bedeutet, dass das Submuster das Supermuster verfeinert, d.h. das Submuster wird als Spezialisierung des anderen, allgemeineren Supermusters aufgefasst. Das Supermuster enthält ein abstrakteres Problem und einen abstrakteren Prozess zur Lösung dieses Problems als das Submuster. Angelehnt an das Paradigma der Objekt-Orientierung ist die Refinement-Beziehung mit der Vererbung zu vergleichen, während die Use-Beziehung mit der Aggregation zu vergleichen ist.

□

Eine Voraussetzung für die Refinement-Beziehung zwischen Prozessmustern ist, dass die Kontexte von Supermuster und Submuster verschieden sein müssen. Eine weitere Voraussetzung ist, dass alle Kontextelemente des Supermusters im Kontext des Submusters enthalten

sein müssen. Über die Verfeinerung der Prozesse von Supermuster und Submuster wird indes keine Aussage getroffen, da es im Ermessensspielraum des Modellierers steht, ob und wie er die Prozesse bzw. ihre Aktivitäten verfeinert.

Die Refinement-Beziehung für Prozessmuster impliziert die Refinement-Beziehung für Probleme.

Definition 3-19: Refinement-Beziehung zwischen Problemen

Die Refinement-Beziehung zwischen Problemen verknüpft ein abstrakteres Problem (sogenanntes Superproblem) und ein detaillierteres Problem (sogenanntes Subproblem). Besteht eine Refinement-Beziehung zwischen Superproblem und Subproblem, dann stellt das Subproblem die Spezialisierung des Superproblems dar. Dies bedeutet, dass alle Elemente des initialen Kontexts des Superproblems im initialen Kontext des Subproblems enthalten sein müssen. Dies gilt analog für den resultierenden Kontext. Darüber hinaus können im Kontext des verfeinernden Subproblems weitere Objekte und Ereignisse enthalten sein, die für die Verfeinerung notwendig sind. □

Beispiel Als Beispiel betrachte man das Prozessmuster „Manual QA“, welches in einer Refinement-Beziehung mit dem Prozessmuster „Review“ (Abbildung 3-6) steht. D.h. wir fassen „Manual QA“ als das Supermuster und „Review“ als das Submuster auf. „Manual QA“ beschreibt allgemein, wie manuelle Prüfmethode durchgeführt werden. Dagegen beschreibt das Prozessmuster „Review“ auf detailliertere Weise als „Manual QA“, wie manuelles Prüfen durchgeführt werden kann (Abbildung 3-13). Der initiale Kontext des Submusters enthält alle Elemente des initialen Kontexts des Supermusters und darüber hinaus ein weiteres Element, das Ereignis „Review Request“. Die resultierenden Kontexte von Supermuster und Submuster stimmen überein.

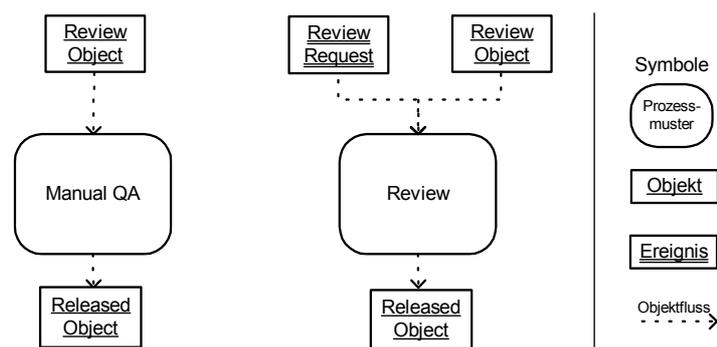


Abbildung 3-12: Supermuster „Manual QA“ und Submuster „Review“

Die Abbildung 3-13 zeigt in einer Übersicht, wie die Verfeinerung von Prozessmustern und Problemen zusammenhängt. Das Problem „How to perform Reviews?“, gelöst durch das Prozessmuster „Review“, ist eine Verfeinerung des Problems „How to perform Manual QA?“, gelöst durch das Prozessmuster „Manual QA“. Die beiden Prozessmuster stehen damit auch selbst in einer Refinement-Beziehung.

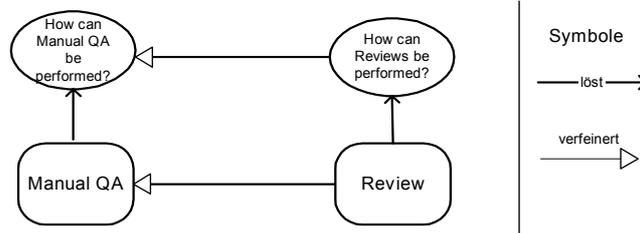


Abbildung 3-13: Verfeinerung von Prozessmuster und Problem

3.3.4 Processvariance-Beziehung

Auch wenn ein Prozessmuster eine erprobte Lösung für ein wiederkehrendes Problem präsentiert, kann es unter Umständen weitere Prozessmuster geben, die ebenfalls eine erprobte Lösung für das gleiche Problem vorhalten. Solche Prozessmuster nennt man Prozessvarianten. Der Muster-Anwender kann zwischen verschiedenen Prozessvarianten die ihm am passendsten erscheinende Variante auswählen. Die Ausgangssituation (d.h. die Problemkontexte) ist für alle Prozessvarianten gleich, unterschiedlich sind die angebotenen Lösungen (d.h. der Prozess). Eine solche Beziehung nennt man auch Processvariance-Beziehung.

Definition 3-20: Processvariance-Beziehung

Die Processvariance-Beziehung verknüpft zwei Prozessmuster, die das gleiche Problem lösen. Die Prozessmuster werden auch als Prozessvarianten bezeichnet. Die Lösung der beiden Prozessvarianten muss also verschieden sein.

□

Als Beispiel betrachte der Leser das Problem „How to implement Tests?“ (Abbildung 3-14). Der initiale Kontext des Problems besteht aus den Objekten „Test Case“, „Tool“ und „Tool Guidelines“. Der resultierende Kontext des Problems besteht aus dem Objekt „Test Skript“. Für dieses Problem existieren zwei Prozessvarianten, nämlich das Prozessmuster „Capture & Replay“ und das Prozessmuster „Program Test“. Beide Prozessmuster beschreiben, wie mit den vorgegebenen Mitteln Testskripte erstellt werden können. Beim Prozessmuster „Capture & Replay“ geschieht dies durch Aufzeichnen bestimmter Arbeitsvorgänge mittels eines geeigneten Tools. Beim Prozessmuster „Program Test“ geschieht dies durch manuelles Programmieren der Testskripte.

Beispiel

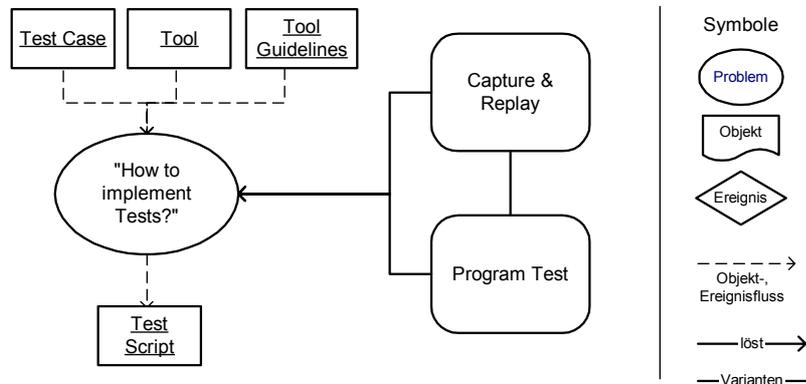


Abbildung 3-14: Prozessvariante Prozessmuster

3.3.5 Einfluss der Objekt- und Ereignis-Komposition auf Prozessmusterbeziehungen

Die Möglichkeit der Komposition von Objekten und Ereignissen hat auch Einfluss auf die Prozessmusterbeziehungen. Prozessmuster, die dem ersten Anschein nach nicht die Bedingungen zum Zustandekommen einer Beziehung erfüllen, können diese Bedingungen durch die kompositorischen Eigenschaften der Objekte und Ereignisse ihrer Kontexte erfüllen. Die Effekte der Komposition werden wir für jede Prozessmusterbeziehung diskutieren und ein Beispiel angeben.

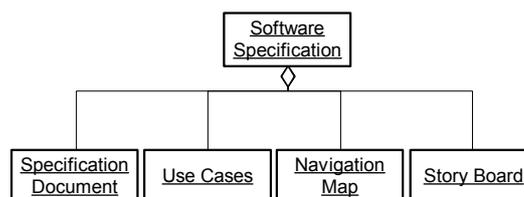


Abbildung 3-15: Komposition einer Softwarespezifikation aus einzelnen Objekten

3.3.5.1 Sequence

Abbildung 3-16 zeigt ein Beispiel, bei dem durch Objektkomposition eine Sequence-Beziehung ermöglicht wird. Augenscheinlich kann zwischen den beiden Prozessmustern keine Sequence-Beziehung gebildet werden, da die Objekte „Navigation Map“ und „Story Board“ nicht im resultierenden Kontext des Prozessmusters „Define Software Specification“ enthalten sind. Berücksichtigt man jedoch, dass das Objekt „Software Specification“ aus verschiedenen Objekten komponiert wird, darunter die Objekte „Navigation Map“ und „Story Board“, kann doch die Sequence-Beziehung zwischen den beiden Prozessmustern gebildet werden.

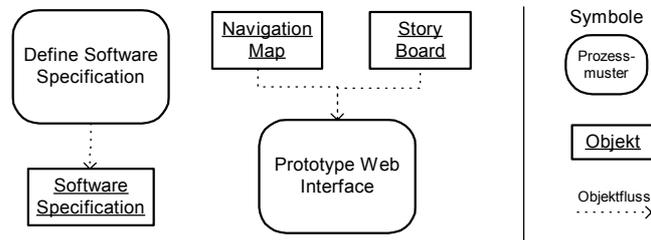


Abbildung 3-16: Sequence-Beziehung ermöglicht durch Objektkomposition

Erweitern wir die Definition der Sequence-Beziehung unter Berücksichtigung der Objekt- und Ereigniskomposition, so gelten folgende Bedingungen für jedes Element des initialen Kontexts des Nachfolgemusters:

- Das Element ist Element der Vereinigung der resultierenden Kontexte aller Vorgängermuster (alte Bedingung) oder
- das Element ist ein Aggregat und alle Aggregatbestandteile sind Elemente der Vereinigung der resultierenden Kontexte aller Vorgängermuster oder
- das Element ist Bestandteil eines Aggregats, welches Element der Vereinigung aller der resultierenden Kontexte aller Vorgängermuster ist.

3.3.5.2 Use

Abbildung 3-17 zeigt ein Beispiel, bei dem durch Objektkomposition eine Use-Beziehung ermöglicht wird. Augenscheinlich kann der Aktivität „Define Test Plan“ das Prozessmuster „Define Test Plan“ nicht zugeordnet werden, da die initialen Kontexte von Aktivität und Prozessmuster außer dem Objekt „Project Plan“ keine Übereinstimmung zeigen. Berücksichtigt man jedoch die Komposition des Objekts „Software Specification“, kann doch die Use-Beziehung zwischen dem Prozessmuster „Define Test Plan“ und dem Prozessmuster, welches die Aktivität „Define Test Plan“ enthält, gebildet werden.

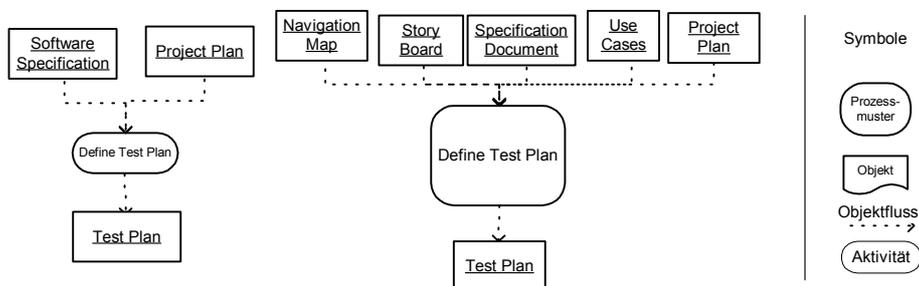


Abbildung 3-17: Use-Beziehung ermöglicht durch Objektkomposition

Erweitern wir die Definition der Use-Beziehung unter Berücksichtigung der Objekt- und Ereigniskomposition, so gelten folgende Bedingungen für jedes Element des Kontexts der Aktivität des Kompositmusters:

- Das Element ist im Kontext des Komponentenmusters enthalten oder
- die Aggregatbestandteile des Elements sind im Kontext des Komponentenmusters enthalten.
- Darüber hinaus enthält der Kontext des Komponentenmusters keine weiteren Elemente.

3.3.5.3 Refinement

Abbildung 3-18 zeigt ein Beispiel, bei dem durch Objektkomposition eine Refinement-Beziehung ermöglicht wird. Augenscheinlich ist das Prozessmuster „Workshop Software Requirements“ keine Verfeinerung des Prozessmusters „Define Software Specification“, da die resultierenden Kontexte der beiden Prozessmuster keinerlei Übereinstimmung zeigen. Berücksichtigt man jedoch die Komposition des Objekts „Software Specification“, kann doch die Refinement-Beziehung zwischen den beiden Prozessmustern gebildet werden.

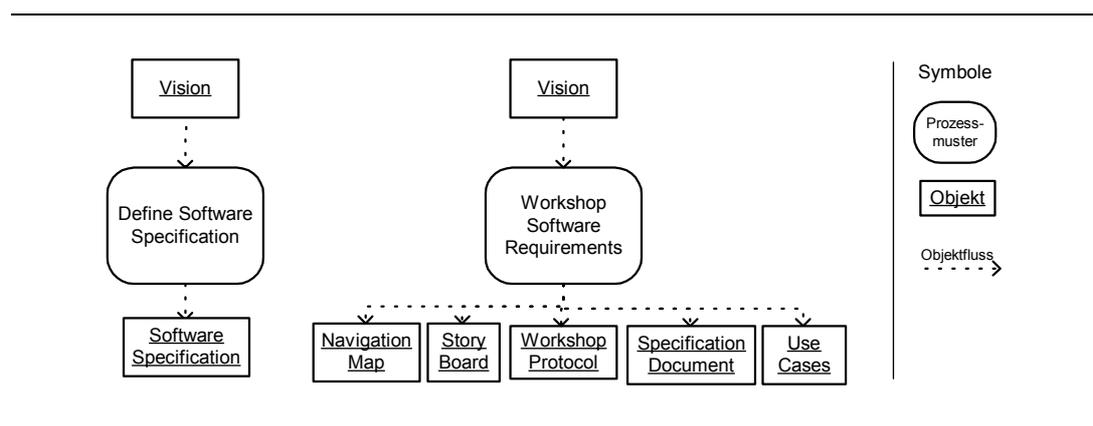


Abbildung 3-18: Refinement-Beziehung ermöglicht durch Objektkomposition

Erweitern wir die Definition der Refinement-Beziehung unter Berücksichtigung der Objekt- und Ereigniskomposition, so gelten folgende Bedingungen für jedes Element des Supermusters:

- Das Element ist im Kontext des Komponentenmusters enthalten oder
- die Aggregatbestandteile des Elements sind im Kontext des Komponentenmusters enthalten.
- Darüber hinaus können im Kontext des Submusters weitere Objekte und Ereignisse enthalten sein, die für die Verfeinerung notwendig sind.

3.3.5.4 Processvariance

Abbildung 3-19 zeigt ein Beispiel, bei dem durch Objektkomposition eine Processvariance-Beziehung ermöglicht wird. Augenscheinlich sind die beiden Prozessmuster „Review Specification“ und „Walkthrough Specification“ keine Prozessvarianten, da die initialen Kontexte

der beiden Prozessmuster keinerlei Übereinstimmung zeigen. Berücksichtigt man jedoch die Komposition des Objekts „Software Specification“, kann doch die Processvariance-Beziehung zwischen den beiden Prozessmustern gebildet werden.

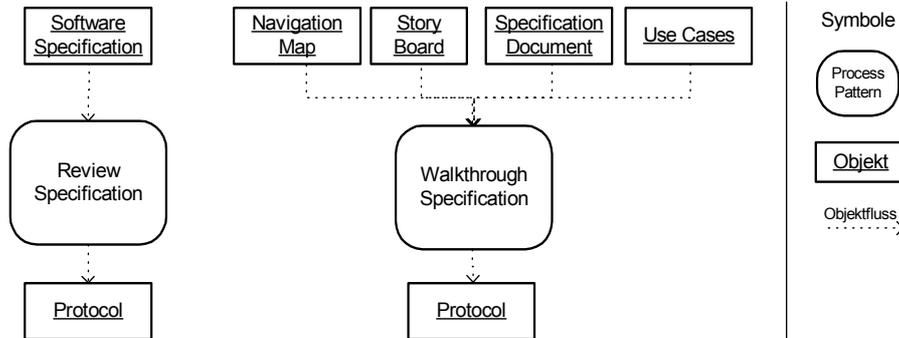


Abbildung 3-19: Processvariance-Beziehung ermöglicht durch Objektkomposition

Erweitern wir die Definition der Processvariance-Beziehung unter Berücksichtigung der Objekt- und Ereigniskomposition, so gelten folgende Bedingungen für jedes Element des Kontexts der ersten Prozessvariante:

- Das Element ist im Kontext der zweiten Prozessvariante enthalten oder
- die Aggregatbestandteile des Elements sind im Kontext der zweiten Prozessvariante enthalten.
- Darüber hinaus enthält der Kontext der zweiten Prozessvariante keine weiteren Elemente.

3.4 Prozessmusterschema und Problemschema

Nachfolgend werden wir das Prozessmusterschema, das wir in dieser Arbeit verwenden, erläutern. Dabei nehmen wir Bezug auf die in den vorhergehenden Abschnitten definierten und erläuterten Konzepte, die in dieses Schema eingebettet werden.

Definition 3-21: Prozessmusterschema

Ein Prozessmusterschema definiert, mit welchen Elementen ein Prozessmuster zu beschreiben ist.

□

Tabelle 3-2 zeigt das Prozessmusterschema von PROPEL:

Element	Bedeutung
Name	Name des Musters
Version	Version des Prozessmusters
Domain	Klassifizierung nach Domäne (Abschnitt 3.1)
Phase	Klassifizierung nach Phase (Abschnitt 3.1)
Aspect	Klassifizierung nach Aspekt (Abschnitt 3.1)
Catalog	Klassifizierung nach Katalog (Abschnitt 3.1)
Synonyms	Synonyme Bezeichnungen für das Prozessmuster.
Problem	Verweis auf das entsprechende Problemschema.
Process	Der Prozess beschreibt die Aktivitäten, die notwendig sind, um das Problem des Prozessmusters zu lösen.
Role	Dieses Element gibt die Rolle an, die verantwortlich für die Ausführung des Prozesses ist.
Related Patterns	Dieses Element gibt an, mit welchen anderen Mustern das Prozessmuster in Beziehung steht (Abschnitt 3.3).
Example	In diesem Element wird die Anwendung, d.h. Instanziierung des Prozessmusters in der Praxis dokumentiert.
Discussion	In diesem Element werden Vor- und Nachteile des Prozessmusters diskutiert.

Tabelle 3-2: Prozessmusterschema

Musterelement Version	Das Musterelement „Version“ wurde von uns hinzugefügt, um verschiedene Versionen eines Prozessmusters unterscheiden zu können. Dies war bislang nicht möglich. Bislang wurde stets ein neues Muster mit neuem Namen definiert, wenn ein Muster verändert wurde.
Nicht berücksichtigte Elemente	<p>Es gibt einige Musterelemente aus dem GoF-Schema, die wir nicht berücksichtigt haben. Dies sind Musterelemente, die stark design- und implementierungsgetrieben sind und für die Beschreibung von Prozessen nicht von Belang sind. Zu den nicht verwendeten Musterelementen gehören:</p> <ul style="list-style-type: none"> • Implementation Präsentiert Fallen, Tipps und Techniken, die man kennen sollte, wenn man das Muster implementiert. Benennt gegebenenfalls sprachspezifische Aspekte und Implementierungsmöglichkeiten. • Sample Code Diskutiert Codefragmente, die veranschaulichen sollen, wie man das Muster in C++ oder Smalltalk implementieren kann. • Collaborations Beschreibt, wie am Muster beteiligte Klassen und Objekte zur Erfüllung der gemeinsamen Aufgabe zusammenarbeiten.

Zusätzlich zum Prozessmusterschema definieren wir das Problemschema, welches dazu dient, Probleme unabhängig von Prozessmustern zu beschreiben.

Problemschema

Definition 3-22: Problemschema

Ein Problemschema definiert, mit welchen Elementen ein Problem zu beschreiben ist.

□

Tabelle 3-3 zeigt das Problemschema von PROPEL:

Element	Bedeutung
Name	Name des Problems
Version	Version des Problems
Initial Context	Der initiale Kontext gibt Objekte und Ereignisse wieder, die als Teil des Problems existieren/auftreten und die Voraussetzung für die Anwendung des Prozessmusters sind.
Resulting Context	Der resultierende Kontext gibt Objekte und Ereignisse wieder, die nach Lösung des Problems existieren/auftreten und die Ergebnis der Anwendung des Prozessmusters sind.
Description	In diesem Abschnitt wird das Problem natürlichsprachig beschrieben.
Solving Process Patterns	Verweis auf Prozessmuster, die dieses Problem lösen

Tabelle 3-3: Problemschema

3.5 Zusammenfassung

Nachfolgend fassen wir die von uns neu eingeführten Konzepte zusammen:

Wir haben ein Prozessmuster als ein Tripel (Problem, Kontext, Prozess) definiert.

Prozessmuster

Ein Problem haben wir als Tupel (Initialer Kontext, Resultierender Kontext) definiert. Die separate Beschreibung von Problemen unabhängig von Prozessmustern ist ein von uns neu eingeführtes Konzept. Hierdurch kann die Modellierung von und die Suche nach Prozessmustern einfacher gestaltet werden.

Problem

Durch Formalisierung des Kontextes eines Prozessmusters kann genau angegeben werden, was die Start- und die Endbedingungen eines Prozessmusters sind. Die Überprüfung, ob ein Prozessmuster ein bestimmtes Problem löst, erfolgt durch Übereinstimmung von Prozess und Kontext. Der Kontext repräsentiert sozusagen eine Schnittstelle des Prozessmusters, die das innere Verhalten des Prozessmusters (d.h. den Prozess) kapselt.

Kontext

Komposition von Objekten und Ereignissen	Bei der Modellierung von Prozessmustern und deren Beziehungen kann auch die Komposition von Objekten und Ereignissen berücksichtigt werden. Hierdurch ergibt sich eine größere Flexibilität bei der Modellierung.
Prozess	Für jedes Prozessmuster wird ein Prozess in Gestalt eines Prozessdiagramms angegeben. Hierdurch wird verhindert, dass, wie bislang, der Prozess eines Prozessmusters natürlichsprachig und damit ungenau modelliert wird. Durch Formalisierung des Prozesses kann die Use-Beziehung formal definiert werden. Ferner haben wir Prozessen und Aktivitäten Rollen zugeordnet, die in der Vorlage, dem UML-Aktivitätsdiagramm, nicht vorhanden ist. Aktivitäten haben wir Rollen und Werkzeuge zugeordnet.
Prozessmusterbeziehungen	Durch die formale Definition von Beziehungen zwischen Prozessmustern besteht nun ein Regelwerk, das für den Aufbau eines Prozessmusterkatalogs oder einfach für die Verknüpfung zweier Prozessmuster genutzt werden kann. Die Definitionen können als Kommunikationsgrundlage zwischen Muster-Modellierern, zwischen Muster-Anwendern und zwischen Muster-Modellierern und Muster-Anwendern eingesetzt werden und damit die bis dahin häufig aufgetretenen Fehlinterpretationen verhindern.
Klassifikationsmerkmale	Durch die Klassifikationsmerkmale können Prozessmuster einheitlich eingeordnet werden. Dies erleichtert die Suche nach Mustern.
Prozessmuster-schema/Problem-schema	Durch Angabe des Prozessmusterschemas und des Problemschemas wird dem Muster-Modellierer ein Vorlage angeboten, mit deren Hilfe er Prozessmuster und Probleme beschreiben kann.

4 Syntax von PROPEL

In diesem Kapitel erläutern wir, wie die in Kapitel 3 eingeführten Konzepte als UML-Erweiterung realisiert werden. Eine Übersicht über das UML-Metamodell kann in Anhang A.3 nachgeschlagen werden. Ergebnis ist eine erweiterte UML, die Process Pattern Description Language PROPEL.

4.1 Struktur des PROPEL-Metamodells

Die Erweiterung der UML zu einer Beschreibungssprache für Prozessmuster findet auf der Metamodellebene der UML statt (Abbildung 4-1, vgl. mit Abbildung 2-11). Die zuunterst liegende Instanz-Ebene von PROPEL enthält instanziierte, d.h. zur Laufzeit eines Projekts angewendete Prozessmuster. Die Elemente dieser Ebene sind Instanzen der Elemente der darüberliegenden Modell-Ebene. Die Modell-Ebene enthält Prozessmuster, die Instanzen der Elemente der darüberliegenden Metamodellebene sind. Die Metamodellebene enthält Elemente wie Metaklassen und Metaassoziationen zur Modellierung von Prozessmustern.

Erweiterung auf der Metamodellebene

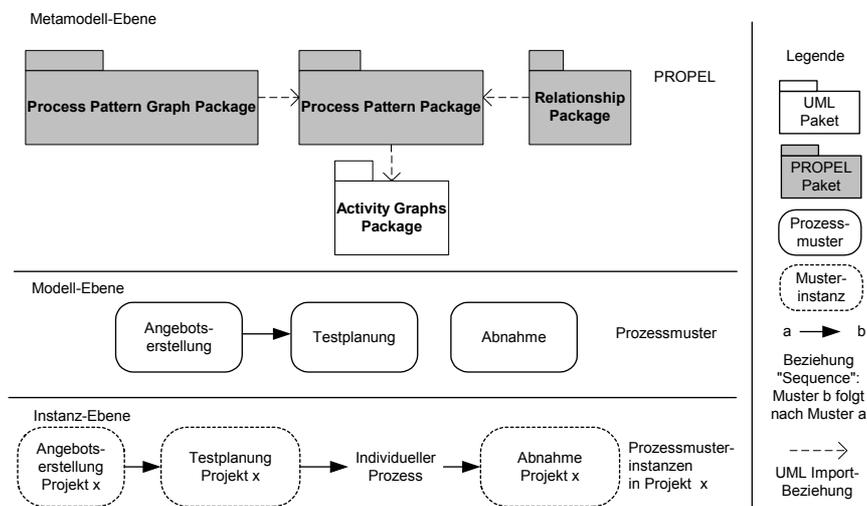


Abbildung 4-1: Die Modellierungsebenen von PROPEL

Die Erweiterungen präsentieren wir analog zur Präsentation des UML-Metamodells in [UML1.5]. Die durch die Erweiterung neu eingeführten Metaklassen, -assoziationen, -attribute und Constraints präsentieren wir also als nahtlose Integration mit den vorhandenen Metamodellelementen. In den Metamodelldiagrammen kennzeichnen wir dies durch Schattierung der neuen Metaklassen und -attribute. Jede neu eingeführte Metaklasse repräsentiert einen UML-Stereotyp, d.h. eine Subklasse einer bestehenden UML-Metaklasse. Basierend auf der

Integration mit UML-Spezifikation

Klassifikation von Berner et al. in [BGJ99] handelt es sich hierbei um restriktive Stereotypen, d.h. Stereotypen, die für neu eingeführte Syntaxelemente auch die Semantik definieren. Eine solche Erweiterung nennt man auch UML-Profil.

Analog zur UML-Spezifikation haben wir die neu eingeführten Konzepte in mehreren Metamodell-Paketen gebündelt (Abbildung 4-1, Metamodell-Ebene):

Process Pattern Package	Das Process Pattern Package enthält die wichtigsten Konzepte zur Beschreibung eines einzelnen Prozessmusters. Hierzu zählt die Definition des Konzepts Prozessmuster sowie seiner Komposition aus Problem, Prozess und Kontext.
Activity Graphs Package	Das Activity Graphs Package als Spezialfall des State Machines Package definiert Prozesse. Wir nehmen Erweiterungen an den bereits bestehenden Metaklassen ActionState und ObjectFlowState vor und führen die Metaklasse OFSComposition zur Modellierung der Objekt- und Ereigniskomposition ein.
Relationship Package	Das Relationship Package enthält die Beziehungskonzepte Sequence, Use, Refinement und Processvariance. Diese Konzepte sind notwendig, um einzelne Prozessmuster zueinander in Beziehung zu setzen. Ferner wird das Konzept des Prozessmuskatalogs syntaktisch definiert.
Process Pattern Graph Package	Das Graph Package enthält Konzepte zur Definition verschiedener Sichten auf Prozessmuster. Diese Sichten werden in Form von Problemdiagrammen und Prozessmuskatalogen hergestellt.
Abbildung auf PROPEL-Metamodell	Die im vorhergehenden Kapitel 3 erläuterten Konzepte von PROPEL werden in diesem Kapitel folgendermaßen syntaktisch auf das PROPEL-Metamodell abgebildet (Tabelle 4-1):

Konzept	wird abgebildet auf Metamodell	in Paket
Prozessmuster	ProcessPattern	Process Pattern Package
Problem	Problem	Process Pattern Package
Kontext	Context	Process Pattern Package
Prozess	Process	Process Pattern Package
Aktivität	ActionState	Activity Graph Package
Objekt, Ereignis	ObjectFlowState	Activity Graph Package
Rolle	Role	Process Pattern Package
Werkzeug	Werkzeug	Process Pattern Package
Sequence	Sequence	Relationship Package
Use	Use	Relationship Package
	ActivityProblemMapping	Process Pattern Package
Refinement	Refinement	Relationship Package

Tabelle 4-1: Zuordnung der Konzepte zu den Syntax-Elementen von PROPEL

Konzept	wird abgebildet auf Metamodell	in Paket
RefineProblem	RefineProblem	Relationship Package
Objektaggregation	OFSComposition	Activity Graph Package
Processvariance	Processvariance	Relationship Package
Prozessmusterkatalog	ProcessPatternCatalog	Relationship Package
Prozessmusterschema	ProcessPatternGraph	Graph Package
Problemschema	ProblemGraph	Graph Package
Klassifikation	Attribute von ProcessPattern	Process Pattern Package
	Aspect	Relationship Package

Tabelle 4-1: Zuordnung der Konzepte zu den Syntax-Elementen von PROPEL (Fortgesetzt)

Abbildung 4-2 zeigt in einer Übersicht alle neu definierten Metaklassen und deren Einbettung in das UML-Metamodell.

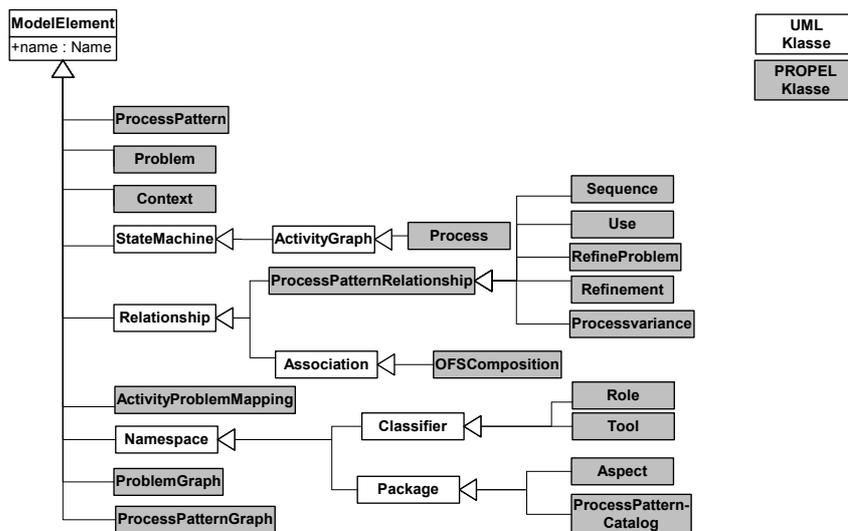


Abbildung 4-2: Integration der PROPEL-Metaklassen in das UML-Metamodell

In Abschnitt 4.2 erläutern wir die grundlegenden Konzepte von PROPEL anhand des Process Pattern Packages, in Abschnitt 4.3 die Erweiterungen des Activity Graphs Packages, in Abschnitt 4.4 die Erweiterungen des StateMachine Packages, in Abschnitt 4.5 die Konzepte des Relationship Packages und in Abschnitt 4.6 die Konzepte des Graph Packages.

Gliederung der nachfolgenden Abschnitte

Analog zur UML-Spezifikation (s. auch Anhang A.3.3) beschreiben wir für jede neue Metaklasse

- die abstrakte Syntax,
- die OCL-Regeln, die die abstrakte Syntax in Form von Constraints einschränken und
- die informale Semantik.

Ergänzung um
Semantik und
Notation

Der Abschnitt informale Semantik wird in der UML-Spezifikation fälschlicherweise „Detailierte Semantik“ genannt. Das Adjektiv „detailliert“ ist insofern irreführend, da man damit eine präzise Semantik verbindet. Die Semantik in diesen Abschnitten ist jedoch nur natürlichsprachiger Natur. Aus diesem Grunde bezeichnen wir diesen Abschnitt mit „Informale Semantik“. In Kapitel 5 geben wir dann die formale Semantik für die Metamodellelemente an. Die abstrakte Syntax ist ferner notationsunabhängig. Eine Abbildung geeigneter Notationselemente auf die abstrakte Syntax erfolgt in Kapitel 6.

Wenn wir den Namen einer UML- oder PROPEL-Metaklasse im Text verwenden, machen wir dies durch Typisierung durch den Arial-Font deutlich. Metaklassen beginnen mit einem Großbuchstaben (z.B. ProcessPattern), während Metattribute und -assoziationen mit einem Kleinbuchstaben beginnen (z.B. version). Die Namen von Metaklassen, -attributen und -assoziationen werden entweder direkt im erläuternden Text verwendet oder in Klammern angegeben.

4.2 Process Pattern Package

Das Process Pattern Package beinhaltet alle notwendigen Konzepte, um ein einzelnes Prozessmuster beschreiben zu können. Die Abbildungen 4-3 und 4-4 zeigen die Syntax der Elemente des Pakets.

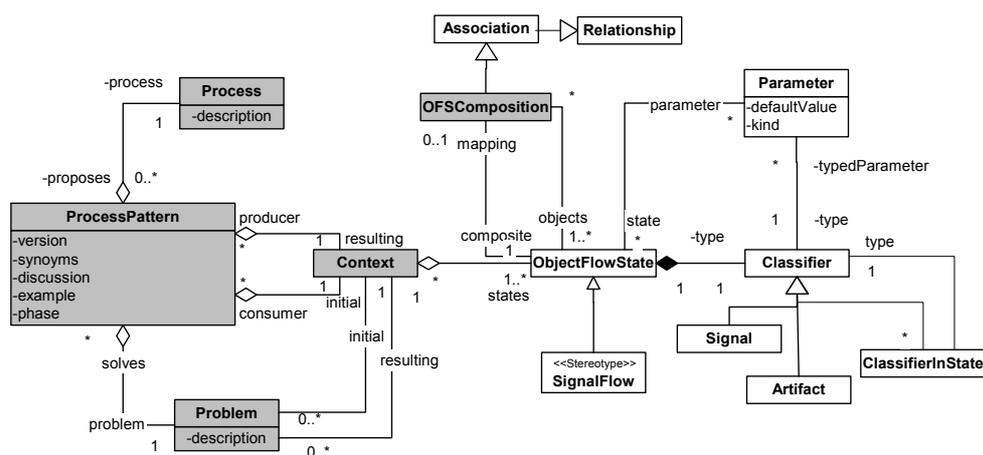


Abbildung 4-3: Abstrakte Syntax von PROPEL – Teil 1

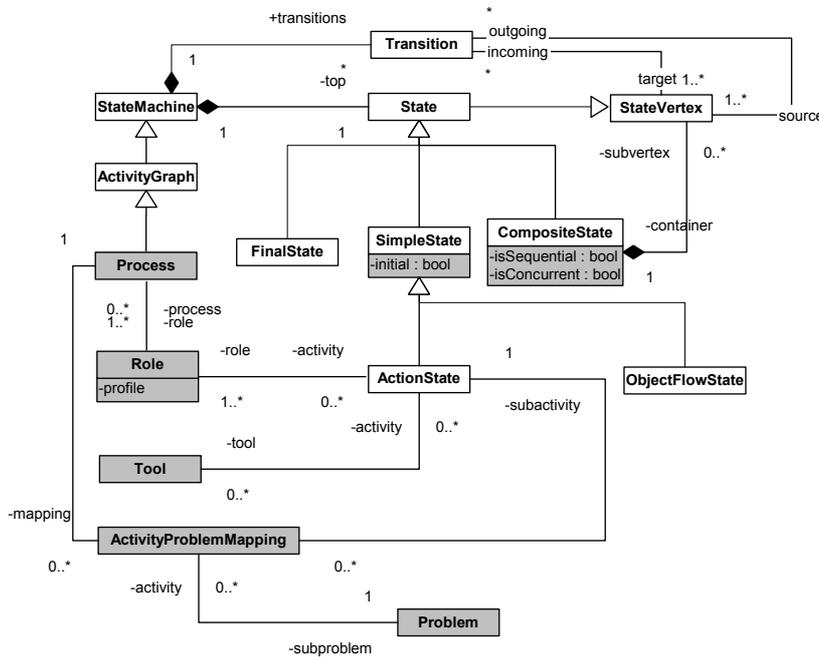


Abbildung 4-4: Abstrakte Syntax von PROPEL – Teil 2

4.2.1 ProcessPattern

Ein Prozessmuster beschreibt einen bewährten Prozess, um ein Problem, das in einem bestimmten Kontext wiederholt aufgetreten ist, zu lösen (Definition 3-1). Das Problem repräsentiert eine Situation, die im Rahmen eines Softwareentwicklungsprojekts auftreten kann (problem). Der Kontext spezifiziert Objekte, die vor und nach der Ausführung des Prozessmusters vorliegen (initial und resulting). Der Prozess spezifiziert Prozessschritte, die notwendig sind, um das Problem zu lösen (process).

Abstrakte Syntax

Die Muster-Elemente Name, Version, Phase, Synonyms, Discussion und Example werden durch Attribute ausgedrückt. Es gibt kein Attribut, um zwischen Prozess- und Ergebnismuster unterscheiden zu können. Da wir nur Prozessmuster beschreiben, ist dieses Attribut obsolet. Auch gibt es kein Attribut für die Domänenausprägung. Da wir ausschließlich Prozessmuster der Domäne „Softwareentwicklung“ beschreiben, ist dieses Attribut ebenfalls obsolet. Für Prozessmuster anderer Domänen müsste dieses Attribut also noch ergänzt werden.

Assoziationen

problem	Ein Prozessmuster hat die Aufgabe, genau ein Problem zu lösen.
process	Ein Prozessmuster offeriert genau eine Lösung (d.h. einen bestimmten Prozess) zur Behandlung des Problems.
initial	Für die Anwendung eines Prozessmusters wird genau ein bestimmter initialer Kontext vorausgesetzt.

resulting	Aus der Anwendung eines Prozessmusters geht genau ein resultierender Kontext hervor.
catalog	Ein Prozessmuster wird genau einem Prozessmusterkatalog zugeordnet.
aspect	Ein Prozessmuster wird genau einem Aspekt zugeordnet.

Attribute

version	Aktuelle Version des Prozessmusters. Dient dazu, mehrere Versionen eines Prozessmusters unterscheiden zu können. Dies rührt daher, dass Prozessmuster lebendiges Wissen verkörpern und es gegebenenfalls von Zeit zu Zeit erforderlich ist, Prozessmuster anzupassen.
phase	Die Phase beschreibt, in welcher Projektphase das Prozessmuster eingesetzt werden kann. Mögliche Ausprägungen sind: Inception, Elaboration, Construction, Transition.
synonyms	Synonyme des Patterns
discussion	Die Diskussion beschreibt Motivation, Konsequenzen und Pro- und Kontraargumente, die mit dem Muster verknüpft sind.
example	Beschreibung, wie das Prozessmuster in der Praxis angewendet wurde. Hieraus ergeben sich Folgerungen (z.B. Pro und Kontra) für die Diskussion.

OCL-Regeln

context: ProcessPattern

- [1] Der initiale Kontext eines Prozessmusters ist identisch mit dem initialen Kontext des zu lösenden Problems.

```
self.initial = self.problem.initial
```

- [2] Der resultierende Kontext eines Prozessmusters ist identisch mit dem resultierenden Kontext des zu lösenden Problems.

```
self.resulting = self.problem.resulting
```

- [3] Stimmen zwei Prozessmuster in Problem bzw. Kontext oder Prozess nicht überein, so handelt es sich um verschiedene Prozessmuster.

```
pp1, pp2: ProcessPattern  
(pp1.problem <> pp2.problem) OR (pp1.process <> pp2.process)  
implies pp1 <> pp2
```

- [4] Alle Objekte und Ereignisse, die zum initialen Kontext gehören, müssen von einer Aktivität konsumiert (sog. Inputparameter) oder konsumiert und modifiziert (sog. Inoutparameter) werden.

```
o : ObjectFlowState  
initialContext->forall(o |  
o.parameter.kind=in OR o.parameter.kind=inout)
```

- [5] Alle Objekte und Ereignisse, die zum resultierenden Kontext gehören, müssen Outputparameter oder Inoutparameter (d.h. modifizierbar) sein.

```
o : ObjectFlowState
resultingContext->forall(o |
o.parameter.kind=out OR o.parameter.kind=inout)
```

- [6] Ein Objekt oder Ereignis, das zum initialen Kontext eines Prozessmusters gehört, muss zuerst konsumiert und kann erst dann produziert werden.

Mit OCL-Constraints nicht ausdrückbar.

Zusätzliche Operationen

- [7] **initialContext** ist die Menge aller Elemente des initialen Kontexts des Prozessmusters, **resultingContext** ist die Menge aller Elemente des resultierenden Kontexts des Prozessmusters

```
initialContext, resultingContext: Set(ObjectFlowState)
initialContext = self.initial.states
resultingContext = self.resulting.states
```

- [8] **setOfA** ist die Menge aller Aktivitäten des Prozesses eines Prozessmusters.

```
setOfA = self.top.subvertex->select(a | a.OclIsTypeOf(ActionS-
tate))
```

Prozessmuster repräsentieren einen abstrakten Prozess, der zur Lösung eines bestimmten Problems in einem bestimmten Kontext verwendet werden kann. Aufgrund der Problembeschreibung, die Teil des Prozessmusters ist, kann ein Anwender erkennen, ob das Prozessmuster für seine spezielle Problemsituation geeignet ist. Das Problem muss mit dem Kontext des Prozessmusters übereinstimmen ([1],[2]). Diese Bedingung gewährleistet, dass das Prozessmuster im Rahmen der spezifizierten Problemsituation angewendet werden kann. Der Kontext repräsentiert die Menge von Objekten und Ereignissen (Metaklasse `ObjectFlowState`), die vor bzw. nach Ausführung des Prozesses vorhanden sein muss. Wäre die Gleichheit von Problem und Kontext nicht gefordert, wäre die Ausführbarkeit des Prozessmusters gefährdet oder das Problem würde nicht entsprechend seiner Spezifikation gelöst.

Informale Semantik

Regel [3] definiert, dass zwei Prozessmuster nur dann identisch sind, wenn es auch ihre Teile sind.

Der initiale Kontext enthält Objekte und Ereignisse, ohne deren Vorkommen in der realen Projektsituation das Prozessmuster nicht angewendet werden kann. Jedes Element des initialen Kontexts muss daher Inputparameter oder Inoutparameter (d.h. das Objekt wird sowohl produziert als konsumiert) sein, um seine Rolle als Vorbedingung zu erfüllen (Regel [4]).

Der resultierende Kontext enthält Objekte und Ereignisse, die nach Anwendung des Prozessmusters in der realen Projektsituation vorkommen müssen. Jedes Element des resultierenden Kontexts muss daher Outputparameter oder Inoutparameter sein, um seine Rolle als Nachbedingung zu erfüllen (Regel [5]).

[6]: Der Modellierer muss ferner darauf achten, dass Objekte zwar sowohl In- als auch Outputparameter sein dürfen, aber nur in einer bestimmten Konstellation. Objekte, die erst im Rahmen des Prozesses erzeugt und anschließend konsumiert werden, dürfen nicht im initia-

len Kontext des Prozessmusters aufgeführt werden (Abbildung 4-5, Objekte o1, o3 und o4). Dagegen ist es erlaubt, ein Objekt, das konsumiert wurde, anschließend zu modifizieren, d.h. zu erzeugen (Abbildung 4-5, Objekt o6).

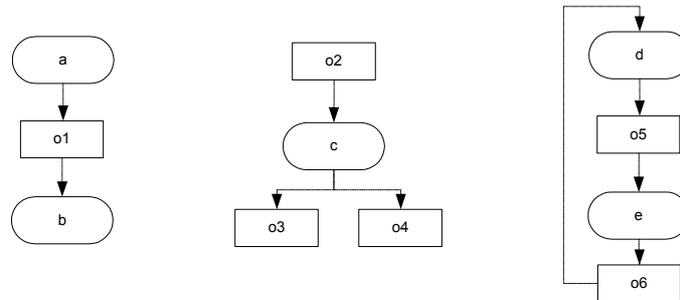


Abbildung 4-5: Modellierung von Inout-Objekten

4.2.2 Problem

Abstrakte Syntax

Ein Problem beschreibt die Problemstellung bzw. die Ziele, die mit dem Einsatz eines Modells gelöst bzw. erreicht werden sollen (Definition 3-10). Im PROPEL-Metamodell wird diese Forderung durch die Metaklasse Problem beschrieben.

Probleme werden durch einen initialen und einen resultierenden Kontext spezifiziert (initial und resulting). Der initiale bzw. resultierende Kontext repräsentiert eine Menge von Objekten (ObjectFlowState) und Ereignissen (Stereotyp SignalFlow).

Die Assoziation solves beschreibt, dass es für ein Problem beliebig viele Lösungen (d.h. Prozessmuster) geben kann. Die ausführliche Beschreibung des Problems wird durch das Attribut description ausgedrückt.

Assoziationen

initial	Ein Problem besitzt genau einen initialen Kontext.
resulting	Ein Problem besitzt genau einen resultierenden Kontext.
solves	Ein Problem kann von beliebig vielen Prozessmustern gelöst werden.

Attribute

description	Weitere Informationen zur Problemstellung. Hierzu gehören z.B. die „Forces“. Dies sind widrige Umstände, die das Problem haben entstehen lassen. Der Prozess des lösenden Prozessmusters muss diese Forces auflösen.
-------------	--

OCL-Regeln

Keine OCL-Regeln vorhanden.

Informale Semantik

Ein Problem beschreibt eine bestimmte Situation (in einem Softwareentwicklungsprojekt), die in eine andere Situation transformiert werden muss. Wir verwenden das Verb „muss“, um deutlich zu machen, dass die Transformation zwingend ist. Ansonsten würde es sich nicht um ein Problem handeln.

Die Ausgangs- und die Zielsituation werden jeweils durch eine Menge von Objekten und Ereignissen (abgebildet durch die Metaklasse `ObjectFlowState` und dessen Stereotyp `<<SignalFlowState>>`) modelliert. Welche Objekte und Ereignisse zum Problemkontext gehören, bestimmt der Problem-Autor.

Probleme sind zwar Teile von Prozessmustern, spielen aber dennoch eine eigenständige Rolle. Dies bedeutet, dass Probleme unabhängig von Prozessmustern spezifiziert werden (z.B. wenn das Problem schon erkannt wurde, aber noch keine geeignete Lösung dafür gefunden wurde). Zu einem Problem können also beliebig viele Prozessmuster (als Prozessvarianten, Abschnitt 4.5.9) spezifiziert werden. Für die Musterspezifikation wird dann der Problemkontext kopiert, d.h. das Problem muss mit dem Kontext des Prozessmusters übereinstimmen (Regel [1] und [2]).

4.2.3 Context

Der Kontext (Context) eines Prozessmusters definiert die Bedingungen, die vor und nach Anwendung des Prozessmusters erfüllt sein müssen (Definition 3-11). Ein Kontext ist Teil eines Prozessmusters. Er wird entweder vom Prozessmuster konsumiert (consumer) oder produziert (producer).

Abstrakte Syntax

Ein Kontext ist eine Menge von Zuständen von Objekten und Ereignissen (states). Ein Objektzustand wird durch die UML-Metaklasse `ObjectFlowState` repräsentiert, dem als Classifier (d.h. als Typ) ein Artifact zugeordnet wird. Ein Ereignis wird durch das Stereotyp `SignalFlowState` repräsentiert, der Typ des Classifiers ist dann ein Signal (s. [UML1.5], Abschnitt 2.13.4.3, S. 2-180: „*For applications requiring that actions or activities bring about an event as their result, use an object flow state with a signal as a classifier.*“).

Assoziationen

consumer	Ein initialer Kontext kann Voraussetzung für beliebig viele Prozessmuster sein.
producer	Ein resultierender Kontext kann Ergebnis beliebig vieler Prozessmuster sein.
states	Ein Kontext ist ein Aggregat eines oder mehrerer Objektzustände bzw. Ereignisse.

context: Context

OCL-Regeln

- [1] Jedes Element des initialen Kontexts muss Element des initialen Kontexts einer Aktivität des Prozessmusters sein.

```
self.states->forall(s |
self.consumer.setofA->forall(a | a.ICA.includes(s)))
```

- [2] Jedes Element des resultierenden Kontexts muss Element des initialen Kontexts einer Aktivität des Prozessmusters sein.

```
self.states->forall(s |
self.producer.setofA->forall(a | a.RCA.includes(s)))
```

Informale Semantik Der Kontext eines Prozessmusters definiert die Bedingungen in Gestalt von Objekten und Ereignissen, die vor und nach Anwendung des Prozessmusters erfüllt sein müssen. Der Kontext wird mittels Objekten und Ereignissen, die einen Zustand besitzen, modelliert. In der realen Situation des Anwenders müssen genau diese Objekte und Ereignisse vorliegen.

Kontexte bilden nicht nur Bedingungen für einzelne Prozessmuster, sondern sind ferner das Bindeglied zwischen verschiedenen Prozessmustern. Sie bilden Schnittstellen, durch die spezifiziert ist, in welcher Reihenfolge Prozessmuster verfeinert oder zusammengesetzt werden können.

4.2.4 Process

Abstrakte Syntax Für die Modellierung von Prozessen greifen wir auf die UML-Konzepte `ActivityGraph` und `StateMachine` zurück. Ein Zustandsautomat (`StateMachine`) besteht aus einer Menge von Transitionen (`Transition`) und einer Menge von Knoten (`StateVertex`). In der Zustandshierarchie gibt es einen ausgezeichneten, obersten Knoten (`top State`), der als zusammengesetzter Zustand (`CompositeState`) weitere Knoten enthält. Eine Transition steht mit einer nichtleeren Menge von Knoten in Beziehung. Ein Knoten hat drei Subklassen, `State` zur Darstellung von Zuständen, `StubState` zur Darstellung von Zuständen, die Transitionen zwischen Submaschinen und dem übergeordneten Zustandsautomaten verbinden, und `PseudoState` zur Verknüpfung mehrerer Transitionen. Ein Zustand hat wiederum drei Subklassen, `CompositeState`, der einen zusammengesetzten Zustand darstellt, `SimpleState`, der einen atomaren Zustand darstellt und `FinalState`, der die Terminierung eines Zustandsautomaten darstellt.

Ein Aktivitätsdiagramm (`ActivityGraph`) ist eine Subklasse eines Zustandsautomaten. Es beschreibt den Kontroll- und den Objektfluss zwischen einer Menge von Aktivitäten. Ein einfacher Zustand besitzt zwei Subklassen, `ActionState`, welcher eine Aktivität darstellt und `ObjectFlowState`, welcher ein Objekt darstellt. Ein Objekt kann als Input- und Outputparameter verwendet werden. Es hat einen Typ (`Classifier`).

Der Prozess beschreibt, wie die vom Prozessmuster adressierte Problemstellung gelöst wird. Im Metamodell ist ein Prozess (`Process`) eine Spezialisierung der Metaklasse `ActivityGraph`. Der Unterschied zwischen einem Prozess und einem Aktivitätsdiagramm ist der, dass ein Prozess zusätzlich einen initialen und einen resultierenden Kontext besitzt, der angibt, welche Objekte und Ereignisse zur Anwendung des Prozesses notwendig sind. Die Kontext ist ein wichtiger Bestandteil, wenn es darum geht, das Verhalten eines oder mehrerer verknüpfter Prozesse zu spezifizieren (Kapitel 5).

Ein Prozess kann Bestandteil beliebig vieler Prozessmuster sein (`proposes`). Für die Ausführung des Prozesses ist eine Rolle verantwortlich (`role`). Die ausführliche Beschreibung der Lösung wird durch das Attribut `description` ausgedrückt.

Die hierarchische Komposition von Prozessen wird durch die Abbildung von Problemen auf Aktivitäten gelöst. Den Aktivitäten (Metaklasse `ActionState`) eines Prozesses können entsprechende Probleme zugeordnet werden (`Assoziation mapping`). Diese Probleme sind somit Subprobleme des Problems des übergeordneten Patterns. Dies weist darauf hin, dass gegebenenfalls weitere Prozessmuster zur Lösung einer Aktivität vorhanden sind. Die Existenz einer oder mehrerer Aktivitäts-Problem-Zuordnungen sind also Voraussetzung für die hierarchische Komposition (`Use-Beziehung`).

		Assoziationen
proposes	Der Prozess kann Bestandteil beliebig vieler Prozessmuster sein.	
role	Einem Prozess ist mindestens eine Rolle zugeordnet.	
tool	Einem Prozess können beliebig viele Werkzeuge zugeordnet werden.	
mapping	Die Aktivitäten eines Prozesses können Probleme adressieren, für deren Lösung gegebenenfalls weitere Prozessmuster existieren.	

description	Natürlichsprachige Beschreibung der Lösung	Attribute
-------------	--	-----------

context: Process OCL-Regeln

- [1] Die Aktivitäten des Prozessmusters besitzen keine Subaktivitäten.

```
self.top.subvertex->forall(v:StateVertex|
v.ocliIsTypeOf(SimpleState) OR v.ocliIsTypeOf(FinalState))
```

- [2] Einer Aktivität kann höchstens ein Problem zugeordnet werden.

```
a: ActionState
m: ActivityProblemMapping
((self.mapping->select(m|m.subactivity=a)) ->collect(subproblem)) ->size<=1
```

- [3] Das Verhalten einer Rolle wird durch den Prozess spezifiziert.

```
self.context.ocliIsTypeOf(Role)
```

Die Klasse Process ist eine Subklasse der Klasse ActivityGraphs. Mit Aktivitätsdiagrammen ist man in der Lage, neben Aktivitäten (und deren Reihenfolge) auch Kontrollflussbedingungen, Zustände, Objekte, Objektflüsse und ihre Eigenschaften sowie Nebenläufigkeit und Synchronisation des Kontrollflusses auszudrücken. Beteiligte Personen in Form von Rollen lassen sich jedoch nicht darstellen. Die UML haben wir daher um die Verknüpfung von Rollen zu Prozessen und Aktivitäten syntaktisch und semantisch erweitert.

Informale Semantik

[1]: Die Zustände eines Prozesses müssen vom Typ SimpleState oder FinalState sein. Dies verhindert, dass den Aktivitäten eines Prozesses Subaktivitäten zugewiesen werden können. Die hierarchische Zerlegung eines Prozesses erfolgt bei PROPEL nämlich durch die Use-Beziehung (Abschnitt 4.5.6).

[2]: Innerhalb eines Prozesses kann einer Aktivität maximal ein Problem zugeordnet werden. Dieses Problem beschreibt genau die Ausgangssituation, die durch die Aktivität vorgegeben wird.

[3]: Prozesse beschreiben das Verhalten von Rollen, nicht von Werkzeugen, da die meisten Prozesse noch nicht automatisiert werden. Auch da, wo schon Aktivitäten durch ein Werkzeug ausgeführt werden (z.B. im Konfigurationsmanagement), müssen diese stets durch eine Rolle ausgelöst oder überwacht werden. Daher werden Werkzeuge als unterstützendes Hilfsmittel betrachtet, aber nicht als kontrollierendes Element.

4.2.5 ActivityProblemMapping

Abstrakte Syntax	Eine ActivityProblemMapping erlaubt es, einer Aktivität (subactivity) ein Problem (subproblem) zuzuordnen. Diesem Problem können dann wiederum Prozessmuster zugeordnet werden.						
Assoziationen	<table border="0"> <tr> <td style="padding-right: 20px;">subproblem</td> <td>Eine Instanz von ActivityProblemMapping verknüpft genau ein Problem mit einer Aktivität.</td> </tr> <tr> <td colspan="2"><hr/></td> </tr> <tr> <td>subactivity</td> <td>Eine Instanz von ActivityProblemMapping verknüpft genau eine Aktivität mit einem Problem.</td> </tr> </table>	subproblem	Eine Instanz von ActivityProblemMapping verknüpft genau ein Problem mit einer Aktivität.	<hr/>		subactivity	Eine Instanz von ActivityProblemMapping verknüpft genau eine Aktivität mit einem Problem.
subproblem	Eine Instanz von ActivityProblemMapping verknüpft genau ein Problem mit einer Aktivität.						
<hr/>							
subactivity	Eine Instanz von ActivityProblemMapping verknüpft genau eine Aktivität mit einem Problem.						
OCL-Regeln	Keine OCL-Regeln vorhanden.						
Informale Semantik	Mit Hilfe der Klasse ActivityProblemMapping können Prozessmuster komponiert werden. Über das Zuweisen eines Problems zu einer Aktivität wird signalisiert, dass über die Ausführung einer Aktivität Unklarheit besteht. Aus diesen Gründen wird ein Problem formuliert, das eine Lösung erfordert, die detaillierter sein muss als die Aktivität.						

4.2.6 Role

Abstrakte Syntax	Eine Rolle beschreibt die notwendigen Erfahrungen, Kenntnisse und Fähigkeiten, um Aktivitäten durchzuführen (Definition 3-14). In der UML-Spezifikation gibt es bereits die ähnlich anmutende Metaklasse Actor. Die Zuordnung von Actors zu anderen Modellelementen ist jedoch auf das zu entwickelnde System beschränkt, d.h. es sind nur Assoziationen zu Use-Cases, Subsystemen und Klassen erlaubt. Wir benötigen jedoch für die Modellierung von Prozessen einen weiter gefassten Begriff und haben deshalb die Metaklasse Role eingeführt. Rollen werden Aktivitäten (activity) und Prozessen (process) zugeordnet. Die Tätigkeiten und Aufgaben einer Rolle können über ein Profil (profile) detailliert beschrieben werden.						
Assoziationen	<table border="0"> <tr> <td style="padding-right: 20px;">process</td> <td>Eine Rolle kann beliebig vielen (also auch null) Prozessen zugeordnet werden.</td> </tr> <tr> <td colspan="2"><hr/></td> </tr> <tr> <td>activity</td> <td>Eine Rolle kann beliebig vielen (also auch null) Aktivitäten zugeordnet werden.</td> </tr> </table>	process	Eine Rolle kann beliebig vielen (also auch null) Prozessen zugeordnet werden.	<hr/>		activity	Eine Rolle kann beliebig vielen (also auch null) Aktivitäten zugeordnet werden.
process	Eine Rolle kann beliebig vielen (also auch null) Prozessen zugeordnet werden.						
<hr/>							
activity	Eine Rolle kann beliebig vielen (also auch null) Aktivitäten zugeordnet werden.						
Attribute	<table border="0"> <tr> <td style="padding-right: 20px;">profile</td> <td>Das Profil einer Rolle beschreibt die notwendigen Erfahrungen, Kenntnisse und Fähigkeiten, um Prozesse und Aktivitäten durchzuführen.</td> </tr> </table>	profile	Das Profil einer Rolle beschreibt die notwendigen Erfahrungen, Kenntnisse und Fähigkeiten, um Prozesse und Aktivitäten durchzuführen.				
profile	Das Profil einer Rolle beschreibt die notwendigen Erfahrungen, Kenntnisse und Fähigkeiten, um Prozesse und Aktivitäten durchzuführen.						

context: Role

OCL-Regeln

- [1] Rollen können nur mit Prozessen und Aktivitäten verknüpft werden. Diese Assoziationen sind binär.

```
self.associations->forAll(a|a.connection->size = 2 AND
a.allConnections->exists(r|r.type.ocIsKindOf(Role)) AND
a.allConnections->exists(r|r.type.ocIsKindOf(ActionState) OR
r.type.ocIsKindOf(Process)))
```

Eine Rolle beschreibt notwendige Erfahrungen, Kenntnisse und Fähigkeiten, um eine ausgewählte Menge an Aktivitäten durchzuführen. Die Aktivitäten beschränken sich nicht nur auf die Entwicklung und Anwendung von Systemen (Actor), sondern auf alle Tätigkeiten, die im Rahmen der Softwareentwicklung anfallen. Eine Rolle kann durch mehrere Personen ausgefüllt werden, und eine Person kann mehrere Rollen innehaben. Jeweils eine Teilmenge dieser Aufgaben, Pflichten und Rechte sind Voraussetzung für die Ausführung eines Prozesses oder einer Aktivität innerhalb eines Softwareentwicklungsprojekts. Die Instanz einer Rolle ist eine projektspezifische Rolle (z.B. Entwickler Komponente 1, Entwickler Komponente 2).

Informale Semantik

4.2.7 Tool

Ein Werkzeug unterstützt die Ausführung einer Aktivität. Bei Werkzeugen handelt es sich meistens um Softwaresysteme (Definition 3-15). Mit dem Einsatz eines Werkzeugs können verschiedene Ziele verknüpft sein, diese Ziele können sich von Aktivität zu Aktivität unterscheiden.

Abstrakte Syntax

activity Ein Werkzeug kann beliebig vielen (also auch null) Aktivitäten zugeordnet werden.

Assoziationen

context: Tool

OCL-Regeln

- [1] Werkzeuge können nur mit Aktivitäten verknüpft werden. Diese Assoziationen sind binär.

```
self.associations->forAll(a|a.connection->size = 2 AND
a.allConnections->exists(r|r.type.ocIsKindOf(Tool)) AND
a.allConnections->exists(r|r.type.ocIsKindOf(ActionState)))
```

Ein Werkzeug umfasst eine Menge von Zielen und Aufgaben, die mit dem Einsatz des Werkzeugs verfolgt werden. Dabei hängt es von der assoziierten Aktivität ab, welche Ziele und Aufgaben des Werkzeugs verfolgt werden. Die Instanz eines Werkzeugs (z.B. Textverarbeitung) ist ein spezifisches Werkzeug (z.B. Winword Version 7.0), das zur Ausübung der Aktivität eingesetzt wird. Werkzeuge werden häufig eingesetzt, um Aktivitäten zu automatisieren, zu beschleunigen oder zu erleichtern.

Informale Semantik

4.3 Activity Graphs Package

Im folgenden Abschnitt werden die Erweiterungen des Activity Graphs Package erläutert. Die bestehende Syntax und Semantik wird nicht verändert.

4.3.1 ObjectFlowState

Abstrakte Syntax	Die Metaklasse <code>ObjectFlowState</code> repräsentiert einen Objektfluss zwischen Aktivitäten (<code>ActionState</code>) in einem Aktivitätsgraphen.		
Assoziationen	<table> <tr> <td><code>mapping</code></td> <td>Für einen <code>ObjectFlowState</code> existiert gegebenenfalls eine Abbildung auf eine Menge von <code>ObjectFlowStates</code>.</td> </tr> </table>	<code>mapping</code>	Für einen <code>ObjectFlowState</code> existiert gegebenenfalls eine Abbildung auf eine Menge von <code>ObjectFlowStates</code> .
<code>mapping</code>	Für einen <code>ObjectFlowState</code> existiert gegebenenfalls eine Abbildung auf eine Menge von <code>ObjectFlowStates</code> .		
OCL-Regeln	Es sind keine weiteren OCL-Regeln außer der in der UML-Spezifikation beschriebenen vorhanden.		
Informale Semantik	Der in der UML-Spezifikation ([UML1.5]) beschriebenen informale Semantik wird folgendes hinzugefügt:		

Für jeden `ObjectFlowState` kann eine Abbildung auf eine Menge von `ObjectFlowStates` vorgesehen werden. Diese Abbildung ist für die Aggregation von Objekten und Ereignissen notwendig. Ein Objekt (z.B. Projektplan) kann gegebenenfalls in weitere Objekte (z.B. Teilprojektpläne) aufgeteilt werden. Diese Erweiterung des `ObjectFlowState` widerspricht nicht der Tatsache, dass `ObjectFlowState` ein `SimpleState` ist, da diese Abbildung `ObjectFlowStates` verschiedener Prozesse verknüpft. Die `ObjectFlowState` sind also stets `SimpleStates`, die Abbildung eine logische Verknüpfung.

4.3.2 ActionState

Abstrakte Syntax	Aktivitäten (<code>ActionState</code>) haben wir aus der UML-Spezifikation übernommen. Als Konzept haben wir die Zuordnung von Rollen zu Aktivitäten ergänzt. Einer Aktivität muss mindestens eine und kann beliebig viele Rolle zugeordnet werden. Wir erweitern die in der UML-Spezifikation ([UML1.5]) beschriebene abstrakte Syntax und informale Semantik um folgende Elemente:				
Assoziationen	<table> <tr> <td><code>role</code></td> <td>Einer Aktivität ist mindestens eine Rolle zugeordnet.</td> </tr> <tr> <td><code>tool</code></td> <td>Einer Aktivität sind beliebig viele (also auch null) Werkzeuge zugeordnet.</td> </tr> </table>	<code>role</code>	Einer Aktivität ist mindestens eine Rolle zugeordnet.	<code>tool</code>	Einer Aktivität sind beliebig viele (also auch null) Werkzeuge zugeordnet.
<code>role</code>	Einer Aktivität ist mindestens eine Rolle zugeordnet.				
<code>tool</code>	Einer Aktivität sind beliebig viele (also auch null) Werkzeuge zugeordnet.				
OCL-Regeln	context: ActionState				

Zusätzliche Operationen

ICA repräsentiert den initialen Kontext der Aktivität. Dies sind die Objekte vom Typ `ObjectFlowState`, die in die Aktivität einfließen.

```
ICA = self.incoming.source->select(o|o.oclIsTypeOf(ObjectFlowState))
```

RCA repräsentiert den resultierenden Kontext der Aktivität. Dies sind die Objekte vom Typ ObjectFlowState, die aus die Aktivität hervorgehen.

```
RCA = self.outgoing.target->select(o|o.oclIsTypeOf(ObjectFlowState))
```

Über die Metaklasse ActivityProblemMapping können Aktivitäten und Probleme miteinander verknüpft werden. Dies ist dann der Fall, wenn über die Ausführung einer Aktivität Unklarheit besteht. Aus diesem Grund wird ein Problem spezifiziert, welches genau den eingehenden und ausgehenden Objekten der Aktivität entspricht. Für dieses Problem können dann ein oder mehrere Prozessmuster spezifiziert werden.

Informale Semantik

Für jede Aktivität muss ferner definiert sein, durch welche Rolle sie ausgeführt wird. Dies bedeutet, dass die mit der Aktivität verknüpften Ziele auch mit der Rolle verknüpft sein müssen.

Für jede Aktivität können ferner null, ein oder mehrere Werkzeuge angegeben werden, die die Ausführung der Aktivität unterstützen. Dies bedeutet, dass die mit der Aktivität verknüpften Ziele auch mit den eingesetzten Werkzeugen verknüpft sein müssen.

4.3.3 OFSComposition

Die Metaklasse OFSComposition repräsentiert eine Abbildung zwischen einem ObjectFlowState und einer Menge von ObjectFlowStates. Diese Abbildung gibt an, wieviele Objekte und Ereignisse benötigt werden, um die Aggregation eines Objekts oder Ereignisses darzustellen (objects).

Abstrakte Syntax

objects	Ein OFSComposition verknüpft einen ObjectFlowState (composite) mit einer nichtleeren Menge von ObjectFlowStates (objects).
composite	Eine OFSComposition ist genau einem kompositen Objekt oder Ereignis zugeordnet.

Assoziationen

context: OFSComposition

OCL-Regeln

- [1] OFSComposition ist eine Subklasse von Association und bezeichnet ausschließlich Aggregationen von ObjectFlowStates.

```
a1, a2: OFSComposition
self.allConnections->forAll(a1, a2 | a1.aggregation=#aggregate or
a2.aggregation=#aggregate)
```

- [2] Ein zusammengesetztes Objekt muss sich wieder aus Objekten zusammensetzen.

```
self.composite.type.oclIsTypeOf(Artifact) implies
self.objects->forAll(o|o.type.oclIsTypeOf(Artifact))
```

- [3] Ein zusammengesetztes Ereignis muss sich wieder aus Ereignissen zusammensetzen.

```
self.composite.type.oclIsTypeOf(Signal) implies
self.objects->forall(o|o.type.oclIsTypeOf(Signal))
```

Informale Semantik

Objekte können sich aus anderen Objekten zusammensetzen. Dies gilt ebenso für Ereignisse. Die Aggregation von Objekten und Ereignissen setzt man aus Gründen der Übersichtlichkeit und der Abstraktion ein. Ein Supermuster kann z. B. ein Objektaggregate in seinem Kontext besitzen, während das Submuster in seinem Kontext die Aggregatbestandteile besitzt. Bei der Verfeinerung eines Problems müssen die Objekte und Ereignisse des Problems verfeinert, d.h. zerlegt werden. Einem Objekt oder Ereignis aus dem Problemkontext wird das gleiche Objekt oder Ereignis zugeordnet (d.h. es hat keine Verfeinerung des Objekts oder des Ereignisses stattgefunden, sondern es wurde nur eine 1:1-Abbildung vorgenommen) oder es wird eine Menge von Objekten oder Ereignissen zugeordnet (Abschnitt 4.5.7). Die Zerlegung bzw. Zusammensetzung eines Objekts oder Ereignisses entspricht dabei immer dem Assoziationstyp „Aggregation“, d.h. ein Objekt kann Element mehrerer Aggregate sein (Regel [1]). Die Aggregatbestandteile eines Aggregats müssen stets den gleichen Typ haben wie das Aggregat, d.h. ein Objekt kann sich nur aus Objekten zusammensetzen, ein Ereignis kann sich nur aus Ereignissen zusammensetzen (Regel [2] und [3]).

4.4 State Machines Package

Aus Vereinfachungsgründen haben wir einige wenige Änderungen am UML-Metamodell vorgenommen. Diese Änderungen betreffen ausschließlich das Metamodell von Zustandsautomaten und damit auch das von Aktivitätsdiagrammen (Abbildung 4-4). Diese Änderungen haben nur geringfügige Auswirkungen auf die Semantik des Metamodells.

4.4.1 PseudoState

Ein Pseudozustand ist kein „wirklicher“ Zustand

In der UML gibt es den Begriff des PseudoStates. Dieser Zustand ist kein „wirklicher“ Zustand, (daher die Bezeichnung Pseudozustand), sondern ein Knoten, um mehrere Transitionen miteinander zu verknüpfen. Dies ist dann notwendig, wenn eine Transition in mehrere Transitionen aufgespalten wird (per fork bzw. decision) oder mehrere Transitionen in eine Transition (per join bzw. junction) zusammengeführt werden. Bei fork und join handelt es sich um die Modellierung von Nebenläufigkeit, bei decision und junction um die Modellierung alternativer Kontrollflüsse. In der UML-Spezifikation ist eine Transition mit genau zwei Knoten (source und target) verknüpft. Komplexere Transitionen (sogenannte compound transitions, s. [UML1.5] S. 2-158 ff.) müssen daher über einen PseudoState modelliert werden.

„syntactic sugar“

PseudoStates stellen also ein Bindeglied mehrerer Transitionen dar (sog. „syntactic sugar“). Die Modellierung solcher zusammengesetzter Transitionen lässt sich vereinfachen, indem eine Transition mit einer Menge von Knoten verknüpft wird. Ein Beispiel hierfür ist bei Störrle in [Stö00] zu finden. Eshuis hat mit ähnlicher Absicht in [Esh02] für zusammengesetzte Transitionen den Begriff der „Hyperedges“ eingeführt. Abstrahiert man also von den einzelnen Transitionen einer zusammengesetzten Transition, können die Werte *join*, *fork*, *junction* und *choice* des Attributs *kind* der Metaklasse *PseudoState* entfallen. Da die Attributwerte *deepHistory* und *ShallowHistory* für unsere Belange keine Rolle spielen, können sie

ebenfalls entfallen. Der Attributwert *initial* wird als neues Attribut in den Metaklassen CompositeState und SimpleState eingeführt und kann daher bei PseudoState entfallen. Auf diese Weise kann die gesamte Metaklasse PseudoState entfallen.

4.5 Relationship Package

Das Relationship Package beinhaltet alle notwendigen Konzepte, um Beziehungen zwischen Prozessmustern beschreiben zu können (Abbildung 4-6).

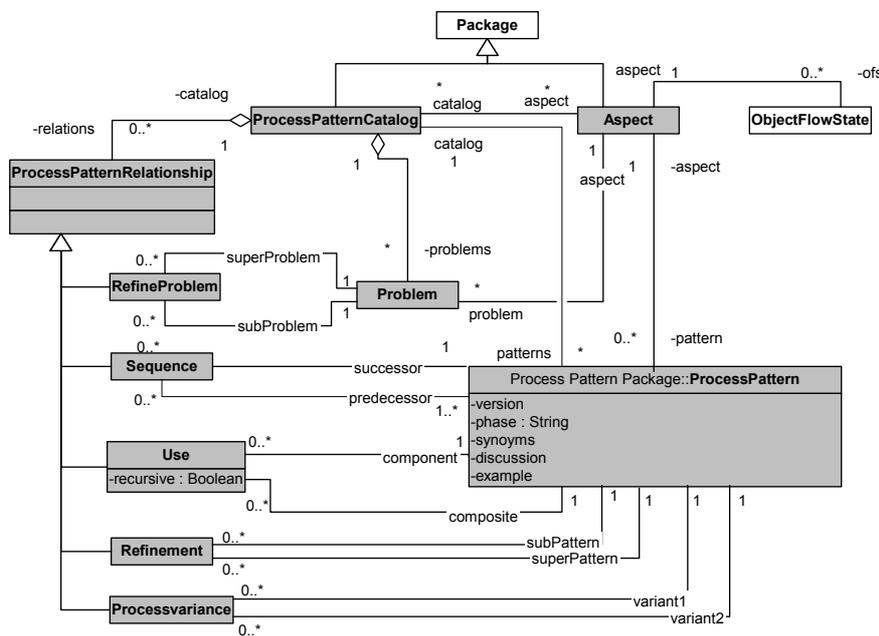


Abbildung 4-6: Abstrakte Syntax des Pakets Process Pattern Relationship

4.5.1 ObjectFlowState

S. Abschnitt 4.3.1.

Abstrakte Syntax

Assoziationen

aspect Ein ObjectFlowState wird genau einem Aspekt zugeordnet.

S. Abschnitt 4.3.1.

OCL-Regeln

S. Abschnitt 4.3.1.

Informale Semantik

4.5.2 Problem

Ein Problem wird thematisch eingeordnet (aspect).

Abstrakte Syntax

Assoziationen	aspect	Ein Problem wird genau einem Aspekt zugeordnet.
OCL-Regeln	S. Abschnitt 4.2.2.	
Informale Semantik	S. Abschnitt 4.2.2.	

4.5.3 ProcessPattern

Abstrakte Syntax	Ein Prozessmuster ist Teil des Aggregats ProcessPatternCatalog, d.h. ein Prozessmuster wird stets einem Prozessmusterkatalog zugeordnet (catalog). Ein Prozessmuster besitzt ferner beliebig viele Assoziationen mit verschiedenen Beziehungen (Relationship), die das Prozessmuster mit einem anderen Prozessmuster verknüpfen. Das Prozessmuster kann zudem thematisch eingeordnet werden (aspect).	
Assoziationen	catalog	Ein Prozessmuster wird genau einem Prozessmusterkatalog zugeordnet.
	aspect	Ein Prozessmuster wird genau einem Aspekt zugeordnet.
OCL-Regeln	S. Abschnitt 4.2.1.	
Informale Semantik	S. Abschnitt 4.2.1.	

4.5.4 ProcessPatternRelationship

Abstrakte Syntax	Die Metaklasse ProcessPatternRelationship ist ein Supertyp für alle Prozessmusterbeziehungen. Eine Beziehung wird einem Prozessmusterkatalog zugeordnet (catalog).	
Assoziationen	catalog	Eine Beziehung wird genau einem Prozessmusterkatalog zugeordnet.
OCL-Regeln	Es sind keine OCL-Regeln vorhanden.	
Informale Semantik	Für die Klasse ProcessPatternRelationship besteht keine spezifische informale Semantik.	

4.5.5 Sequence

Abstrakte Syntax	Die Sequence-Beziehung verknüpft ein oder mehrere vorausgehende Prozessmuster (Vorgänger) und ein nachfolgendes Prozessmuster (Nachfolger). Zwischen mehreren Prozessmustern besteht eine solche Beziehung, wenn die Vorgängermuster alle Objekte und Ereignisse produzieren, die das Nachfolgermuster für seine Ausführung benötigt (Definition 3-16). Die Verknüpfung zu den Prozessmustern einer Sequence-Beziehung wird durch die beiden Assoziationen predecessor und successor realisiert.	
Assoziationen	predecessor	Zu einer Sequence-Beziehung gehören mindestens ein Vorgänger.
	successor	Zu einer Sequence-Beziehung gehört genau ein Nachfolger.

OCL-Regeln

- [1] Besteht zwischen zwei oder mehreren Prozessmustern eine Sequence-Beziehung, so muss für jedes Element des initialen Kontexts des Nachfolgers folgendes gelten:
 Das Element ist Element der Vereinigungsmenge der resultierenden Kontexte aller Vorgänger oder
 das Element ist ein Aggregat und alle Aggregatbestandteile sind Elemente der Vereinigungsmenge der resultierenden Kontexte aller Vorgänger oder
 das Element ist Bestandteil eines Aggregats, welches Element der Vereinigungsmenge aller resultierenden Kontexte aller Vorgänger ist.

```

s, t: ObjectFlowState
self.successor.initial.states->forall(s |
self.predecessor->allStates->includes(s) OR
self.predecessor->allStates->includesAll(s.mapping.objects) OR
self.predecessor->allStates->exists(t | t.mapping.objects->includes(s))

```

Zusätzliche Operationen

- [2] allStates ist die Menge aller Elemente des resultierenden Kontexts aller Vorgängermusters.

```

allStates : Set(ObjectFlowState)
allStates->union(self.predecessor->resultingContext)

```

Eine Sequence-Beziehung repräsentiert eine Verknüpfung zwischen mehreren Prozessmustern. Die Prozessmuster stellen eine geordnete Menge von Elementen dar, d.h. ein Tupel ($\text{predecessor}_1, \dots, \text{predecessor}_n, \text{successor}$). Alle Objekte und Ereignisse, die das Nachfolgermuster benötigt (definiert durch den initialen Kontext des Nachfolgermusters) werden durch den resultierenden Kontext der Vorgängermuster erzeugt. Hieraus ergeben sich zwei Bedeutungen für ein ($\text{predecessor}_1, \dots, \text{predecessor}_n, \text{successor}$)-Tupel: 1. Wurden alle Vorgängermuster angewendet, ist die Sequence-Beziehung ein Hinweis, dass als nächstes Prozessmuster das Nachfolgermuster angewendet werden kann. 2. Wird die Anwendung des Nachfolgermusters verlangt und sind nicht alle Eingangsvoraussetzungen (d.h. des initialen Kontexts) vorhanden, ist die Sequence-Beziehung ein Hinweis, dass zur Herstellung der fehlenden Eingangsvoraussetzungen die Vorgängermuster angewendet werden können.

Informale Semantik

4.5.6 Use

Die Use-Beziehung setzt jeweils zwei Prozessmuster (Komposit- und Komponentenmuster) zueinander in Beziehung. Zwischen zwei Prozessmustern besteht eine solche Beziehung, wenn das Komponentenmuster einen Teilprozess des Kompositmusters (d.h. einen Teil der Lösung) beschreibt (Definition 3-17). Die Verknüpfung zu den beiden Prozessmustern einer Sequence-Beziehung wird durch die beiden Assoziationen `composite` und `component` realisiert.

Abstrakte Syntax

<code>composite</code>	Zu einer Use-Beziehung gehört genau ein Kompositmuster.
<code>component</code>	Zu einer Use-Beziehung gehört genau ein Komponentenmuster.

Assoziationen

`recursive` Das Attribut zeigt an, ob es sich bei der Use-Beziehung um eine rekursive Beziehung handelt.

Attribute

OCL-Regeln **context: Use**

- [1] Voraussetzung für eine Use-Beziehung ist, dass mindestens ein Aktivitäts-Problem-Paar im Prozess des Kompositmusters vorkommt.

```
self.composite.process.mapping->notEmpty()
```

- [2] Das Problem eines dieser Aktivitäts-Problem-Paare entspricht dann dem Problem des Komponentenmusters.

```
q: ActivityProblemMapping
self.composite.process.mapping->exists(q|q.subproblem =
self.component.problem)
```

- [3] Das Komponentenmuster beschreibt einen bestimmten Ausschnitt aus dem Prozessgraphen des Kompositmusters und zwar eine Aktivität und deren eingehenden und ausgehenden Objekte und Ereignisse (ObjectFlowState). Für jeden Eingangszustand der Aktivität gilt: Der Eingangszustand (oder seine Aggregatbestandteile) ist (sind) im initialen Kontext des Komponentenmusters enthalten. Diese Annahme gilt analog für die Ausgangszustände der Aktivität und den resultierenden Kontext des Komponentenmusters. Desweiteren enthält der Kontext des Komponentenmusters keine weiteren Elemente.

```
c1, c2: ProcessPattern
t: Transition
s: StateVertex
a: ActionState
c1 = self.composite
c2 = self.component
```

```
a = c1.process.top.subvertex->any(x|x.ocliIsTypeOf(ActionState)
AND c1.process.mapping->exists(q|q.subactivity=x))
```

```
aInput = a.incoming->collect(s|s=t.source AND s.ocliIsTypeOf(ObjectFlowState))
```

```
aOutput = a.outgoing->collect(s|s=t.target AND s.ocliIsTypeOf(ObjectFlowState))
```

```
aInput->forAll(x|c2.initialContext->includes(x) OR c2.initialContext->includesAll(x.mapping.objects)) AND
c2.initialContext->forAll(y|aInput->includes(y) OR aInput->exists(z|z.mapping.objects->includes(x))
```

```
aOutput->forAll(x|c2.resultingContext->includes(x) OR c2.resultingContext->includesAll(x.mapping.objects)) AND
c2.resultingContext->forAll(y|aOutput->includes(y) OR aOutput->exists(z|z.mapping.objects->includes(x))
```

- [4] Gibt es in einem Musterkatalog zwei Prozessmuster, die über ActivityProblemMapping miteinander verknüpft sind, so besteht zwischen diesen beiden Mustern eine Use-Beziehung.

```

c1, c2: ProcessPattern
ppc: ProcessPatternCatalog
r: Relationship

(ppc.patterns->includes(c1) AND ppc.patterns->includes(c2) AND
c1.process.mapping->select(m|m.subproblem=c2.problem))

implies

(ppc.relations->select(r|r.oclIsKindOf(Use) AND r.composite=c1
AND r.component=c2)->size=1)

```

- [5] Steht ein Prozessmuster mit sich selbst in einer Use-Beziehung, so handelt es sich um eine rekursive Use-Beziehung.

```

self.composite = self.component implies self.recursive = true

```

Eine Use-Beziehung repräsentiert eine Verknüpfung zwischen jeweils zwei Prozessmustern. Diese zwei Prozessmuster stellen eine geordnete Menge von Elementen dar, d.h. ein Tupel (composite, component). Zwischen Kompositmuster und Komponentenmuster besteht eine Use-Beziehung, wenn das Komponentenmuster einen Teilprozess des Kompositmusters, d.h. einen Teil der Lösung, beschreibt.

Informale Semantik

Hieraus ergibt sich folgende Bedeutung für ein (composite, component)-Tupel: Stellt der Muster-Anwender im Rahmen der Anwendung des Kompositmusters fest, dass er für die Ausführung einer Aktivität mehr Detailwissen notwendig ist, kann zur Erlangung dieses Detailwissens das Komponentenmuster angewendet werden. Nachdem das Komponentenmuster vollständig angewendet wurde, kann mit der Anwendung des Kompositmusters fortgefahren werden. Bei einer rekursiven Use-Beziehung bedeutet dies, dass mit der Anwendung der Instanz des Prozessmusters begonnen wird, dann eine weitere Instanz dieses Prozessmusters erzeugt wird, und erst wenn deren Anwendung abgeschlossen wurde, mit der ersten Instanz fortgefahren werden kann.

Für das Bestehen einer Use-Beziehung gibt es verschiedene Voraussetzungen: [1] Jeder Prozess besitzt eine beliebige Menge von Aktivitäts-Problem-Paaren. Ein Aktivitäts-Problem-Paar bedeutet, dass einer Aktivität ein entsprechendes Problem zugeordnet werden konnte. Das Problem spezifiziert, welche Objekte und Ereignisse notwendig sind, um die Aktivität durchzuführen bzw. welche Objekte und Ereignisse Ergebnis der Aktivitätsdurchführung sind. D.h. das Problem spezifiziert den Kontext der Aktivität. Diese Problemzuordnung ist die Voraussetzung, um ein Prozessmuster zuordnen zu können, welches beschreibt, wie das Problem gelöst und die Aktivität durchgeführt werden kann (s. auch Abschnitt 4.2.5). Voraussetzung für eine Use-Beziehung ist daher, dass mindestens ein Aktivitäts-Problem-Paar mit dem Prozess des Kompositmusters assoziiert werden kann. [2] Das Problem eines dieser Aktivitäts-Problem-Paare entspricht dann dem Problem des Komponentenmusters. Voraussetzung für eine Use-Beziehung ist daher, dass ein Prozessmuster nur dann ein Komponentenmuster in einer Use-Beziehung ist, wenn dieses Prozessmuster ein Problem löst, welches durch eine Aktivität des Kompositmusters adressiert wird. [3] Das Komponentenmuster beschreibt einen bestimmten Ausschnitt aus dem Prozess des Kompositmusters und zwar

eine Aktivität und deren eingehenden und ausgehenden Objekte und Ereignisse (Object-FlowState). Die Eingangszustände der Aktivität entsprechen dem initialen Kontext des Komponentenmusters. Die Ausgangszustände der Aktivität entsprechen analog dem resultierenden Kontext des Komponentenmusters. Voraussetzung für eine Use-Beziehung ist also, dass ein Kompositmuster nur dann von einem Komponentenmuster „genutzt“ werden kann, wenn das Kompositmuster genau den gleichen Kontext besitzt wie die Aktivität. [4] Gibt es in einem Musterkatalog zwei Prozessmuster *c1* und *c2*, wobei ein Subproblem von *c1* identisch mit dem Problem von *c2* ist, so besteht zwischen diesen beiden Prozessmustern eine Use-Beziehung. [5] Steht ein Prozessmuster mit sich selbst in einer Use-Beziehung, so handelt es sich um eine rekursive Use-Beziehung. Ein Prozessmuster kann sich also selbst nutzen. Dies bedeutet, dass vom einem Prozessmuster (mindestens) zwei Instanzen erzeugt werden. Der Anwender beginnt mit der Ausführung der ersten Instanz. Diese Instanz benutzt einer weitere Instanz des Prozessmusters. Diese zweite Instanz muss zunächst ausgeführt werden, bevor mit der Ausführung der ersten Instanz fortgefahren werden kann.

4.5.7 RefineProblem

Abstrakte Syntax Die RefineProblem-Beziehung setzt jeweils zwei Probleme (Superproblem und Subproblem) zueinander in Beziehung. Zwischen zwei Problemen besteht eine solche Beziehung, wenn das Subproblem das Superproblem verfeinert (Definition 3-19). Die Verknüpfung zu den beiden Problemen einer RefineProblem-Beziehung wird durch die beiden Assoziationen *superProblem* und *subProblem* realisiert.

Assoziationen

<i>superProblem</i>	Zu einer RefineProblem-Beziehung gehört genau ein Superproblem.
<i>subProblem</i>	Zu einer RefineProblem-Beziehung gehört genau ein Subproblem.

OCL-Regeln

context: RefineProblem

[1] Zwei Probleme, die in einer RefineProblem-Beziehung stehen, sind verschiedene Probleme.

```
self.subProblem <> self.superProblem
```

[2] Im Rahmen der Problemverfeinerung wird der initiale Kontext des Superproblems auf den initialen Kontext des Subproblems abgebildet. Die Abbildung sieht vor, dass jedes Element oder alle seine Aggregatbestandteile des initialen Superproblemkontexts im initialen Subproblemkontext enthalten sind. Darüber hinaus können im initialen Subproblemkontext weitere Objekte und Ereignisse enthalten sein, die für die Verfeinerung notwendig sind.

```
x:ObjectFlowState
self.superProblem.initial.states->forall (x |
self.subProblem.initial.states->includes(x) or
self.subProblem.initial.states->includesAll(x.mapping.objects))
```

- [3] Im Rahmen der Problemverfeinerung wird der resultierende Kontext des Superproblems auf den resultierenden Kontext des Subproblems abgebildet. Die Abbildung sieht vor, dass jedes Element oder alle seine Aggregatbestandteile des resultierenden Superproblemkontexts im resultierenden Subproblemkontext enthalten sind. Darüber hinaus können im resultierenden Subproblemkontext weitere Objekte und Ereignisse enthalten sein, die für die Verfeinerung notwendig sind.

```
x:ObjectFlowState
self.superProblem.resulting.states->forAll (x |
self.subProblem.resulting.states->includes(x) or
self.subProblem.resulting.states->includesAll(x.map-
ping.objects))
```

Eine RefineProblem-Beziehung repräsentiert eine Verknüpfung zwischen jeweils zwei Problemen. Diese zwei Probleme stellen eine geordnete Menge von Elementen dar, d.h. ein Tupel (superProblem, subProblem). Zwischen Superproblem und Subproblem besteht eine RefineProblem-Beziehung, wenn das Subproblem eine Verfeinerung des Superproblems beschreibt. Für das Bestehen einer RefineProblem-Beziehung gibt es verschiedene Voraussetzungen:

Informale Semantik

[1]: Die Probleme einer RefineProblem-Beziehung müssen verschiedene Probleme sein. Dies liegt darin begründet, dass ein Problem niemals seine eigene Verfeinerung sein kann.

[2] und [3]: Im Rahmen der Problemverfeinerung wird ein Objekt oder Ereignis des Kontexts des zu verfeinernden Superproblems auf eine Menge von Objekten und Ereignissen des Kontexts des verfeinernden Subproblems abgebildet. Durch diese Abbildung von ObjectFlowStates auf eine Menge von ObjectFlowStates können Objekte und Ereignisse z.B. zum Zwecke ihrer Zerlegung verfeinert werden. Beispielsweise kann das Objekt „Projektplan“ auf die Objekte „Iterationsplan“, „Risikomanagementplan“, Konfigurationsmanagementplan“ und „Testplan“ abgebildet werden. Im Kontexts des Subproblems können zusätzliche Objekte und Ereignisse enthalten sein, die für die Verfeinerung notwendig sind.

4.5.8 Refinement

Die Refinement-Beziehung setzt jeweils zwei Prozessmuster (Superpattern und Subpattern) zueinander in Beziehung. Zwischen zwei Prozessmustern besteht eine solche Beziehung, wenn das Subpattern die Spezialisierung des Superpatterns darstellt (Definition 3-18). Die Instanz einer Refinement-Beziehung ist ein geordnetes Paar von Prozessmuster-Instanzen. Die Verknüpfung zu den beiden Prozessmustern einer Refinement-Beziehung wird durch die beiden Assoziationen *superPattern* und *subPattern* realisiert.

Abstrakte Syntax

superPattern	Zu einer Refinement-Beziehung gehört genau ein superPattern.
subPattern	Zu einer Refinement-Beziehung gehört genau ein subPattern.

Assoziationen

context: Refinement

OCL-Regeln

- [1] Zwei Prozessmuster, die in einer Refinement-Beziehung stehen, lösen verschiedene Probleme

```
not (self.subPattern.problem = self.superPattern.problem)
```

- [2] superPattern und subPattern besitzen zwei unterschiedliche Prozesse.

```
not (self.subPattern.process = self.superPattern.process)
```

- [3] Die Verfeinerung eines Prozessmusters setzt die Verfeinerung des zugehörigen Problems voraus.

```
r: ProcessPatternRelationship  
self.catalog.relations->exists (r|r.oc1IsTypeOf (RefineProblem)  
AND  
r.superProblem = self.superPattern.problem AND  
r.subProblem = self.subPattern.problem)
```

Informale Semantik

Eine Refinement-Beziehung repräsentiert eine Verknüpfung zwischen jeweils zwei Prozessmustern. Diese zwei Prozessmuster stellen eine geordnete Menge von Elementen dar, d.h. ein Tupel (superPattern, subPattern). Zwischen Superpattern und Subpattern besteht eine Refinement-Beziehung, wenn das Subpattern eine Verfeinerung des Superpatterns beschreibt. Für das Bestehen einer Refinement-Beziehung gibt es verschiedene Voraussetzungen: [1] Die Probleme, die jeweils von Superpattern und Subpattern gelöst werden, müssen verschiedene Probleme sein. Dies bedeutet ferner, dass Superpattern und Subpattern verschiedene Prozessmuster sind (Abschnitt 4.2.1). Dies liegt darin begründet, dass ein Problem bzw. ein Prozessmuster niemals seine eigene Verfeinerung sein kann. [2] Die Prozesse von Superpattern und Subpattern müssen ebenfalls verschieden sein. Dies liegt daran, dass durch Forderung [1] für die beiden Prozesse verschiedene Kontexte gelten. Ferner kann es sein, dass der Prozess des Superpatterns andere Aktivitäten besitzt als das Subpattern. Dies wird jedoch nicht explizit als Forderung modelliert, da die Veränderung (Hinzufügen neuer Aktivitäten, Aufsplitten vorhandener Aktivitäten) der Aktivitäten im Rahmen der Verfeinerung eine Entscheidung des Modellierers ist. [3] Die Verfeinerung eines Prozessmusters setzt die Verfeinerung des zugehörigen Problems voraus.

4.5.9 Processvariance

Abstrakte Syntax

Die Processvariance-Beziehung setzt jeweils zwei Prozessmuster (Variante1 und Variante2) zueinander in Beziehung. Zwischen zwei Prozessmustern besteht eine solche Beziehung, wenn Variante1 und Variante2 zwar das gleiche Problem lösen, dafür aber unterschiedliche Lösungen anbieten (Definition 3-20). Die Verknüpfung zu den beiden Prozessmustern einer Processvariance-Beziehung wird durch die beiden Assoziationen variant1 und variant2 realisiert.

Assoziationen

variant1	Erste Variante in der Processvariance-Beziehung.
variant2	Zweite Variante in der Processvariance-Beziehung.

context: Processvariance

OCL-Regeln

- [1] Beide Prozessvarianten lösen das gleiche Problem oder es existiert ein Mapping der beiden Probleme aufeinander.

```

s,t: ObjectFlowState
p1,p2: Problem
p1 = self.processvariant1.problem
p2 = self.processvariant2.problem

p1 = p2
OR (
  p1.initial.states->forAll(s|
    p2.initial.states->includes(s) OR
    p2.initial.states->includesAll(s.mapping.objects) OR
    p2.initial.states->exists(t|t.mapping.objects->includes(s))
  )
  AND
  p1.resulting.states->forAll(s|
    p2.resulting.states->includes(s) OR
    p2.resulting.states->includesAll(s.mapping.objects) OR
    p2.resulting.states->exists(t|t.mapping.objects->includes(s))
  )
  AND
  p2.initial.states->forAll(s|
    p1.initial.states->includes(s) OR
    p1.initial.states->includesAll(s.mapping.objects) OR
    p1.initial.states->exists(t|t.mapping.objects->includes(s))
  )
  AND
  (p2.resulting.states->forAll(s|
    p1.resulting.states->includes(s) OR
    p1.resulting.states->includesAll(s.mapping.objects) OR
    p1.resulting.states->exists(t|t.mapping.objects->includes(s))
  )
)

```

- [2] Der Prozess der beiden Varianten ist verschieden.

```

not (self.processVariant1.process = self.processVariant2.pro-
cess)

```

Eine Processvariance-Beziehung repräsentiert eine Verknüpfung zwischen jeweils zwei Prozessmustern. Diese zwei Prozessmuster stellen eine geordnete Menge von Elementen dar, d.h. ein Tupel (variant1, variant2). Zwischen variant1 und variant2 besteht eine Processvariance-Beziehung, wenn beide Prozessmuster variante Prozesse zu dem gleichen Problem anbieten. Für das Bestehen einer Processvariance-Beziehung gibt es verschiedene Voraussetzungen: [1] Die Probleme, die jeweils von den beiden Prozessvarianten gelöst werden, müssen identisch sein. [2] Die Prozesse der beiden Prozessvarianten müssen jedoch verschieden sein. Diese Bedingung impliziert ferner, dass die beiden Prozessvarianten verschiedene Prozessmuster sind (Abschnitt 4.2.1).

Informale Semantik

4.5.10 ProcessPatternCatalog

Abstrakte Syntax	Ein Prozessmusterkatalog repräsentiert eine Menge von Prozessmustern und Beziehungen, die Prozessmuster miteinander verknüpfen.								
Assoziationen	<table> <tr> <td>patterns</td> <td>Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Prozessmuster.</td> </tr> <tr> <td>problems</td> <td>Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Probleme.</td> </tr> <tr> <td>relations</td> <td>Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Beziehungen.</td> </tr> <tr> <td>aspect</td> <td>Ein Prozessmusterkatalog kann beliebig viele Aspekte besitzen.</td> </tr> </table>	patterns	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Prozessmuster.	problems	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Probleme.	relations	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Beziehungen.	aspect	Ein Prozessmusterkatalog kann beliebig viele Aspekte besitzen.
patterns	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Prozessmuster.								
problems	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Probleme.								
relations	Innerhalb eines Prozessmusterkatalogs gibt es beliebig viele Beziehungen.								
aspect	Ein Prozessmusterkatalog kann beliebig viele Aspekte besitzen.								
OCL-Regeln	Keine OCL-Regeln vorhanden.								
Informale Semantik	Ein Prozessmusterkatalog präsentiert eine Menge von Prozessmustern und deren Beziehungen. Prozessmusterkataloge dienen dazu, Mengen von Prozessmustern zu kapseln. Diese Kapselung dient der besseren Auffindbarkeit und Wiederverwendung von Prozessmustern. Das Katalogdiagramm dient daher als strukturelle Übersicht über den Katalog. So könnte es beispielsweise einen Prozessmusterkatalog für die komponentenbasierte oder für die weborientierte Softwareentwicklung geben. Prozessmuster innerhalb eines Prozessmusterkatalogs sollten daher thematisch gleich ausgerichtet sein, d.h. einer Domäne angehören (Definition 3-3).								

4.5.11 Aspect

Abstrakte Syntax	Der Aspekt ist Subklasse der Metaklasse Package und bündelt eine Menge von Prozessmustern, Problemen und ObjectFlowStates. Der Aspekt gibt Aufschluss über die thematische Ausrichtung dieser Elemente.								
	Mögliche Ausprägungen des Attributs sind z.B.: project acquisition, project management, risk management usw.								
Assoziationen	<table> <tr> <td>pattern</td> <td>Ein Aspekt kann beliebig vielen Prozessmustern zugeordnet werden.</td> </tr> <tr> <td>problem</td> <td>Ein Aspekt kann beliebig vielen Problemen zugeordnet werden.</td> </tr> <tr> <td>catalog</td> <td>Ein Aspekt kann beliebig vielen Prozessmusterkatalogen zugeordnet werden.</td> </tr> <tr> <td>ofs</td> <td>Ein Aspekt kann beliebig vielen ObjectFlowStates zugeordnet werden.</td> </tr> </table>	pattern	Ein Aspekt kann beliebig vielen Prozessmustern zugeordnet werden.	problem	Ein Aspekt kann beliebig vielen Problemen zugeordnet werden.	catalog	Ein Aspekt kann beliebig vielen Prozessmusterkatalogen zugeordnet werden.	ofs	Ein Aspekt kann beliebig vielen ObjectFlowStates zugeordnet werden.
pattern	Ein Aspekt kann beliebig vielen Prozessmustern zugeordnet werden.								
problem	Ein Aspekt kann beliebig vielen Problemen zugeordnet werden.								
catalog	Ein Aspekt kann beliebig vielen Prozessmusterkatalogen zugeordnet werden.								
ofs	Ein Aspekt kann beliebig vielen ObjectFlowStates zugeordnet werden.								
OCL-Regeln	Es sind keine OCL-Regeln vorhanden.								
Informale Semantik	Ein Aspekt fasst eine Menge von Prozessmustern, Problemen und Objekten und Ereignissen zusammen, die thematisch zusammengehören. Der Aspekt ist ein Hilfsmittel, um einen Prozessmusterkatalog thematisch zu strukturieren.								

4.6 Process Pattern Graph Package

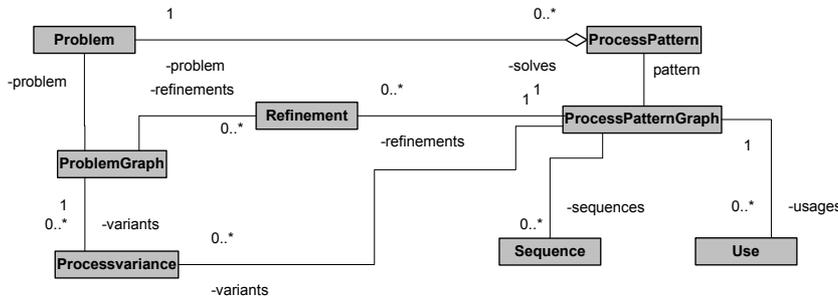


Abbildung 4-7: Abstrakte Syntax des Pakets Process Pattern Graphs – Graphs

4.6.1 ProblemGraph

Das Problemdiagramm (ProblemGraph) verkörpert ein bestimmtes Problem (problem) und zeigt auf, welche Prozessmuster dieses Problem lösen. Das Problemdiagramm zeigt diese Prozessmuster unter Zuhilfenahme der Processvariance-Beziehung (variants) und der Refinement-Beziehung (refinements).

Abstrakte Syntax

problem	Das Problemdiagramm beschreibt genau ein Problem.
variants	Das Problemdiagramm kann beliebige viele Prozessvarianten enthalten.
refinements	Das Problemdiagramm kann beliebige viele Patternverfeinerungen enthalten.

Assoziationen

context: ProblemGraph

OCL-Regeln

- [1] Enthält ein Problemdiagramm Prozessvarianten, so sind die Probleme aller Prozessvarianten identisch mit dem Problem, welches dem Problemdiagramm zugeordnet ist.

```
v: Processvariance
self.variants->forAll (v |
v.processvariant1.problem = self.problem AND
v.processvariant2.problem = self.problem)
```

- [2] Enthält ein Problemdiagramm Musterverfeinerungen, so sind die Probleme der Supermuster identisch mit dem Problem, welches dem Problemdiagramm zugeordnet ist. Die Probleme der Submuster müssen sich jedoch von dem Problem des Problemdiagramms unterscheiden, da sie laut Refinement-Beziehung Verfeinerungen dieses Problems darstellen.

```
r: Relationship
self.refinements->forAll (r |
r.superpattern.problem = self.problem AND
r.subpattern.problem <> self.problem)
```

Informale Semantik Ein Problemdiagramm repräsentiert alle zu einem Problem zugehörigen Informationen. Hierzu zählt der Name des Problems, Kontexts des Problems, die natürlichsprachige Beschreibung des Problems, sowie die Darstellung aller Prozessmuster, die dieses Problem lösen. Um alle in Frage kommenden Prozessmuster zu ermitteln, werden Instanzen der Processvariance- und der Refinement-Beziehung überprüft. D.h. alle Prozessvarianten und alle Verfeinerungen dieser Prozessvarianten werden im Problemdiagramm aufgeführt.

4.6.2 ProcessPatternGraph

Abstrakte Syntax Das Prozessmusterdiagramm (ProcessPatternGraph) verkörpert ein bestimmtes Prozessmuster (pattern). Ferner zeigt das Prozessmusterdiagramm, an welchen Beziehungen das Prozessmuster beteiligt ist (Assoziationen sequences, usages, refinements, variants).

Assoziationen

pattern	Das Prozessmusterdiagramm stellt alle Informationen bzgl. eines Prozessmusters dar.
sequences	Das Prozessmusterdiagramm kann beliebige viele Sequence-Beziehungen enthalten.
usages	Das Prozessmusterdiagramm kann beliebige viele Use-Beziehungen enthalten.
refinements	Das Prozessmusterdiagramm kann beliebige viele Refinements-Beziehungen enthalten.
variants	Das Prozessmusterdiagramm kann beliebige viele Processvariance-Beziehungen enthalten.

OCL-Regeln

context: ProcessPatternGraph

- [1] Enthält ein Prozessmusterdiagramm Sequence-Beziehungen, so muss das dem Prozessmusterdiagramm zugeordnete Prozessmuster das Vorgänger- oder das Nachfolgermuster in diesen Beziehungen repräsentieren.

```
s : Sequence
self.sequences->forAll (s |
s.predecessor->exists(self.pattern) OR s.successor = self.pattern)
```

- [2] Enthält ein Prozessmusterdiagramm Use-Beziehungen, so muss das dem Prozessmusterdiagramm zugeordnete Prozessmuster das Komposit- oder das Komponentenmuster in diesen Beziehungen repräsentieren.

```
u : Use
self.usages->forAll (u |
u.composite = self.pattern OR u.component = self.pattern)
```

- [3] Enthält ein Prozessmusterdiagramm Refinement-Beziehungen, so muss das dem Prozessmusterdiagramm zugeordnete Prozessmuster das Super- oder das Submuster in diesen Beziehungen repräsentieren.

```
r : Refinement  
self.refinements->forall(r|  
r.superpattern = self.pattern OR r.subpattern = self.pattern)
```

- [4] Enthält ein Prozessmusterdiagramm Processvariance-Beziehungen, so muss das dem Prozessmusterdiagramm zugeordnete Prozessmuster eine der beiden Varianten in diesen Beziehungen repräsentieren.

```
v : Processvariance  
self.variants->forall(v|  
v.processvariant1 = self.pattern OR v.processvariant2 =  
self.pattern)
```

Ein Prozessmusterdiagramm repräsentiert eine Teilmenge der zu einem Prozessmuster zugehörigen Informationen (Abschnitt 3.4), und zwar Name, Aspekt, Problem, Related Patterns, Prozess und die Rolle. Der Kontext des Prozessmusters wird über das zugehörige Problemdiagramm geliefert. Weitere Informationen wie Version, Synonyms, die Discussion und Examples können natürlichsprachlich ergänzt werden.

Informale Semantik

5 Semantik

In diesem Kapitel formalisieren wir die Semantik der von uns betrachteten und neu eingeführten UML-Konzepte. Dies ist notwendig, da die UML keine formale Semantik besitzt (Abschnitt 2.3). Die von uns formulierte Semantik ist von denotationaler Art. Im Rahmen einer denotationalen Semantik werden für jede syntaktische Domäne semantische Funktionen definiert, welche die syntaktischen Elemente auf mathematische Objekte abbilden (s. [NN99] und [SK95] für eine Einführung). Die denotationale Definition einer Sprache besteht also aus fünf Komponenten (Abbildung 5-1): Syntaktische Domänen bezeichnen Mengen von syntaktischen Objekten. Syntaktische Regeln geben an, wie Objekte der syntaktischen Domänen strukturiert und miteinander verknüpft werden. Semantische Domänen bezeichnen Mengen mathematischer Objekte. Semantische Funktionen bilden die syntaktischen Domänen auf die semantischen Domänen ab. Semantische Gleichungen schließlich spezifizieren die semantischen Funktionen für einzelne syntaktische und semantische Objekte.

Vorgehen

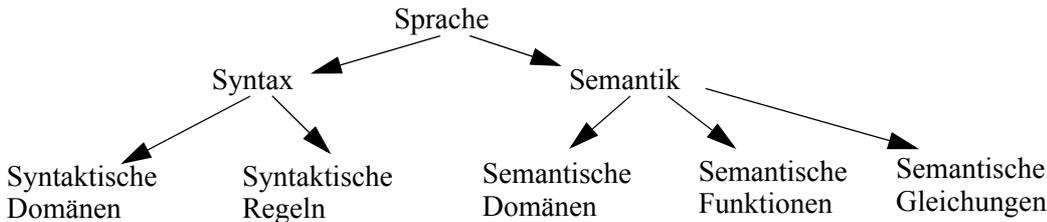


Abbildung 5-1: Aufteilung einer Sprache in Syntax und Semantik

5.1 Syntaktische Regeln

Die Syntax der UML wird mit Klassendiagrammen und OCL-Constraints beschrieben. Da wir die semantische Abbildung nicht direkt auf dem PROPEL-Metamodell definieren können, müssen wir syntaktische Domänen definieren, die den Definitionsbereich der semantischen Funktionen repräsentieren. Die Menge der syntaktischen Domänen ist im Anhang B.1 aufgeführt. Die syntaktischen Regeln für die syntaktischen Domänen, welche die Struktur von PROPEL repräsentieren, definieren wir in diesem Abschnitt.

Syntaktische Domänen und Regeln

Vom PROPEL-Metamodell leiten wir eine kontextfreie Grammatik in der erweiterten Backus-Naur-Form (EBNF) ab. Eine ähnliche Vorgehensweise verwendet Störrle in [Stö00] zur Ableitung einer kontextfreien Grammatik für Zustandsautomaten.

Ableiten einer kontextfreien Grammatik

Wir unterscheiden Nichtterminale, **Terminale** und *Werte* typographisch. Die Darstellung von Nichtterminalen entspricht bzgl. Font, Schriftgröße und Schriftweise der Darstellung von PROPEL-Metaklassen. Diese Übereinstimmung ist gewollt, da die Nichtterminale entweder PROPEL-Metaklassen oder abstrakte Superklassen wie z.B. Z repräsentieren.

Schreibweisen

Regelschema Zunächst geben wir ein Schema für die Beschreibung der Metaklassen an. Die jeweils linke Seite einer Regel besteht aus einem Nichtterminal *MetaClass*, welches genau eine syntaktische Domäne verkörpert. Die jeweils rechte Seite einer Grammatikregel definiert dann die Struktur dieses Nichtterminals. Die Regel beginnt auf der rechten Seite mit dem Terminalsymbol **mc**, das die Metaklasse identifiziert. Attribute einer Metaklasse werden durch das Terminalsymbol **attribute** repräsentiert, der Typ der Attribute durch das Terminalsymbol *type*. Assoziationen einer Metaklasse werden durch das Terminalsymbol **rolename** repräsentiert, der Typ der Assoziation durch das Terminalsymbol *type*. Bool'sche Attribute werden durch das Terminalsymbol **[booleanAttribute]** dargestellt, wobei der boolsche Wert durch die Optionalität der eckigen Klammern ausgedrückt wird. Die Produktion wird mit der Angabe des Nichtterminalsymbols *SuperClass* für die Superklasse der Metaklasse abgeschlossen. Nichtterminalsymbole werden also stets in Normalschrift, Terminalsymbole in Fett- oder Kursivschrift dargestellt. Die Regel zur Definition der abstrakten Syntax von Metaklassen lautet somit:

$MetaClass ::= mc : \mathbf{attribute} \textit{type} [\mathbf{booleanAttribute}] \mathbf{rolename} \textit{type} \dots SuperClass$

Für die Abbildung von Spezialisierungsbeziehungen (d.h. der Darstellung einer Superklasse und ihren Subklassen) verwenden wir folgende Regel:

$MC ::= MetaClass \mid SubClass_1 \mid \dots \mid Subclass_n$

Syntaxregeln Die nachfolgenden Syntaxregeln (Tabelle 5-1) wurden von dem PROPEL-Metamodell abgeleitet (Abbildungen 4-3, 4-4, 4-6 und 4-7).

ActionState	::=	aState: role Role ⁺ tool Tool* SimpleState
ActivityGraph	::=	ag: StateMachine
ActivityProblem-Mapping	::=	apm: process Process subproblem Problem subactivity ActionState ModelElement
Aspect	::=	asp: pattern ProcessPattern* problem Problem catalog ProcessPattern-Catalog* aspect ObjectFlowState Package
Association	::=	ass: Relationship
ClassifierInState	::=	cistate: type Classifier inState State* Classifier
Classifier	::=	classifier: Namespace
CompositeState	::=	composite: [isConcurrent] [isSequential] subvertex Z* State
Context	::=	c: states ObjectFlowState* producer ProcessPattern* consumer ProcessPattern* ModelElement
Element	::=	el:
FinalState	::=	final: State
ModelElement	::=	me: name <i>name</i> Element
Namespace	::=	ns: ModelElement
ObjectFlowState	::=	oState: [isSynch] type Classifier parameter Parameter* [mapping OFSComposition] SimpleState
OFSComposition	::=	ofsc: composite ObjectFlowState objects ObjectFlowState ⁺ Association

Tabelle 5-1: Syntaxregeln

Package	::=	pack: importedElement* ModelElement Namespace
Parameter	::=	par: ModelElement
Problem	::=	p: description <i>string</i> initial Context resulting Context activity Activity-ProblemMapping ModelElement
ProblemGraph	::=	pgraph: problem Problem variants ProcessVariance* refinements Refinement* ModelElement
Process	::=	s: description <i>string</i> role Role proposes ProcessPattern mapping ActivityProblemMapping* ActivityGraph
ProcessPattern	::=	pp: version <i>int</i> phase <i>string</i> synonyms <i>string</i> discussion <i>string</i> example <i>string</i> problem Problem initial Context resulting Context process Process ModelElement
ProcessPattern-Catalog	::=	ppk: patterns ProcessPattern* relations ProcessPatternRelationship* aspect Aspect* Package
ProcessPattern-Graph	::=	ppgraph: pattern ProcessPattern refinements Refinement* variants ProcessVariance* sequences Sequence* usages Use* ModelElement
PPR	::=	ProcessPatternRelationship Sequence Use Refinement RefineProblem ProcessVariance
ProcessPattern-Relationship	::=	ppr: catalog ProcessPatternCatalog ModelElement
ProcessVariance	::=	pvar: variant1 ProcessPattern variant2 ProcessPattern ProcessPattern-Relationship
Refinement	::=	ref: subPattern ProcessPattern superPattern ProcessPattern ProcessPatternRelationship
RefineProblem	::=	refp: subProblem Problem superProblem Problem ProcessPatternRelationship
Relationship	::=	rel: ModelElement
Role	::=	rle: profile <i>string</i> process Process* activity ActionState* ModelElement
Sequence	::=	seq: successor ProcessPattern predecessor ProcessPattern* ProcessPatternRelationship
SimpleState	::=	simple: [initial] State
State	::=	state: StateVertex
StateMachine	::=	sm: top Z transitions Transition* ModelElement
StateVertex	::=	sv: ModelElement
ST	::=	SimpleState ActionState ObjectFlowState
Tool	::=	tl: activity ActionState* ModelElement
Transition	::=	transition: source StateVertex* target StateVertex* ModelElement
Use	::=	use: [recursive] component ProcessPattern composite ProcessPattern ProcessPatternRelationship
Z	::=	CompositeState ST FinalState

Tabelle 5-1: Syntaxregeln (Fortgesetzt)

In diesem Abschnitt haben wir einige Metamodellklassen nicht berücksichtigt, da sie für die Formulierung der formalen Semantik von PROPEL keine Rolle spielen. Zu diesen Metaklassen gehören: Guard, Procedure, Event, StubState, SubmachineState, SubactivityState, CallState und Partition.

Nicht berücksichtigte Klassen

Beispiel Abbildung 5-2 zeigt an einem Beispiel die Anwendung der Syntaxregeln. Auf der unteren Seite der Abbildung ist ein Beispiel-Aktivitätsdiagramm zu sehen, auf der oberen Seite die zugehörige Syntax. *A* ist der Name des zusammengesetzten (obersten) Zustands, *ActivityGraph1* der Name des Aktivitätsdiagramms.

```

sm:
  top
    composite:
      isSequential subvertex
        simple: initial state: sv: me: name initial el:
        aState: simple: initial state: sv: me: name a el:
        oState: simple: initial state: sv: me: name o el:
        bState: simple: initial state: sv: me: name b el:
        simple: initial state: sv: me: name final el:
      state: sv: me: name A el:
    transitions
      transition:
        source sv: me: name initial el: target sv: me: name a el:
      me: name t1 el:
      transition:
        source sv: me: name a el: target sv: me: name o el:
      me: name t2 el:
      transition:
        source sv: me: name o el: target sv: me: name b el:
      me: name t3 el:
      transition:
        source sv: me: name b el: target sv: me: name final el:
      me: name t4 el:
  me: name ActivityGraph1 el:

```

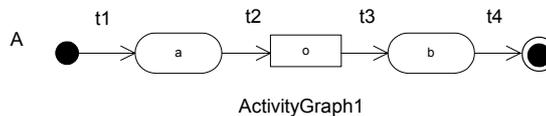


Abbildung 5-2: Beispiel: Abstrakte Syntax eines Aktivitätsdiagramms

5.2 Semantische Domänen

Vorgehen Für die Definition der semantischen Domänen verwenden wir mathematische Grundbegriffe über Mengen und Relationen (s. z.B. [EMC01] für eine Einführung in die Thematik). Einige grundlegende Konzepte definieren wir in Abschnitt 5.2.1 explizit. In Abschnitt 5.2.2 definieren wir die semantische Domäne der Petri-Netze und B/E-Systeme (eine ausführliche Einführung in die Welt der Petri-Netze findet der Leser z.B. in [Bau96]). In Abschnitt 5.2.3 definieren wir schließlich Operatoren zur Manipulation von Petri-Netzen.

5.2.1 Grundlegende Konzepte

Die Projektion benötigen wir bei den semantischen Gleichungen in Abschnitt 5.3 zur Auswahl von Zuständen aus Tupeln von Zuständen.

Definition 5-1: Projektion

Sei $t \in A_1 \times \dots \times A_n$ ein n -stelliges Tupel. Dann liefert die Projektion $\pi_i: A_1 \times \dots \times A_i \times \dots \times A_n \rightarrow A_i$ für $1 \leq i \leq n$ das i -te Element des Tupels t .

□

Für die semantischen Funktionen (Überblick in Anhang B.2) verwenden wir die Wertebereiche Name, Type, Graph und eB/E-Systeme (Abschnitt 5.2.2). Nachfolgend definieren wir diese Wertebereiche.

Definition 5-2: String und Name

Sei A das deutsche Alphabet. Dann ist String definiert als die Menge der Wörter w über A :

$$\text{String} = \{ w \mid w \in A^* \}$$

$w = \lambda$ bezeichnet das leere Wort. „Name“ ist ein im Rahmen der UML definierter Datentyp, der einen String repräsentiert.

□

Definition 5-3: Type

$\text{Type} = \{\text{simple}, \text{final}, \text{concurrent}, \text{sequential}\}$ bestimmt den Typ der Metaklasse State.

□

Definition 5-4: Graph

Sei X eine endliche nichtleere Menge und R eine Menge geordneter Paare (x, y) mit $x, y \in X$, also $R \subset X \times X$. Dann heißt $G = (X, R)$ ein endlicher gerichteter Graph. Die Elemente von X heißen Knoten, die Elemente von R heißen Kanten. Ist (x, y) eine Kante, so heißt x Vorgänger von y , und y heißt Nachfolger von x . Ein Knoten, der keinen Vorgänger hat, heißt Quelle; ein Knoten, der keinen Nachfolger hat, heißt Senke oder Blatt.

□

5.2.2 Petri-Netze und eB/E-Systeme

Zunächst definieren wir den Begriff Petri-Netze.

Definition 5-5: Petri-Netz, Vorbereich, Nachbereich, Teilnetz, Rand

Ein Petri-Netz ist ein Tripel $N=(S,T,F)$ mit $S \cap T = \emptyset$ und $F \subseteq (S \times T) \cup (T \times S)$.

Die Elemente von S heißen Stellen, die Elemente von T heißen Transitionen. Beide zusammen werden als Knoten bezeichnet. Die Elemente der Flussrelation F heißen Kanten.

Der Vorbereich eines Netzknotens ist die Menge $\bullet x := \{y \mid (y,x) \in F\}$. Der Nachbereich eines Netzknotens ist die Menge $x\bullet := \{y \mid (x,y) \in F\}$.

Ein Netz $N'=(S',T',F')$ ist Teilnetz von N , wenn $S' \subseteq S$, $T' \subseteq T$, $F' = F \cap ((S' \times T') \cup (T' \times S'))$.

Der relative Rand $\text{Rand}(N',N)$ eines Teilnetzes N' bzgl. des Gesamtnetzes N sind diejenigen Knoten, die über Kanten mit dem Restnetz verbunden sind:

$$\text{Rand}(N',N) = \{x \in S' \cup T' \mid (x\bullet \cup \bullet x) \setminus (S' \cup T') \neq \emptyset\}.$$

Stellenberandet heißt das Teilnetz N' , wenn $\text{Rand}(N',N) \subseteq S'$. □

Aufbauend auf der Petri-Netz-Definition können wir nun eine Definition für B/E-Systeme angeben.

Definition 5-6: B/E-System

Ein 6-Tupel $Y=(S,T,F,K,W,M_0)$ heißt Bedingungs-/Ereignis-System (B/E-System), falls gilt:

- (S,T,F) ist ein Petri-Netz aus Stellen S und Transitionen T
- $K:S \rightarrow 1$ erklärt die Kapazität 1 für jede Stelle.
- $W:F \rightarrow 1$ bestimmt zu jedem Pfeil des Netzes das Gewicht 1.
- $M_0:S \rightarrow \{0, 1\}$ ist eine Anfangsmarkierung, die die Kapazitäten respektiert, d.h. für jede Stelle $s \in S$ gilt: $M_0(s) \leq K(s)$.

Statt $Y=(S,T,F,K,W,M_0)$ schreiben wir auch $Y=(N,M_0)$. □

Auf B/E-Systemen definieren wir folgende Eigenschaften:

Definition 5-7: Markierung, Aktiviertheit, Folgemarkierungsrelation, Etikettierung

Y sei ein B/E-System. Eine Abbildung $M: S \rightarrow \mathbb{N}_0$ mit $\forall s \in S: M(s) \leq K(s)$ heißt Markierung von Y . $\mathcal{M}(Y)$ bezeichnet die Menge aller Markierungen auf Y . Eine Transition $t \in T$ heißt aktiviert unter M (geschrieben $M[t >$), wenn

$$\begin{aligned} \forall s \in \bullet t: M(s) &\geq W(s, t) \\ \forall s \in t \bullet: M(s) &\leq K(s) - W(t, s) \end{aligned}$$

Wir sagen t schaltet von M nach M' (geschrieben $M[t > M'$), wenn t unter M aktiviert ist und M' aus M durch Entnahme von Marken aus den Eingangsstellen und Ablage von Marken auf die Ausgangsstellen gemäß den Kantengewichten entsteht:

$$M'(s) = \begin{cases} M(s) - W(s, t) & \text{falls } s \in \bullet t \setminus t \bullet \\ M(s) + W(t, s) & \text{falls } s \in t \bullet \setminus \bullet t \\ M(s) - W(s, t) + W(t, s) & \text{falls } s \in t \bullet \cap \bullet t \\ M(s) & \text{sonst} \end{cases}$$

Die Relation $\Phi(Y) = \{(M_1, M_2) \in \mathcal{M}(Y)^2 \mid \exists t \in T: M_1[t > M_2\}$ ist die Folgemarkierungsrelation auf Y .

(Y, h) heißt etikettiertes System (eB/E-System), wenn eine Abbildung h von $S \cup T$ auf einen String existiert. Darüber hinaus existiert eine Abbildung H von $(S \cup T)^*$ auf eine Menge von Strings, d.h. $H(X) = \{h(x) \mid x \in X, X \text{ Teilmenge von } (S \cup T)^*\}$.

□

Bemerkung. Im Vergleich zur gängigen Petri-Netz-Literatur erweitern wir den Etikettierung von Transitionen um die Etikettierung von Stellen, da wir etikettierte Stellen für die Definition der Petri-Netz-Operatoren (Abschnitt 5.2.3) benötigen.

Erweiterung der Etikettierung

Definition 5-8: Deadlockfreiheit

Ein B/E-System $Y = (S, T, F, K, W, M_0)$ heißt deadlockfrei, wenn unter jeder erreichbaren Markierung mindestens eine Transition aktiviert ist, d.h.

$$\forall M_1 \in [M_0 >: \exists t \in T: M_1[t >$$

□

Definition 5-9: Konflikt

Zwei Transitionen t_1 und t_2 eines B/E-Systems mit $K=\infty$ sind im Konflikt, wenn $M[t_1>$ und $M[t_2>$, aber nicht $M[\{t_1,t_2>$.

□

5.2.3 Petri-Netz-Operatoren

Die in diesem Abschnitt definierten Petri-Netz-Operatoren dienen dazu, Petri-Netze bzw. eB/E-Systeme auf bestimmte Weise zu manipulieren. Bei Petri-Netzen sprechen wir in Analogie zu Vorgänger-, Nachfolger-, Super-, Sub-, Komposit-, Komponenten- und Variantenmustern von Vorgänger-, Nachfolger, Super-, Sub-, Komposit-, Komponenten- und Variantennetzen.

Connect-Operator

Der Connect-Operator verknüpft zwei B/E-Systeme auf sequentielle Weise zu einem dritten B/E-System. Das Beispiel in Abbildung 5-3 zeigt, wie zwei Systeme Y und Y' mit Hilfe des Connect-Operators miteinander zu dem System Y'' verknüpft werden. Die Verknüpfung wird erzielt, indem die Stellen, Transitionen und Kanten der beiden Ausgangssysteme durch Umbenennung disjunkt vereinigt werden. Darüber hinaus werden die Stellen, an denen die Systeme Y und Y' verknüpft werden, miteinander verschmolzen, d.h. doppelte Stellen werden entfernt.

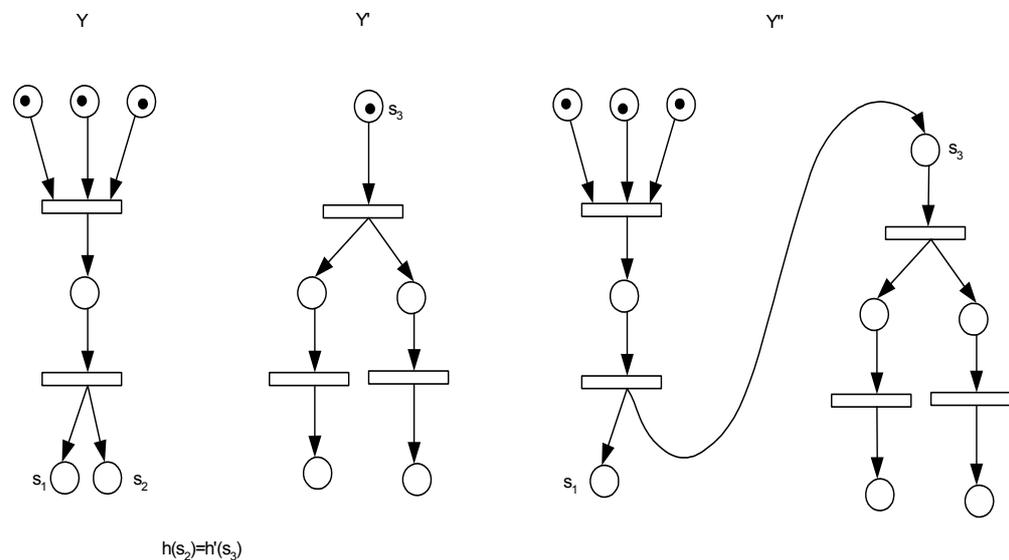


Abbildung 5-3: Verknüpfung von B/E-Systemen

Definition 5-10: Connect-Operator

Seien $Y=(S,T,F,K,W,M,h)$, $Y'=(S',T',F',K',W',M',h')$ und $Y''=(S'',T'',F'',K'',W'',M'',h'')$ etikettierte B/E-Systeme. Die Mengen S , S' und die Mengen T, T' gelten jeweils als disjunkt. Wir definieren $\text{Connect}(Y, I, R, Y', I', R') = Y''$ mit

- $\text{CON} := (H'(I') \cap H(R)) \neq \emptyset$, mit $I, R \subseteq S$ und $I', R' \subseteq S'$
Stellen wir uns ein Petri-Netz Y als Prozesselement eines Prozessmusters vor, so entspricht R dem resultierenden Kontext und I dem initialen Kontext des Prozessmusters. Um zwei Petri-Netze miteinander verknüpfen zu können, darf die Schnittmenge von den Etiketten der Mengen R und I' (bezeichnet als CON) daher nicht die leere Menge sein.
- $S'' = (S \setminus \{s \mid h(s) \in \text{CON}\}) \cup (S' \setminus \{s' \mid h'(s) = \text{„start“}\})$,
d.h. die Stellen von Y und Y' werden miteinander vereinigt, bis auf die Schnittmenge der Mengen R und I' , identifiziert durch die Menge CON . D.h. Duplikate, die durch die Verknüpfung entstehen, werden entfernt. Ferner wird die „start“-Stelle des Nachfolgenernetzes entfernt.
- $T'' = T \cup T'$,
d.h. die Transitionen von Y und Y' werden miteinander vereinigt.
- $F'' = F \setminus F_{\text{elim}} \cup F' \setminus F'_{\text{elim}} \cup F''_{\text{new}}$
mit $F_{\text{elim}} = \{(x, y) \mid h(y) \in \text{CON}\}$, $F'_{\text{elim}} = \{(v, w) \mid h'(v) = \text{„start“}\}$ und $F''_{\text{new}} = \{(m, n) \mid \exists (x, y) \in F_{\text{elim}}: m=x, n \in I', h''(n)=h(y)\}$
d.h. die Kanten von Y und Y' werden miteinander vereinigt. Elimiert werden Kanten des Vorgängernetzes zu den (ebenfalls zu eliminierenden) Stellen aus der Menge CON und Kanten des Nachfolgenernetzes, die von der „start“-Stelle wegführen. Hinzugefügt werden Kanten von den Transitionen aus dem Vorbereich der zu eliminierenden Stellen des Vorgängernetzes hin zu den Stellen des Nachfolgenernetzes, über die Vorgänger- und Nachfolgenernetz verknüpft werden.

Dann ist die Anfangsmarkierung von Y'' definiert als $M''_0(x) = \begin{cases} 1 & \forall x \in I \\ 0 & \text{sonst} \end{cases}$.

Ferner existieren Markierungen, die die Inputstellen des Nachfolgenernetzes mit einer Marke belegen, d.h. $\forall x \in I': \exists M''_i \in [M''_0 > \text{ mit } M''_i(x)=1$

Die Terminalmarkierung von Y'' ist definiert als $M''_n(x) = \begin{cases} 1 & \forall x \in R' \\ 0 & \text{sonst} \end{cases}$

Das System Y'' gilt als deadlockfrei, wenn die Systeme Y und Y' deadlockfrei sind.

Bemerkung. $N=(S,T,F)$ und $N'=(S',T',F')$ sind stellenberandete Teilnetze von $N''=(S'',T'',F'')$.

Die Verknüpfung ein oder mehrerer B/E-Systeme (entsprechend einer Menge von Vorgängermustern) mit einem B/E-System (entsprechend einem Nachfolgermuster) wird über den Operator Connect^* folgendermaßen definiert:

$\text{Connect}^*(\emptyset, \emptyset, \emptyset, Y, I, R) = Y$ und

$$\text{Connect}^*(\{Y_1, \dots, Y_n\}, \bigcup_{i=1}^n I_i, \bigcup_{i=1}^n R_i, Y_{n+1}, I_{n+1}, R_{n+1}) =$$

$$\text{Connect}^*(\{Y_2, \dots, Y_n\}, \bigcup_{i=2}^n I_i, \bigcup_{i=2}^n R_i, \text{Connect}(Y_1, I_1, R_1, Y_{n+1}, I_{n+1}, R_{n+1}), I_{n+1}, R_{n+1}).$$

□

Include-Operator

Der Include-Operator verfeinert eine Stelle eines B/E-Systems durch ein zweites B/E-System, wodurch wiederum ein drittes B/E-System entsteht. Abbildung 5-4 zeigt ein Beispiel für eine stellenweise Verfeinerung eines Systems Y durch ein System Y' zu einem System Y'' . Die Stelle b und alle einlaufenden und ausgehenden Kanten werden entfernt und durch Y' ersetzt.

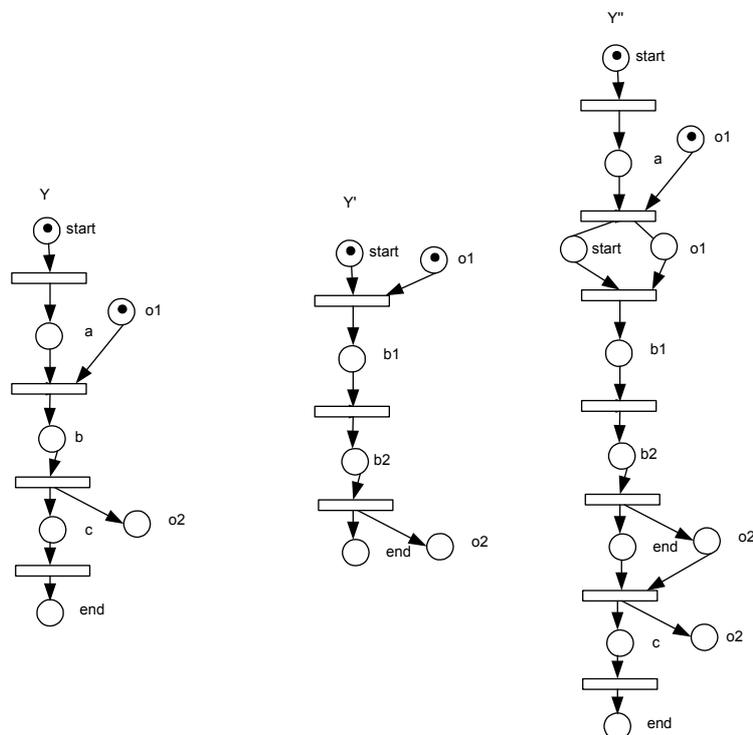


Abbildung 5-4: Verfeinerung/Inklusion von B/E-Systemen

Definition 5-11: Include-Operator

Seien $Y=(S,T,F,K,W,M,h)$, $Y'=(S',T',F',K',W',M',h')$ und $Y''=(S'',T'',F'',K'',W'',M'',h'')$ etikettierte B/E-Systeme. Die Mengen S , S' und die Mengen T, T' gelten jeweils als disjunkt. Sei $s_{\text{replace}} \in S$ die Stelle, die durch Y' ersetzt wird. D.h. durch die Verfeinerung der Stelle s_{replace} entsteht das System Y'' . Diese Verfeinerung (oder auch Inklusion) geben wir durch den Operator

$$\text{Include}(Y, Y', I', R') = Y''$$

an, wobei $I', R' \subseteq S'$. Dann ist Y' ein stellenberandetes Teilnetz von Y'' .

Wir definieren die Verfeinerung von Y über den Operator φ :

- $\forall (x, s_{\text{replace}}) \in F: \varphi(x, s_{\text{replace}}) = \{(x,y) \mid y \in \text{Rand}(Y') \wedge (y \in I' \vee h'(y) = \text{“start“})\}$
D.h. der Operator φ ersetzt die Kanten des Systems Y , die die Stelle s_{replace} als Ziel haben, durch eine Menge von Kanten, die als Ziel diejenigen Stellen des Systems Y' haben, die entweder zum initialen Kontext des abgebildeten Prozessmusters gehören oder zum Startzustand des Systems Y' zählen.
- $\forall (s_{\text{replace}}, x) \in F: \varphi(s_{\text{replace}}, x) = \{(y,x) \mid y \in \text{Rand}(Y') \wedge (y \in R' \vee h'(y) = \text{“end“})\}$
D.h. der Operator φ ersetzt die Kanten des Systems Y , die die Stelle s_{replace} als Quelle haben, durch eine Menge von Kanten, die als Quelle diejenigen Stellen des Systems Y' haben, die entweder zum resultierenden Kontext des abgebildeten Prozessmusters gehören oder zum Endzustand des Systems Y' zählen.
- $\forall (x, y) \in F: \varphi(x, y) = \{(x,y) \mid x, y \in S \cup T \setminus \{s_{\text{replace}}\}\}$
D.h. alle anderen Kanten des Systems Y bleiben erhalten.

Dann ist das System Y'' definiert mit

- $S'' = (S \setminus \{s_{\text{replace}}\} \cup S')$,
d.h. die Stellen von Y und Y' werden miteinander vereinigt, wobei die Stelle s_{replace} aus der Menge S entfernt wurde.
- $T'' = T \cup T'$
d.h. die Transitionen von Y und Y' werden miteinander vereinigt.
- $F'' = \varphi(F) \cup F'$
d.h. die durch den Operator φ erzeugten Kanten und die Kanten von Y' werden miteinander vereinigt.

eine Verfeinerung von Y .

Die initiale Markierung M_0'' von Y'' ist definiert als

$$\forall s \in S'': M_0''(s) = \begin{cases} 1 & M_0(s) = 1 \\ 0 & \text{sonst} \end{cases}$$

Dies bedeutet, dass die initiale Markierung des Systems Y' zunächst verloren geht. Durch die durch den Operator φ erzeugten Kanten wird jedoch sichergestellt, dass alle Stellen aus I' in einer bestimmten Markierung des System Y'' eine Marke tragen, d.h.

$$\exists M \in \mathcal{M}(Y'') : \forall s \in I' : M(s) = 1$$

Das System Y'' gilt als deadlockfrei, wenn die Systeme Y und Y' deadlockfrei sind.

□

Conflict-Operator

Der Conflict-Operator verknüpft zwei B/E-Systeme zu einem dritten B/E-System, wobei die Verknüpfung so gewählt wird, dass nur jeweils eines der beiden Teilsysteme (die die zu verknüpfenden Einzelnetze darstellen) durchlaufen werden kann. Das Beispiel in Abbildung 5-5 zeigt, wie zwei Systeme Y und Y' mit Hilfe des Conflict-Operators miteinander zu dem System Y'' verknüpft werden. Durch Einführung von zwei Transitionen $t_{variant1}$ und $t_{variant2}$, die miteinander in Konflikt stehen, und einer gemeinsamen Inputstelle $s_{conflict}$ kann nur jeweils eine der beiden Teilnetze durchlaufen werden. Initial wird dabei zunächst nur die Stelle $s_{conflict}$ markiert. Ferner wird einer Menge neuer Kanten eingeführt, die als Quelle jeweils eine der konfligierenden Transitionen haben und als Ziel eine Stelle, die dem initialen Kontext des abgebildeten Prozessmusters angehört. Hierdurch wird sichergestellt, dass es Markierungen in dem System Y'' gibt, unter denen die Stellen des initialen Kontexts eines der Teilnetze mit einer Marke versehen sind.

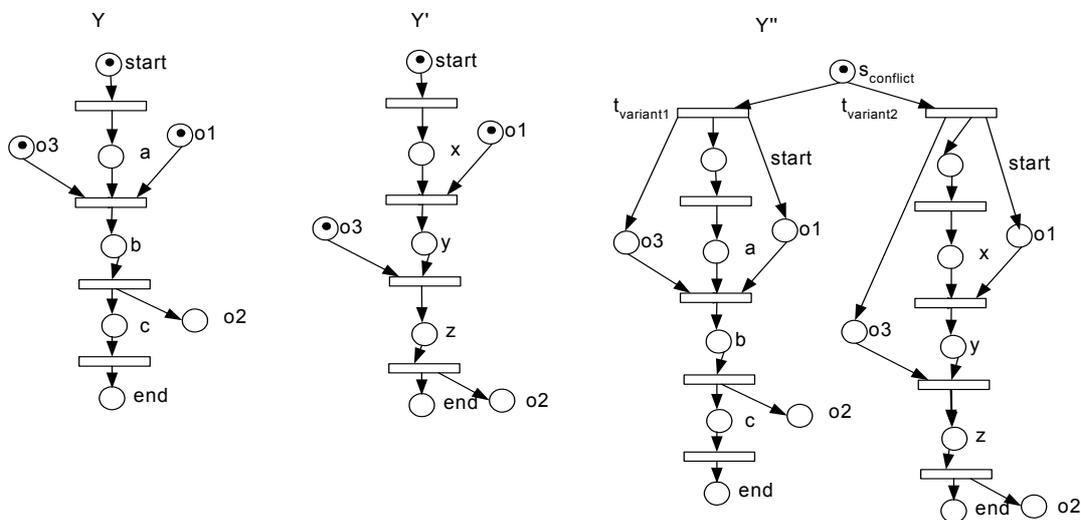


Abbildung 5-5: Zwei Systeme (Y und Y'), die zu einem System Y'' mit konfligierenden Transitionen ineinandergefügt werden

Definition 5-12: Conflict-Operator

Seien $Y=(S,T,F,K,W,M,h)$, $Y'=(S',T',F',K',W',M',h')$ und $Y''=(S'',T'',F'',K'',W'',M'',h'')$ etikettierte B/E-Systeme. Die Mengen S , S' und die Mengen T,T' gelten jeweils als disjunkt. s_{conflict} eine neu eingeführte Stelle mit $s_{\text{conflict}} \notin S \cup S'$ und t_{variant1} , t_{variant2} zwei neu eingeführte Transitionen mit $t_{\text{variant1}} \notin T \cup T'$ und $t_{\text{variant2}} \notin T \cup T'$, die miteinander im Konflikt stehen. $I \subseteq S$ und $I' \subseteq S'$ seien Mengen von Stellen.

Wir definieren $\text{Conflict}(Y, Y', I, I') = Y''$ mit

- $S'' = S \cup S' \cup \{s_{\text{conflict}}\}$,
d.h. die Stellen von Y und Y' werden miteinander vereinigt, wobei die Stelle s_{conflict} neu hinzugefügt wird.
- $T'' = T \cup T' \cup \{t_{\text{variant1}}\} \cup \{t_{\text{variant2}}\}$,
d.h. die Transitionen von Y und Y' werden miteinander vereinigt, wobei die Transitionen t_{variant1} und t_{variant2} neu hinzugefügt werden.
- $F'' = F \cup F' \cup \{(s_{\text{conflict}}, t_{\text{variant1}}), (t_{\text{variant1}}, s), (s_{\text{conflict}}, t_{\text{variant2}}), (t_{\text{variant2}}, s') \mid s \in I, s' \in I'\}$
d.h. die Kanten von Y und Y' werden miteinander vereinigt. Hinzu kommen noch zwei Kanten von der gemeinsamen Inputstelle zu den jeweiligen konfligierenden Transitionen sowie Transitionen ausgehend von den jeweiligen konfligierenden Transitionen zu den Stellen des initialen Kontexts des abgebildeten Prozessmusters.

- $M''_0(x) = \begin{cases} 1 & x = s_{\text{conflict}} \\ 0 & \text{sonst} \end{cases}$

d.h. die gemeinsame Inputstelle der konfligierenden Transitionen erhält initial eine Marke.

- $\exists M_1, M_2 \in \mathcal{M}(Y'')$: $\begin{cases} M_1(s) = 1 & \forall s \in I \\ M_2(s') = 1 & \forall s' \in I' \end{cases}$

d.h. es gibt zwei Markierungen, die sicherstellen, dass die Markierungen der Stellen des initialen Kontexts beider Teilnetze in einer späteren Markierung berücksichtigt werden.

Das System Y'' gilt als deadlockfrei, wenn die Systeme Y und Y' deadlockfrei sind. \square

5.3 Verknüpfung von Syntax und Semantik

Nachdem wir die syntaktischen und semantischen Domänen definiert haben, müssen wir diese aufeinander abbilden. Diese Abbildung definieren wir durch die nachfolgend definierte semantische Gleichungen. Definitions- und Wertebereiche der semantischen Gleichungen werden durch die semantischen Funktionen festgelegt (Anhang B.2). Die OCL-Constraints werden nicht auf die Semantik abgebildet, da bei der semantischen Abbildung einzelne syn-

Semantische Gleichungen und Funktionen

taktische Elemente auf eine formale Semantik abgebildet werden. Die OCL-Constraints repräsentieren jedoch Einschränkungen der Syntax, die unabhängig von der semantischen Abbildung gelten.

Komposition der Gleichungen

Über die kontextfreie Grammatik geben wir einen Syntaxbaum an. Dieser Syntaxbaum enthält Basiselemente (d.h. die Blätter) sowie zusammengesetzte Elemente (d.h. die inneren Knoten), für die eine eindeutige Dekomposition existiert. Die semantischen Gleichungen werden also kompositional definiert. Für jedes Basiselement wird zunächst eine semantische Gleichung angegeben. Anschließend wird für innere Knoten des Syntaxbaums eine semantische Gleichung angegeben, die aus semantischen Gleichungen der direkten Nachfolger im Syntaxbaum komponiert wird.

Eine semantische Gleichung ist folgendermaßen aufgebaut: Auf der linken Seite der Gleichung erfolgt die Angabe der semantischen Funktion. Durch Indizierung können verschiedene Elemente einer Regelfamilie unterschieden werden. Die Gleichung wird auf einen bestimmten Ausdruck der syntaktischen Domäne angewendet. Auf der rechten Seite wird das Ergebnis der Regelanwendung angegeben. Dies repräsentiert die semantische Bedeutung des syntaktischen Ausdrucks.

Gleichung_{index}[[Ausdruck]] = Mathematisches Objekt

Für Mengen von Objekten einer Klasse Y , d.h. Y^* , verwenden wir die gleiche semantische Gleichung g . Auf diese Weise müssen wir nicht separate Gleichungen für Y und Y^* definieren, d.h.

$$g[[Y^*]] ::= g[[Y]] \cup g[[Y^*]] \mid g[[\lambda]] \text{ und}$$

$$g[[\lambda]] ::= \emptyset$$

Abkürzung von Ausdrücken

Wir verwenden das Symbol „...“, um Ausdrücke der kontextfreien Grammatik abzukürzen, also z.B. `[[pp: ... initial Context ...]]` statt `[[pp: version int phase string synonyms string discussion string example string problem initial Context resulting Context process Process ModelElement]]`.

Indexierung von Elementen

Um Elemente, die mehrmals in einem Ausdruck verwendet werden, unterscheiden zu können, versehen wir diese Element mit einem Index (z.B. `pp: ... initial Context1 resulting Context2 ...`). Dabei können beim Ersetzen eines Nichtterminals durch seine rechte Seite der Index an die Elemente des Ausdrucks der rechten Seite – wenn nötig – weitergegeben werden (z.B. `pp: ... initial c: states ObjectFlowState*1 ... resulting Context2 ...`).

Wir definieren die Gleichungen immer in zwei Schritten: Zunächst ersetzen wir den syntaktischen Ausdruck (z.B. `ProcessPattern`) mit der entsprechenden Regel (d.h. `[[pp: ... ModelElement]]`), dann definieren wir für diesen Ausdruck die semantische Bedeutung (z.B. `name`). Häufig ist ein Ausdruck in mehreren Schritten aufzulösen (z.B. `[[pp: ... me: name name ...]]`).

5.3.1 Bezeichner für syntaktische Elemente

In der folgenden Tabelle werden die semantischen Bezeichner für syntaktische Elemente definiert. Die Bezeichner sind alle vom Typ `Name`, welcher einen String repräsentiert.

id_1 [[ProcessPattern]]	= id_1 [[pp: ... ModelElement]] := <i>name</i>
id_2 [[PPR]]	= id_2 [[ProcessPatternRelation Sequence Use Refinement RefineProblem Processvariance]] := id_{10} [[Sequence]], falls PPR->Sequence id_{11} [[Use]], falls PPR->Use id_{12} [[Refinement]], falls PPR->Refinement id_{13} [[Processvariance]], falls PPR->Processvariance
id_3 [[SimpleState]]	= id_3 [[simple: ... State]] := <i>name</i>
id_4 [[FinalState]]	= id_4 [[final: State]] := <i>name</i>
id_5 [[CompositeState]]	= id_5 [[composite: ... State]] := <i>name</i>
id_6 [[Z]]	= id_6 [[CompositeState ST FinalState]] := id_3 [[SimpleState]], falls Z->SimpleState id_4 [[FinalState]], falls Z->FinalState id_5 [[CompositeState]], falls Z->CompositeState
id_7 [[Transition]]	= id_7 [[transition: ... ModelElement]] := <i>name</i>
id_8 [[ObjectFlowState]]	= id_8 [[oState: ... SimpleState]] := <i>name</i>
id_9 [[Context]]	= id_9 [[c: states ObjectFlowState* ...]] := id_8 [[ObjectFlowState*]]
id_{10} [[Sequence]]	= id_{10} [[seq: successor ProcessPattern predecessor ProcessPattern* ...]] := $\{(x, id_1$ [[ProcessPattern]]) $x \in id_1$ [[ProcessPattern*]]\}
id_{11} [[Use]]	= id_{11} [[use: ... component ProcessPattern ¹ composite ProcessPattern ² ...]] := (id_1 [[ProcessPattern ²]], id_1 [[ProcessPattern ¹]])
id_{12} [[Refinement]]	= id_{12} [[ref: subPattern ProcessPattern ² superPattern ProcessPattern ¹ ...]] := (id_1 [[ProcessPattern ²]], id_1 [[ProcessPattern ¹]])
id_{13} [[Processvariance]]	= id_{13} [[pvar: variant1 ProcessPattern ¹ variant2 ProcessPattern ² ...]] := (id_1 [[ProcessPattern ¹]], id_1 [[ProcessPattern ²]])
id_{14} [[StateVertex]]	= id_{12} [[sv: ModelElement]] := <i>name</i>

5.3.2 Prozessmusterkatalog

Ein Prozessmusterkatalog ist ein Graph, wobei die Prozessmuster die Knoten und die Prozessmusterbeziehungen die Kanten des Graphen bilden.

$$\begin{aligned} \text{structure-} & & = & \text{structure}[\mathbf{ppk: patterns} \text{ ProcessPattern* } \mathbf{relations} \text{ Proc-} \\ \text{re}[\text{ProcessPatternCatalog}] & & & \text{essPatternRelationship* ...}] \\ & & := & (\text{id}_1[\text{ProcessPattern*}], \text{id}_2[\text{ProcessPatternRelationship*}]) \end{aligned}$$

5.3.3 Komposition von Objekten und Ereignissen und Kontexten

Die dec-Gleichungen definieren die Dekomposition von Objekten und Ereignissen. Die erste Gleichung dekomponiert ein Aggregat in rekursiver Weise, wobei die dritte Gleichung als Hilfsgleichung fungiert. Bei der zweiten Gleichung handelt es sich nicht um ein Aggregat, die rechte Seite besteht also nur aus einem Bezeichner. Die vierte Gleichung identifiziert Aggregate eines Kontexts. Die fünfte Gleichung gibt die Menge aller Kontextelemente an, wobei Aggregate in ihre Bestandteile aufgelöst werden. Die sechste Gleichung dient dazu, die Kontextdekomposition auf die entsprechenden Petri-Netze zu übertragen. Hierzu werden

- (i) die Stellen entfernt, die Aggregaten entsprechen und die Stellen hinzugefügt, die den (rekursiv) identifizierten Stellen entsprechen
- (ii) die Transitionen beibehalten
- (iii) und die Kanten entfernt, die Kontextaggregate enthalten und durch Kanten ersetzt, die die entsprechenden Aggregatbestandteilen enthalten

$$\begin{aligned} \text{dec}_1[\text{ObjectFlowState}] & = \text{dec}_1[\mathbf{oState: ... mapping} \text{ OFSComposition...}] \\ & := \text{dec}_3[\text{OFSComposition}] \\ \hline \text{dec}_2[\text{ObjectFlowState}] & = \text{dec}_2[\mathbf{oState: ... SimpleState}] \\ & := \text{id}_3[\text{SimpleState}] \\ \hline \text{dec}_3[\text{OFSComposition}] & = \text{dec}_3[\mathbf{ofsc: ... objects} \text{ ObjectFlowState}^+ \text{ ...}] \\ & := \text{dec}_i[\text{ObjectFlowState}^+] \\ \hline \text{dec}_4[\text{Context}] & = \text{dec}_4[\mathbf{c: states} \text{ ObjectFlowState* ...}] \\ & := \{x \mid x \in \text{id}_8[\text{ObjectFlowState*}], \text{ObjectFlowState} \rightarrow \mathbf{oState:} \\ & \quad \dots \mathbf{mapping} \text{ OFSComposition...}\} \\ \hline \text{dec}_5[\text{Context}] & = \text{dec}_5[\mathbf{c: states} \text{ ObjectFlowState* ...}] \\ & := \text{dec}_i[\text{ObjectFlowState*}] \end{aligned}$$

$$\begin{aligned}
\text{dec}_6\llbracket\text{ProcessPattern}\rrbracket &= \text{dec}_6\llbracket\text{pp: ... initial Context}^1 \text{ resulting Context}^2 \text{ ...}\rrbracket \\
&:= (S, T, F, K, W, M_0, h) \text{ mit} \\
&\quad \text{(i) } S = S' \setminus S_{\text{composite}} \cup S_{\text{component}} \\
&\quad \bullet (S', T', F', K', W', M'_0, h') = \text{pr}_2\llbracket\text{ProcessPattern}\rrbracket \\
&\quad \bullet S_{\text{composite}} = \text{dec}_4\llbracket\text{Context}^1\rrbracket \cup \text{dec}_4\llbracket\text{Context}^2\rrbracket \\
&\quad \bullet S_{\text{component}} = \text{dec}_5\llbracket\text{Context}^1\rrbracket \cup \text{dec}_5\llbracket\text{Context}^2\rrbracket \\
&\quad \text{(ii) } T = T' \\
&\quad \text{(iii) } F = \psi(F') \setminus \{(x, y) \mid (x \vee y) \in S_{\text{composite}}\}
\end{aligned}$$

Für die Transformation der Kanten gilt

$\psi(x, y) =$

$$\left\{ \begin{array}{l} (x, y) \quad x, y \notin S_{\text{composite}} \\ (x, z) \quad y \in S_{\text{composite}}, z \in \text{dec}_1[\text{ObjectFlowState}], \text{id}_g[\text{ObjectFlowState}] = y \\ (z, y) \quad x \in S_{\text{composite}}, z \in \text{dec}_1[\text{ObjectFlowState}], \text{id}_g[\text{ObjectFlowState}] = x \end{array} \right.$$

5.3.4 Semantik von Aktivitätsdiagrammen

Aktivitätsdiagramme (Metaklasse *ActivityGraph*) dienen dazu, das dynamische Verhalten im allgemeinen von Systemen zu beschreiben. Aktivitätsdiagramme sind spezielle Zustandsautomaten (Metaklasse *StateMachine*), in denen die Zustände durch Aktivitäten dargestellt werden. Aktivitätsdiagramme ähneln in Struktur und Verhalten den Petri-Netzen. In der kommenden UML2-Version (s. auch Abschnitt A.3.4) wird die Spezifikation von Aktivitätsdiagrammen von der Spezifikation von *StateMachines* klar getrennt und orientiert sich explizit, jedoch informal an Petri-Netzen ([Boc03a], [Boc03b], [Boc03c]). Wir bilden Aktivitätsdiagramme angelehnt an die Vorgehensweise von Gehrke et al. ([GGW98]) auf Petri-Netze bzw. Bedingungs-Ereignis-Systeme ab. Eine ähnliche Strategie verfolgte Störrle in [Stö00] zur Umwandlung von Zustandsautomaten in Petri-Netze.

Abbildung auf Petri-Netz-Semantik

Wir verwenden Bedingungs-Ereignis-Systeme (B/E-Systeme), da dies für unsere Modellierungszwecke ausreicht. B/E-Systeme modellieren das Gelten oder Nichtgelten von Bedingungen, die durch Ereignisse eintreten oder beendet werden. Dies entspricht der Modellierung von Objekten und Ereignissen mit Zustandsautomaten bzw. Aktivitätsdiagrammen. Für ein Prozessmuster müssen wir das Vorhandensein oder den „Verbrauch“ eines Objekts oder Ereignisses modellieren können. Die Modellierung von Mengen von Objekten oder Ereignissen des gleichen Typs (z.B. fünf Projektpläne) ist nicht notwendig.

Bedingungs-Ereignis-Systeme ausreichend

Abbildung 5-6 zeigt exemplarisch, wie Aktivitätsdiagramme in Petri-Netze umgewandelt werden. A zeigt ein einfaches Aktivitätsdiagramm, B eine Aufspaltung des Kontrollflusses und C einen bedingten Kontrollfluss.

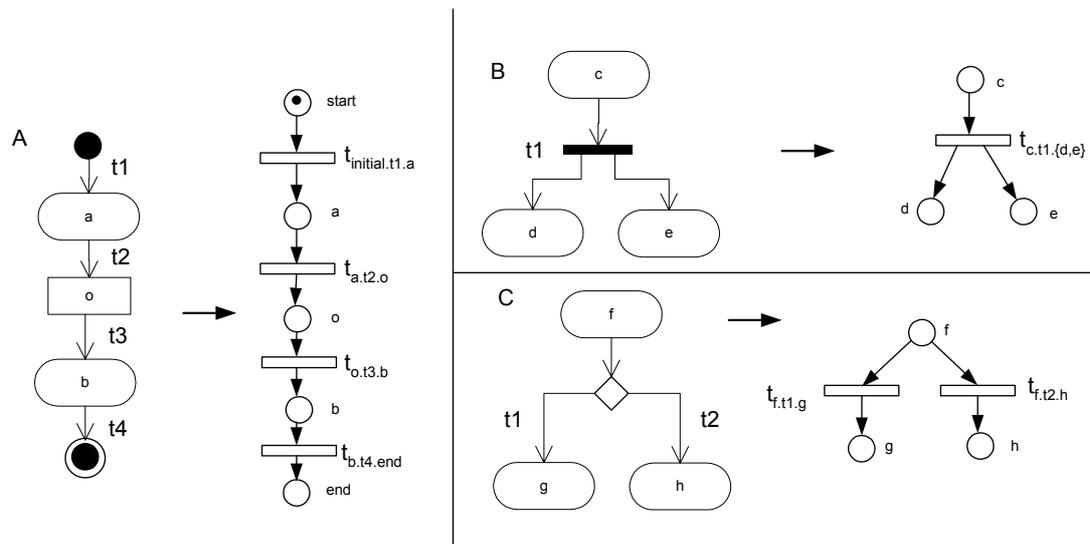


Abbildung 5-6: Abbildung von Aktivitätsdiagrammen auf Elemente eines Petri-Netzes

Abbildung von
Zuständen aus
Stellen

Ein Aktivitätsdiagramm wird auf ein B/E-Netz wie folgt abgebildet: Zustände des Typs **simple** und **final** (SimpleState (ActionState, ObjectFlowState), FinalState) werden auf Stellen abgebildet. Zusammengesetzte Zustände (CompositeState) vom Typ **concurrent** oder **sequential** werden zu Stellen des Typs **simple** oder **final** aufgelöst. Die Namen der Zustände werden als Etiketten von Stellen interpretiert. Initiale und finale Zustände werden durch Stellen repräsentiert, die das Etikett „start“ oder „end“ tragen. Die entsprechenden semantischen Gleichungen sind in Abschnitt 5.3.4.1 aufgeführt.

Abbildung von
Transitionen auf
Transitionen

PROPEL-Transitionen (Transition) werden auf Petri-Netz-Transitionen abgebildet. Anschließend werden die Kanten zwischen den Stellen und Transitionen berechnet. Die Aufspaltung eines Kontrollflusses wird durch eine Transition auf zwei Stellen abgebildet (Abbildung 5-6, oben rechts). Entscheidungen werden durch Transitionen, die miteinander in Konflikt stehen, abgebildet (Abbildung 5-6, unten rechts). Die Inputstelle dieser Transitionen erlaubt den Konflikt, da es sich ja um eine Stelle mit Kapazität 1 handelt. Die entsprechenden semantischen Gleichungen sind in Abschnitt 5.3.4.2 aufgeführt.

Kapazitäten,
Gewichte, Anfangs-
markierung

Kapazitäten und Gewichte sind stets 1, da es sich ja um B/E-Netze handelt. Die Anfangsmarkierung eines Netzes ergibt sich, in dem alle Stellen, die das Etikett „start“ tragen, mit einer Marke belegt werden (Abbildung 5-6, links).

Integration der
einzelnen
Gleichungen

Die semantischen Einzeldefinitionen werden schließlich für die Semantikdefinition von Aktivitätsdiagrammen zusammengeführt. Die entsprechenden semantischen Gleichungen sind in Abschnitt 5.3.4.3 aufgeführt.

5.3.4.1 Abbildung von Zuständen auf Stellen

type Die type-Gleichungen bestimmen den Typ (**simple**, **final**, **concurrent**, **sequential**) eines Zustands.

$\text{type}_1 \llbracket \text{SimpleState} \rrbracket$	$= \text{type}_1 \llbracket \mathbf{simple: [initial] State} \rrbracket$ $:= \{ \text{id}_3 \llbracket \text{SimpleState} \rrbracket \rightarrow \mathbf{simple} \}$
$\text{type}_2 \llbracket \text{FinalState} \rrbracket$	$= \text{type}_2 \llbracket \mathbf{final: State} \rrbracket$ $:= \{ \text{id}_4 \llbracket \text{FinalState} \rrbracket \rightarrow \mathbf{final} \}$
$\text{type}_3 \llbracket \text{CompositeState} \rrbracket$	$= \text{type}_3 \llbracket \mathbf{composite: isConcurrent subvertex Z^* State} \rrbracket$ $:= \{ \text{id}_5 \llbracket \text{CompositeState} \rrbracket \rightarrow \mathbf{concurrent} \} \cup \text{type}_5 \llbracket Z^* \rrbracket$
$\text{type}_4 \llbracket \text{CompositeState} \rrbracket$	$= \text{type}_4 \llbracket \mathbf{composite: isSequential subvertex Z^* State} \rrbracket$ $:= \{ \text{id}_5 \llbracket \text{CompositeState} \rrbracket \rightarrow \mathbf{sequential} \} \cup \text{type}_5 \llbracket Z^* \rrbracket$
$\text{type}_5 \llbracket Z \rrbracket$	$= \text{type}_5 \llbracket \text{CompositeState} \mid \text{ST} \mid \text{FinalState} \rrbracket$ $:= \text{type}_1 \llbracket \text{SimpleState} \rrbracket, \text{ falls } Z \rightarrow \text{SimpleState}$ $\text{type}_2 \llbracket \text{FinalState} \rrbracket, \text{ falls } Z \rightarrow \text{FinalState}$ $\text{type}_3 \llbracket \text{CompositeState} \rrbracket \vee \text{type}_4 \llbracket \text{CompositeState} \rrbracket, \text{ falls } Z \rightarrow \text{CompositeState}$

Die init-Gleichungen bestimmen die initialen Zustände aus einer Menge von Zuständen.

init

$\text{init}_1 \llbracket \text{SimpleState} \rrbracket$	$= \text{init}_1 \llbracket \mathbf{simple: [initial] State} \rrbracket$ $:= \textit{name}, \text{ if initial=true}$ $\quad \emptyset, \text{ if initial=false}$
$\text{init}_2 \llbracket \text{FinalState} \rrbracket$	$= \text{init}_2 \llbracket \mathbf{final: State} \rrbracket$ $:= \emptyset$
$\text{init}_3 \llbracket \text{CompositeState} \rrbracket$	$= \text{init}_3 \llbracket \mathbf{composite: [isConcurrent] [isSequential] subvertex Z^* State} \rrbracket$ $:= \text{init}_4 \llbracket Z^* \rrbracket$
$\text{init}_4 \llbracket Z \rrbracket$	$= \text{init}_4 \llbracket \text{CompositeState} \mid \text{ST} \mid \text{FinalState} \rrbracket$ $:= \text{init}_1 \llbracket \text{SimpleState} \rrbracket, \text{ falls } Z \rightarrow \text{ST}$ $\text{init}_2 \llbracket \text{FinalState} \rrbracket, \text{ falls } Z \rightarrow \text{FinalState}$ $\text{init}_3 \llbracket \text{CompositeState} \rrbracket, \text{ falls } Z \rightarrow \text{CompositeState}$

Die terminal-Gleichungen bestimmen die finalen Zustände aus einer Menge von Zuständen.

terminal

$\text{terminal}_1 \llbracket \text{SimpleState} \rrbracket$	$= \text{terminal}_1 \llbracket \mathbf{simple: [initial] State} \rrbracket$ $:= \emptyset$
$\text{terminal}_2 \llbracket \text{FinalState} \rrbracket$	$= \text{terminal}_2 \llbracket \mathbf{final: State} \rrbracket$ $:= \textit{name}$
$\text{terminal}_3 \llbracket \text{CompositeState} \rrbracket$	$= \text{terminal}_3 \llbracket \mathbf{composite: [isConcurrent] [isSequential] subvertex Z^* State} \rrbracket$ $:= \text{terminal}_4 \llbracket Z^* \rrbracket$

$$\text{terminal}_4[[Z]] \quad := \quad \{s \in \pi_1(\text{type}_5[[Z]]) \mid \pi_2(\text{type}_5[[Z]]) = \mathbf{final} \}$$

sub Die sub-Gleichungen bestimmen alle Teilstände (Metaklasse SubState) eines Zustands. Das Ergebnis sind 2-Tupel der Art (z', z) , wobei z' Teilstand des Zustands z ist.

$$\text{sub}_1[[\text{SimpleState}]] \quad := \quad (\emptyset, \text{id}_3[[\text{SimpleState}]])$$

$$\text{sub}_2[[\text{FinalState}]] \quad := \quad (\emptyset, \text{id}_4[[\text{FinalState}]])$$

$$\text{sub}_3[[\text{CompositeState}]] \quad = \quad \text{sub}_3[[\mathbf{composite: [isConcurrent] [isSequential] subvertex Z^* State}]]$$

$$:= \quad \{(z, \text{id}_5[[\text{CompositeState}]] \mid z \in \text{id}_6[[Z^*]]\} \cup \text{sub}_4[[Z^*]]$$

$$\text{sub}_4[[Z]] \quad = \quad \text{sub}_4[[\text{CompositeState} \mid \text{ST} \mid \text{FinalState}]]$$

$$:= \quad \begin{array}{l} \text{sub}_1[[\text{SimpleState}]], \text{ falls } Z \rightarrow \text{ST} \\ \text{sub}_2[[\text{FinalState}]], \text{ falls } Z \rightarrow \text{FinalState} \\ \text{sub}_3[[\text{CompositeState}]], \text{ falls } Z \rightarrow \text{CompositeState} \end{array}$$

5.3.4.2 Abbildung von PROPEL-Transitionen auf Petri-Netz-Transitionen

Wie bereits weiter oben erläutert, werden PROPEL-Transitionen in Transitionen von Petri-Netzen übersetzt. Eine einfache PROPEL-Transition wird auf eine einfache Petri-Netz-Transition abgebildet. Eine zusammengesetzte („compound“, s. Abschnitt 4.4.1) Transition, welche eine Menge von Transitionen repräsentiert, die von einer Menge von Zuständen herrühren und zu einer Menge von Zuständen streben, wird auf zweierlei Art abgebildet. Bei Nebenläufigkeit (fork oder join) wird die zusammengesetzte PROPEL-Transition in eine einfache Petri-Netz-Transition umgewandelt. Im Falle alternativer Kontrollflüsse (choice und junction) wird jede Transition der zusammengesetzten PROPEL-Transition auf eine einfache Petri-Netz-Transition abgebildet. Die resultierenden Petri-Netz-Transitionen werden nach dem Schema *from.id.to* etikettiert, mit *from* die Menge der Zustände, von der die Transition herrührt, *id* der Name der ursprünglichen PROPEL-Transition und *to* die Menge von Zuständen, zu denen die Transition strebt.

trans

$$\begin{aligned} \text{trans}[[\text{Transition}]] &= \text{trans}[[\mathbf{transition: source StateVertex}^*1 \mathbf{target StateVertex}^*2 \dots]] \\ &:= \{t_{\text{from.id.to}} \mid \text{from} = \text{id}_{12}[[\text{StateVertex}^*1]], \text{id} = \text{id}_7[[\text{Transition}]], \\ &\quad \text{to} = \text{id}_{12}[[\text{StateVertex}^*2]]\} \end{aligned}$$

Beispiel In Abbildung 5-6 zeigten wir bereits die Umwandlung von UML-Transitionen in Petri-Netz-Transitionen. Die Petri-Netz-Transitionen werden dabei folgendermaßen berechnet:

Für das Aktivitätsdiagramm A:

- $\text{trans}[[t1]] = \{t_{\text{initial.t1.a}}\}$
- $\text{trans}[[t2]] = \{t_{\text{a.t2.o}}\}$

- $\text{trans}[[t3]] = \{t_{o,t3,b}\}$
- $\text{trans}[[t4]] = \{t_{b,t4,final}\}$

Für das Aktivitätsdiagramm B:

- $\text{trans}[[t1]] = \{t_{c,t1,\{d,e\}}\}$

Für das Aktivitätsdiagramm C:

- $\text{trans}[[t1]] = \{t_{f,t1,g}\}$
- $\text{trans}[[t2]] = \{t_{f,t2,h}\}$

5.3.4.3 Abbildung von Aktivitätsdiagrammen auf Petri-Netze

Die semantischen Definitionen der vorhergehenden Abschnitte können wir nun verwenden, um die Semantik von Aktivitätsdiagrammen zu definieren. Ein Aktivitätsdiagramm ist definiert als B/E-System, wobei

- (i) die Stellen von den PROPEL-Zuständen mit Typ `simple` und `final` abgeleitet werden,
- (ii) die Transitionen von den PROPEL-Transitionen berechnet werden,
- (iii) die Kanten zwischen diesen Stellen und Transitionen berechnet werden,
- (iv) Kapazitäten und Gewichte stets 1 sind,
- (v) die Anfangsmarkierung auf jede Stelle, die einem initialen Zustand entspricht, eine Marke legt,
- (vi) die Terminalmarkierung auf jede Stelle, die einem finalen Zustand entspricht, eine Marke legt,
- (vii) die initiale Stellen mit „start“ etikettiert sind,
- (viii) die terminalen Stellen mit „end“ etikettiert sind und
- (ix) alle anderen Stellen mit dem Namen des korrespondierenden PROPEL-Zustands etikettiert sind.

$$\begin{aligned}
\text{agraph}[\llbracket \text{ActivityGraph} \rrbracket] &= \text{agraph}[\llbracket \text{ag: sm: top Z transitions Transition* ...} \rrbracket] \\
&:= (S, T, F, K, W, M_0, h) \text{ mit} \\
&\quad \text{(i) } S = \{s \in \pi_1(\text{type}_5[\llbracket Z \rrbracket]) \mid \pi_2(\text{type}_5[\llbracket Z \rrbracket]) \in \{\mathbf{simple}, \mathbf{final}\}\} \\
&\quad \text{(ii) } T = \text{trans}[\llbracket \text{Transition*} \rrbracket] \\
&\quad \text{(iii) } F = \{(s, t_{\text{from.id.to}}) \mid t_{\text{from.id.to}} \in T, s \in S, s \in \text{from}\} \cup \\
&\quad \quad \{(t_{\text{from.id.to}}, s) \mid t_{\text{from.id.to}} \in T, s \in S, s \in \text{to}\} \\
&\quad \text{(iv) } \forall s \in S: K(s)=1, W(s)=1 \\
&\quad \text{(v) } M_0(\text{init}_4[\llbracket Z \rrbracket]) = 1 \wedge M_0(S \setminus \text{init}_4[\llbracket Z \rrbracket]) = 0 \\
&\quad \text{(vi) } \exists M_n \in M(N): \exists \text{kein } t \in T: M_n [t > M_{n+1}: \forall x \in \text{terminal}_4[\llbracket Z \rrbracket]: \\
&\quad \quad M_n(x) = 1 \\
&\quad \text{(vii) } \forall x \in \text{init}_4[\llbracket Z \rrbracket]: h(x) = \text{start} \\
&\quad \text{(viii) } \forall x \in \text{terminal}_4[\llbracket Z \rrbracket]: h(x) = \text{end} \\
&\quad \text{(ix) } \forall x \in \text{id}_6[\llbracket Z \rrbracket]: h(x) = \text{id}_3[\llbracket x \rrbracket]
\end{aligned}$$

5.3.4.4 Vernachlässigung der run-to-completion-Annahme

Im Rahmen der StateMachines-Spezifikation, auf der die ActivityGraphs-Spezifikation aufbaut, gibt es den Begriff der Ereignisverarbeitung, besser bekannt als „run-to-completion-Annahme“: Zu einem bestimmten Zeitpunkt wird ein Ereignis von einer Zustandsmaschine erzeugt oder verarbeitet. Die Reihenfolge der Abarbeitung ist nicht spezifiziert. Der Modellierer kann daher beliebige Prioritäten spezifizieren. Die „run-to-completion-Annahme“ bedeutet dabei, dass ein Ereignis erst dann erzeugt oder verarbeitet werden darf, wenn das vorhergehende Ereignis vollständig abgearbeitet wurde. Als „run-to-completion-step“ wird dann der Schritt zwischen zwei stabilen StateMachine-Zuständen bezeichnet, d.h. zwischen StateMachines, deren Transitionen alle beendet sind, die durch das Ereignis ausgelöst wurden. Wir vernachlässigen die „run-to-completion-Annahme“ aus zwei Gründen: Zum einen reicht die von uns definierte Semantik von Aktivitätsdiagrammen für unsere Zwecke aus. Die von uns benötigten Aktivitätsdiagramme setzen wir zur Modellierung von Geschäftsprozessen ein, die durch Individuen und nicht von Systemen ausgeführt werden. Eine Einschränkung durch die „run-to-completion-Annahme“ ist daher nicht notwendig, da die Entscheidung über die Abarbeitung von Ereignissen von den Individuen vorgenommen wird. Zum anderen würde die Definition einer solch „strikten“ Semantik den Rahmen unserer Arbeit sprengen. Arbeiten, die sich mit einer solchen Semantik befassen sind z.B. für StateMachines und für Aktivitätsdiagramme verfügbar (Abschnitt 2.3).

5.3.5 Semantik von PROPEL-Prozessen

Ein Prozess besitzt die gleiche Semantik wie ein Aktivitätsdiagramm mit zwei Erweiterungen (Abbildung 5-7): Zusätzlich zu den initialen Stellen erhalten auch diejenigen Stellen eine Anfangsmarkierung, die dem initialen Kontext des Prozessmusters entsprechen. Alle anderen Stellen besitzen keine Marke. Zusätzlich zu den terminalen Stellen erhalten auch diejenigen Stellen eine Terminalmarkierung, die dem resultierenden Kontext des Prozessmusters entsprechen. Alle anderen Stellen besitzen keine Marke.

$$\begin{aligned}
pr_1[\text{Process}] &= pr_1[\text{s: ... proposes ProcessPattern ... ActivityGraph}] \\
&:= \text{agraph}[\text{ActivityGraph}] \text{ mit} \\
&\bullet M_0(\text{init}_4[\text{Z}] \cup \text{ic}[\text{ProcessPattern}]) = 1 \wedge \\
&\quad M_0(\text{Sinit}_4[\text{Z}] \cup \text{ic}[\text{ProcessPattern}]) = 0 \\
&\bullet \exists M_n \in M(\mathbb{N}): \exists \text{kein } t \in \mathbb{T}: M_n[t > M_{n+1}: \\
&\quad \forall x \in (\text{rc}[\text{ProcessPattern}] \cup \text{terminal}_4[\text{Z}]): M_n(x) = 1
\end{aligned}$$

$$\begin{aligned}
pr_2[\text{ProcessPattern}] &= pr_2[\text{pp: ... process Process...}] \\
&:= pr_1[\text{Process}]
\end{aligned}$$

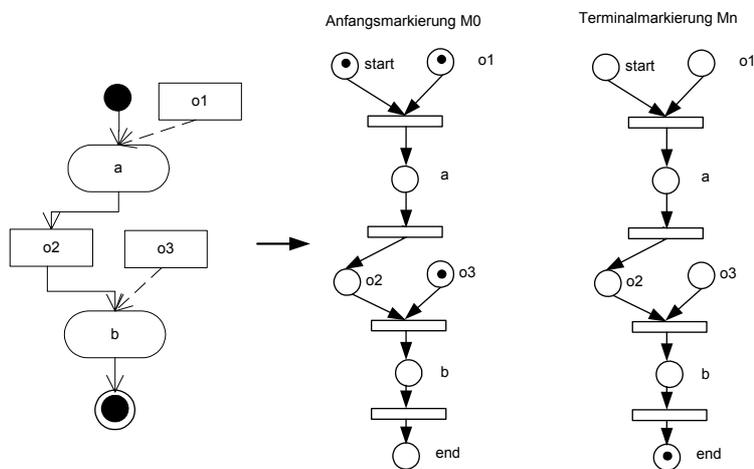


Abbildung 5-7: Abbildung des Prozesses eines Prozessmusters auf ein etikettiertes B/E-System

5.3.6 Semantik von Kontexten

Der initiale und der resultierende Kontext eines Prozessmusters ist jeweils eine Menge von Namen, d.h. die Namen der ObjectFlowStates, die einem Kontext angehören. Die jeweils ersten Gleichungen für den initialen und resultierenden Kontext bezeichnen eine Menge von Kontextelementbezeichnern, während die jeweils zweiten Gleichungen gegebenenfalls aggregierte Kontextelemente durch ihre Aggregatbestandteile ersetzen.

$$\begin{aligned}
ic_1[\text{ProcessPattern}] &= ic_1[\text{pp: ... initial Context ... process Process...}] \\
&:= id_9[\text{Context}], \text{ wobei } id_9[\text{Context}] \subseteq S, S \in pr_1[\text{Process}]
\end{aligned}$$

$$\begin{aligned}
ic_2[\text{ProcessPattern}] &= ic_2[\text{pp: ... initial Context ...}] \\
&:= dec_5[\text{Context}], \text{ wobei } dec_5[\text{Context}] \subseteq S, S \in dec_6[\text{ProcessPattern}]
\end{aligned}$$

$$\begin{aligned}
rc_1[\text{ProcessPattern}] &= rc_1[\text{pp: ... resulting Context process Process ...}] \\
&:= id_9[\text{Context}], \text{ wobei } id_9[\text{Context}] \subseteq S, S \in pr_1[\text{Process}]
\end{aligned}$$

$$\begin{aligned}
 rc_2 \llbracket \text{ProcessPattern} \rrbracket &= rc_2 \llbracket \text{pp: ... resulting Context ...} \rrbracket \\
 &:= dec_5 \llbracket \text{Context} \rrbracket, \text{ wobei } dec_5 \llbracket \text{Context} \rrbracket \subseteq S, S \in dec_6 \llbracket \text{ProcessPattern} \rrbracket
 \end{aligned}$$

5.3.7 Semantik von Beziehungen

Sequence-Beziehung

Die Sequence-Beziehung verknüpft ein oder mehrere vorausgehende Prozessmuster (**predecessor**) mit einem nachfolgenden Prozessmuster (**successor**). Diese Verknüpfung wird mittels des Connect-Operators aus Abschnitt 5.2.3 spezifiziert. Er verschmilzt die Stellen, die in der Schnittmenge von dem resultierenden Kontext des vorausgehenden Prozessmusters und dem initialen Kontext des nachfolgenden Prozessmusters liegen. Auf diese Weise erhält man ein B/E-System, das mehrere Teilsysteme (die einzelnen Prozessmuster) enthält.

$$\begin{aligned}
 rel_1 \llbracket \text{Sequence} \rrbracket &= rel_1 \llbracket \text{seq: successor ProcessPattern predecessor ProcessPattern}^* \\
 &\quad \dots \rrbracket \\
 &:= \text{Connect}(pr_2 \llbracket \text{ProcessPattern}^* \rrbracket, ic_1 \llbracket \text{ProcessPattern}^* \rrbracket, \\
 &\quad rc_1 \llbracket \text{ProcessPattern}^* \rrbracket, pr_2 \llbracket \text{ProcessPattern} \rrbracket, ic_1 \llbracket \text{ProcessPattern} \rrbracket, \\
 &\quad rc_1 \llbracket \text{ProcessPattern} \rrbracket)
 \end{aligned}$$

Abbildung 5-8 zeigt ein Beispiel für eine Sequence-Beziehung der Aktivitätsdiagramme A und B. Aktivitätsdiagramm A besitzt in seinem resultierenden Kontext das Objekt o3, welches das Aktivitätsdiagramm B in seinem initialen Kontext benötigt. Durch die Abbildung der Aktivitätsdiagramme auf Petri-Netze und Verschmelzung dieser Petri-Netze entsteht das Petri-Netz Y". Hierbei werden die „start“-Stellen des nachfolgenden Petri-Netzes sowie die wegführenden Kanten eliminiert. Die Entfernung dieser Stellen ist unproblematisch, da ja die Kontexte eines Prozessmusters nie leer sein dürfen (Abbildung 4-3 in Kapitel 4) und verknüpfende Stellen vorhanden sind.

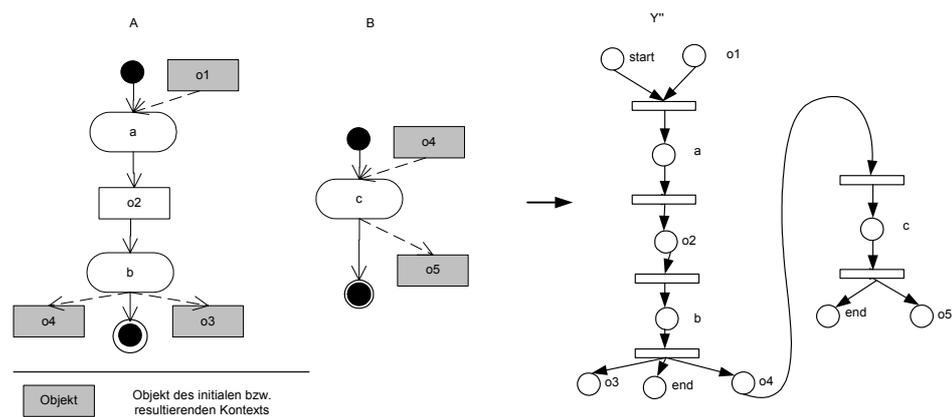


Abbildung 5-8: Beispiel für eine Sequence-Beziehung

Die „end“-Stellen der vorausgehenden Petri-Netze müssen bestehen bleiben, um den Fall abzufangen, dass die „letzte“ Transition (Abbildung 5-9, Transition $t_{b,t4.end}$) im Erreichbarkeitsgraphen auch schalten kann. Ferner kann auf diese Weise die Transition $t_{o4,t5.c}$ schalten, sobald die Stelle $o4$ mit einer Marke versehen ist. Dies bedeutet, dass die Aktivität c (bzw. die Transition $t_{c,t6.\{end, o5\}}$) schon ausgeführt werden kann, bevor die Aktivität b ausgeführt wird. Für die Prozessmuster bedeutet dies, dass nicht erst alle Aktivitäten der Vorgängermuster abgearbeitet sein müssen, bevor das Nachfolgemuster starten kann.

„end“-Stellen der vorausgehenden Petri-Netze bleiben bestehen

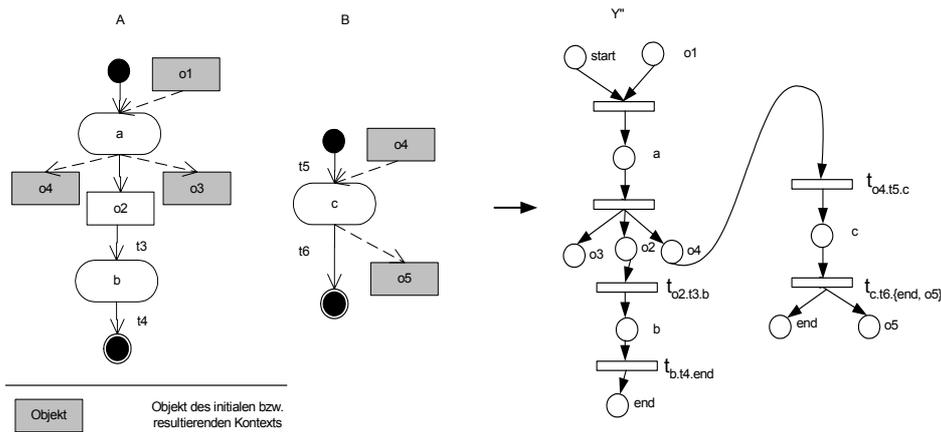


Abbildung 5-9: Aktivität b mit leerem resultierendem Kontext

Abbildung 5-10 zeigt ein weiteres Beispiel für den Fall, dass ein Objekt des resultierenden Kontexts des Vorgängermusters mehrfach im initialen Kontext des Nachfolgemusters vorkommt. In diesem Fall werden Kanten von der Transition (hier $t_{a,t2.\{o2,o3,o4,o4\}}$) aus dem Vorbereich der eliminierten Stelle zu diesem mehrfach verwendeten Objekt hinzugefügt.

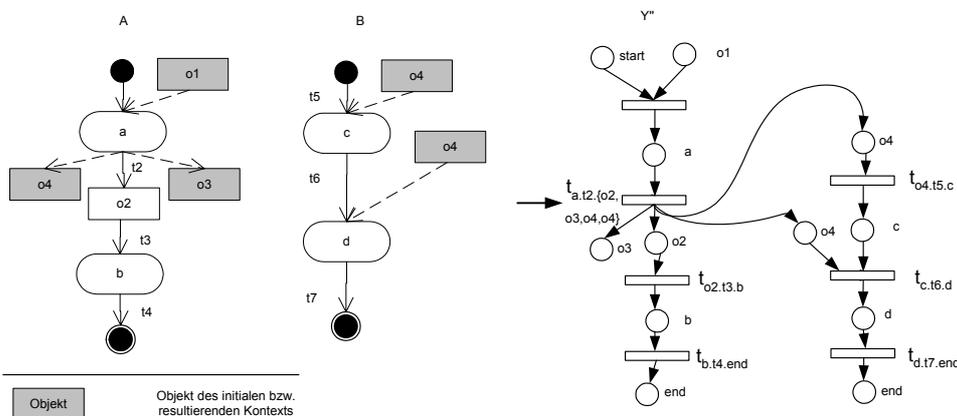


Abbildung 5-10: Mehrfache Verwendung des Objekts o4 im initialen Kontext des Nachfolgemusters

Das Verhalten der einzelnen B/E-Systeme wird nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn alle seine Teilsysteme deadlockfrei sind. Diese Aussage kann dadurch erläutert werden, dass beim Verknüpfen der Netze das Nachfolgernetz gar nicht und das Vorgängernetz nur geringfügig manipuliert werden. Beim Vorgängernetz werden lediglich die redundanten Stellen eliminiert, die bereits im initialen Kontext des Nachfolgernetzes enthalten sind. Dies bedeutet, dass die Transitionen, die vor der Eliminierung mit den eliminierten Stellen verbunden sind, jetzt mit den entsprechenden Stellen im Nachfolgernetz verbunden sind. Diese Transitionen können also weiterhin (zu einem bestimmten Zeitpunkt) schalten. Sie sorgen dafür, dass die Stellen, über die Vorgänger- und Nachfolgernetz verknüpft sind, mit einer Marke belegt sind. Diese Aussage gilt auch für weitere mögliche Vorgängernetze, die mit dem Nachfolgernetz verknüpft sind. Dadurch, dass irgendwann alle Stellen des initialen Kontexts des verknüpften Nachfolgernetzes belegt sind, weist dieses die gleichen Eigenschaften und Verhalten auf, wie das einzelne Nachfolgernetz.

Use-Beziehung Die Use-Beziehung verknüpft ein übergeordnetes und ein untergeordnetes Prozessmuster. Hierzu wird ein Petri-Netz definiert, welches durch Verfeinerung (im Sinne von Petri-Netzen) einer Stelle des übergeordneten Prozessmusters durch das untergeordnete Prozessmuster entsteht. Wie auch bei der Sequence-Beziehung wird das ursprüngliche Verhalten der beiden Ausgangsnetze nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn das übergeordnete und das untergeordnete Netz deadlockfrei sind.

$$\begin{aligned} \text{rel}_2[\text{Use}] &= \text{rel}_2[\text{use: ... component ProcessPattern}^1 \text{ composite ProcessPattern}^2 \text{ ...}] \\ &:= \text{Include}(\text{pr}_2[\text{ProcessPattern}^2], \text{pr}_2[\text{ProcessPattern}^1], \\ &\quad \text{ic}_1[\text{ProcessPattern}^1], \text{rc}_1[\text{ProcessPattern}^1]) \end{aligned}$$

Abbildung 5-11 zeigt ein Beispiel für eine Use-Beziehung der Aktivitätsdiagramme der Prozessmuster A und B. A repräsentiert das Kompositmuster, B das Komponentenmuster. Die Aktivität b von A soll durch das Prozessmuster B ersetzt werden. Durch die Abbildung der Aktivitätsdiagramme A und B auf Petri-Netze Y und Y' und Verfeinerung des Petri-Netzes Y durch das Netz Y' entsteht das Petri-Netz Y''. Hierbei werden die Stelle b des Netzes Y und alle davon ausgehenden und einlaufenden Kanten entfernt und durch Y ersetzt. Darüber hinaus werden weitere Kanten hinzugefügt, die von der Transition (hier $t_{02,t3,b}$), die ehemals zu b hinführte, zu der „start“-Stelle und den Stellen des initialen Kontexts des Komponentenmusters führen und die von der „end“-Stelle und dem resultierenden Kontext zu der Transition (hier: $t_{b,t3.\{c,o2\}}$) führen, die im Nachbereich von b lag.

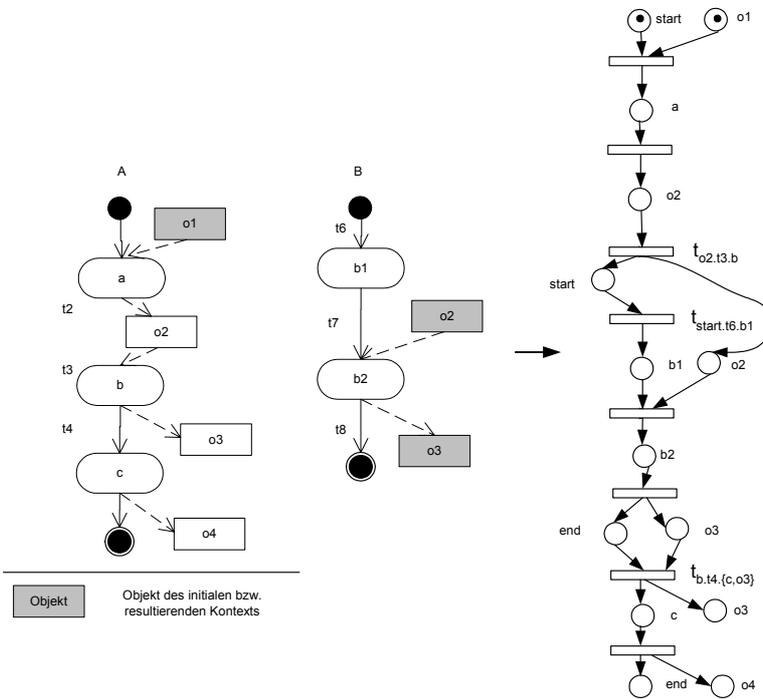


Abbildung 5-11: Beispiel für eine Use-Beziehung

Diese Lösung erscheint zunächst wenig elegant, da die „start“- und „end“-Stelle des Komponentenmusters erhalten bleiben und auch die Elemente des Kontexts des Komponentenmusters redundant zu den entsprechenden Elementen des Kontext der zu verfeinernden Aktivität (hier: b) sind. Diese Vorgehensweise lässt sich jedoch nicht vermeiden, wenn man das Verhalten und die Eigenschaften der beiden Ursprungnetze erhalten will. Würde man beispielsweise die „start“-Stelle des Komponentennetzes und deren ein- und ausgehenden Kanten entfernen, müsste man zusätzlich die nachfolgende Transition $t_{start.t6.b1}$ und alle davon ausgehenden Kanten entfernen. Von der Transition $t_{o2.t3.b}$ müssten Kanten zu allen Stellen des initialen Kontexts des Komponentennetzes und zusätzlich zur Stelle b1 hinzugefügt werden. Würde unser Beispiel jedoch so aussehen, dass die Aktivität b1 ebenfalls ein Inputobjekt verlangt, dürfte man die Transition $t_{start.t6.b1}$ nicht löschen. Die Aussagen gelten analog für den resultierenden Kontext des Komponentennetzes. Eine solche Fallunterscheidung ist recht komplex und fehleranfälliger als die von uns gewählte einfachere (aber weniger elegantere) Lösung.

Das Verhalten der einzelnen B/E-Systeme wird wie bei der Sequence-Beziehung nicht verändert. Dies bedeutet wiederum, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn alle seine Teilsysteme deadlockfrei sind. Diese Aussage kann dadurch erläutert werden, dass beim Verknüpfen der Netze das Kompositnetz nur geringfügig und das Komponentennetz gar nicht manipuliert werden. Beim Vorgängernetz werden lediglich die zu ersetzende Stelle und die davon hin- und wegführenden Kanten entfernt. Darüber hinaus werden neue Kanten hinzugefügt, die zu den Stellen des initialen Kontexts des Komponentennetzes führen und diese beim Schalten der Transition aus dem Vorbereich der zu ersetzenden Stelle mit einer Marke versehen.

„start“- und „end“-Stelle des Komponentenmusters bleiben erhalten

Verhalten der einzelnen B/E-Systeme wird nicht verändert

Refinement-
Beziehung

Die Refinement-Beziehung verknüpft ein abstrakteres und detaillierteres Prozessmuster. Hierzu wird ein Petri-Netz definiert, das diese beiden Prozessmuster als Teilnetze enthält. Durch Einführung zweier Transitionen, die in Konflikt stehen, kann nur eines der beiden Teilnetze durchlaufen werden. Wie auch bei vorhergehenden Beziehungen wird das ursprüngliche Verhalten der beiden Ausgangsnetze nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn beide Teilnetze deadlockfrei sind.

$$\begin{aligned}
 \text{rel}_3[[\text{Refinement}]] &= \text{rel}_3[[\text{ref: subPattern ProcessPattern}^1 \text{ superPattern ProcessPattern}^2 \dots]] \\
 &:= \text{Conflict}(\text{pr}_2[[\text{ProcessPattern}^1]], \text{pr}_2[[\text{ProcessPattern}^2]], \text{init}_4[[Z]] \cup \text{ic}_1[[\text{ProcessPattern}^1]], \text{init}_4[[Z]] \cup \text{ic}_1[[\text{ProcessPattern}^2]])
 \end{aligned}$$

Abbildung 5-12 zeigt ein Beispiel für eine Refinement-Beziehung der Aktivitätsdiagramme der Prozessmuster A und B.

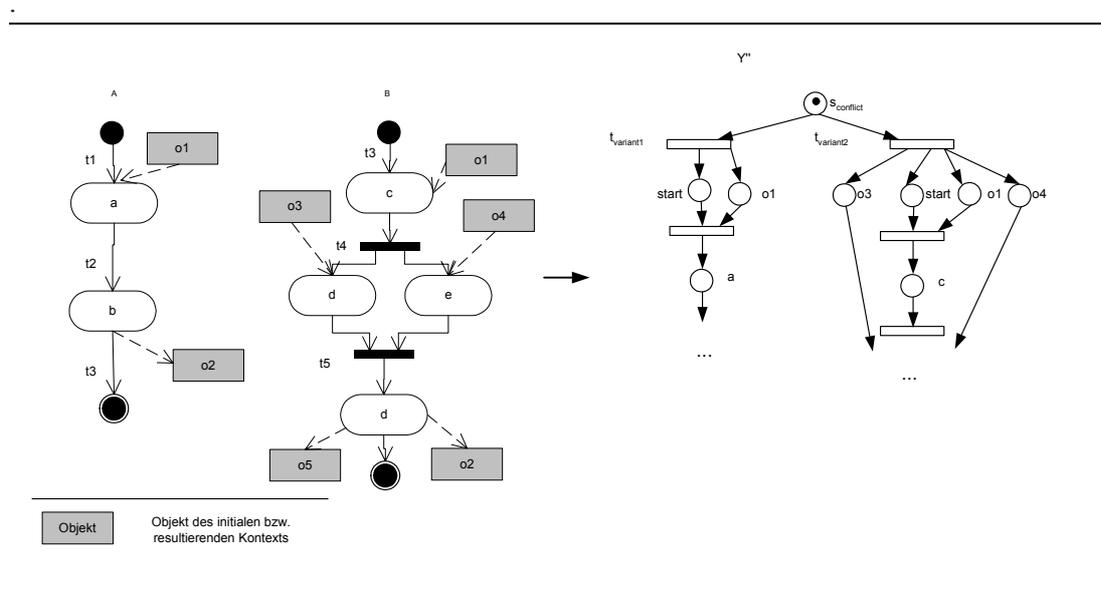


Abbildung 5-12: Beispiel für eine Refinement-Beziehung

Durch die Abbildung der Aktivitätsdiagramme A und B auf Petri-Netze Y und Y' und die Verknüpfung der beiden Netze entsteht das Petri-Netz Y''. Hierbei wird ein Konflikt eingeführt, d.h. es werden zwei neue Transitionen eingeführt, die über eine gemeinsame Inputstelle miteinander in Konflikt stehen. Darüber hinaus werden weitere Kanten hinzugefügt, die von diesen beiden Transition zu den „start“-Stellen und den Stellen des initialen Kontexts der beiden Netze führen.

Das Verhalten der einzelnen B/E-Systeme wird nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn alle seine Teilsysteme deadlockfrei sind. Diese Aussage kann dadurch erläutert werden, dass beim Verknüpfen der Netze die beiden Netze nicht manipuliert werden. Durch die Einführung der Stelle s_conflict wird lediglich sichergestellt, dass nur eines der beiden Teilnetze durchlaufen wird. Über die neu eingeführten Transitionen und Kanten wird sichergestellt, dass alle Stellen des initialen

Kontexts eine Marke erhalten. Schaltet eine der konfligierenden Transitionen, wird eine Markierung erreicht, die der Anfangsmarkierung einer der Ursprungsnetze Y und Y' entspricht.

Die Processvariance-Beziehung verknüpft zwei alternative Prozessmuster. Hierzu wird ein Petri-Netz definiert, das diese beiden Prozessmuster als Teilnetze enthält (analog Refinement-Beziehung). Durch Einführung zweier Transitionen, die in Konflikt stehen, kann nur eines der beiden Teilnetze durchlaufen werden. Wie auch bei den vorhergehenden Beziehungen wird das ursprüngliche Verhalten der beiden Ausgangsnetze nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn beide Teilnetze deadlockfrei sind.

Processvariance-
Beziehung

$$\begin{aligned}
 \text{rel}_4[\text{Processvariance}] &= \text{rel}_4[\text{pvar: variant1 ProcessPattern}^1 \text{ variant2 ProcessPattern}^2 \\
 &\quad \dots] \\
 &:= \text{Conflict}(\text{pr}_2[\text{ProcessPattern}^1], \text{pr}_2[\text{ProcessPattern}^2], \text{init}_4[\text{Z}] \cup \\
 &\quad \text{ic}_1[\text{ProcessPattern}^1], \text{init}_4[\text{Z}] \cup \text{ic}_1[\text{ProcessPattern}^2])
 \end{aligned}$$

Abbildung 5-13 zeigt ein Beispiel für eine Processvariance-Beziehung der Aktivitätsdiagramme der varianten Prozessmuster A und B.

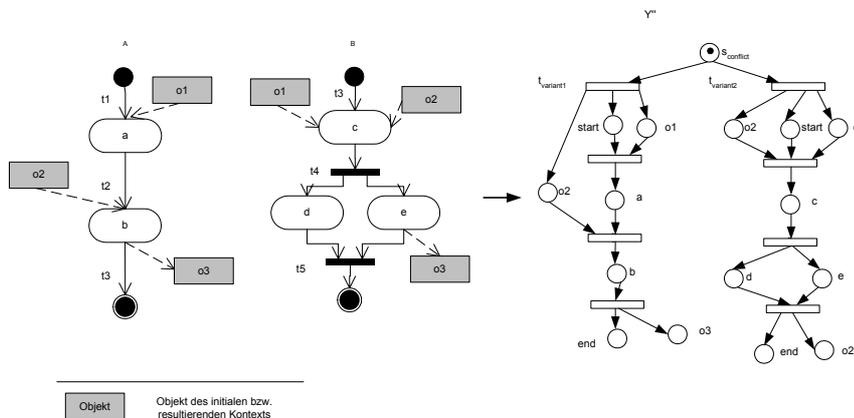


Abbildung 5-13: Beispiel für eine Processvariance-Beziehung

Durch die Abbildung der Aktivitätsdiagramme A und B auf Petri-Netze Y und Y' und die Verknüpfung der beiden Netze entsteht das Petri-Netz Y''. Hierbei wird ein Konflikt eingeführt, d.h. es werden zwei neue Transitionen eingeführt, die über eine gemeinsame Inputstelle miteinander in Konflikt stehen. Darüber hinaus werden weitere Kanten hinzugefügt, die von diesen beiden Transitionen zu den „start“-Stellen und den Stellen des initialen Kontexts der beiden Netze führen.

Das Verhalten der einzelnen B/E-Systeme wird nicht verändert. Dies bedeutet insbesondere, dass das auf diese Weise erzeugte B/E-System deadlockfrei ist, wenn alle seine Teilsysteme deadlockfrei sind. Diese Aussage kann dadurch erläutert werden, dass beim Verknüpfen der Netze die beiden Netze nicht manipuliert werden. Durch die Einführung der Stelle s_conflict

wird lediglich sichergestellt, dass nur eines der beiden Teilnetze durchlaufen wird. Über die neu eingeführten Transitionen und Kanten wird sichergestellt, dass alle Stellen des initialen Kontexts eine Marke erhalten. Schaltet eine der konfligierenden Transitionen, wird eine Markierung erreicht, die der Anfangsmarkierung einer der Ursprungsnetze Y und Y' entspricht.

5.3.8 Berücksichtigung der Objekt- und Ereigniskomposition bei den Prozessmusterbeziehungen

Berücksichtigt man die Objekt- und Ereigniskomposition bei den Prozessmusterbeziehungen, so müssen die Kontexte der Prozessmuster von ihren Aggregaten befreit und durch deren Aggregatbestandteile ersetzt werden. Dies wird durch die folgenden vier semantischen Gleichungen ausgedrückt. Für die Erläuterung der semantischen Gleichungen greifen wir auf die Beispiele aus Abschnitt 3.3.5 zurück.

Dekomposition bei
Sequence

$$\begin{aligned} \text{rel}_5[\text{Sequence}] &= \text{rel}_5[\text{seq: successor ProcessPattern predecessor ProcessPattern*} \\ &\quad \dots] \\ &:= \text{Connect}(\text{dec}_6[\text{ProcessPattern*}], \text{ic}_2[\text{ProcessPattern*}], \\ &\quad \text{rc}_2[\text{ProcessPattern*}], \text{dec}_6[\text{ProcessPattern}], \\ &\quad \text{ic}_2[\text{ProcessPattern}], \text{rc}_2[\text{ProcessPattern}]) \end{aligned}$$

Beispiel Ein Beispiel für die Dekomposition bei der Sequence-Beziehung zeigt Abbildung 5-14.

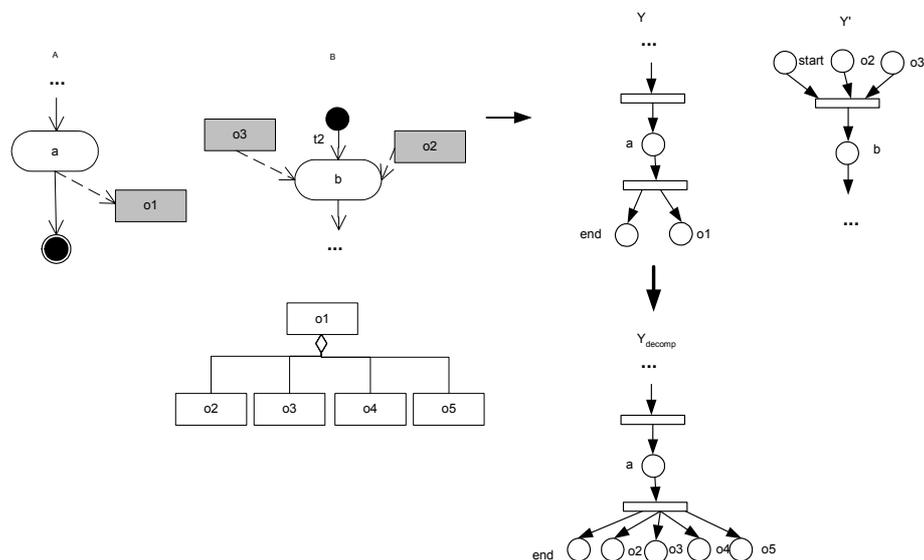


Abbildung 5-14: Dekomposition bei der Sequence-Beziehung

A und B sind zwei Prozessmuster, die aufgrund der Beschaffenheit nicht in einer Sequence-Beziehung stehen können (s. auch nebenstehende, korrespondierende Petri-Netze Y und Y'). Dekomponiert man jedoch das Objekt $o1$ in seine Aggregatbestandteile $o2$, $o3$, $o4$ und $o5$,

erhält man das Petri-Netz Y_{decomp} . Durch diese Modifikation ist nun eine Sequence-Beziehung zwischen den Petri-Netzen Y_{decomp} als Vorgängernetz und Y' als Nachfolgernetz erlaubt.

$$\begin{aligned}
 rel_6[[Use]] &= rel_6[[\mathbf{use}: \dots \mathbf{component} \text{ ProcessPattern}^1 \mathbf{composite} \text{ ProcessPattern}^2 \dots]] \\
 &:= Include(dec_6[[\text{ProcessPattern}^2]], dec_6[[\text{ProcessPattern}^1]], \\
 &\quad ic_2[[\text{ProcessPattern}^1]], rc_2[[\text{ProcessPattern}^1]])
 \end{aligned}$$

Dekomposition bei Use

Ein Beispiel für die Dekomposition bei der Use-Beziehung zeigt Abbildung 5-15. A und B sind zwei Prozessmuster, die aufgrund der Beschaffenheit nicht in einer Use-Beziehung stehen können (s. auch nebenstehende, korrespondierende Petri-Netze Y und Y'). Dekomponiert man jedoch das Objekt $o1$ in seine Aggregatbestandteile $o2, o3, o4$ und $o5$, erhält man das Petri-Netz Y_{decomp} . Durch diese Modifikation ist nun eine Use-Beziehung zwischen den Petri-Netzen Y_{decomp} als Kompositnetz (hier ist nur die Aktivität, die durch Y' gelöst wird dargestellt, nicht das vollständige, dem gesamten Prozessmuster entsprechende Netz) und Y' als Komponentennetz erlaubt.

Beispiel

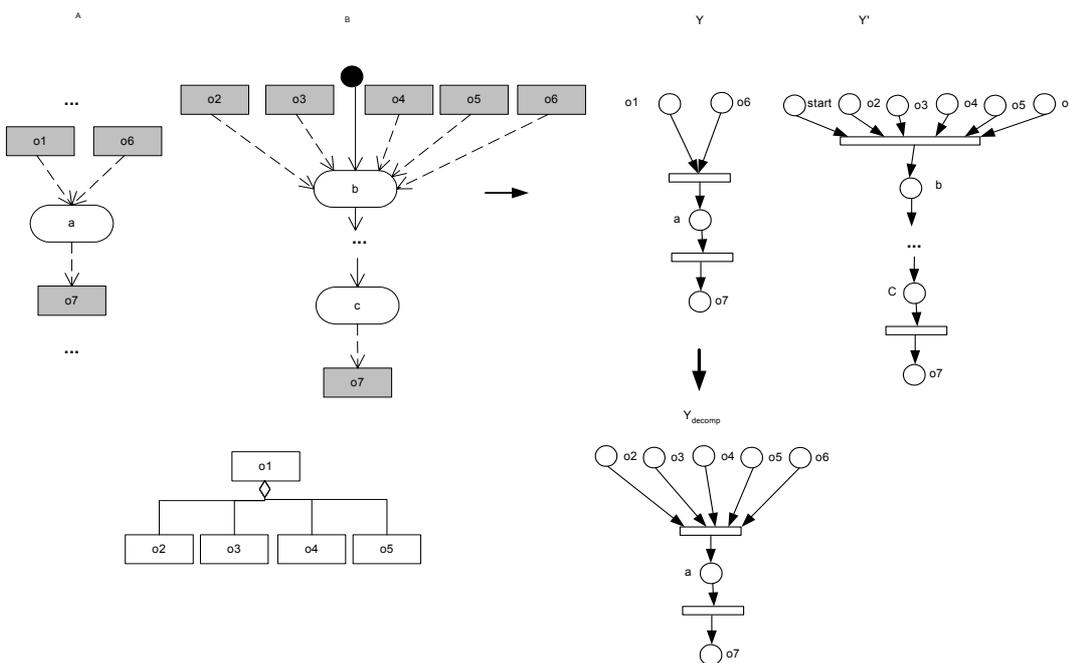


Abbildung 5-15: Dekomposition bei der Use-Beziehung

Dekomposition bei
Refinement

$$\begin{aligned} \text{rel}_7[[\text{Refinement}]] &= \text{rel}_3[[\text{ref: subPattern ProcessPattern}^1 \text{ superPattern ProcessPattern}^2 \dots]] \\ &:= \text{Conflict}(\text{dec}_6[[\text{ProcessPattern}^1]], \text{dec}_6[[\text{ProcessPattern}^2]], \text{init}_4[[Z^1]] \\ &\quad \cup \text{ic}_2[[\text{ProcessPattern}^1]], \text{init}_4[[Z^2]] \cup \text{ic}_2[[\text{ProcessPattern}^2]]) \end{aligned}$$

Beispiel

Ein Beispiel für die Dekomposition bei der Refinement-Beziehung zeigt Abbildung 5-16. A und B sind zwei Prozessmuster, die aufgrund der Beschaffenheit nicht in einer Refinement-Beziehung stehen können (s. auch nebenstehende, korrespondierende Petri-Netze Y und Y'). Dekomponiert man jedoch das Objekt o1 in seine Aggregatbestandteile o2, o3, o4 und o5, erhält man das Petri-Netz Y_{decomp} . Durch diese Modifikation ist nun eine Refinement-Beziehung zwischen den Petri-Netzen Y_{decomp} als Supernetz und Y' als Subnetz erlaubt.

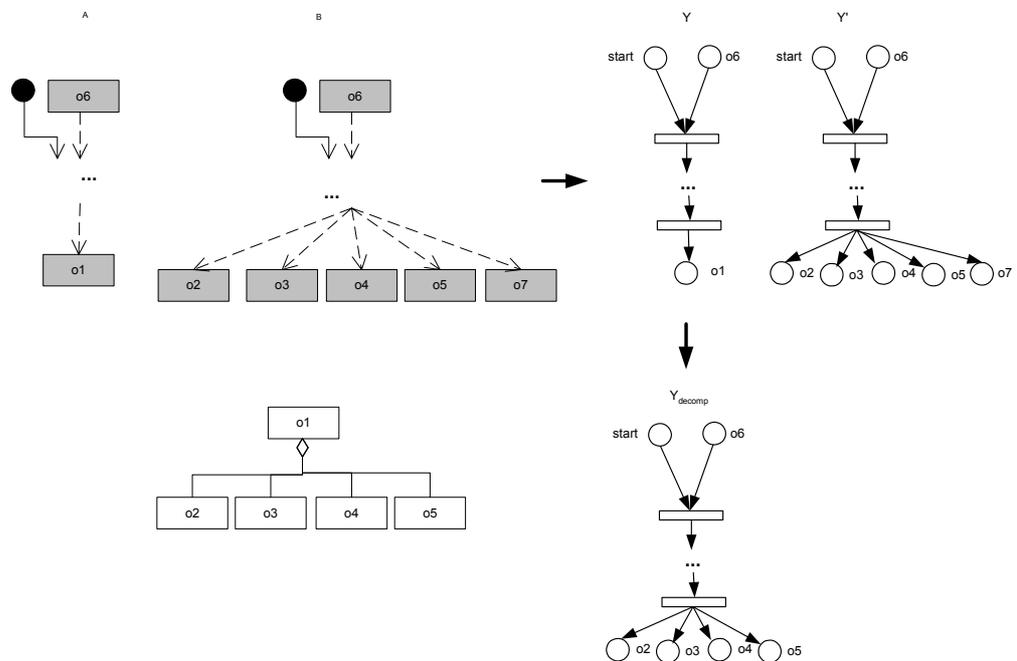


Abbildung 5-16: Dekomposition bei der Refinement-Beziehung

$$\begin{aligned}
 \text{rel}_8[\text{Processvariance}] &= \text{rel}_4[\text{pvar: variant1 ProcessPattern}^1 \text{ variant2 ProcessPattern}^2 \\
 &\quad \dots] \\
 &:= \text{Conflict}(\text{dec}_6[\text{ProcessPattern}^1], \text{dec}_6[\text{ProcessPattern}^2], \text{init}_4[\text{Z}^1]) \\
 &\quad \cup \text{ic}_2[\text{ProcessPattern}^1], \text{init}_4[\text{Z}^2] \cup \text{ic}_2[\text{ProcessPattern}^2])
 \end{aligned}$$

Dekomposition bei Processvariance

Ein Beispiel für die Dekomposition bei der Processvariance-Beziehung zeigt Abbildung 5-17. A und B sind zwei Prozessmuster, die aufgrund der Beschaffenheit nicht in einer Processvariance-Beziehung stehen können (s. auch nebenstehende, korrespondierende Petri-Netze Y und Y'). Dekomponiert man jedoch das Objekt o1 in seine Aggregatbestandteile o2, o3, o4 und o5, erhält man das Petri-Netz Y_{decomp}. Durch diese Modifikation ist nun eine Processvariance-Beziehung zwischen den Petri-Netzen Y_{decomp} und Y' als Netzvarianten erlaubt.

Beispiel

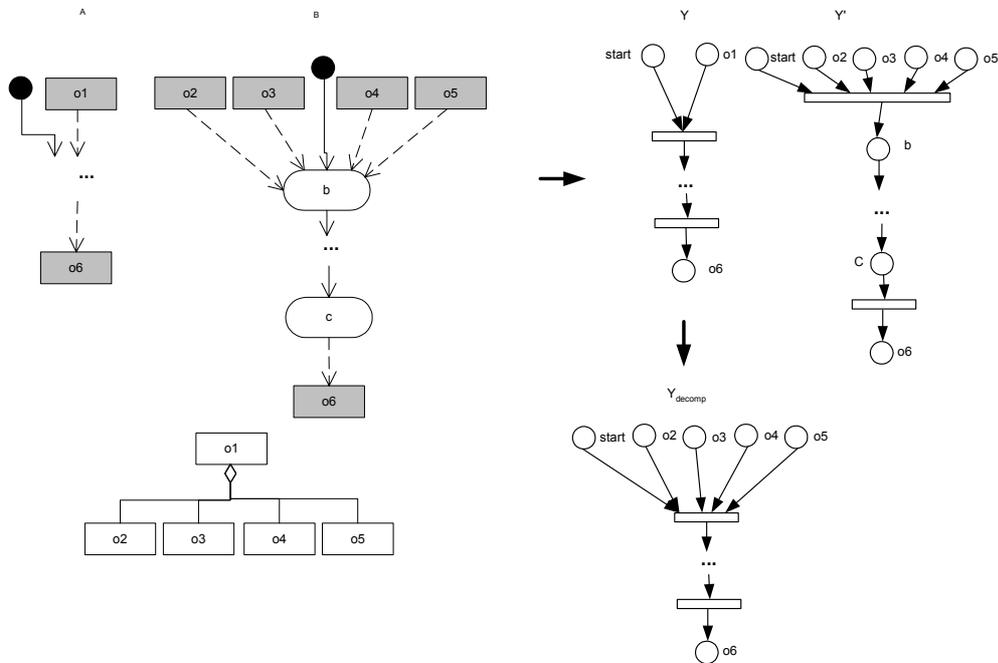


Abbildung 5-17: Dekomposition bei der Processvariance-Beziehung

6 Notation

In diesem Abschnitt werden die Notationselemente eingeführt und erläutert, die zur Darstellung der in Kapitel 4 und Kapitel 5 definierten syntaktischen und semantischen Elementen dienen. Durch Abbildung dieser Notation auf die abstrakte Syntax entsteht die sogenannte konkrete Syntax. Die abstrakte Syntax und die zugehörige Notation werden also analog zur UML-Spezifikation separat definiert. Dies unterstreicht die Tatsache, dass die Syntax notationsunabhängig und die Notation jederzeit auswechselbar ist. Für alle anderen Elemente der UML, die wir nicht geändert haben, verwenden wir die vorhandenen Notationen.

Notation wird separat definiert

Alle folgenden Abschnitte besitzen die gleiche Struktur: Zunächst erfolgt die Erläuterung der Notation und optionaler Präsentationsalternativen. Ein Beispiel erläutert die Anwendung der Notation. Schließlich wird erläutert, wie die Notationselemente auf die abstrakte Syntax abgebildet werden (s. Tabelle 6-1 für eine Übersicht). In Abschnitt 6.1 werden Notationselemente zur Darstellung einzelner Prozessmuster spezifiziert, in Abschnitt 6.2 Notationselemente zur Darstellung von Prozessmusterbeziehungen.

Aufbau und Inhalt des Kapitels

Abschnitt	Notationselement	wird abgebildet auf Metamodell
6.1.1	Prozessmuster	ProcessPattern
6.1.2	Problem	Problem
6.1.3	Objekte und Ereignisse	ObjectFlowState
	Aggregation von Objekten und Ereignissen	OFSComposition
6.1.4	Spezialisierung/Generalisierung von Objekten und Ereignissen	GeneralizableElement, Generalization
6.1.5	Problemdiagramm	ProblemGraph
6.1.6	Kontext	Context
6.1.7	Action State	ActionState
6.1.8	Rolle	Role
6.1.9	Werkzeug	Tool
6.1.10	Prozessdiagramm	Process
6.2.1	Sequence	Sequence
6.2.2	Use	Use
		ActivityProblemMapping
6.2.3	Problemverfeinerungsdiagramm	RefineProblem

Tabelle 6-1: Abbildung von Notationslementen auf Metamodellelemente

Abschnitt	Notationselement	wird abgebildet auf Metamodell
6.2.4	Refinement	Refinement
6.2.5	Processvariance	Processvariance
6.2.6	Beziehungsdiagramm	Sequence, Use, Refinement, Processvariance
6.2.7	Prozessmusterdiagramm	ProcessPatternGraph
6.2.8	Katalogdiagramm	ProcessPatternCatalog

Tabelle 6-1: Abbildung von Notationslementen auf Metamodellelemente (Fortgesetzt)

6.1 Notationselemente zur Darstellung einzelner Prozessmuster

6.1.1 Prozessmuster

Notation Ein Prozessmuster wird als Rechteck mit abgerundeten Ecken präsentiert. In der Mitte des Rechtecks wird der Name des Prozessmusters angezeigt. Ein Prozessmuster wird mit anderen Prozessmustern über Beziehungen verknüpft (Abschnitte 6.2.1, 6.2.2, 6.2.4 und 6.2.5).

Präsentationsoptionen Der Aspekt eines Prozessmusters kann in der rechten unteren Ecke des Prozessmuster-Symbols angegeben werden.

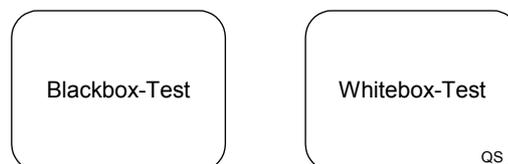


Abbildung 6-1: Notationsbeispiel Prozessmuster

Abbildung auf das Metamodell Ein Prozessmustersymbol wird auf die Metaklasse `ProcessPattern` abgebildet. Der Name des Symbols entspricht dem Attribute `name` der Metaklasse `ProcessPattern`. Der Aspekt wird auf die Metaklasse `Aspect` abgebildet.

6.1.2 Problem

Notation Ein Problem wird als Kreis präsentiert. In der Mitte des Kreises wird der Name des Problems angezeigt. Das Problemsymbol wird in zwei Diagrammtypen verwendet, im Problem Graph und im Katalogdiagramm.

Präsentationsoptionen Der Aspekt eines Problems kann im rechten unteren Abschnitt des Problem-Symbols angegeben werden.



Abbildung 6-2: Notationsbeispiel Problem

Das Problem-Symbol wird auf die Metaklasse Problem abgebildet. Der Name des Symbols entspricht dem Attribute name der Metaklasse Problem. Der Aspekt wird auf die Metaklasse Aspect abgebildet.

Abbildung auf das Metamodell

6.1.3 Objekte und Ereignisse

Für die Darstellung von Objekten wird wie in der UML-Spezifikation das Objektsymbol (Rechteck mit unterstrichenem Objektnamen in der Mitte, Objektzustand gegebenenfalls in eckigen Klammern unter dem Objektnamen) verwendet. Für Ereignisse führen wir ein neues Symbol ein: Wir benutzen das gleiche Symbol wie für Objekte, wobei der Ereignisname doppelt unterstrichen wird. Für die Aggregation von Objekten oder Ereignissen verwenden wir das in der UML übliche Aggregationssymbol (Kante mit einer hohlen Raute am Aggregat).

Notation

Die Angabe des Objekt- bzw. Ereigniszustands ist optional. Der Aspekt eines Objekts oder Ereignisses kann in der rechten unteren Ecke des ObjectFlowState-Symbols angegeben werden.

Präsentationsoptionen

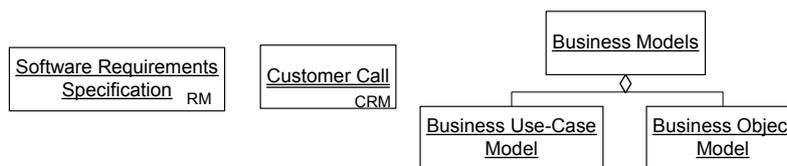


Abbildung 6-3: Notationsbeispiele für Objekte (links), Ereignisse (Mitte) und Aggregation (rechts)

Die Notationen für Objekte und Ereignisse werden auf die Metaklasse ObjectFlowState abgebildet. Der Aspekt wird auf die Metaklasse Aspect abgebildet. Die Aggregation wird auf die Metaklasse OFSComposition abgebildet.

Abbildung auf das Metamodell

6.1.4 Spezialisierung/Generalisierung von Objekten und Ereignissen

Objekte können andere Objekte konkretisieren. Dies gilt ebenso für Ereignisse. Die Spezialisierung/Generalisierung von Objekten und Ereignissen setzt man wie bei der Aggregation aus Gründen der Übersichtlichkeit und der Abstraktion ein. Ein Supermuster kann z.B. ein abstraktes Objekt in seinem Kontext besitzen, während das Submuster in seinem Kontext ein konkreteres Objekt besitzt.

Notation

Für die Darstellung der Spezialisierung wird wie in der UML-Spezifikation eine Kante mit einem leeren Dreieck verwendet, wobei das Dreieck an das spezialisierte Objekt/Ereignis anschließt.

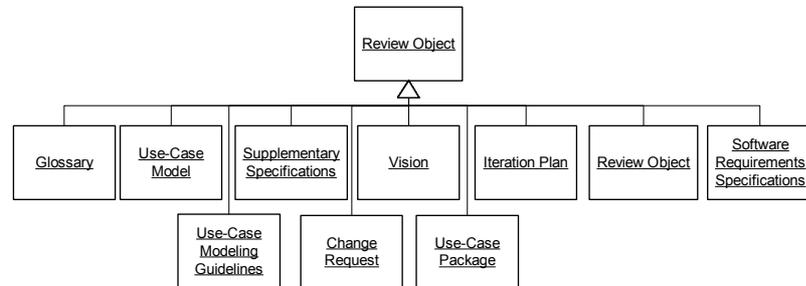


Abbildung 6-4: Notationsbeispiel für Objekt- und Ereignisspezialisierung

Abbildung auf das Metamodell

Die Notationen für Objekte und Ereignisse werden auf die UML-Metaklasse `GeneralizableElement` abgebildet. Das Symbol für die Spezialisierung wird auf die UML-Metaklasse `Generalization` abgebildet.

6.1.5 Problemdiagramm

Notation

Ein Problemdiagramm präsentiert zwei Sichten auf das zugehörige Problem: Die Kontextsicht auf der oberen Hälfte des Problemdiagramms präsentiert, welche Elemente zum Kontext des Problems gehören. Die Kontextelemente werden durch `ObjectFlowStates` dargestellt. Mit dem Problem werden die Kontextelemente über eine gerichtete Kante verbunden. Die Pattern-Sicht auf der unteren Hälfte des Problemdiagramms präsentiert, welche Prozessmuster das Problem lösen. Die Prozessmuster werden über die Notationselemente für die Prozessvarianz-Beziehung und die Refinement-Beziehung verbunden.

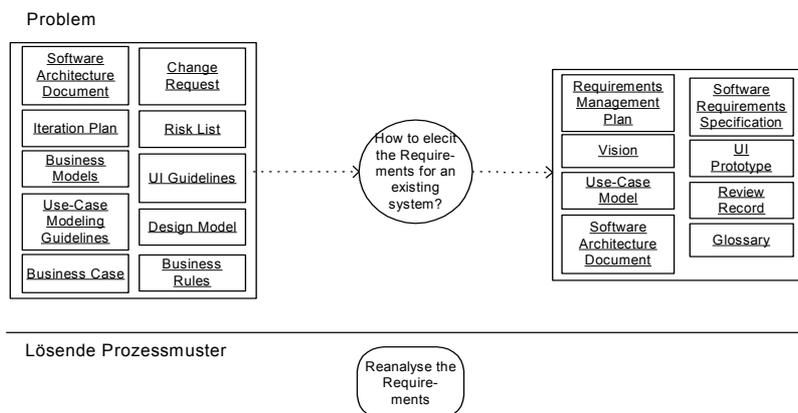


Abbildung 6-5: Notationsbeispiel für ein Problemdiagramm

Bei einer Vielzahl von Kontextelementen kann ein Kasten um die Elemente des initialen oder resultierenden Kontexts gezogen werden. Es wird dann nur diese Kasten mit der Kante verknüpft, nicht jedes einzelne Element.

Präsentationsoptionen

Das Problemdiagramm wird auf die Metaklasse ProblemGraph abgebildet. Die Abbildung aller anderen Modellelemente (Problem, ObjectFlowState, Prozessmuster, Processvariance, Refinement) findet man in den jeweiligen Abschnitten in diesem Kapitel.

Abbildung auf das Metamodell

6.1.6 Kontext

Der initiale Kontext präsentiert, welche Objekte und Ereignisse Voraussetzung für die Anwendung des Prozessmusters sind. Diese Objekte und Ereignisse, repräsentiert durch ObjectFlowStates, werden mit einer gerichteten Kante (Pfeilrichtung Prozessmuster) mit dem Prozessmuster, repräsentiert durch das Prozessmuster-Symbol, verbunden. Der resultierende Kontext präsentiert, welche Objekte und Ereignisse Resultat der Anwendung des Prozessmusters sind. Diese Objekte und Ereignisse, repräsentiert durch ObjectFlowStates, werden mit einer gerichteten Kante (Pfeilrichtung Objekte und Ereignisse des resultierenden Kontexts) mit dem Prozessmuster verbunden.

Notation

Ist der Kontext nicht eindeutig (z.B. dann, wenn es weitere Objekte und Ereignisse gibt, die nicht zum Kontext gehören), werden die Objekte und Ereignisse des Kontexts schattiert (Abbildung 6-10).

Präsentationsoptionen

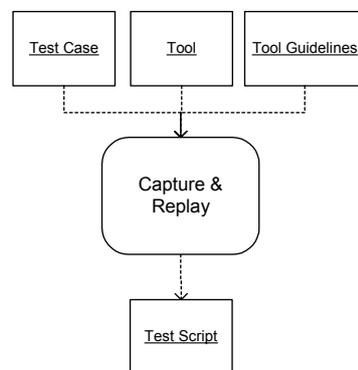


Abbildung 6-6: Notationsbeispiel für Kontext

Der Kontext wird auf die Metaklasse Context abgebildet. Die Abbildung aller anderen Modellelemente (ObjectFlowState, ProcessPattern) findet man in den jeweiligen Abschnitten in diesem Kapitel.

Abbildung auf das Metamodell

6.1.7 Action State

Ein Action State wird als Rechteck mit gewölbten Querkanten dargestellt. Wir haben zwei zusätzliche Notationen für einen Action State hinzugefügt:

Notation

- Action States, für die ein Problem im Musterkatalog existiert. Für Action States dieses Typs wird das Symbol von links unten nach rechts oben schraffiert dargestellt,

- Action States, für die ein Problem und mindestens ein Prozessmuster im Katalog existiert. Für Action States dieses Typs wird das Symbol zusätzlich von rechts unten nach links oben.



Abbildung 6-7: Notationsbeispiele für Action States (links), Action States mit zugehörigem Problem (mitte) und Action States mit zugehörigem Problem und Prozessmuster (rechts)

Abbildung auf das Metamodell

Alle drei Notationssymbole werden auf die Metaklassen `ActionState` und `ActivityProblemMapping` abgebildet.

6.1.8 Rolle

Notation Für Rollen wird das Personensymbol verwendet. Der Name der Rolle wird unterhalb des Symbols angegeben. Eine Rolle wird durch eine gestrichelte Linie mit der zugeordneten Aktivität verknüpft. Eine Rolle wird mit einem Prozess verknüpft, indem das Rollensymbol in der rechten oberen Ecke des Prozessdiagramms angegeben wird (Abbildung 6-10).

Präsentationsoptionen

Statt eine gestrichelte Linie zwischen Rolle und Aktivität zu ziehen, kann die Rolle auch direkt neben der Aktivität platziert werden. Die gestrichelte Linie entfällt dann. Führt eine Rolle mehrere Aktivitäten innerhalb eines Prozessdiagramms aus, so ist es erlaubt, die Rolle nur einmal abzubilden und von dieser Rolle mehrere gestrichelte Linien zu den zugehörigen Aktivitäten zu ziehen.



Abbildung 6-8: Notationsbeispiele für Rolle

Abbildung auf das Metamodell

Das Rollensymbol wird auf die Metaklasse `Role` abgebildet. Der Name der Rolle entspricht dem Attribut `name` der Metaklasse `Role`.

6.1.9 Werkzeug

Notation Für Werkzeuge wird das Geodreiecksymbol verwendet. Der Name des Werkzeugs wird unterhalb des Symbols angegeben. Ein Werkzeug wird durch eine gestrichelte Linie mit der zugeordneten Aktivität verknüpft.

Präsentationsoptionen

Statt eine gestrichelte Linie zwischen Werkzeug und Aktivität zu ziehen, kann das Werkzeug auch direkt neben der Aktivität platziert werden. Die gestrichelte Linie entfällt dann. Wird ein Werkzeug bei mehreren Aktivitäten innerhalb eines Prozessdiagramms eingesetzt, so ist es erlaubt, das Werkzeug nur einmal abzubilden und von diesem Werkzeug mehrere gestrichelte Linien zu den zugehörigen Aktivitäten zu ziehen.



Abbildung 6-9: Notationsbeispiele für Werkzeuge

Das Werkzeugsymbol wird auf die Metaklasse Tool abgebildet. Der Name des Werkzeugs entspricht dem Attribute name der Metaklasse Tool.

Abbildung auf das Metamodell

6.1.10 Prozessdiagramm

Ein Prozessdiagramm ist eine Erweiterung des UML-ActivityGraphs (Aktivitätsdiagramm). Wie im Aktivitätsdiagramm werden Aktivitäten modelliert, welche die Erstellung bestimmter Ergebnisse (in Gestalt von Objekten und Ereignissen) zu Ziel haben. Der Prozess und die Aktivitäten werden jedoch keiner Klasse zugeordnet, sondern einer Rolle, die für die Ausführung des Prozesses verantwortlich ist und die durch Menschen ausgefüllt wird. Ferner können Werkzeuge den Aktivitäten zugeordnet werden.

Notation

Das die Prozessrolle repräsentierende Personensymbol ist in der oberen rechten Ecke angeordnet. Die je Aktivität spezifizierte Rolle wird wie in Abschnitt 6.1.8 neben der Aktivität abgebildet. Im Rahmen von Aktivitäten einzusetzende Werkzeuge werden durch Anzeige des Werkzeugsymbols neben der Aktivität dargestellt. Die Objekte des initialen und resultierenden Kontexts werden durch Schattierung hervorgehoben.

Ist die Prozessrolle mit sämtlichen Aktivitätenrollen identisch, so reicht die Angabe der Prozessrolle aus, die Anzeige der Aktivitätenrollen kann entfallen.

Präsentationsoptionen

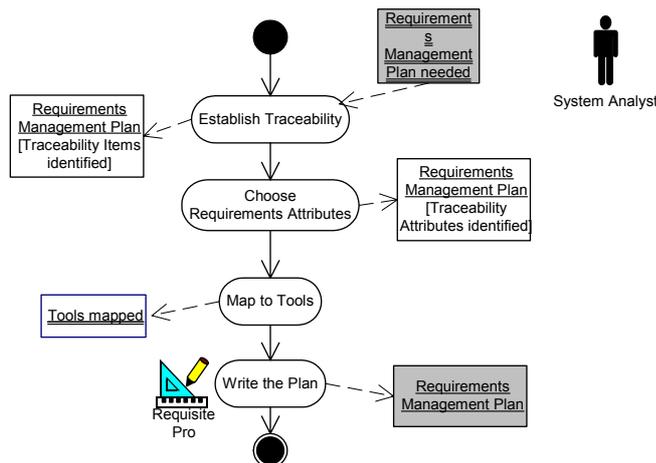


Abbildung 6-10: Notationsbeispiel für ein Prozessdiagramm

Abbildung auf das Metamodell Ein Prozessdiagramm wird auf die Metaklasse `Process` abgebildet. Die Abbildung aller anderen Modellelemente (Rolle, Werkzeug, Aktivität, Transitionen) findet man in den jeweiligen Abschnitten in diesem Kapitel.

6.2 Notationselemente zur Darstellung von Prozessmusterbeziehungen

6.2.1 Sequence

Notation Die Sequence-Beziehung wird als durchgezeichnete, gerichtete Kante mit einem kleinen senkrechten Strich am Ausgangspunkt der Kante dargestellt. Die Kante beginnt bei den Vorgängermustern und endet bei dem Nachfolgermuster. Bei einem Vorgängermuster wird eine Kante vom Vorgängermuster zum Nachfolgermuster gezogen (Beispiel A). Bei mehreren Vorgängermustern wird eine Kante von jedem Vorgängermuster zum Nachfolgermuster gezogen, wobei die Pfeilenden am Nachfolgermuster vereinigt werden (Beispiel B). Alternativ kann – empfehlenswert bei einer großen Anzahl von Vorgängermustern – ein Kasten um die Vorgängermuster gezogen werden, von dem eine Kante zum Nachfolgermuster gezogen wird (Beispiel C). Mehrere Sequence-Beziehungen können in abgekürzter Schreibweise angegeben werden, indem von den Vorgängermustern eine Kante zu einem Kasten gezogen wird, in dem sich alle Nachfolgermuster befinden (Beispiel D). Das Prozessmuster „Develop Requirements Management Plan“ steht also jeweils mit den Prozessmustern „Detail a Use-Case“, „Detail the Software Requirements“ usw. in einer Sequence-Beziehung.

Präsentationsoptionen Gibt es alternative Prozessmuster, die zur Erzeugung bestimmter Objekte oder Ereignisse zur Verfügung stehen, so können diese alternativen Muster durch überlappende Prozessmustersymbole angezeigt werden (Beispiel E).

Die möglichen Nachfolger eines Prozessmusters sind eine wichtige Information. Ist das betrachtete Prozessmuster das einzige Vorgängermuster eines Nachfolgermusters, so kann dies wie in Beispiel A dargestellt werden. Ist das Prozessmuster ein Element aus einer Menge von Vorgängermustern, wird dies wie in Beispiel F mit einem zweimal gestrichenen Pfeil gekennzeichnet. Die Angabe aller anderen Elemente aus dieser Menge ist nicht sinnvoll, da hierunter die Übersichtlichkeit leidet. Darüber hinaus ist diese Information bereits in den Prozessmusterdiagrammen der nachfolgenden Prozessmuster vorhanden. Der Anwender erhält also lediglich die Information, mit welchen Prozessmustern er fortfahren kann und dass das betrachtete Prozessmuster nicht das einzige Vorgängermuster des jeweiligen Nachfolgermusters ist.

Abbildung auf das Metamodell Das Symbol für die Sequence-Beziehung wird auf die Metaklasse `Sequence` abgebildet. Die Prozessmustersymbole werden auf die Metaklasse `ProcessPattern` abgebildet.

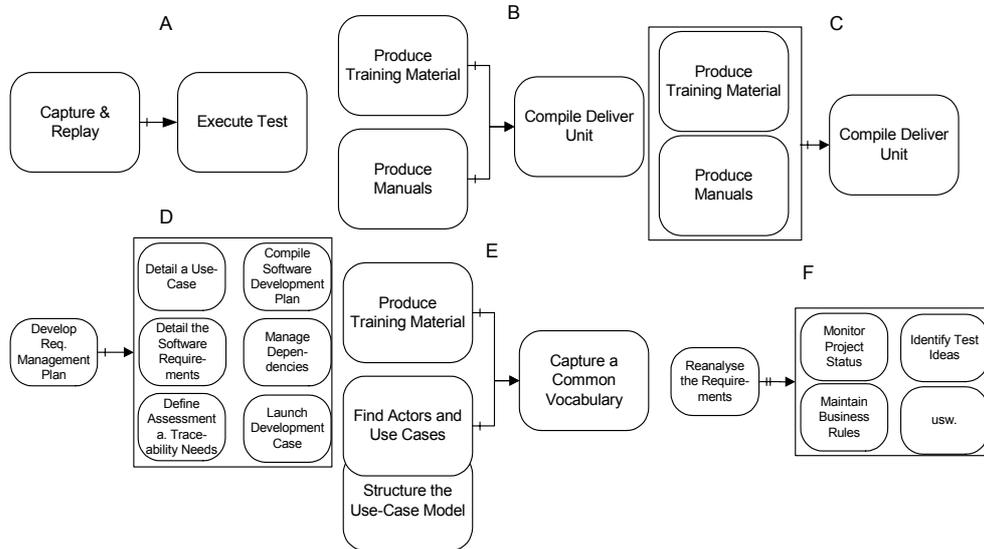


Abbildung 6-11: Notationsbeispiele: Einfache Sequence-Notation (A), mit vereinigten Pfeilenden (B) oder alternativ mit Kästen (C), abgekürzte Schreibweise für mehrere Sequence-Beziehungen (D), für alternative Vorgängermuster (E) und für die Darstellung eines Vorgängermusters aus ein Menge von Vorgängermustern (E)

6.2.2 Use

Die Use-Beziehung wird als durchgezeichnete, gerichtete Kante mit einem kleinen Quadrat am ihrem Anfang dargestellt, die zwei Prozessmustersymbole miteinander verknüpft. Die Kante beginnt bei dem abstrakteren Muster und endet bei dem detaillierteren Muster.

Notation

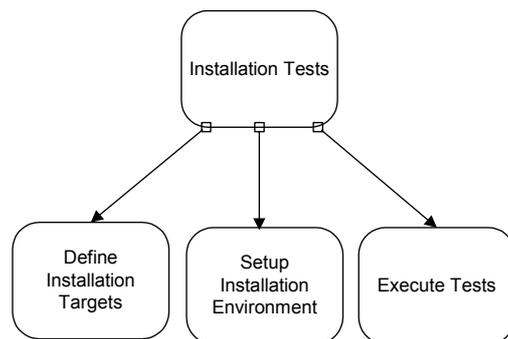


Abbildung 6-12: Notationsbeispiel Use

Das Symbol für die Use-Beziehung wird auf die Metaklasse Use abgebildet. Die Prozessmustersymbole werden auf die Metaklasse ProcessPattern abgebildet.

Abbildung auf das Metamodell

6.2.3 Problemverfeinerungsdiagramm

Notation Die RefineProblem-Beziehung wird durch zwei Sichten beschrieben. Die erste Sicht auf der linken Seite zeigt, welche Probleme durch eine RefineProblem-Beziehung verknüpft sind. Die Verfeinerung wird als durchgezeichnete, gerichtete Kante mit einer hohlen Pfeilspitze am Pfeilende dargestellt, die zwei Problem-Symbole miteinander verknüpft. Die Kante beginnt bei dem Subproblem und endet bei dem Superproblem. Die zweite Sicht zeigt, wie die Objekte und Ereignisse der Probleme aufeinander abgebildet werden. Hierbei wird nach initialem und resultierendem Kontext des Problems unterschieden, gekennzeichnet durch die Schlüsselwörter IN und OUT. Jeweils ein Objekt oder Ereignis des Superproblems wird mit einem oder mehreren Objekten oder Ereignissen verknüpft. Diese Verknüpfung zeigt, wie ein Objekt oder Ereignis verfeinert wird.

Beispiel Im folgenden Beispiel wird das Objekt „Requirements Documents“ zu den Objekten „Vision“, „Business Models“ usw. verfeinert. Das Objekt „Glossary“ des Problemoutputs wird nicht verfeinert.

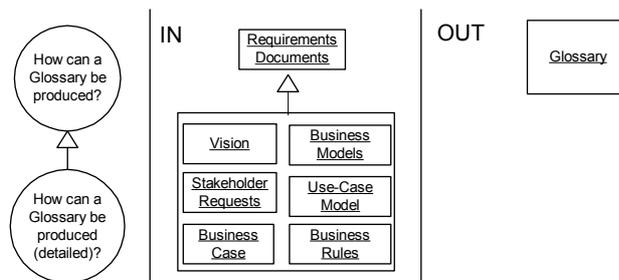


Abbildung 6-13: Notationsbeispiel für Problemverfeinerung

Abbildung auf das Metamodell Das Symbol für die RefineProblem-Beziehung wird auf die Metaklasse RefineProblem abgebildet. Die Problem-Symbole werden auf die Metaklasse Problem abgebildet. Die Objekt- und Ereignissymbole werden auf die Metaklasse ObjectFlowState abgebildet.

6.2.4 Refinement

Notation Die Refinement-Beziehung wird als durchgezeichnete, gerichtete Kante mit einer hohlen Pfeilspitze am Pfeilende dargestellt, die zwei Prozessmustersymbole miteinander verknüpft. Die Kante beginnt bei dem Submuster und endet bei dem Supermuster.

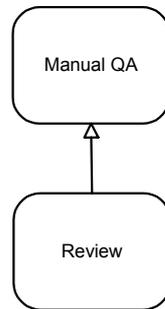


Abbildung 6-14: Notationsbeispiel für Refinement

Das Symbol für die Refinement-Beziehung wird auf die Metaklasse Refinement abgebildet. Die Prozessmustersymbole werden auf die Metaklasse ProcessPattern abgebildet.

Abbildung auf das Metamodell

6.2.5 Processvariance

Die Processvariance-Beziehung wird als durchgezeichnete, ungerichtete Kante dargestellt, die zwei Prozessmustersymbole miteinander verknüpft.

Notation



Abbildung 6-15: Notationsbeispiel für Prozessvarianten

Das Symbol für die Processvariance-Beziehung wird auf die Metaklasse Processvariance abgebildet. Die Prozessmustersymbole werden auf die Metaklasse ProcessPattern abgebildet.

Abbildung auf das Metamodell

6.2.6 Beziehungsdiagramm

Ein Beziehungsdiagramm repräsentiert alle zu einem Prozessmuster in Beziehung stehenden Prozessmuster. D.h. es werden alle Prozessmuster aufgeführt, die Vorgänger- oder Nachfolgermuster, Komposit- oder Komponentenmuster, Super- oder Submuster oder eine Prozessvariante des Prozessmusters sind.

Notation

Für die Darstellung der Beziehungen werden die in den Abschnitten 6.2.1 bis 6.2.5 definierten Notationssymbole verwendet. Für die Prozessmuster wird das Prozessmustersymbol verwendet.

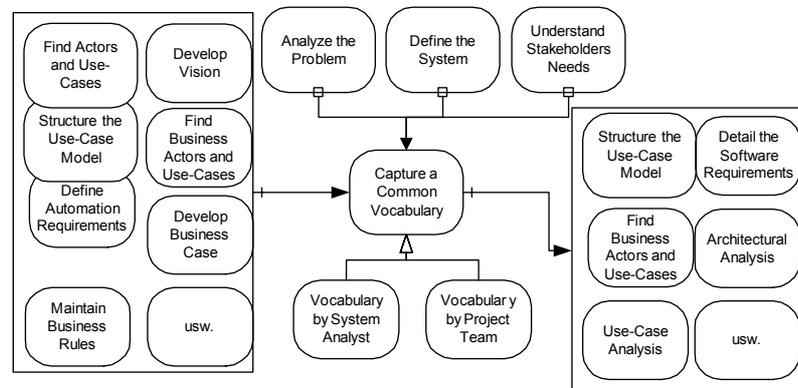


Abbildung 6-16: Notationsbeispiel für ein Beziehungsdiagramm

Abbildung auf das Metamodell Das Relationship-Diagramm wird auf die Metaklassen für die Prozessmusterbeziehungen abgebildet. Die Prozessmustersymbole werden auf die Metaklasse ProcessPattern abgebildet.

6.2.7 Prozessmusterdiagramm

Notation Ein Prozessmusterdiagramm präsentiert drei Sichten auf das zugehörige Prozessmuster: Die Prozessmuster-Sicht auf der oberen Seite des Prozessmusterdiagramms präsentiert das durch das Prozessmuster zu lösende Problem. Die Beziehungssicht in der Mitte des Prozessmusterdiagramms präsentiert, mit welchen Prozessmustern das Prozessmuster in Beziehung steht. Die Prozess-Sicht im unteren Teil des Prozessmusterdiagramms präsentiert den Prozess des Prozessmusters.

Abbildung auf das Metamodell Das Prozessmusterdiagramm wird auf die Metaklasse ProcessPatternGraph abgebildet. Die Abbildung aller anderen Modellelemente (Prozessmuster, Problem, Prozessdiagramm, Beziehungsdiagramm) findet man in den jeweiligen Abschnitten in diesem Kapitel.

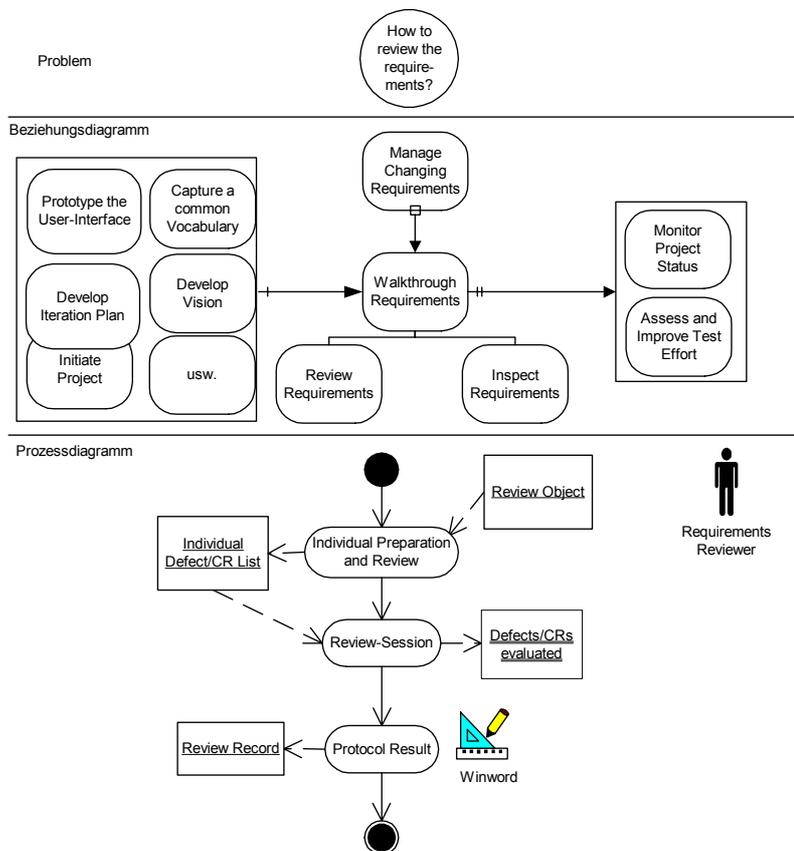


Abbildung 6-17: Notationsbeispiel für ein Prozessmusterdiagramm

6.2.8 Katalogdiagramm

Notation

Ein Katalogdiagramm ist ein gerichteter Graph mit Knoten (Prozessmuster) und Bögen (Beziehungen). Für die Darstellung der Graphenelemente werden die den einzelnen Elementen entsprechenden Notationselemente verwendet. Um die Übersichtlichkeit zu erhöhen, wird der Katalog in verschiedenen Sichten angegeben. Für jede Beziehung gibt es eine Sicht.

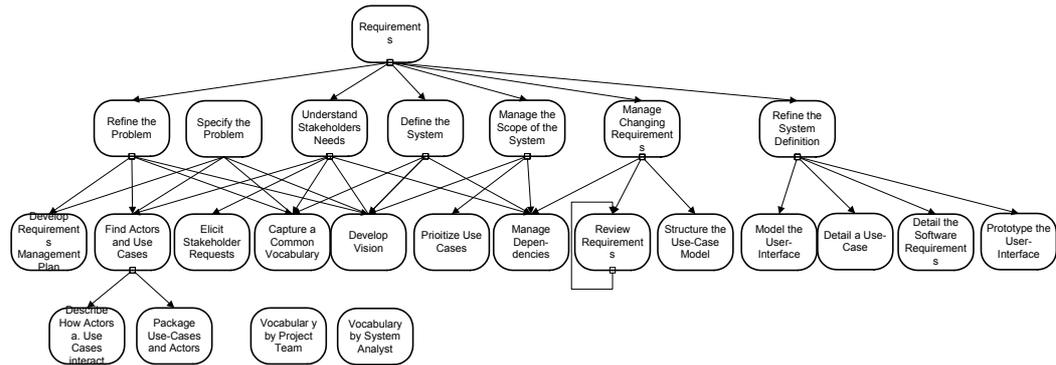


Abbildung 6-18: Notationsbeispiel für ein Katalogdiagramm – Use-Sicht

Abbildung auf das
Metamodell

Das Katalogdiagramm wird auf die Metaklasse `ProcessPatternCatalog` abgebildet. Die Abbildung aller anderen Modellelemente findet man in den jeweiligen Abschnitten.

7 Der Prozessmusterkatalog CADs

In diesem Kapitel präsentieren wir einen Ausschnitt aus dem Prozessmusterkatalog CADs (Catalog for the Development of Software), der die Anwendung der in den Kapiteln 3 bis 6 entwickelten Konzepte veranschaulicht. Im nachfolgenden Kapitel 8 wird ein Ausschnitt des Katalogs in elektronischer Form auf der Process Pattern Workbench präsentiert.

Inhalt des Kapitels

Abschnitt 7.1 legt dar, wie wir den Prozessmusterkatalog entwickelt und welchen Ausschnitt des RUPs wir als Vorlage verwendet haben. In Abschnitt 7.2 werden zum Zwecke der Gesamtschau aller Prozessmuster alle Sichten des Prozessmusterkatalogs angegeben. Abschnitt 7.3 zeigt die von den Problemen und Prozessmustern verwendete Objekt- und Ereignisstruktur. In Abschnitt 7.4 werden die detailliert betrachteten Probleme und Prozessmuster angegeben. Strategien zur Anwendung von Prozessmusterkatalogen werden in Abschnitt 7.5 erläutert. Schließlich erfolgt in Abschnitt 7.6 eine Zusammenfassung.

Gliederung des Kapitels

7.1 Ableitung des Prozessmusterkatalogs

Für die Erstellung des Prozessmusterkatalogs CADs greifen wir auf ein bestehendes Vorgehensmodell, den Rational Unified Process [Kru00], zurück. Die Wahl fiel auf dieses Vorgehensmodell aus verschiedenen Gründen:

- Der RUP gehört zu den zur Zeit am weit verbreitetsten Vorgehensmodellen. Da das Beispiel an den RUP angelehnt ist, ist die Wahrscheinlichkeit höher, dass der Leser die Prozesse bereits kennt. Daher ist nur noch der Transfer in die Darstellung der Prozessmuster nachzuvollziehen. Ziel dieses Kapitels ist deswegen nicht, ein Verständnis über den Inhalt der Prozessmuster zu erlangen, sondern über die Modellierung der Prozessmuster.
- Die Akzeptanz des RUPs lässt darauf schließen, dass sich die RUP-Prozesse in der Praxis bewährt haben. Der RUP kann also als praxisnah bezeichnet werden. Durch Darstellung unserer Konzepte an einem praxisnahem Beispiel kann aber gerade die Praxistauglichkeit unserer Konzepte bewiesen werden. Desweiteren können die Vorteile von PROPEL besser herausgearbeitet werden, da eine Vergleichbarkeit der Prozessbeschreibungen in RUP und PROPEL gegeben ist. Desweiteren können anhand des RUPs Unzulänglichkeiten in der Prozessbeschreibung identifiziert werden, und mit PROPEL verglichen werden.
- Der RUP ist bereits hierarchisch vorstrukturiert. Durch die Beschreibung von Prozessen auf Disziplin-, Workflowdetail- und Aktivitätenebene ist eine Komposition der Prozesse vorhanden. Diese erleichtert es, Use-Beziehungen zwischen den Prozessmustern zu identifizieren.

Fokus auf Prozessbeschreibung, nicht Prozessinhalt

Praxisnahes Beispiel

Hierarchische Vorstrukturierung des RUP

V-Modell 97 als
Alternative

Obwohl der RUP also als ein sehr geeignetes Objekt zur Darstellung unserer Konzepte scheint, sei der Leser darauf hingewiesen, dass das Beispiel auch mit anderen Prozessmodellen (z.B. V-Modell 97) hätte durchgeführt werden können.

RUP: mächtig und
komplex

Der RUP ist ein sehr mächtiges Vorgehensmodell. Er beinhaltet neun thematisch orientierte Disziplinen, die sich aus knapp sechzig Workflowdetails und rund einhundertfünfzig Aktivitäten zusammensetzen. Zu der Mächtigkeit des RUPs kommt noch die Komplexität durch die enge Verknüpfung von Disziplinen, Workflowdetails und Aktivitäten hinzu. Aus diesen Gründen präsentieren wir in diesem Kapitel nur einen Ausschnitt des RUPs. Für die Darstellung der in dieser Arbeit eingeführten Konzepte haben wir eines der sechs Workflowdetails der Disziplin „Requirements“ ausgewählt (Abbildung 7-1). Mit Hilfe dieses Ausschnitts können jedoch alle PROPEL-Konzepte gezeigt werden.

Abbildung 7-1 zeigt den hierarchischen Aufbau der RUP-Disziplin „Requirements“. Die Disziplin besteht aus Workflowdetails, diese bestehen wiederum aus Aktivitäten. Die grau unterlegten Symbole geben an, welche Elemente wir zur Ableitung von Prozessmustern detaillierter betrachten.

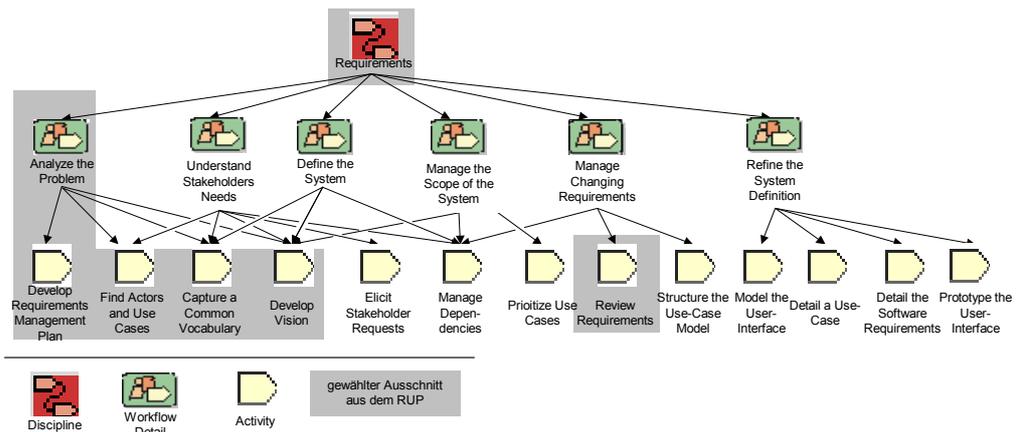


Abbildung 7-1: Aufbau der RUP-Disziplin „Requirements“

7.2 Sichten des Prozessmuskatalogs

Ziel Nachfolgend zeigen wir den von uns gewählten Ausschnitt des Prozessmuskatalogs mit Hilfe verschiedener Sichten. Die Sichten sind notwendig, um den Aufbau und die Struktur des Katalogs und eine Übersicht über alle vorhandenen Probleme und Prozessmuster vermitteln zu können. Prinzipiell können auch alle Sichten in einer Sicht (das entspräche dann der Darstellung des Gesamtkatalogs) vereinigt werden. Wie in diesem Beispiel jedoch deutlich wird, führt dies zu einer unübersichtlichen und damit unbrauchbaren Darstellung.

Wir haben so weit möglich von RUP-Disziplinen, -Workflowdetails und Aktivitäten den Namen für die Prozessmuster übernommen. Beispielweise haben wir von der Aktivität „Develop Vision“ das Prozessmuster „Develop Vision“ abgeleitet. Wenn wir die Namen nicht übernommen haben, so haben wir dies im Text angegeben. Wenn nicht anders angegeben haben wir für die Darstellung der Sichten die Notation aus Kapitel 6 übernommen.

Bezeichnung von Problemen und Prozessmustern

7.2.1 Sicht „Use-Beziehung“

Abbildung 7-2 zeigt den hierarchischen Aufbau des Katalogs CADs mit Hilfe der Use-Beziehung. Dieser hierarchische Aufbau wurde weitestgehend aus dem RUP übernommen (vgl. mit Abbildung 7-1). Das Prozessmuster „Reanalyse the Requirements“ (abgeleitet von der RUP-Disziplin „Requirements“) benutzt sechs Prozessmuster, nämlich „Refine the Problem“ (abgeleitet von RUP-Workflowdetail „Analyze the Problem“), „Understand Stakeholder Needs“, „Define the System“, „Manage the Scope of the System“, „Manage Changing Requirements“ und „Refine the System Definition“. In Abweichung zum RUP wurden für die Disziplin „Requirements“ zwei Prozessmuster („Reanalyse the Requirements“ und „Specify the Requirements“ (hier nicht weiter betrachtet)) geschaffen. Dies war notwendig, da das Workflowdetail implizit für zwei unterschiedliche Situationen zwei unterschiedliche Kontexte beschreibt. Diese unterschiedlichen Kontexte werden durch die beiden Prozessmuster abgebildet.

Hierarchischer Aufbau des Katalogs

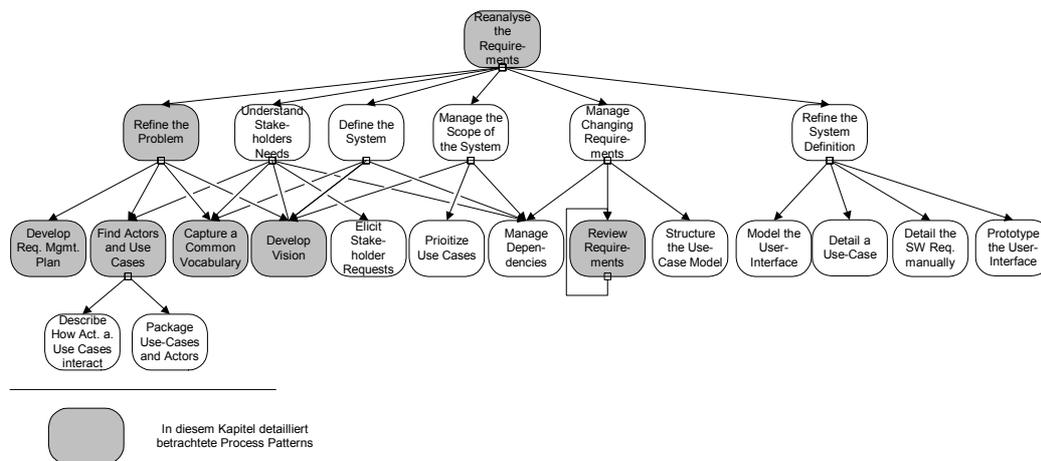


Abbildung 7-2: Katalogdiagramm – Sicht „Use-Beziehung“

Das Prozessmuster „Refine the Problem“ nutzt weitere Prozessmuster, nämlich „Develop Requirements Management Plan“, „Find Actors and Use Cases“, „Capture a Common Vocabulary“ und „Develop Vision“.

In dieser Sicht sind natürlich nicht alle Prozessmuster des Katalogs sichtbar. Dies gilt für Prozessmuster, die mit keinem anderen Prozessmuster in Beziehung stehen. Hierzu gehören die Prozessmuster „Capture Vocabulary by Project Team“, „Capture Vocabulary by System Analyst“, „Walkthrough Requirements“ und „Inspect Requirements“.

Ausblenden von Prozessmustern

Mehrfachnutzung, Rekursivität Anhand der Abbildung wird deutlich, dass einige Prozessmuster mehrfach genutzt werden, wie z.B. das Prozessmuster „Develop Vision“. Das Prozessmuster „Find Actors and Use Cases“ wiederum nutzt zwei weitere Prozessmuster, nämlich „Describe how Actors and Use-Cases interact“ und „Package Use-Cases and Actors“. Ferner betrachten wir die Prozessmuster „Review Requirements“, „Walkthrough Requirements“ und „Inspect Requirements“ genauer. Das Prozessmuster „Review Requirements“ steht mit sich selbst in einer (rekursiven) Use-Beziehung.

Der Leser beachte, dass wir in diesem Kapitel aus Platzgründen nur die grau eingefärbten Prozessmuster im Detail angeben. Mit diesen Prozessmustern können jedoch alle eingeführten Konzepte veranschaulicht werden.

Betrachteter Aspekt Die Darstellung der in diesem Kapitel zitierten oder detaillierter beschriebenen Prozessmuster beschränkt sich auf Prozessmuster, die dem Aspekt „Requirements Management“ angehören (verdeutlicht durch das Kürzel „RM“ rechts unten im Prozessmuskensymbol, s. auch Abschnitt 6.1.1). Prozessmuster, die anderen Aspekten angehören (z.B. „Business Modeling, Kürzel „BM“), tauchen bei der Angabe der Prozessmusterbeziehungen auf (z.B. Prozessmuster „Find Business Actors and Use-Cases“ in Abbildung 7-3).

7.2.2 Sicht „Sequence-Beziehung“

Abbildung 7-3 zeigt, ob und mit wem die betrachteten (d.h. die grau eingefärbten) Prozessmuster in einer Sequence-Beziehung stehen. Die Sequence-Beziehung bedeutet die mögliche direkte Abfolge von einem oder mehreren Vorgängermustern und einem Nachfolgemuster. Der Muster-Anwender erfährt über diese Sicht, welche Prozessmuster er direkt vor oder nach einem bestimmten Prozessmuster anwenden kann. Z.B. kann nach Anwendung der Prozessmuster „Find Business Workers and Entities“, „Find Business Actors and Use-Cases“, „Maintain Business Rules“, „Develop Business Case“, „Develop Use-Case Modeling Guidelines“ das Prozessmuster „Refine the Problem“ angewendet werden. Diese Information ist im RUP nur implizit vorhanden (Abschnitt 2.4.2).

Die Darstellung der Sequence-Beziehung ist ein Vorteil gegenüber der Darstellung des RUPs, in der nur jeweils die Input- und die Outputartefakte eines Workflowdetails bzw. einer Aktivität angegeben werden, aber nicht mögliche Reihenfolgen. Gerade weil im RUP die Aktivitäten, Workflowdetails und Disziplinen eng miteinander verzahnt sind, ist eine solche informative Darstellung hilfreich.

Die Anzahl der Sequence-Beziehungen mit nur einem Vorgängermuster ist eher gering. Dies liegt darin begründet, dass der RUP in neun verschiedene Disziplinen aufgeteilt ist, aber meistens Prozesse (d.h. Workflowdetails oder Aktivitäten) mehrerer Disziplinen zusammenwirken, um ein Objekt oder ein Ereignis zu produzieren. Wegen dieser engen Verknüpfung der Prozesse ist das Vorkommen von Sequence-Beziehungen mit mehr als einem Vorgängermuster viel höher.

Auf die einzelnen Vorkommen der Sequence-Beziehung wird in den Abschnitten eingegangen, in denen die jeweiligen Prozessmuster beschrieben werden.

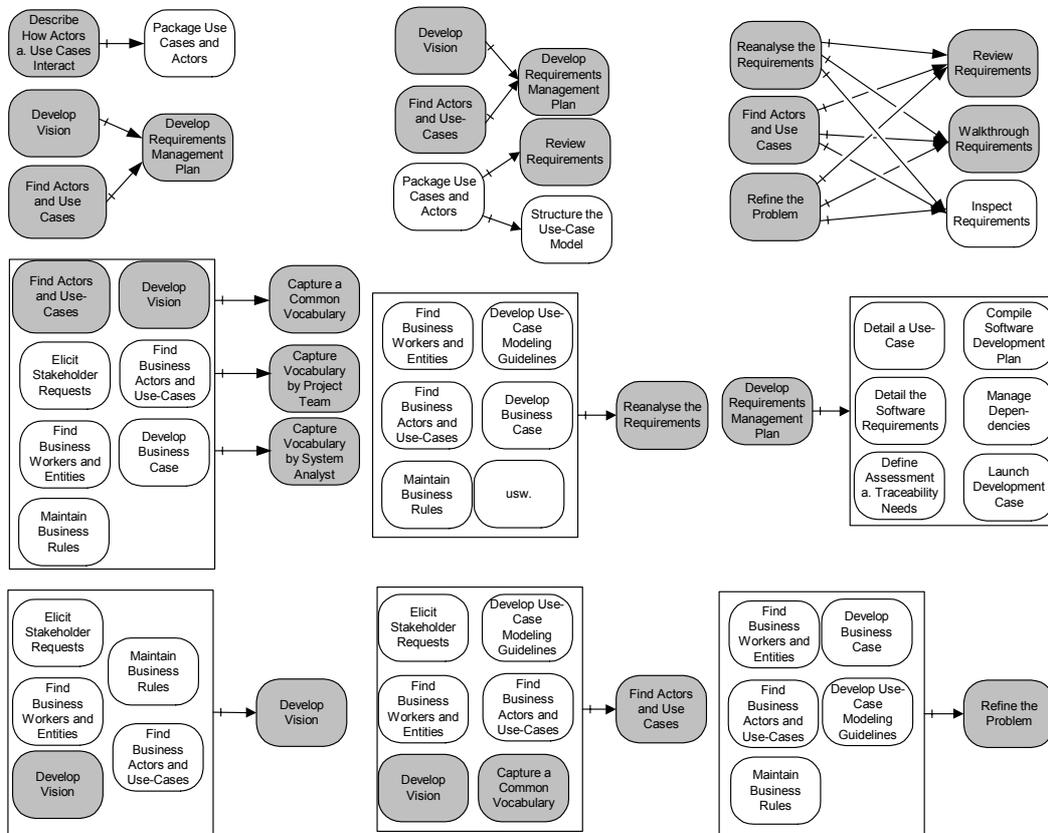


Abbildung 7-3: Katalogdiagramm – Sicht „Sequence-Beziehung“

7.2.3 Sicht „Refinement-Beziehung“

Abbildung 7-4 zeigt in einer Übersicht alle Vorkommen der Refinement-Beziehung. Z.B. existiert zu dem Prozessmuster „Capture a Common Vocabulary“ die von uns hinzugefügten Prozessmuster „Vocabulary by Project Team“ und „Vocabulary by System Analyst“. Diese beiden Prozessmuster beschreiben detaillierter als das abstrakte Muster, wie ein Projektglossar erstellt werden kann. Detaillierter sind hierbei jeweils Kontext und Prozess der verfeinernden Prozessmuster. Der Anwender kann sich also bei Vorliegen einer Refinement-Beziehung entscheiden, wieviel Unterstützung (Supermuster – weniger Unterstützung, Submuster – mehr Unterstützung) er zur Lösung des Problems benötigt. Der RUP bietet keine Möglichkeiten zur Beschreibung von Prozessverfeinerungen an.

Auf die einzelnen Vorkommen der Refinement-Beziehung wird in den Abschnitten eingegangen, in denen die jeweiligen Prozessmuster beschrieben werden.

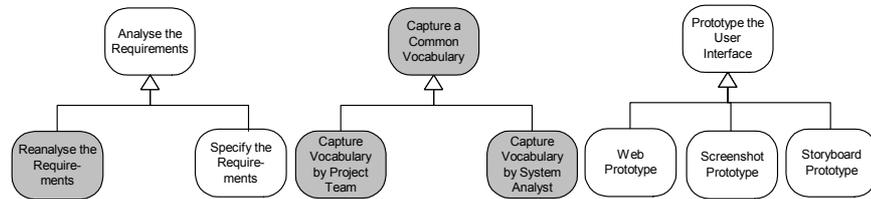


Abbildung 7-4: Katalogdiagramm – Sicht „Refinement-Beziehung“

7.2.4 Sicht „Processvariance-Beziehung“

Abbildung 7-5 zeigt in einer Übersicht alle Vorkommen der Processvariance-Beziehung. Prozessvariante Muster lösen jeweils das gleiche Problem. Hierzu gehören beispielsweise die drei Prozessmuster „Review Requirements“, „Walkthrough Requirements“ und „Inspect Requirements“. Diese drei Prozessmuster zeigen, wie auf unterschiedliche Art eine manuelle Prüfung durchgeführt werden kann. Der Anwender kann sich also bei Vorliegen einer Processvariance-Beziehung entscheiden, wie er das Problem lösen möchte. Auf variante Prozesse wird im RUP nicht eingegangen. Dies ist ein Nachteil für den Anwender, da er somit stets auf ein bestimmtes Vorgehen festgelegt ist. Gegebenenfalls kennt er auch eine variante Vorgehensweise, die er der Prozessbeschreibung des RUPs gerne hinzufügen würde. Dies ist jedoch nicht möglich. Die Beispiele für Prozessvarianten sind daher jeweils von uns hinzugefügt worden.

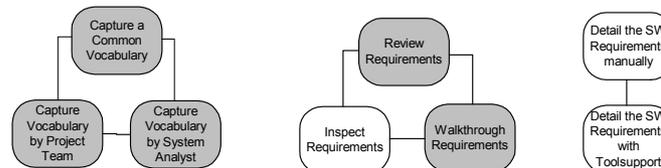


Abbildung 7-5: Katalogdiagramm – Sicht „Processvariance-Beziehung“

Auf die einzelnen Vorkommen der Processvariance-Beziehung wird in den Abschnitten eingegangen, in denen die jeweiligen Prozessmuster beschrieben werden.

7.3 Objekt- und Ereignisstruktur

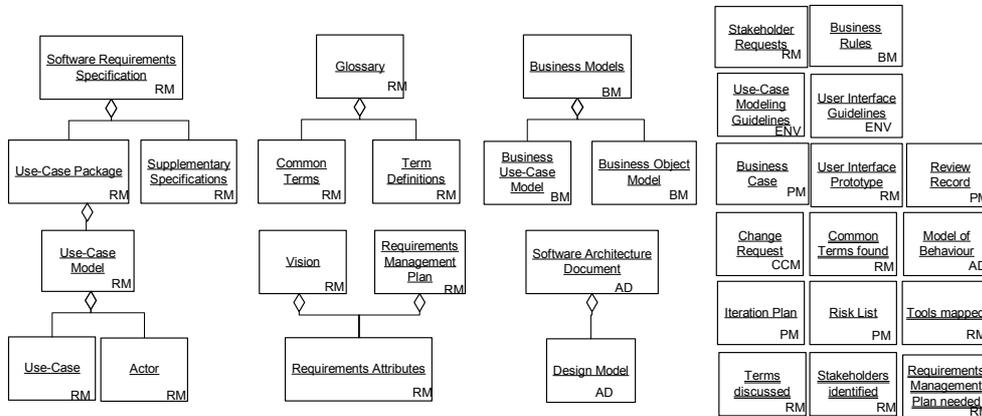


Abbildung 7-6: Objekt- und Ereignisstruktur – Aggregation

In diesem Abschnitt wird die Objekt- und Ereignisstruktur, die der Prozessmusterkatalog CADs benutzt, erläutert. Die Objekt- und Ereignisstruktur führt alle Objekte und Ereignisse auf, die von den Prozessmustern des Katalogs konsumiert oder produziert werden.

Darstellung aller Objekte und Ereignisse

Neben der Darstellung aller Objekte und Ereignisse wird die Aggregation von Objekten und Ereignissen dargestellt. Durch die Aggregation kann die Übersichtlichkeit erhöht werden, da in einem Prozessmuster statt einer Menge von Objekten oder Ereignissen auch das aggregierte Objekt/Ereignis angegeben werden kann. Durch die Objekt- und Ereignisaggregation werden mehr Prozessmusterbeziehungen ermöglicht und dadurch eine größere Flexibilität bei der Modellierung erlaubt (Abschnitt 3.3.5).

Aggregation

Die Objekt- und Ereignisaggregation wird häufig im Rahmen der Refinement-Beziehung verwendet. Im Rahmen der Verfeinerung eines Prozessmusters kann nicht nur der Prozess, sondern auch der Kontext des Prozessmusters verfeinert, d.h. aufgespalten werden. Beispielsweise enthält der initiale Kontext des Prozessmusters „Capture a Common Vocabulary“ das Aggregat „Requirements Documents“, das verfeinernde Muster „Vocabulary by Project Team“ enthält dagegen im initialen Kontext die Aggregatbestandteile. Die Aggregation kann aber auch in den anderen Prozessmusterbeziehungen berücksichtigt werden. Produziert beispielsweise ein Vorgängermuster alle Aggregatbestandteile eines Aggregat, welches ein Nachfolgermuster zur Ausführung benötigt, so können diese beiden Prozessmuster miteinander in einer Sequence-Beziehung stehen.

Ferner können Objekte und Ereignisse spezialisiert bzw. generalisiert werden (Abbildung 7-7). Statt der Aufspaltung eines Objekt bzw. eines Ereignisses in seine Bestandteile kann das Objekt bzw. Ereignis vor seiner Anwendung durch ein konkreteres ersetzt werden. Auf diese Weise kann ein Prozessmuster häufiger eingesetzt werden. Beispielsweise kann das Prozessmuster „Review Requirements“ für jedes „Review Object“ angewendet werden, also z.B. für „Glossary“, „Use-Case Model“ usw.

Generalisierung

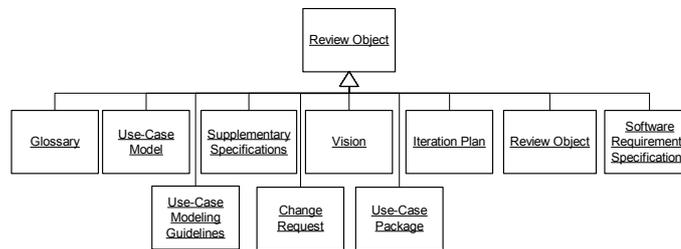


Abbildung 7-7: Objekt- und Ereignisstruktur – Generalisierung

7.4 Probleme und Prozessmuster

Im folgenden beschreiben wir die ausgewählten Probleme und Prozessmuster mit den in dieser Arbeit entwickelten Konzepten. Zunächst werden jeweils die Probleme beschrieben. Für jedes Problem werden dann die lösenden Prozessmuster angegeben. Im folgenden Abschnitt beschreiben wir ein Problem und ein Prozessmuster ausführlich. In den darauffolgenden Abschnitten überlassen wir diese ausführliche Interpretation dem Leser.

7.4.1 Problem „How to elicit the Requirements for an existing system?“

Das Requirements Management hat zur Aufgabe, die Anforderungen der Kunden und aller weiteren Interessenbeteiligten an das (weiter) zu entwickelnde System ermitteln. Diese Aufgabe wird durch das Problem „How to elicit the Requirements for an existing system?“ repräsentiert (Abbildung 7-8). Dieses Problem haben wir von der RUP-Disziplin „Requirements“ abgeleitet (Abbildung 2-14). Zu Beginn des Disziplinworkflows wurde eine Entscheidung modelliert. Da diese Entscheidung unterschiedliche Problemkontexte erfordert, haben wir zwei Probleme modelliert, nämlich „How to elicit the Requirements for an existing system?“ (dieser Abschnitt) und „How to elicit the Requirements for a new system?“ (hier nicht weiter betrachtet). Um zu zeigen, dass die beiden Probleme den gleichen Ursprung haben, haben wir die beiden Probleme als Verfeinerung des abstrakten Problems „How to analyse the Requirements“ modelliert (Abbildung 7-9).

Problemstellung

Die Problemstellung wird charakterisiert durch den initialen Problemkontext (linker Kasten), aus denen u.a. die Anforderungen abgeleitet werden können. Diese Anforderungen sollen u.a. in Gestalt der „Vision“, des „Use-Case-Models“ und des „User Interface Prototypes“ festgehalten werden (resultierender Kontext, rechter Kasten). Der Leser beachte, dass das Objekt „Business Models“ ein Aggregat ist, das die beiden Objekte „Business Use-Case Model“ und „Business Object Model“ ersetzt. Das Problem wird durch das Prozessmuster „Reanalyse the Requirements“ gelöst (Abbildung 7-10).

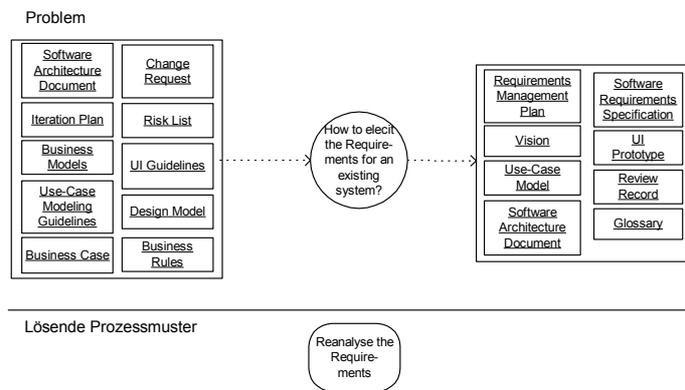


Abbildung 7-8: Problemdiagramm „How to elicit the Requirements for an existing system?“

Das Problem „How to elicit the Requirements for an existing system?“ ist eine Verfeinerung des Problems „How to analyse the Requirements?“ (Abbildung 7-9, RefineProblem-Diagramm), welches eine zweite Verfeinerung „How to elicit the Requirements for a new system?“ besitzt.

Verfeinerung

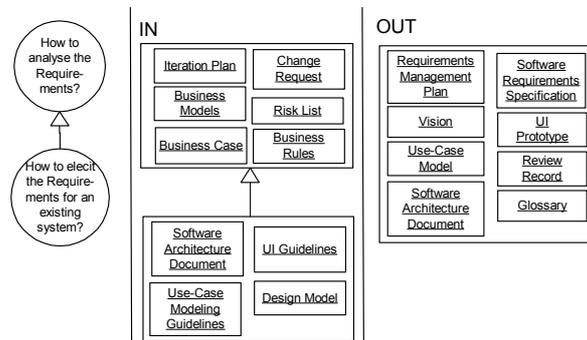


Abbildung 7-9: Problemverfeinerungsdiagramm „How to analyse the Requirements?“

Abbildung 7-10 zeigt das Prozessmusterdiagramm des Prozessmusters „Reanalyse the Requirements“. Das Prozessmuster löst das Problem „How to elicit the Requirements for an existing system?“. Das Prozessmuster wurde von der RUP-Disziplin „Requirements“ abgeleitet (Abbildung 2-14). Man beachte, dass das Prozessmuster „Reanalyse the Requirements“ von der Definition der RUP-Disziplin „Requirements“ abweicht. Die Abweichung liegt darin begründet, dass der Workflow zu Beginn eine Entscheidung aufzeigt. Da je nach Entscheidung ein unterschiedlicher initialer Kontext für das Prozessmuster gilt, der Kontext aber immer eindeutig sein muss, haben wir für die Disziplin zwei Prozessmuster definiert, ein Prozessmuster zur Analyse existierender Systeme und eines zur Analyse neuer Systeme.

Prozessmuster
„Reanalyse the Requirements“

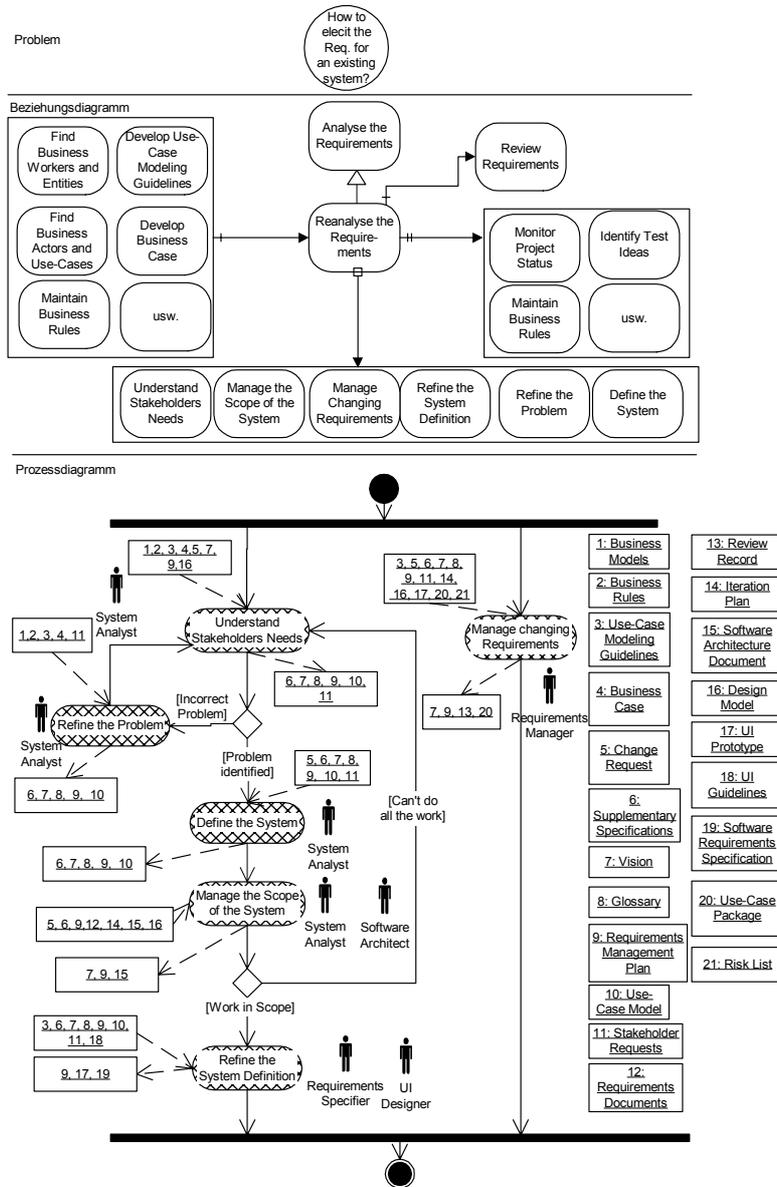


Abbildung 7-10: Prozessmusterdiagramm „Reanalyse the Requirements“

Vorausgehende
Muster

Wie das Problemdiagramm (Abbildung 7-8) zeigt, besteht der initiale Kontext des Prozessmusters aus den Objekten „Software Architecture Document“, „Change Request“, „Iteration Plan“, „Risk List“, „Business Models“, „UI Guidelines“, „Use-Case Modeling Guidelines“, „Design Model“, „Business Rules“ und „Business Case“. Damit das Prozessmuster „Reanalyse the Requirements“ diese Objekte konsumieren kann, müssen diese vorher erzeugt werden. Für die Erzeugung der Objekte steht eine Vielzahl von Prozessmustern zur Verfügung, darunter „Find Business Workers and Entities“, „Find Business Actors and Use-Cases“, „Maintain Business Rules“, „Develop Use-Case Modeling Guidelines“ und „Develop Business Case“ (Beziehungsdiagramm, linker Kasten). Werden all diese Prozessmuster angewen-

det, kann anschließend das Prozessmuster „Reanalyse the Requirements“ angewendet werden. Aus diesem Grund steht das Prozessmuster als Nachfolgemuster in einer Sequence-Beziehung mit diesen Vorgängermustern.

Der resultierende Kontext des Prozessmusters enthält die Objekte „Vision“ und „Software Requirements Specification“, „UI Prototype“, „Requirements Management Plan“, „Use-Case Model“, „Review Record“, „Glossary“ und „Software Architecture Document“. „Review Record“ ist einziges Element des initialen Kontexts des Prozessmusters „Review Requirements“. Aus diesem Grund gibt es eine Sequence-Beziehung zwischen den beiden Prozessmustern. Das Prozessmuster „Reanalyse Requirements“ ist ferner eines von mehreren Vorgängermustern jeweils für die Prozessmuster „Monitor Project Status“, „Identify Test Ideas“ usw.

Nachfolgende
Muster

Das Prozessmuster nutzt ferner die Prozessmuster „Refine the Problem“, „Refine the Problem“, „Define the System“, „Understand Stakeholders Needs“, „Manage the Scope of the System“, „Manage Changing Requirements“ und „Refine the System Definition“ (Beziehungsdiagramm, unterer Kasten).

Genutzte Prozess-
muster

Ferner haben wir ergänzend zum RUP die Rolle „Requirements Manager“ eingeführt, die verantwortlich für die Ausführung des Prozesses des Prozessmusters „Reanalyse the Requirements“ ist.

Neue Rolle

7.4.2 Problem „How to refine the Problem?“

Im Rahmen der Anforderungsanalyse ist die Analyse des Problems, welches durch das zu entwickelnde System gelöst werden soll, zu verfeinern. Diese Aufgabe wird durch das Problem „How to refine the Problem?“ repräsentiert (Abbildung 7-11). Das Problem wird durch das Prozessmuster „Refine the Problem“ gelöst (Abbildung 7-12).

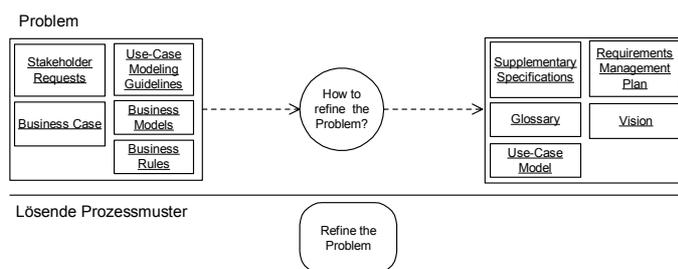


Abbildung 7-11: Problem „How to refine the Problem“

Prozessmuster
„Refine the
Problem“

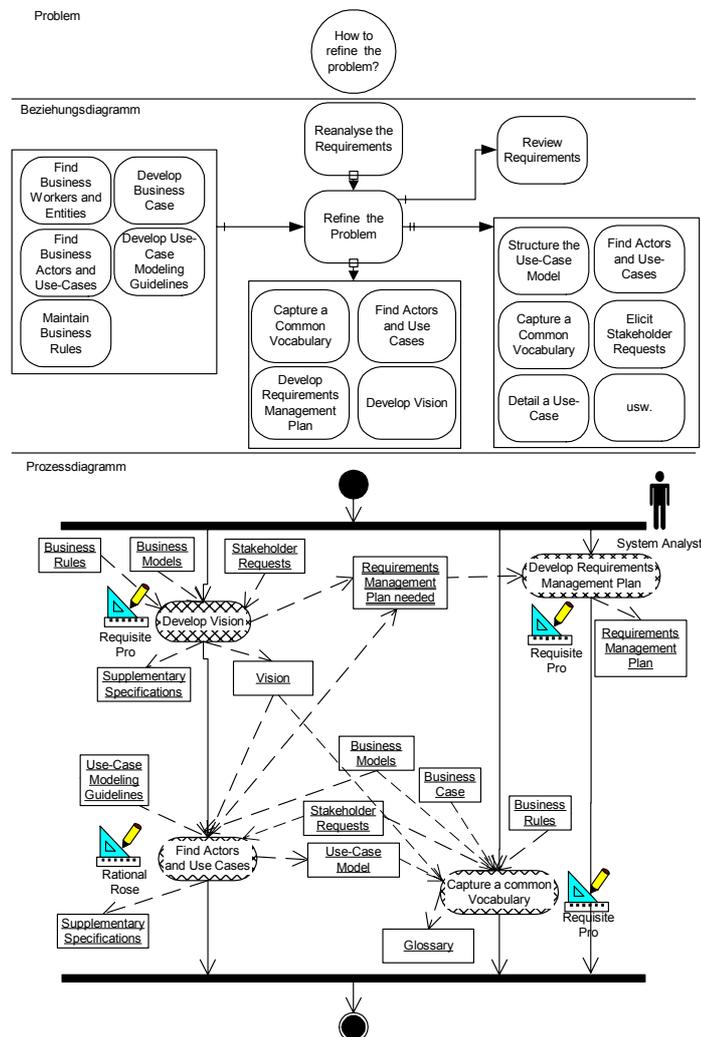


Abbildung 7-12: Prozessmusterdiagramm „Refine the Problem“

Abbildung 7-12 zeigt das Prozessdiagramm des Prozessmusters „Refine the Problem“. Das Prozessmuster löst das Problem „How to refine the Problem“.

7.4.3 Problem „How to manage the requirements?“

Im Rahmen des Requirements Management muss festgehalten werden, wie die Anforderungen dokumentiert, klassifiziert und nachverfolgt werden können. Diese Aufgabe wird durch das Problem „How to manage the Requirements?“ repräsentiert. Die Problemstellung wird charakterisiert durch das Ereignis „Requirements Management Plan needed“. Das Problem wird durch das Prozessmuster „Develop Requirements Management Plan“ gelöst. Ziel ist die Erstellung des Dokuments „Requirements Management Plan“.

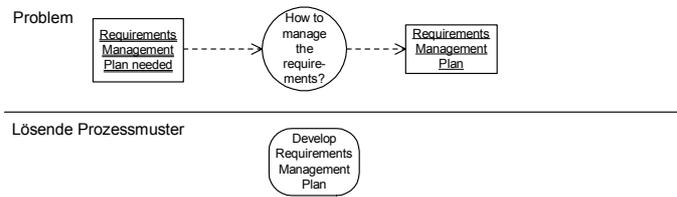


Abbildung 7-13: Problem „How to manage the requirements?“

Abbildung 7-14 zeigt das Prozessdiagramm des Prozessmusters „Develop Requirements Management Plan“. Das Prozessmuster löst das Problem „How to manage the requirements?“.

Prozessmuster „Develop Requirements Management Plan“

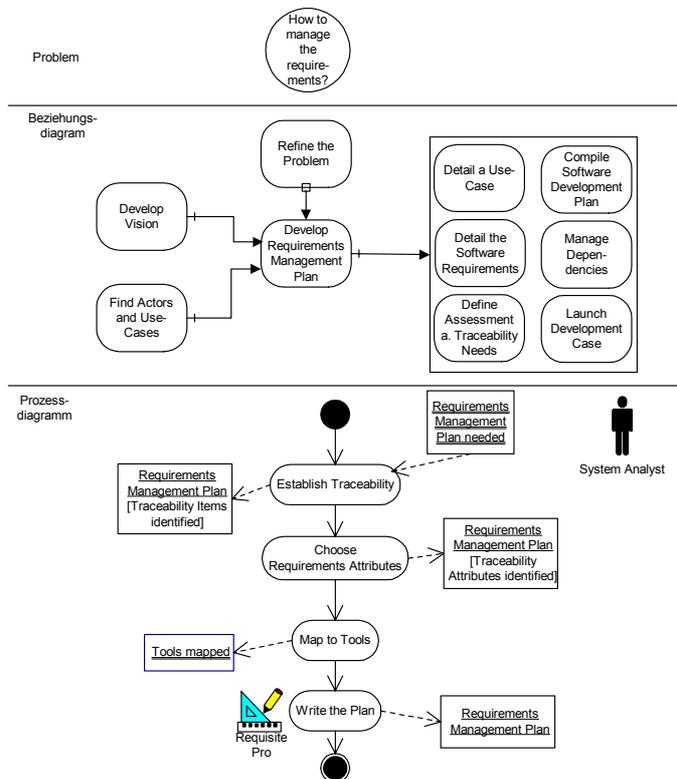


Abbildung 7-14: Prozessmusterdiagramm „Develop Requirements Management Plan“

7.4.4 Problem „How can Actors and Use-Cases be found?“

Für die Anforderungsermittlung kann die Use-Case Modellierungstechnik eingesetzt werden. In Gestalt von Use-Cases wird die Interaktion von Akteuren mit dem zu entwickelnden System beschrieben. Diese Aufgabe wird durch das Problem „How can Actors and Use-Cases be found?“ repräsentiert. Die Problemstellung wird charakterisiert durch eine Menge

von Objekten, aus denen die Akteure und die Use-Cases abgeleitet werden können. Das Problem wird durch Erstellung der Objekte „Use-Case Model“ und „Supplementary Specifications“ gelöst. Man beachte, dass wir abweichend vom RUP das Objekt „Glossary“ aus dem initialen Kontext entfernt haben, da sonst ein Zykel mit dem Problem „How can Vocabulary be produced?“ besteht. Glossary ist daher ein optionales Objekt und wird nicht im initialen Kontext aufgeführt. Man beachte ferner, dass wir abweichend vom RUP die Objekte „Use-Case und Akteur“ aus dem resultierenden Kontext entfernt haben, da diese redundant zum Objekt „Use-Case Model“ sind. Das Problem wird durch das Prozessmuster „Find Actors and Use-Cases“ gelöst.

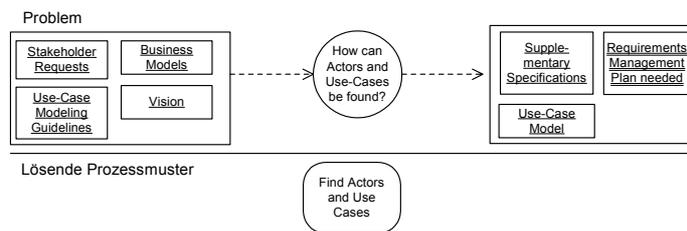


Abbildung 7-15: Problem „How can Actors and Use-Cases be found?“

Zu dem Problem existiert die Generalisierung „How to model the system behaviour?“.

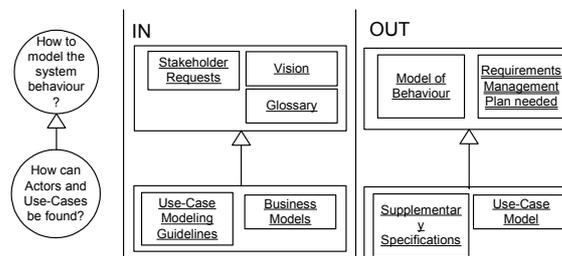


Abbildung 7-16: Verfeinerung des Problems „How to model the system behaviour?“

Prozessmuster „Find Actors and Use Cases“

Abbildung 7-17 zeigt das Prozessdiagramm des Prozessmusters „Find Actors and Use Cases“. Das Prozessmuster löst das Problem „How can Actors and Use Cases be found?“. Die entsprechende RUP-Aktivität kann in Abschnitt A.4.1 im Anhang nachgeschlagen werden.

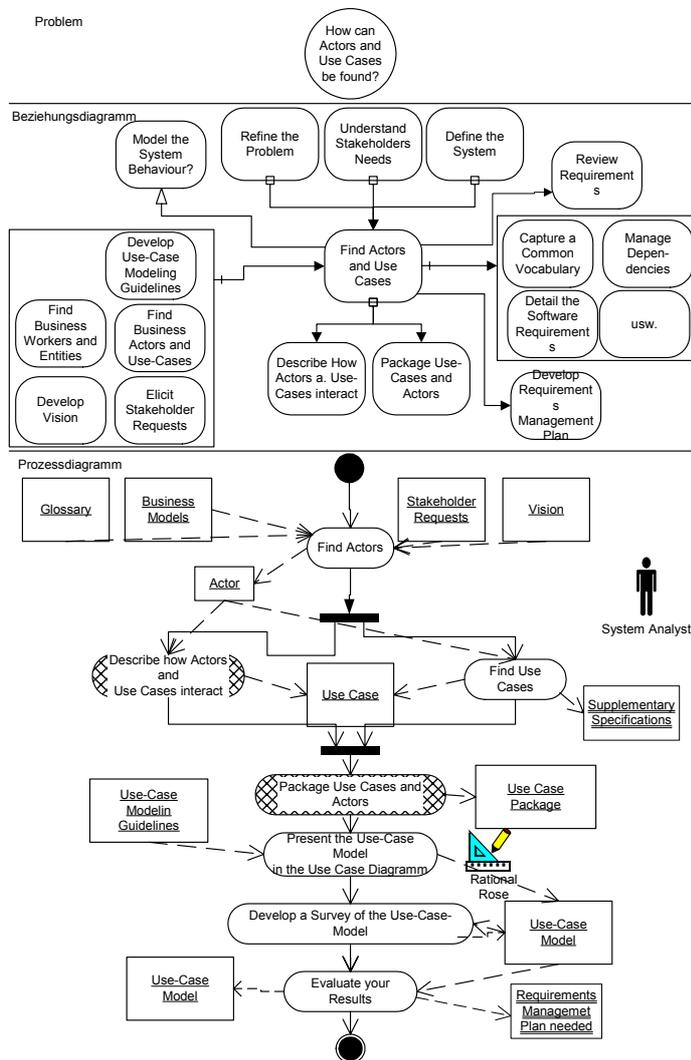


Abbildung 7-17: Prozessmusterdiagramm „Find Actors and Use Cases“

7.4.5 Problem „How can a Glossary be produced?“

Innerhalb eines Projekts sollte ein Projekt-Vokabular entwickelt werden, um die Kommunikation zwischen den Mitarbeitern zu verbessern und eindeutige Formulierungen in den Dokumenten zu erreichen. Diese Aufgabe wird durch das Problem „How can a Glossary be produced?“ repräsentiert.

Das Problem wird durch das Prozessmuster „Capture a Common Vocabulary“ (Abbildung 7-19) gelöst. Zu diesem Prozessmuster gibt es zwei Verfeinerungen, Prozessmuster „Vocabulary by System Analyst“ (Abbildung 7-22) und Prozessmuster „Vocabulary by Project

Team“ (Abbildung 7-23). Diese beiden Prozessmuster sind zugleich Prozessvarianten, da sie den gleichen Kontext besitzen. Der Anwender kann also zwischen diesen beiden Prozessmustern wählen.

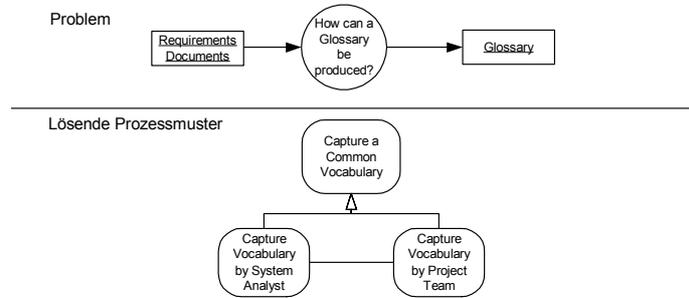


Abbildung 7-18: Problem „How can a Glossary be produced?“

Prozessmuster
„Capture a Common
Vocabulary“

Abbildung 7-19 zeigt das Prozessdiagramm des Prozessmusters „Capture a Common Vocabulary“. Das Prozessmuster löst das Problem „How can a Glossary be produced?“ (Abschnitt 7.4.5).

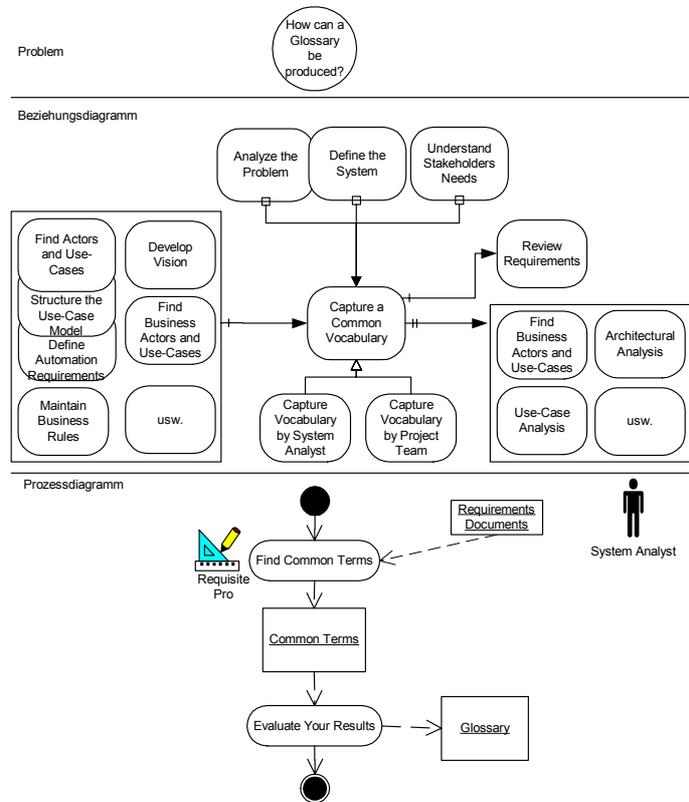


Abbildung 7-19: Prozessmusterdiagramm „Capture a Common Vocabulary“

7.4.6 Problem „How can a Glossary be produced (detailed)?“

Das Problem „How can a Glossary be produced?“ (Abschnitt 7.4.5) besitzt eine Verfeinerung, nämlich das Problem „How can a Glossary be produced (detailed)?“. Das Problem wird durch die Prozessmuster „Vocabulary by System Analyst“ (Abbildung 7-22) und „Vocabulary by Project Team“ (Abbildung 7-23) gelöst.

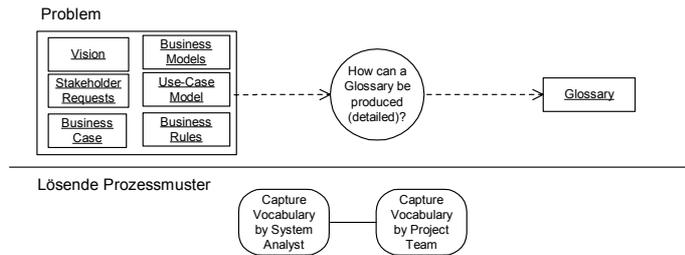


Abbildung 7-20: Problem „How can a Glossary be produced (detailed)?“

Abbildung 7-21 zeigt, wie der Kontext des Problems „How can a Glossary be produced?“ verfeinert wird.

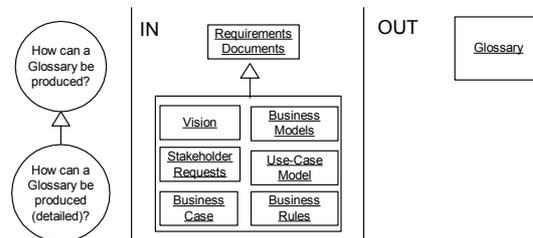


Abbildung 7-21: Verfeinerung des Problems „How can a Glossary be produced?“

Abbildung 7-22 zeigt das Prozessdiagramm des Prozessmusters „Capture a Vocabulary by System Analyst“. Das Prozessmuster löst das Problem „How can a Glossary be produced (detailed)?“ (Abschnitt 7.4.6).

Prozessmuster „Capture Vocabulary by System Analyst“

Der resultierende Kontext des Prozessmusters enthält wie auch das Supermuster das Objekt „Glossary“. Das Prozessmuster „Capture a Vocabulary by System Analyst“ steht daher mit den gleichen Prozessmustern in einer Sequence-Beziehung wie das Supermuster „Capture a Common Vocabulary“.

Zu dem Prozessmuster „Capture a Vocabulary by System Analyst“ existiert ferner eine Prozessvariante, nämlich „Capture a Vocabulary by Project Team“. Dies bedeutet, dass die Probleme und die Kontexte der beiden Prozessmuster identisch sind (Abbildung 7-22 und Abbildung).

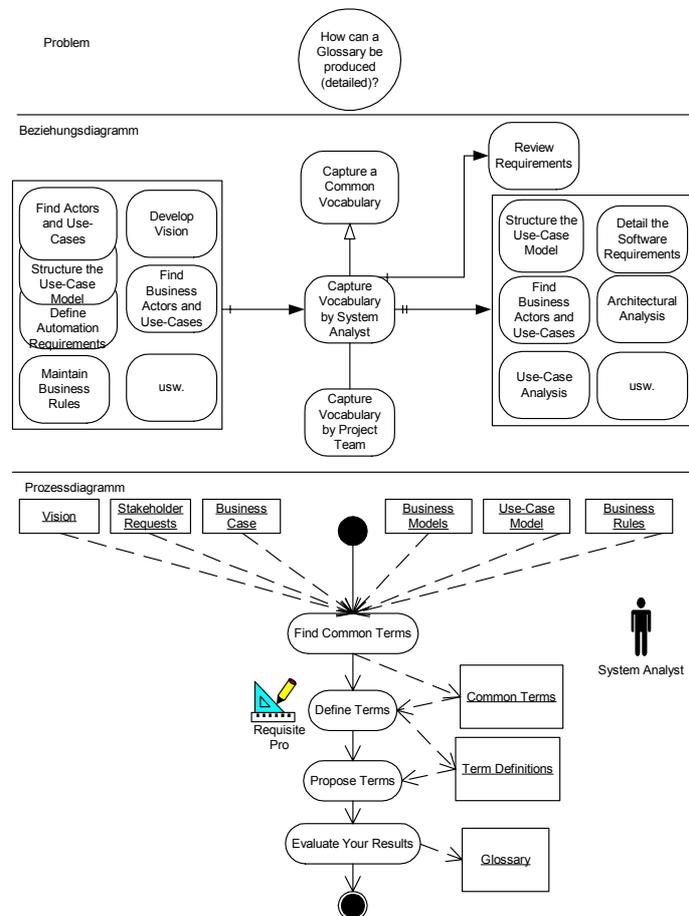


Abbildung 7-22: Prozessmusterdiagramm „Capture Vocabulary by System Analyst“

Prozessmuster
„Capture Vocabulary
by Project Team“

Abbildung zeigt das Prozessdiagramm des Prozessmusters „Capture a Vocabulary by Project Team“. Das Prozessmuster löst das Problem „How can a Glossary be produced (detailed)?“. Dieses Problem ist identisch mit dem Problem des Prozessmusters „Capture a Vocabulary by System Analyst“. Diese Übereinstimmung muss gegeben sein, da die beiden Prozessmuster Prozessvarianten sind. Für das Prozessmuster „Capture a Vocabulary by Project Team“ gelten daher die gleichen Aussagen wie für die Prozessmuster „Capture a Vocabulary by System Analyst“ und „Capture a common Vocabulary“.

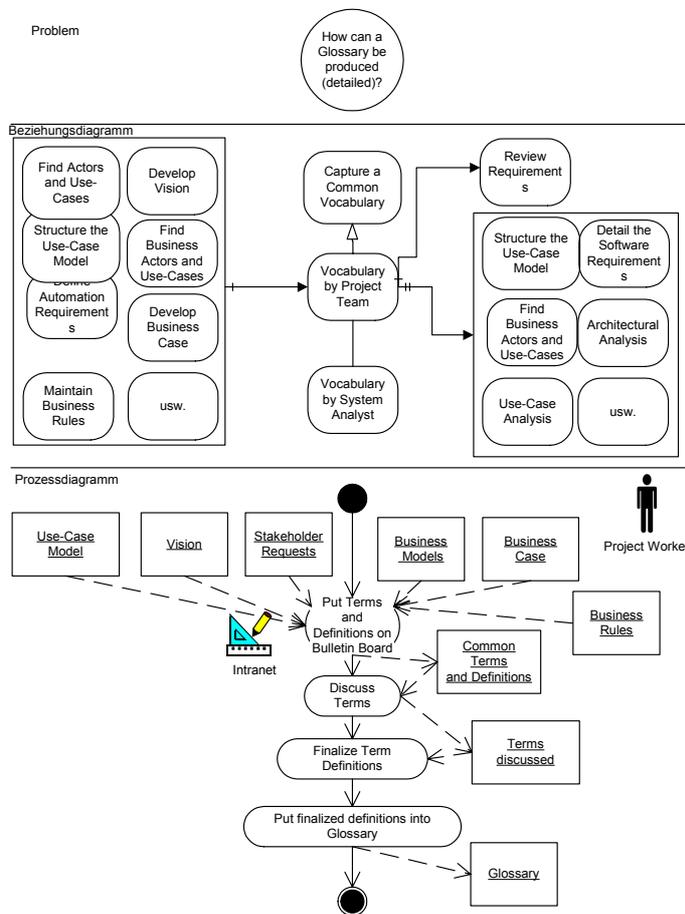


Abbildung 7-23: Prozessmusterdiagramm „Capture Vocabulary by Project Team“

7.4.7 Problem „How to envision the system?“

Für die Anforderungsanalyse muss zunächst ermittelt werden, welches Problem von dem zu entwickelnden System gelöst werden soll, wer von dem System betroffen ist (Stakeholder) und welche Aufgaben (in einer groben Skizze) das System erfüllen soll. All dies wird in der Vision festgehalten. Diese Aufgabe wird durch das Problem „How to envision the system?“ repräsentiert.

Man beachte, dass wir abweichend vom RUP das Objekt „Vision“ aus dem initialen Problemkontext entfernt haben. Dadurch lässt man offen, ob bereits eine Vision erstellt wurde (in späteren Iterationen) oder noch nicht (in früheren Iterationen).

Das Problem wird durch das Prozessmuster „Develop Vision“ gelöst.

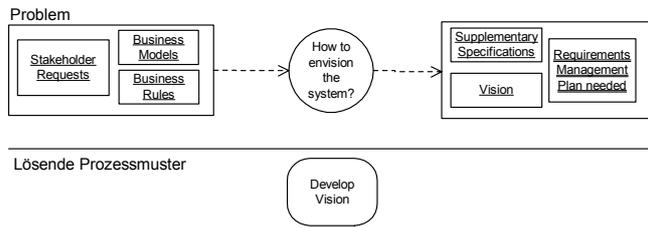


Abbildung 7-24: Problem „How to envision the system?“

Abbildung 7-25 zeigt das Prozessdiagramm des Prozessmusters „Develop Vision“.

Prozessmuster „Develop Vision“

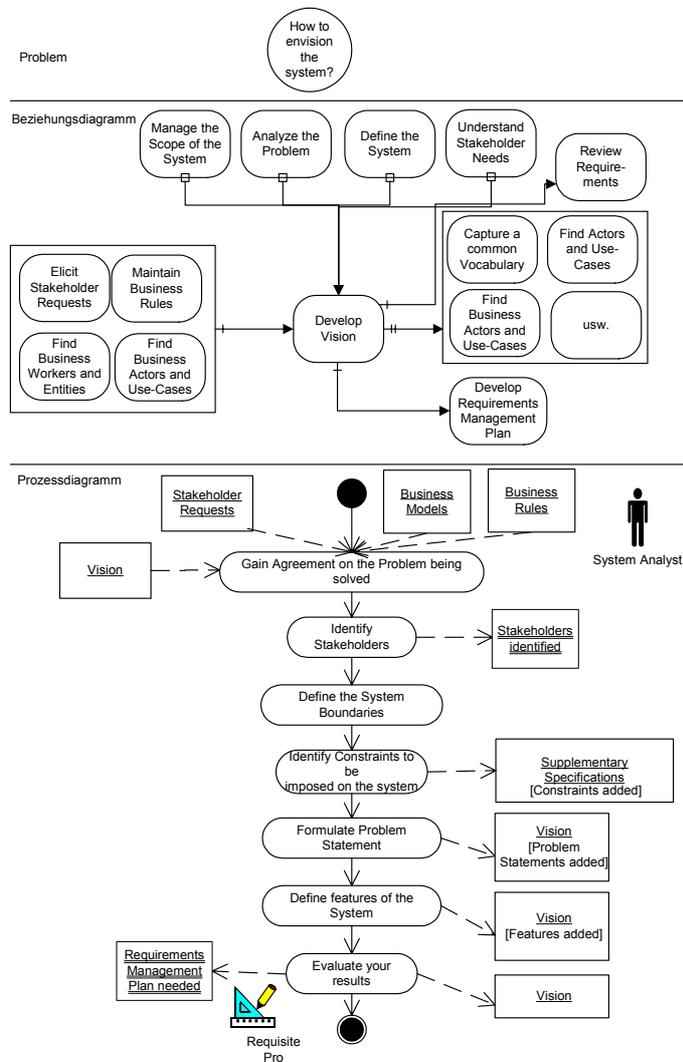


Abbildung 7-25: Prozessmusterdiagramm „Develop Vision“

Das Prozessmuster löst das Problem „How to envision the system?“. Die entsprechende RUP-Aktivität kann in Abschnitt A.4.1 im Anhang nachgeschlagen werden.

7.4.8 Problem „How to review Requirements?“

Jedes Dokument, in dem Anforderungen festgelegt werden, muss einem Review unterzogen werden. Diese Aufgabe wird durch das Problem „How to review the Requirements?“ repräsentiert. Der initiale Problemkontext besteht aus dem Objekt „Review Object“. Dieses Objekt ist eine Generalisierung von einer Menge von Objekten, die jeweils ein spezielles Review-Objekt darstellen, nämlich „Glossary“, „Use-Case Model“, „Supplementary Specifications“, „Vision“, „Iteration Plan“, „UI Prototype“, „Software Requirements Specification“, „Use-Case Modeling Guidelines“, „Change Request“ und „Use-Case Package“ (Abbildung 7-6).

Der Vorteil dieser Problemschreibung liegt darin, dass für den Anwender eindeutig ist, dass jeweils eines der Review-Objekte dem Review unterzogen wird. Aus der bisherigen RUP-Schreibweise geht dies nicht hervor; dort werden alle Review-Objekte im initialen Kontext der Aktivität genannt. Der Anwender muss also zunächst davon ausgehen, dass er alle Artefakte des initialen Kontexts benötigt.

Das Problem wird durch die prozessvarianten Prozessmuster „Review Requirements“ (Abbildung 7-27), „Inspect Requirements“ (Detailbetrachtung nicht vorhanden) und „Walkthrough Requirements“ (Abbildung 7-28) gelöst.

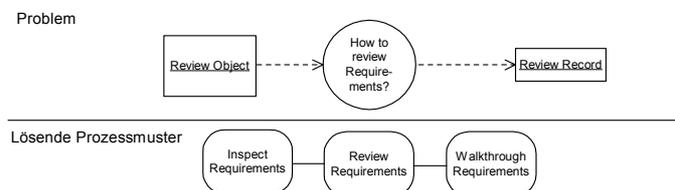


Abbildung 7-26: Problem „How to review Requirements?“

Abbildung 7-27 zeigt das Prozessdiagramm des Prozessmusters „Review Requirements“. Das Prozessmuster löst das Problem „How to review Requirements?“ (Abschnitt 7.4.8). Die entsprechende RUP-Aktivität kann in Abschnitt A.4.1 im Anhang nachgeschlagen werden. „Review Requirements“ steht mit sich selbst in einer rekursiven Use-Beziehung.

Prozessmuster
„Review
Requirements“

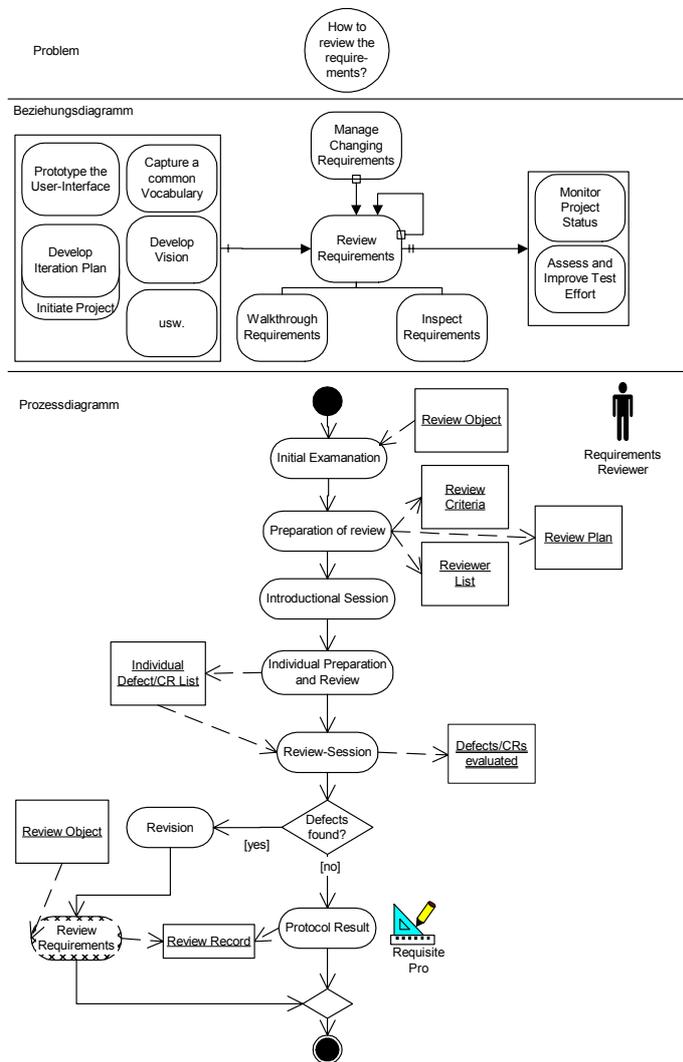


Abbildung 7-27: Prozessmusterdiagramm „Review Requirements“

Prozessmuster
„Walkthrough
Requirements“

Abbildung 7-28 zeigt das Prozessdiagramm des Prozessmusters „Walkthrough Requirements“. Das Prozessmuster löst das Problem „How to review Requirements?“ (Abschnitt 7.4.8). Es ist somit eine Prozessvariante zu dem Prozessmuster „Review Requirements“. Ein Walkthrough als Prüfungsmethode ist weniger formal als ein Review und besitzt deswegen weniger Aktivitäten.

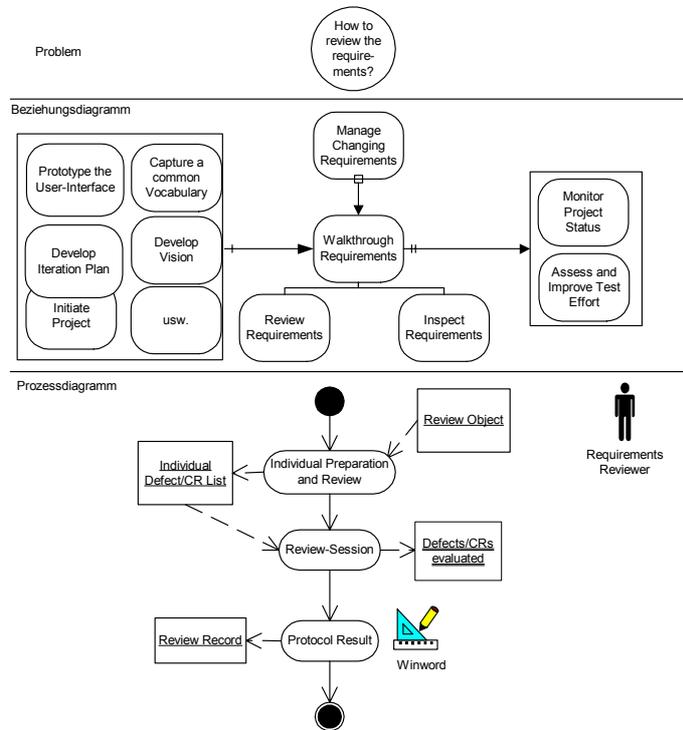


Abbildung 7-28: Prozessmusterdiagramm „Walkthrough Requirements“

7.5 Anwendung des Prozessmuskatalogs

Nachdem wir nun einen beispielhaften Prozessmuskatalog beschrieben haben, ist noch zu klären, wie ein Prozessmuskatalog im Rahmen von Projekten eingesetzt werden kann. Für die Anwendung gibt es zwei verschiedene Strategien, die Top-Down- und die From-Within-Strategie.

Bei der Top-Down-Strategie wählt man die Prozessmuster aus dem Prozessmuskatalog top-down aus. Dies bedeutet, dass man die Use-Sicht betrachtet und zunächst Prozessmuster auswählt, die ganz oben der Hierarchie angesiedelt sind. Damit ist schon ein Teil der Prozesse grob vorgegeben. Die Top-Down-Strategie ist daher bei eher großen Projekten einzusetzen, bei denen eine Festlegung der Prozesse vor Projektbeginn wichtig ist. Nach und nach können nun weitere Prozessmuster ausgewählt werden, die weiter unten in der Prozessmuskataloghierarchie angesiedelt sind. Parallel hierzu können Prozessmuster ausgewählt werden, die mit den bereits ausgewählten Prozessmustern über die drei Beziehungen „Sequence“, „Refinement“ und „Processvariance“ verknüpft sind. Die Top-Down-Strategie kann mit den Tailoring-Mechanismen – z.B. bekannt aus dem V-Modell 97⁹ – verglichen werden. Der Unterschied ist jedoch, dass das Streichen und Hinzufügen von Prozessmustern bzw. Prozes-

Top-Down-Strategie

9. Ausschreibungsrelevantes Tailoring für das Tailoring vor Projektbeginn und Technisches Tailoring für das Tailoring während des Projekts.

sen durch die formale Definition der Beziehungen vereinfacht wird. Wird im V-Modell ein Prozess weggelassen, so ist nur durch aufwändige Recherche ersichtlich, ob Konsistenzen zwischen Prozessen dadurch verletzt werden.

From-Within-
Strategie

Bei der From-Within-Strategie wählt man die Prozessmuster aus dem Prozessmusterkatalog bottom-up aus. Dies bedeutet, dass man zu Projektbeginn durch Auswahl eines abstrakten Prozessmusters keine grobe Richtung vorgibt wie bei der Top-Down-Strategie, sondern dass man je nach Bedarf im Projekt Prozessmuster zur Unterstützung auswählt. Dies bedeutet, dass man Prozessmuster zunächst unabhängig von ihrer hierarchischen Einordnung auswählt. Natürlich können nach Auswahl eines Prozessmusters genutzt, d.h. niedrig angesiedeltere Prozessmuster ausgewählt werden. Auf diese Weise ist eine flexible Prozessunterstützung möglich. Die From-Within-Strategie ist daher bei eher kleinen Projekten einzusetzen, bei denen eine Festlegung der Prozesse vor Projektbeginn nicht so wichtig ist wie eine flexible Prozessunterstützung und -anpassung. Auch größere Projekte können von dieser Strategie nutzen, indem man z.B. ein Standard-Vorgehensmodell einsetzt und bei Änderungen im Prozess Prozessmuster aus einem Katalog hinzugenommen werden können. Nach Durchführung der Prozessmuster kehrt man dann zum Standard-Vorgehensmodell zurück. Durch die Definition der Beziehungen ist bei der Bottom-Up-Strategie jederzeit leicht entscheidbar, welche weiteren Prozessmuster ausgewählt werden können.

Auswahl von
Prozessmustern und
Problemen im Detail

Die Auswahl von Prozessmustern und Problemen im Detail vollzieht sich in zwei Schritten. Zunächst wird über den Namen des Problems oder über den vorliegenden und den zu produzierenden Kontext ein passendes Problem ausgewählt. Anschließend wird – wenn vorhanden – eines der lösenden Prozessmuster ausgewählt.

7.6 Zusammenfassung

Mit den Mitteln der Process Pattern Description Language PROPEL haben wir einen Prozessmusterkatalog basierend auf Prozessen des Rational Unified Process beschrieben. Für die Modellierung haben wir einige wenige Tage benötigt. Der größte Teil des Modellierungsaufwands bestand allerdings darin, Inkonsistenzen der RUP-Prozesse als solche zu erkennen und zu entfernen. Hierdurch konnte gleichzeitig der Vorteil von PROPEL genutzt werden, die es ja Regeln für Prozesskonsistenzen vorgibt. Der Aufwand hat sich also aus zwei Gründen gelohnt: Zum einen wurde eine konsistente Prozessbeschreibung eines RUP-Ausschnitts erreicht, zum anderen konnte die Komplexität des Ursprungsmodells durch Aufteilung in Prozessmuster reduziert werden.

In den vorhergehenden Abschnitten haben wir bereits die Eigenschaften von PROPEL hervorgehoben und fassen sie an dieser Stelle zusammen.

Prozess

PROPEL besitzt alle notwendigen Mittel, um Prozesse und Prozessmuster zu beschreiben. Hierzu gehört die Beschreibung von Problemen, Prozessmustern, Objekten und Ereignissen, Rollen und Werkzeugen.

Prozessmusterbe-
ziehungen

Besonders hervorzuheben ist die Möglichkeit, Beziehungen zwischen Prozessmustern und damit zwischen Prozessen zu beschreiben. Die meisten Vorgehensmodelle besitzen keine Möglichkeit zur Darstellung von Prozess-Beziehungen. Die hierarchische Komposition von

Prozessen ist dort zwar implizit vorhanden. Für die Komposition existieren jedoch keine syntaktischen oder semantischen Definitionen. Auch Prozessabfolgen werden angegeben, allerdings ebenfalls ohne formale syntaktische oder semantische Definitionen.

Im RUP werden lediglich Disziplinen als Workflow dargestellt. In welcher Reihenfolge (RUP-)Aktivitäten auszuführen sind, ist aus der Beschreibung der Workflowdetails jedoch nicht ersichtlich. Die PROPEL Prozessmuster beinhalten dagegen stets eine Prozessbeschreibung in Gestalt eines Prozessdiagramms. Ein Ablauf der (RUP-)Aktivitäten in Form von Prozessen haben wir für jedes Prozessmuster ergänzt.

Modellierung von Prozessen

Prozessmodelle im RUP suggerieren oft eine nicht mögliche Prozessabfolge. Beispiel: Die Aktivitäten „Analyze the Problem“ und „Understand Stakeholder Needs“ können nur dann (Workflow der Disziplin „Requirements“) hintereinander ausgeführt werden, wenn gleichzeitig auch die Aktivität „Manage Change Requests“ ausgeführt wurde. Denn die Aktivität „Manage Change Requests“ erzeugt das Objekt „Change Request“, welches ein notwendiges Inputobjekt des Prozessmusters „Understand Stakeholder Needs“ ist. Mit Hilfe der Sequence-Beziehung von PROPEL können dagegen konkret die sequentielle Ausführbarkeit zweier oder mehrerer Prozessmuster angegeben werden.

Modellierung von Prozessabfolgen

Anhand der Use-Sichten des Katalogdiagramms wird deutlich, ob Prozessmuster mehrfach genutzt werden (z.B. das Prozessmuster „Develop Vision“ wird von vier Prozessmustern genutzt, s. Abbildung 7-2). Durch diese Darstellung wird die Modularisierung und die Wiederverwendung der Prozessmuster unterstützt. Im RUP werden Nutzungs-Beziehungen nur teilweise angegeben. Es werden lediglich pro Aktivität die nutzenden Workflowdetails angegeben.

Modularität

Wie viele andere Vorgehensmodelle auch unterscheidet der RUP nicht zwischen zusammengesetzten und atomaren Artefakten. Da die Komposition von Objekten und Ereignissen jedoch ein wichtiges Hilfsmittel für die Abstraktion von Prozessmustern ist, haben wir dieses Konzept PROPEL hinzugefügt.

Modellierung der Komposition von Objekten und Ereignissen

Mit PROPEL können Ereignisse modelliert werden (z.B. Abbildung 7-27, Ereignis „Defect/CRs evaluated“). Der RUP und auch andere Vorgehensmodelle besitzen nicht diese Möglichkeit.

Modellierung von Ereignissen

Der RUP verwendet optionale Artefakte (z.B. „Stakeholder Requests“, Abbildung 2-15). Bei optionalen Artefakten lassen sich jedoch keine eindeutigen Prozessrelationen identifizieren. Da aufgrund unserer Prozessmuster-Definition Kontexte stets nur benötigte Objekte und Elemente aufweisen, haben wir entweder die optionalen Artefakte in Muss-Artefakte umgewandelt oder haben weitere Prozessmuster erzeugt, welches die optionalen und die obligatorischen Artefakte in seinem Kontext beinhaltet (Prozessmuster „Reanalyze the Requirements“, Abbildung 7-10).

Vernachlässig optionaler Artefakte

PROPEL Prozessmuster kapseln den Prozess. Dies bedeutet, dass zunächst nur die Kontexte eines Prozessmusters via der Problembeschreibung bekannt ist. Zwischenergebnisse sind daher für die Prozessmuster Beschreibung irrelevant. Der RUP nennt jedoch Zwischenergebnisse im Aktivitätsinput und -output. Z.B. enthält der Abschnitt „Outputartefakte“ der Aktivität „Find Actors and Use Cases“ die Artefakte „Use Case“, „Actor“ und „Use-Case Model“. Da jedoch Ziel der Aktivität die Erstellung des Use-Case Modells ist, welches die Artefakte „Use Case“ und „Actor“ enthält, sind diese Artefakte als Zwischenergebnisse zu

Kapselung des Prozesses

betrachten. Diese Schreibweise kann zu Problemen führen, da manche Zwischenergebnisse nur im Rahmen der Prozessdurchführung benötigt werden. Führt man diese im Kontext auf, so hat diese gegebenenfalls unerwünschte Implikationen auf die Musterbeziehungen.

Freiheit vs.
Genauigkeit

Abschließend wollen wir noch darauf hinweisen, dass eine Prozessbeschreibung, wie sie vom RUP verwendet wird, dem Modellierer und dem Anwender mehr Freiheiten (z.B. optionale Artefakte) lässt, als es die Syntax und Semantik von PROPEL zulassen. Durch diese Freiheiten wachsen allerdings auch die Interpretationsmöglichkeiten. Die Anzahl der Interpretationsmöglichkeiten wollen wir jedoch gerade durch Angabe einer semiformalen Syntax und Semantik beschränken. Durch Verwendung des Prozessmuster-Ansatzes zielen wir jedoch darauf ab, dem Modellierer und Anwender durch Unterstützung flexibler Prozesse möglichst viele Freiheiten wieder zurückzugeben.

8 Die Process Pattern Workbench

In diesem Kapitel stellen wir die Process Pattern Workbench vor, welche die in dieser Arbeit vorgestellten Konzepte (Kapitel 3 bis Kapitel 6) realisiert. Die Process Pattern Workbench erläutern wir anhand des in Kapitel 7 präsentierten Prozessmuskatalogs CADS.

In Abschnitt 8.1 definieren wir die von der Process Pattern Workbench zu erfüllenden Anforderungen. In Abschnitt 8.2 stellen wir den Softwareentwurf vor. Anschließend erläutern wir in Abschnitt 8.3, wie die Process Pattern Workbench implementiert wurde und illustrieren das Ergebnis anhand zahlreicher Screenshots. Abschnitt 8.4 gibt einen Ausblick auf mögliche Ausbaustufen.

8.1 Anforderungsspezifikation

Im Rahmen der Anforderungsspezifikation definieren wir funktionale und nichtfunktionale Eigenschaften der Process Pattern Workbench. Die funktionalen Anforderungen definieren wir in Abschnitt 8.1.1 anhand von Use Cases. Die nicht-funktionalen Anforderungen definieren wir in Abschnitt 8.1.2.

8.1.1 Funktionale Anforderungen

Ein Use Case definiert einen spezifischen Teil des funktionalen Verhaltens. Dabei ist nur das Verhalten relevant, welches für einen Benutzer des Systems erkennbar ist. Die interne Struktur des System ist nicht relevant. Die identifizierten Use Cases haben wir verschiedenen Komponenten zugeordnet: Benutzerverwaltung, Katalogverwaltung, Problemdefinition, Patterndefinition, Reporting und Projektverwaltung.

Use Cases

Benutzer können in den Use Cases verschiedene Rollen annehmen. Die Rollen können den Abstraktionsschichten von PROPEL, nämlich der Metamodell-Ebene, der Modell-Ebene oder der Instanz-Ebene zugeordnet werden (Abbildung 4-1). Der Workbench Developer arbeitet auf der Metamodell-Ebene. Er modifiziert oder erweitert die Syntax und Semantik von PROPEL und übersetzt diese Änderungen in die Process Pattern Workbench. Diese Rolle wird im Verlauf des weiteren Kapitels nicht weiter betrachtet, da dessen Aufgaben nicht mit der Anwendung der Prozessmuster in Verknüpfung stehen. Der Pattern Designer arbeitet auf der Modell-Ebene. Er ist für die Definition und Modellierung von Prozessmustern und deren Pflege zuständig. Der Pattern User arbeitet auf der Instanz-Ebene. Er initiiert Projekte, sucht Muster aus Prozessmuskatalogen und instanziiert und verknüpft diese zu Prozessen innerhalb des Entwicklungsprojektes. Eine weitere Rolle, die sowohl der Modell-ebene als auch der Instanzebene zuzuordnen ist, ist der Workbench Administrator. Dieser administriert die Process Pattern Workbench.

Rollen

8.1.1.1 Benutzerverwaltung

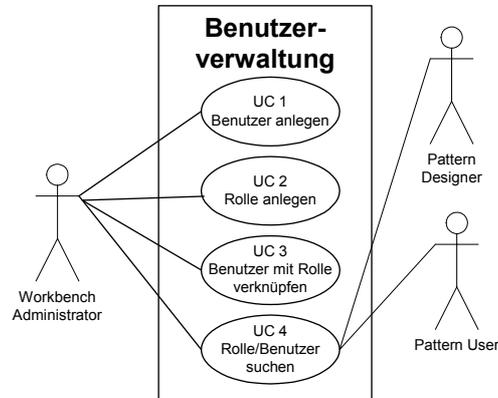


Abbildung 8-1: Use Cases der Benutzerverwaltung

Use Case 1 **Benutzer anlegen**

Kurzbeschreibung: Der Akteur legt den Benutzer an (Benutzername, Vor-, Nachname).
 Primärer Akteur: Workbench Administrator

Use Case 2 **Rolle anlegen**

Kurzbeschreibung: Der Akteur legt die Rolle an (Rollenname, Beschreibung).
 Primärer Akteur: Workbench Administrator

Use Case 3 **Benutzer mit Rolle verknüpfen**

Kurzbeschreibung: Der Akteur weist dem Benutzer eine Rolle zu.
 Primärer Akteur: Workbench Administrator

Rolle/Benutzer suchen

Use Case 4

Kurzbeschreibung: Der Akteur kann sich alle Rollen bzw. alle Benutzer anzeigen lassen. Alternativ kann er über einen regulären Ausdruck nach einer Rolle bzw. nach einem Benutzer suchen. Ferner kann sich der Akteur alle Rollen eines bestimmten Benutzers oder alle Benutzer mit einer bestimmten Rolle anzeigen lassen.

Primärer Akteur: Workbench Administrator, Pattern Designer, Pattern User

8.1.1.2 Problemdefinition

Die Use Cases der Problemdefinition beziehen sich auf die Erstellung und Modifikation von Problemen, Prozessmustern, Objekten und Ereignissen und Prozessmuskatalogen.

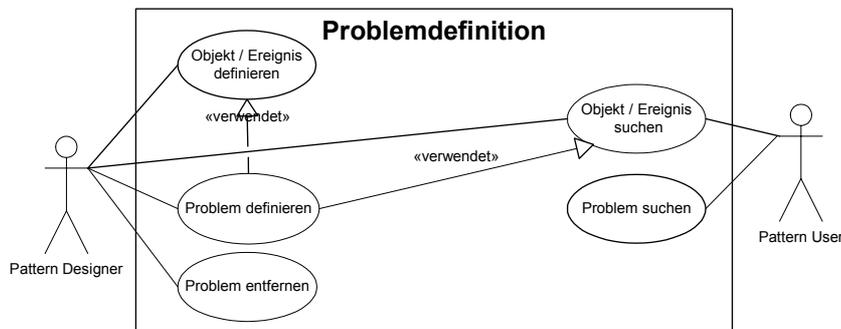


Abbildung 8-2: Use Cases der Problemdefinition

Objekt/Ereignis definieren

Use Case 5

Kurzbeschreibung: Der Akteur erzeugt ein neues Objekt oder Ereignis (eindeutiger Name), weist einen Aspekt zu und eine Beschreibung.

Primärer Akteur: Pattern Designer

Use Case 6 Problem definieren

- Kurzbeschreibung: Der Akteur definiert ein Problem, indem er den Problemkontext angibt. Hierbei handelt es sich um Objekte und Ereignisse, die
- für die Anwendung des Patterns notwendig sind (initialer Kontext) und
 - nach der Anwendung der Patterns vorhanden sind (resultierender Kontext)

Hierzu kann er aus einer Dropdownliste alle verfügbaren Objekte und Ereignisse auswählen. Zusätzlich kann er noch eine Beschreibung des Problems ergänzen. Das Problem erhält die Versionsnummer 1.

Primärer Akteur: Pattern Designer

Erweiterungen: Dieser Use Case verwendet die Use Cases UC 5 und UC 9.

Use Case 7 Problem suchen

- Kurzbeschreibung: Der Akteur kann ein Problem über den Namen suchen oder kann sich alle existierenden Probleme anzeigen lassen. Von der Ergebnisliste der Suche gelangt man zu der Detailansicht eines Problems.

Primärer Akteur: Pattern Designer, Pattern User

Use Case 8 Problem entfernen

- Kurzbeschreibung: Der Akteur entfernt ein Problem mitsamt seines Kontexts und allen Versionen. Das Problem kann nur dann entfernt werden, falls diesem Problem kein Prozessmuster zugewiesen wurde.

Primärer Akteur: Pattern Designer

Use Case 9 Objekt/Ereignis suchen

- Kurzbeschreibung: Der Akteur kann sich alle Objekte bzw. Ereignisse anzeigen lassen. Alternativ kann er sich alle Objekte bzw. Ereignisse eines Aspekts anzeigen lassen.

Primärer Akteur: Pattern Designer, Pattern User

8.1.1.3 Patterndefinition

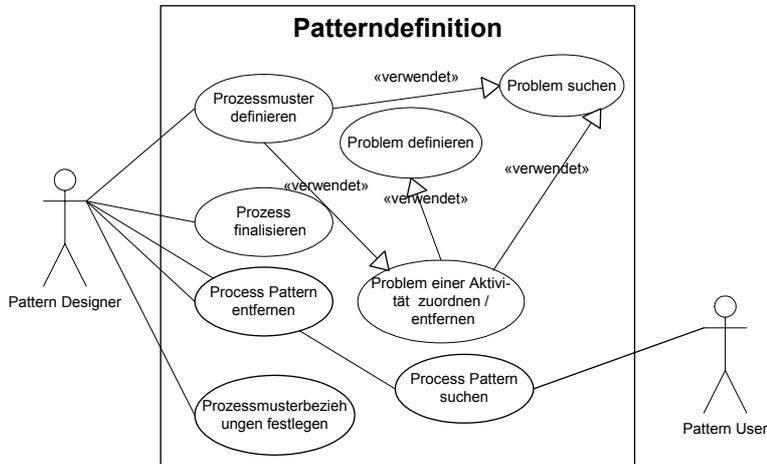


Abbildung 8-3: Patterndefinition

Prozessmuster definieren

Use Case 10

Kurzbeschreibung: Der Akteur definiert ein Prozessmuster durch Auswahl eines zu lösenden Problems. Der Problemkontext wird in den Musterkontext automatisch übernommen. Anschließend definiert der Akteur den Namen, eine Beschreibung sowie den Aspekt.

Der Akteur wählt die Funktion „Prozess editieren“ und gelangt in das entsprechende Prozessdiagramm. Dieses enthält bereits alle Objekte und Ereignisse des Patternkontexts. Der Akteur fügt die gewünschten Aktivitäten hinzu und gibt für jede der Aktivitäten einen Kontext an.

Primärer Akteur: Pattern Designer

Erweiterungen: Diese Use Case verwendet die Use Cases „Problem suchen“ und „Problem einer Aktivität zuordnen/entfernen“.

Prozess finalisieren

Use Case 11

Kurzbeschreibung: Solange die Version eines Prozesses noch nicht finalisiert ist, gilt die vorherige Version als die aktuelle. Dies bedeutet, dass noch nicht finalisierte Versionen für den Pattern User nicht sichtbar sind. Finalisiert der Akteur die Arbeitsversion, so wird sie zur aktuellen Version im Prozessmuskatalog.

Primärer Akteur: Pattern Designer

Use Case 12 Problem einer Aktivität zuordnen/entfernen

Kurzbeschreibung: Der Akteur weist eine Aktivität im Prozessdiagramm ein Problem zu. Hierzu kann er entweder aus den bestehenden Problemen auswählen oder ein neues Problem erstellen.

Primärer Akteur: Pattern Designer

Erweiterungen: Diese Use Case verwendet die Use Cases Problem suchen und Problem definieren.

Use Case 13 Prozessmuster suchen

Kurzbeschreibung: Der Akteur kann ein Prozessmuster suchen, indem er sich die Liste aller Prozessmuster anzeigen lässt, oder indem er sich alle Prozessmuster eines Aspekts anzeigen lässt oder indem er per regulärem Ausdruck nach dem Namen des Prozessmusters sucht.

Primärer Akteur: Pattern Designer, Pattern User

Use Case 14 Prozessmuster entfernen

Kurzbeschreibung: Der Akteur entfernt ein Prozessmuster mitsamt seines Kontexts und des Prozesses und allen Versionen. Wird das Prozessmuster gelöscht, werden auch alle seine Beziehungen aus dem Prozessmusterkatalog gelöscht.

Primärer Akteur: Pattern Designer

Use Case 15 Prozessmusterbeziehungen festlegen

Kurzbeschreibung: Der Akteur wählt aus, welche Prozessmusterbeziehung er festlegen möchte. In zwei Auswahllisten kann der Akteur jeweils diejenigen Prozessmuster auswählen, die über die Beziehung verknüpft werden sollen. Anschließend findet eine automatische Überprüfung durch das System statt, ob die Beziehung bzgl. der syntaktischen Regeln erlaubt ist. Falls ja, wird die Beziehung angelegt, falls nein, wird die Beziehung nicht angelegt und es erscheint eine Fehlermeldung.

Primärer Akteur: Pattern Designer

8.1.1.4 Projektverwaltung

Die Anwendungsfälle der Projektverwaltung befassen sich mit der Instanziierung der durch die Prozessmuster definierten Modellprozesse. Dies bedeutet, dass für ein konkretes Entwicklungsprojekt dokumentiert wird, welche Prozessmuster ausgewählt und angewendet wurden.

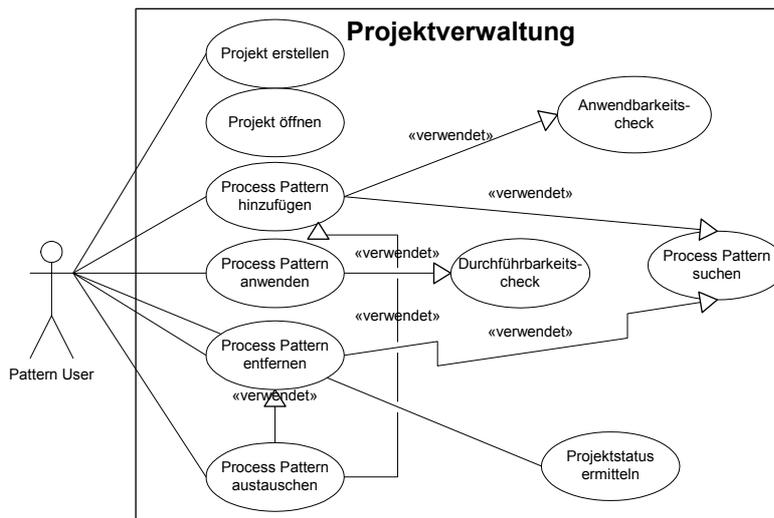


Abbildung 8-4: Anwendungsfälle der Projektverwaltung

Projekt erstellen

Use Case 16

Kurzbeschreibung: Der Akteur legt für ein Entwicklungsprojekt ein Workbenchprojekt an. Dieses Workbenchprojekt besitzt initial keine Prozessmuster.

Primärer Akteur: Pattern User

Projekt öffnen

Use Case 17

Kurzbeschreibung: Der Akteur öffnet ein Workbenchprojekt. Der aktuelle Stand der Auswahl und Anwendung von Prozessmustern wird angezeigt.

Primärer Akteur: Pattern User

Initiales Prozessmuster hinzufügen

Use Case 18

Kurzbeschreibung: Einem Workbenchprojekt wird ein initiales Prozessmuster hinzugefügt. Das Prozessmuster gilt zu diesem Zeitpunkt als noch nicht ausgeführt.

Primärer Akteur: Pattern User

Use Case 19 **Prozessmustersequenz hinzufügen**

Kurzbeschreibung: In einem Workbenchprojekt wird zu einem schon ausgewählten Projekt ein Nachfolgermuster zugeordnet. Falls die syntaktischen Regeln diese Verknüpfung erlauben (oder durch eine Prozessmusterbeziehung schon festgelegt wurde), wird die Verknüpfung angelegt, andernfalls erscheint eine Fehlermeldung.

Primärer Akteur: Pattern User

Use Case 20 **Prozessmuster anwenden**

Kurzbeschreibung: Nachdem ein Prozessmuster ausgewählt wurde, kann es ausgeführt werden. Bei der Durchführung kann vom Akteur eine Aktion innerhalb eines Prozesses gewählt und ausgeführt werden. Bei der Durchführung einer Aktion werden im Zielzustand der Aktion die Artefakt- und Event-Instanzen erzeugt (d.h. sichtbar gemacht), deren Typen durch den Aktionstypen (=Aktivität) beschrieben wird.

Für die Anwendung einer Aktion wird die Zuordnung einer Person, die die vorgegebene Rolle besitzt, verlangt. Die Anwendung der Aktion wird farblich markiert.

Primärer Akteur: Pattern User

8.1.2 Nicht-funktionale Anforderungen

NF 1 **Teamfähigkeit der Software**

Um Versionskonflikte und Inkonsistenzen zu vermeiden, werden alle definierten Muster in einem zentralen Katalog abgelegt. Dieser Katalog ist nun für sämtliche Verwaltungsfunktionen (Kreation, Versionierung, Destruktion, Auffinden, ...) verantwortlich.

NF 2 **Grafische Benutzerschnittstelle**

Die Workbench soll mit einer grafischen Benutzerschnittstelle ausgestattet werden, die ein bequemes und unkompliziertes Erstellen von Mustern ermöglicht. Weiterhin soll ein grafisches Bearbeiten definierter Entwicklungsprozesse unterstützt werden.

NF 3 **Persistenz einzelner Prozessmuster**

Einzelne Prozessmuster sollen auf einem nichtflüchtigen Speicher persistent abgelegt werden können. Über eine Katalog-Komponente können diese Muster wieder ermittelt werden.

NF 4 **Persistenz definierter Prozesse**

Aus einzelnen Prozessmuster-Instanzen definierte Workbenchprojekte sind auf einem nichtflüchtigen Speicher persistent abzulegen.

8.2 Entwurf

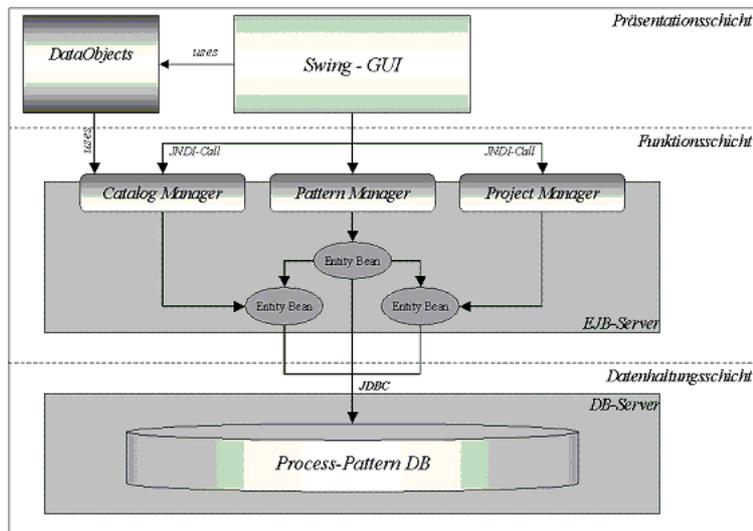


Abbildung 8-5: Architektur der Process Pattern Workbench

In diesem Abschnitt beschreiben wir die Architektur der Process Pattern Workbench. Aus Platzgründen geben wir nur einen Überblick über die Architektur. Eine detaillierte Beschreibung des Entwurfs findet der Leser in [Sch03].

Die Architektur entspricht der bekannten 3-Schichten-Architektur. Hierbei wird eine Gesamtanwendung in Präsentationsschicht, Funktionsschicht und Datenhaltungsschicht (Persistenzschicht) unterteilt (Abbildung 8-5).

Die Präsentationsschicht besteht im wesentlichen aus einer Client-Anwendung, welche eine Darstellungsfunktionalität übernimmt und dem Benutzer die Daten auf dem Bildschirm präsentiert. Implementiert wird der Client durch einen Swing-Client, der über RMI mit dem Pattern-Server kommuniziert und die entsprechende Business-Logik mittels eines JNDI-Aufrufs auf dem Server sucht. Die Informationen werden in Form von Schnittstellenobjekten gekapselt, die beiderseits vom Server als auch auf Client-Seite bekannt gemacht werden müssen.

Präsentationsschicht

Die Business-Logik liegt vollständig auf dem in der Funktionsschicht angesiedelten EJB-Server. Aufgrund der Komplexität wird dieser Bereich weiterhin in die Schichten Session-Facade, Business-Control und Entities unterteilt (Abbildung 8-6).

Business-Logik

Die Schicht Session-Facade beinhaltet die Komponenten, welche unmittelbar mit dem Client kommunizieren. Ihre Aufgabe besteht in der Entgegennahme von Client-Requests und der Abarbeitung übergreifender Geschäftsvorfälle. Die Business-Control-Schicht beinhaltet Prüfmethode, die von den jeweiligen Facade-Beans benötigt werden. Komponenten der Business-Controll Schicht liegen genau wie die Komponenten der Entities-Schicht hinter der Session-Facade-Schicht und können daher nicht direkt vom Client aufgerufen werden. Die Entities-Schicht enthält Komponenten, welche die persistenten Daten innerhalb des Systems darstellen und speichern diese auf einem Persistenzspeicher ab.

Session-Facade,
Business-Control
und Entities

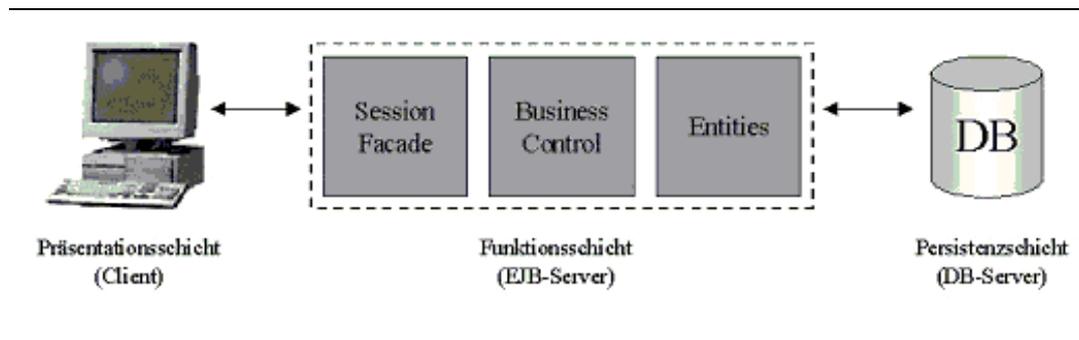


Abbildung 8-6: Aufteilung der Funktionsschicht auf dem EJB-Server

Datenhaltungs-
schicht

Die Datenhaltungsschicht wird durch einen im Backend gekoppelten Datenbankserver realisiert. Komponenten der Entities-Schicht greifen auf eine Datenbankinstanz zu und können die dort gespeicherten Daten auslesen und modifizieren.

8.3 Implementierung

In diesem Abschnitt betrachten wir die softwaretechnische Realisierung von PROPEL, die Process Pattern Workbench. Implementierungstechnische Details werden in Anhang C und in [Sch03] erläutert.

Zuordnung Use
Cases zu Dialogen

Abbildung 8-7 zeigt in einer Übersicht die Zuordnung von Use Cases zu softwaretechnisch realisierten Dialogen. Der Menüpunkt Project wurde aus Platzgründen ausgeblendet.

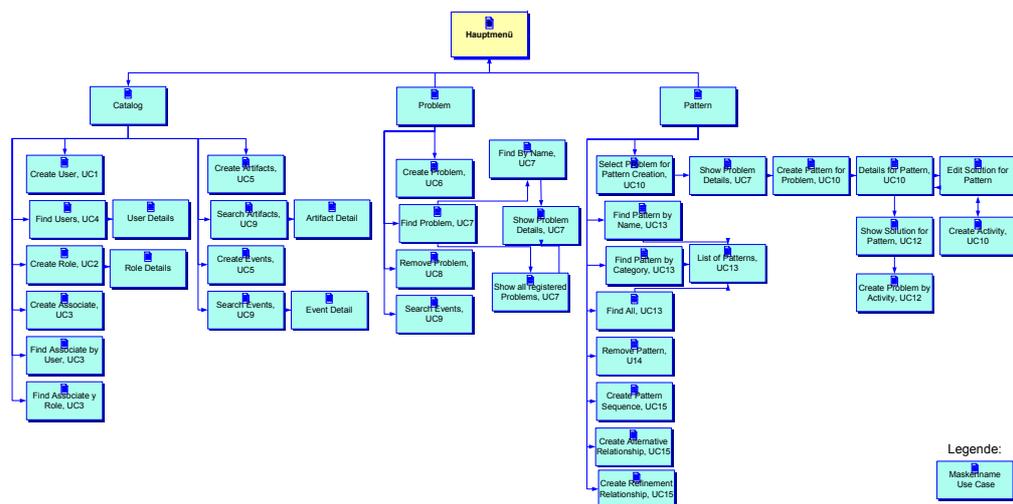


Abbildung 8-7: Navigation der Process Pattern Workbench

8.3.1 Das Menü

Beim Start der Process Pattern Workbench präsentiert sich dem Nutzer zunächst die Menüleiste der Workbench (Abbildung 8-8). Es gibt die Menüpunkte Server, Catalog, Problem, Pattern, Project und Info. Der Menüpunkt Server dient dazu, den Client mit dem Server zu verbinden. Der Menüpunkt Info informiert über den Versionsstand der Workbench. Ein detaillierte Darstellung der Menüpunkte Catalog, Problem, Pattern und Project erfolgt in den nachfolgenden Abschnitten.

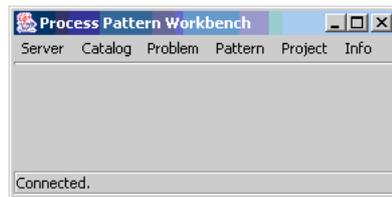


Abbildung 8-8: Menüleiste der Process Pattern Workbench

8.3.2 Der Menüpunkt Catalog

Der Menüpunkt Catalog bietet Funktionalitäten, um die Grundelemente eines Prozessmusterkatalogs zu definieren. Dies sind die Benutzer (hier: User) der Workbench, Rollen (hier: Roles), die jeweils eine Menge von Berechtigungen zur Ausübung bestimmter Funktionen umfassen, die Zuordnung (hier: Associate) von Rollen zu Benutzern, die Objekte (hier: Artifacts) und die Ereignisse (hier: Events), die später bei der Modellierung von Problemen und Prozessmustern verwendet werden können.

Für das Anlegen eines neuen Benutzers sind der Benutzername (hier: Username), sowie sein Vor- und Nachname (hier: Firstname bzw. Surname) anzugeben. Über die Funktion „find“ können alle angelegten Benutzer angezeigt und – wenn gewünscht – modifiziert werden. Beim Klick auf den Benutzer wird die Detailansicht des Benutzers angezeigt.

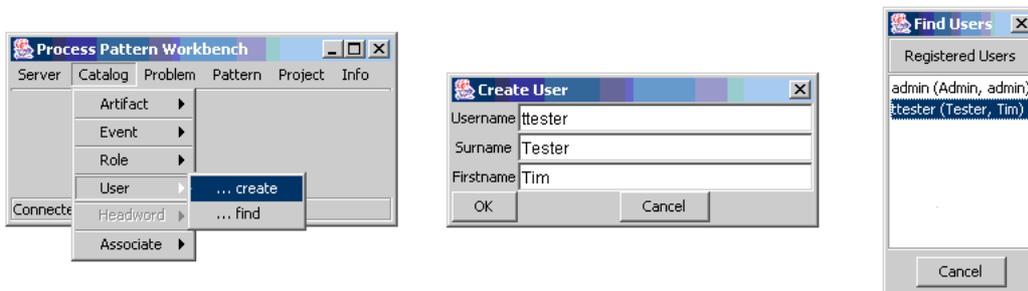


Abbildung 8-9: Benutzer anlegen (links, mittig) und Benutzer suchen (mittig, rechts)

Für das Anlegen einer neuen Rolle sind der Rollenname (hier: Role Name), sowie eine Beschreibung der mit der Rolle verknüpften Tätigkeiten und Berechtigungen erforderlich. Über die Funktion „find“ können alle angelegten Rollen angezeigt und – wenn gewünscht – modifiziert werden. Alternativ können per Wildcards Rollen gesucht werden. Beim Klick auf die Rolle wird die Detailansicht der Rolle angezeigt.

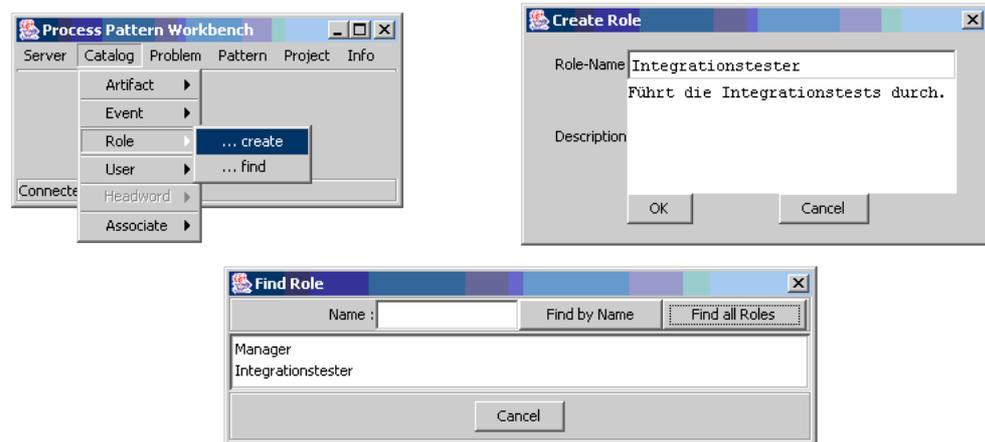


Abbildung 8-10: Rolle anlegen (oben) und Rolle suchen (unten)

Einem Benutzer ist eine Rolle zuzuweisen. Über den Dialog „Create Associate“ kann dies durchgeführt werden. Ferner können über die Funktionen „Find by User“ bzw. „Find by Role“ jeweils alle Rollen eines Benutzers bzw. alle Benutzer mit einer bestimmten Rolle angezeigt werden.

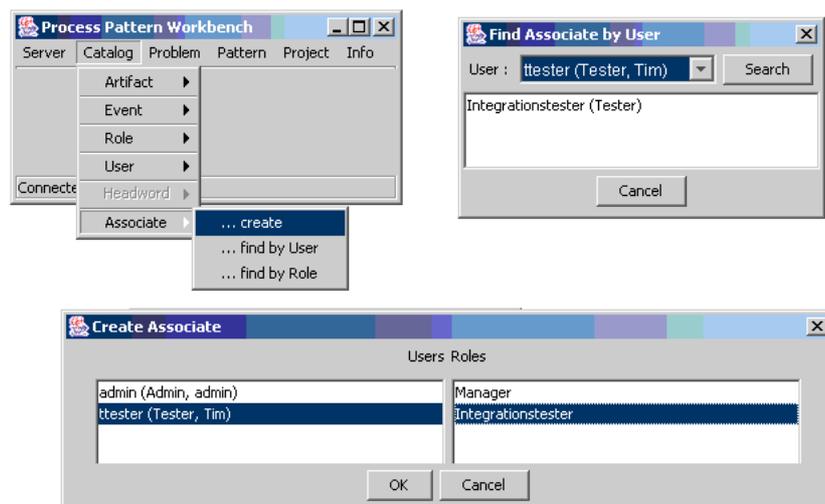


Abbildung 8-11: Benutzer-Rollen-Verknüpfung (links oben, unten) und Benutzer-Rollen-Verknüpfungen suchen (oben rechts)

Für das Anlegen eines neuen Objekts sind der Name des Objekts (hier: Artifact Name), der Aspekt (hier: Category), und eine Beschreibung anzugeben. Über die Funktion „find“ können alle Objekte angezeigt und – wenn gewünscht – modifiziert werden. Die Suche lässt sich auch auf alle Objekte eines Aspekts einschränken. Beim Klick auf das Objekt wird die Detailansicht des Objekts angezeigt. Alle Dialoge existieren analog für Ereignisse. Auf die Darstellung dieser Dialoge wird aus diesem Grund verzichtet.

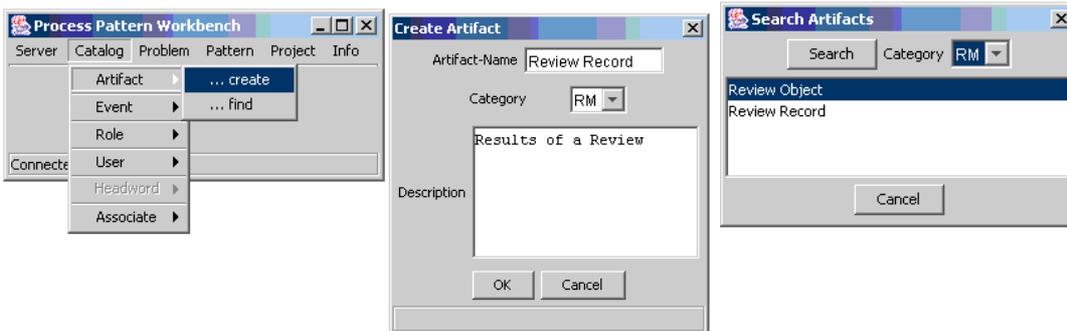


Abbildung 8-12: Anlegen eines Objekts (links, Mitte) und Suchen eines Objekts (rechts)

8.3.3 Der Menüpunkt Problem

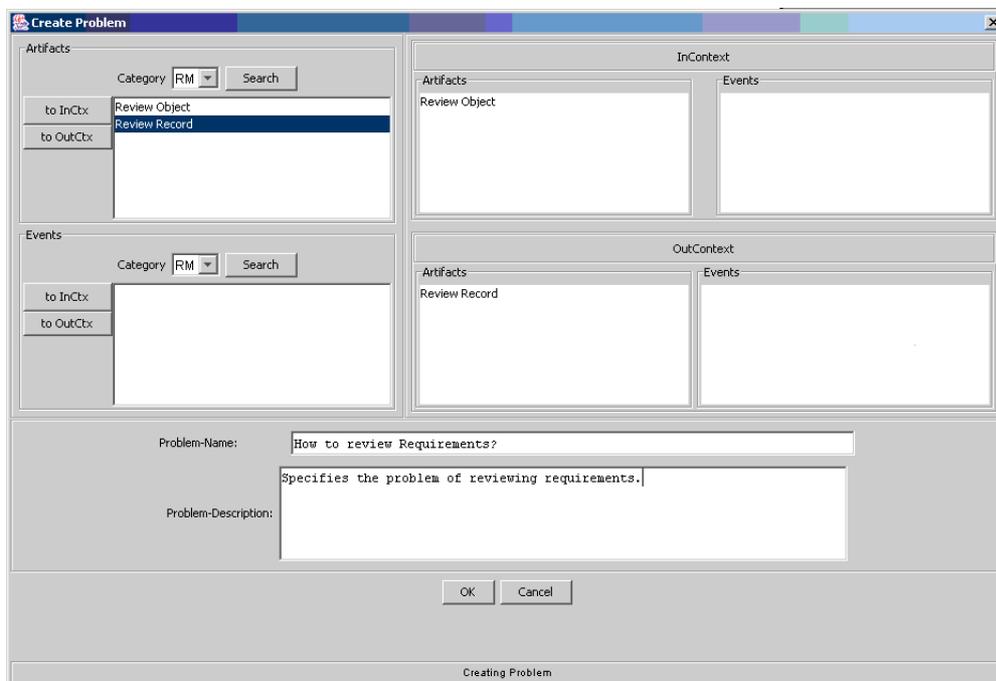


Abbildung 8-13: Definition eines Problems

Menüpunkt Problem	Der Menüpunkt Problem bietet Funktionalitäten, um Probleme zu definieren, zu suchen und wieder zu löschen. Diesen Problemen können anschließend über das Menü „Pattern“ geeignete Prozessmuster zugeordnet werden.
Probleme definieren	Ein Problem wird über den Dialog „Create Problem“ definiert (Abbildung 8-13). Hierzu werden zunächst Objekte und Ereignisse ausgewählt und dem initialen und resultierenden Kontexts des Problems zugewiesen. Ferner wird dem Problem ein Name und eine Beschreibung zugewiesen.
Probleme suchen	Über den Menüpunkt „Find Problem“ kann ein Problem gesucht werden, entweder, indem man die Liste aller Probleme anfordert, oder indem man per Wildcard nach dem Namen sucht (Abbildung 8-14, links). Ausgehend von der Ergebnisliste der Suche kann dann die Detailansicht des Problems angefordert werden. In dieser Ansicht werden bereits die lösenden Prozessmuster (Abbildung 8-14, rechts, Feld „Solving Patterns“) angegeben.

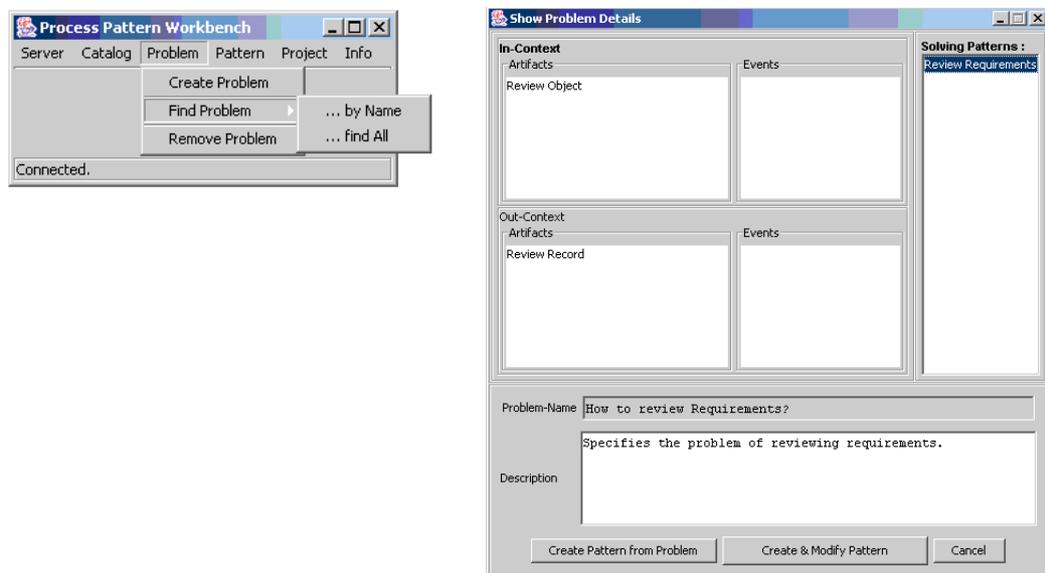


Abbildung 8-14: Suche und Detailansicht eines Problems

Probleme löschen	Über den Menüpunkt „Remove Problem“ kann ein Problem aus dem Prozessmusterkatalog entfernt werden (Abbildung 8-15).
------------------	---

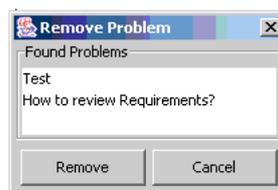


Abbildung 8-15: Entfernen eines Problems

8.3.4 Der Menüpunkt Pattern

Der Menüpunkt Pattern bietet Funktionalitäten, um Prozessmuster zu definieren, zu suchen und wieder zu löschen. Ferner können Beziehungen zwischen Prozessmustern definiert werden.

Menüpunkt Pattern

Über den Menüpunkt „Create Pattern by Problem“ kann ein neues Prozessmuster angelegt werden (Abbildung 8-16). Für die Musterdefinition muss ein vorab definiertes Problem ausgewählt werden. Anschließend wird der schon weiter oben erläuterte Dialog „Show Problem Details“ (Abbildung 8-14, rechts) angezeigt. Dort kann über die Funktion „Create Pattern from Problem“ das Prozessmuster mit Namen, Aspekt und Beschreibung angelegt werden.

Prozessmuster definieren

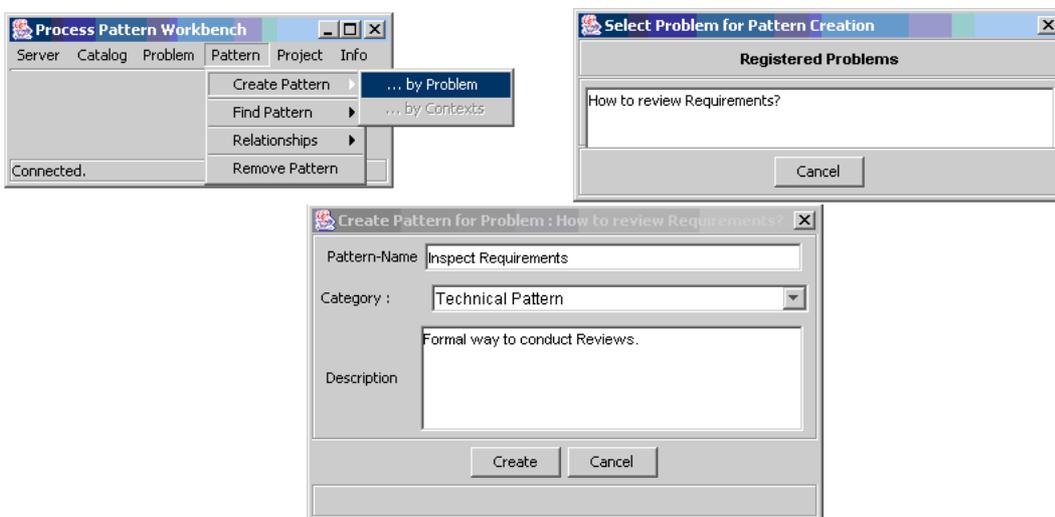


Abbildung 8-16: Erzeugen eines Prozessmusters (links oben) über die Auswahl eines Problems (rechts oben) und Vergabe eines Namens (unten)

In der Detailansicht eines Prozessmusters kann anschließend der Prozess modelliert werden. Hierzu wählt der Benutzer die Funktion „Edit Solution“, die ihn in das Modellierungsfenster führt (Abbildung 8-17). Im dem Modellierungsfenster sind bereits der initiale und der resultierende Kontext des Prozessmusters vorgegeben. Der Benutzer kann nun neue Aktivitäten anlegen. Für jede Aktivität wird ein initialer und resultierender Kontext definiert (Abbildung 8-18).

Prozess modellieren

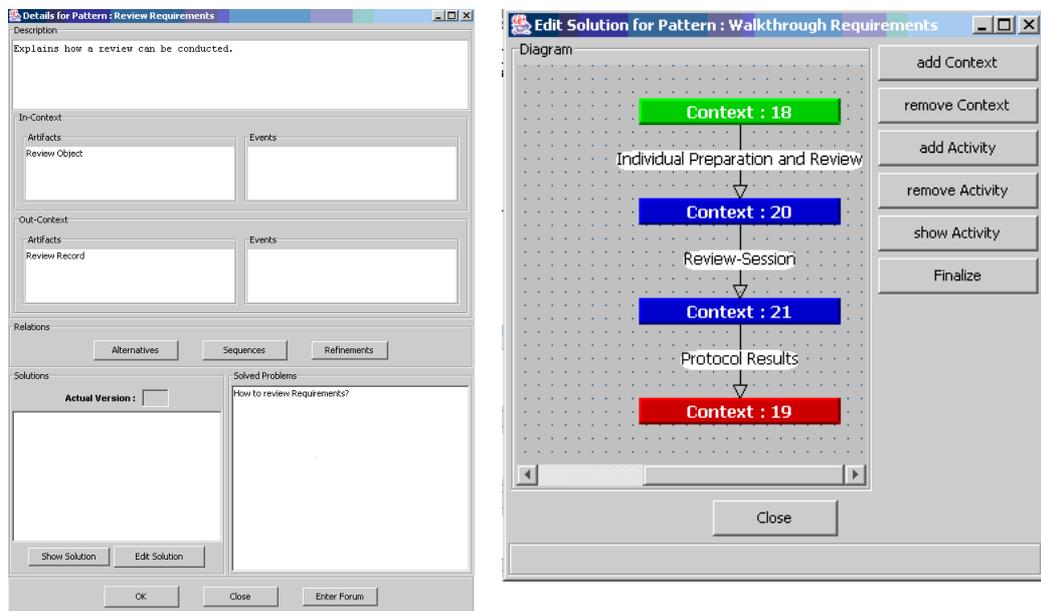


Abbildung 8-17: Modellierung eines Prozesses durch Auswahl der Funktion „Edit Solution“, (links), Definition von Aktivitäten und Kontexten (rechts),

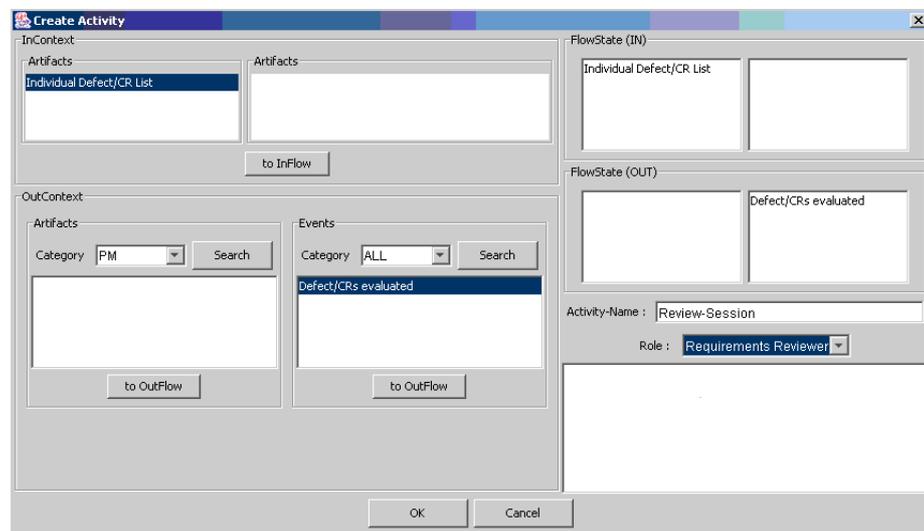


Abbildung 8-18: Dialog zur Modellierung einer einzelnen Aktivität

Prozessversion
einfrieren

Der Benutzer kann durch die Funktion „Finalize“ (Abbildung 8-17, oben rechts) die Version eines Prozesses einfrieren. Weitere Änderungen an dem Prozess werden dann stets an der nächsthöheren Version des Prozesses durchgeführt. Über die Funktion „Show Solution“ im Dialog „Details for Pattern“ (Abbildung 8-17, oben links) kann eine eingefrorene Version eines Prozess angezeigt werden.

Im gleichen Dialog kann man einer Aktivität ein Problem zuordnen bzw. definieren (Dialog „Create Problem by Activity“, Abbildung 8-19). Durch die spätere Zuordnung von Prozessmustern zu diesem Problem wird eine Use-Beziehung automatisch zwischen dem übergeordneten und dem untergeordneten Muster erstellt.

Zuordnung von Aktivität und Problem

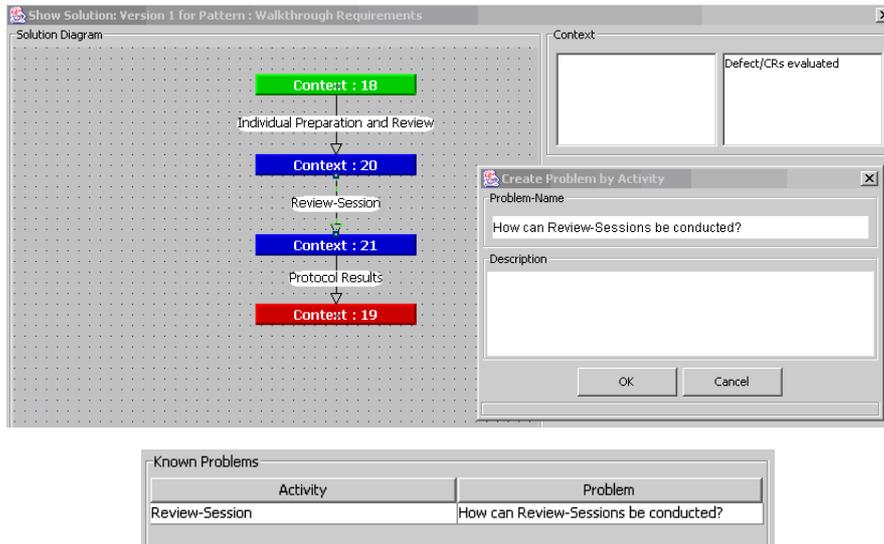


Abbildung 8-19: Betrachtung eines finalisierten Prozesses und Zuordnung von Problemen zu Aktivitäten (oben), Ansicht der Aktivitäts-Problem-Paare (unten)

Über den Menüpunkt „Find Pattern“ kann ein Prozessmuster gesucht werden, entweder, indem man die Liste aller Probleme anfordert, oder indem man sich alle Muster eines Aspekt anzeigen lässt oder indem man per Wildcard nach dem Namen sucht (Abbildung 8-20). Ausgehend von der Ergebnisliste der Suche kann dann die Detailansicht des Prozessmusters angefordert werden (Dialog „Details for Pattern“ in Abbildung 8-17, oben links).

Prozessmuster suchen

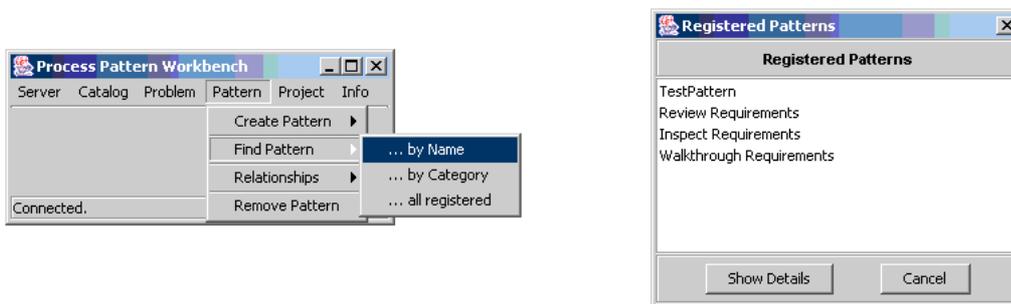


Abbildung 8-20: Suche nach Prozessmustern (links), Ergebnis der Suche (rechts)

Über den Menüpunkt „Remove Pattern“ können Prozessmuster aus dem Prozessmusterkatalog gelöscht werden.

Prozessmuster löschen

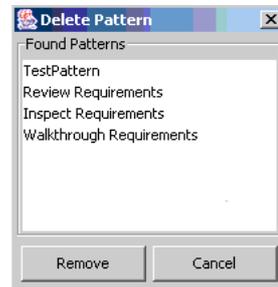


Abbildung 8-21: Entfernen eines Prozessmusters aus dem Prozessmusterkatalog

Prozessmusterbeziehungen festlegen

Der Akteur wählt über die Funktion „Relationship -> Create“ aus, welche Prozessmusterbeziehung er festlegen möchte (Abbildung 8-22). In zwei Auswahllisten kann der Akteur jeweils diejenigen Prozessmuster auswählen, die über die Beziehung verknüpft werden sollen. Anschließend findet eine automatische Überprüfung durch das System statt, ob die Beziehung bzgl. der syntaktischen Regeln erlaubt ist. Falls ja, wird die Beziehung angelegt, falls nein, wird die Beziehung nicht angelegt und es erscheint eine Fehlermeldung. Die Ansicht, mit welchen anderen Prozessmustern ein Prozessmuster in Beziehung steht, ist über die Funktionen „Sequences“, „Refinements“ und „Alternatives“ im Dialog „Details for Pattern“ (Abbildung 8-17, oben links) zugänglich. Eine Übersicht über die Use-Beziehung ist noch nicht verfügbar.

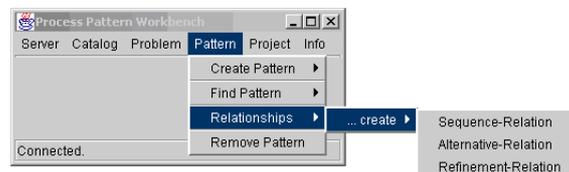


Abbildung 8-22: Definition von Prozessmusterbeziehungen – Teil 1

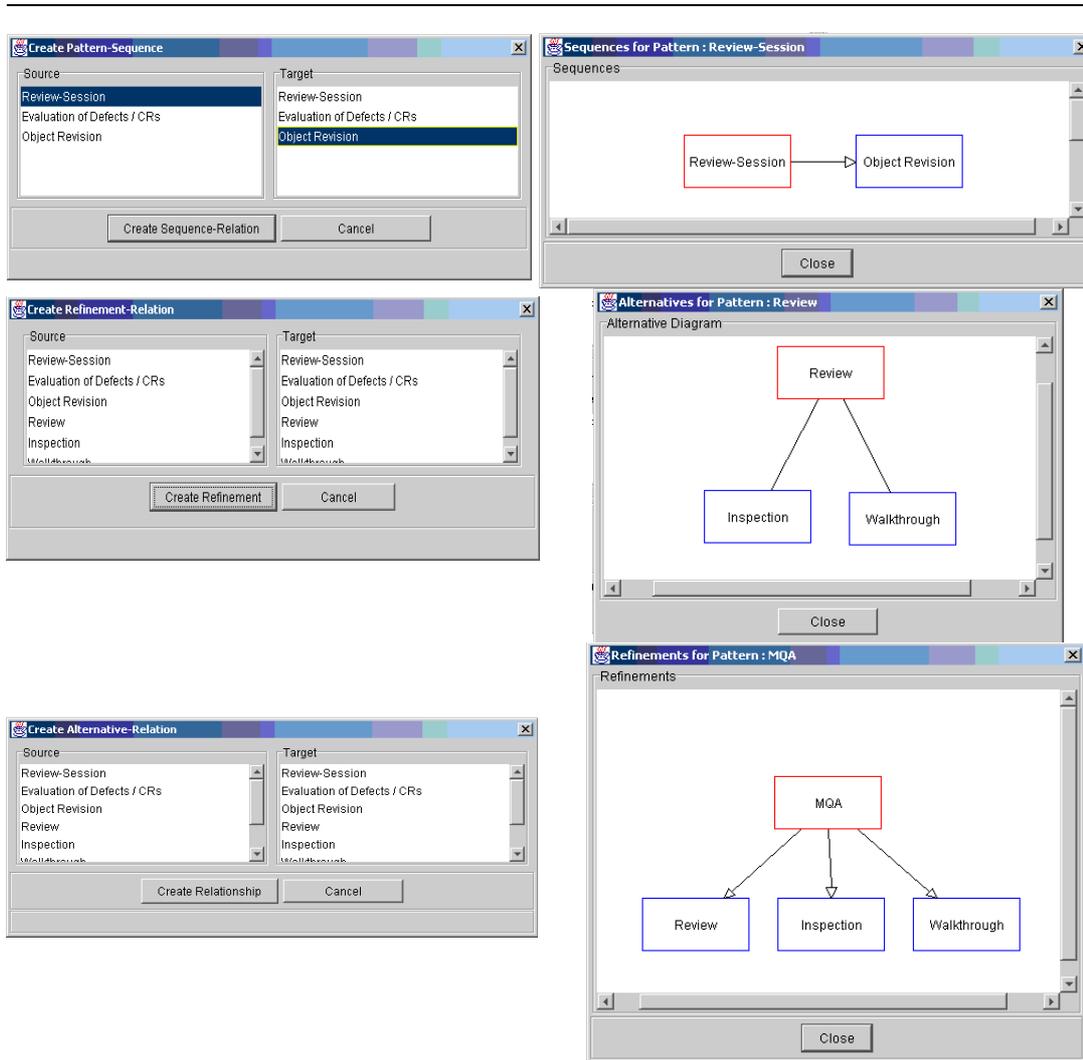


Abbildung 8-23: Definition von Prozessmusterbeziehungen – Teil 1

8.3.5 Der Menüpunkt Project

Der Menüpunkt Project bietet Funktionalitäten, um die Anwendung von Prozessmustern in einem Projekt dokumentieren zu können. Über die Funktion „Project ->Create“ wird ein neues Workbenchprojekt angelegt (Abbildung 8-24). Hier muss der Projektname und der für das Workbenchprojekt verantwortliche Mitarbeiter angegeben werden. Wählt man die Funktion „Project->Load“, kann aus einer Liste ein Workbenchprojekt ausgewählt werden.

Menüpunkt Project

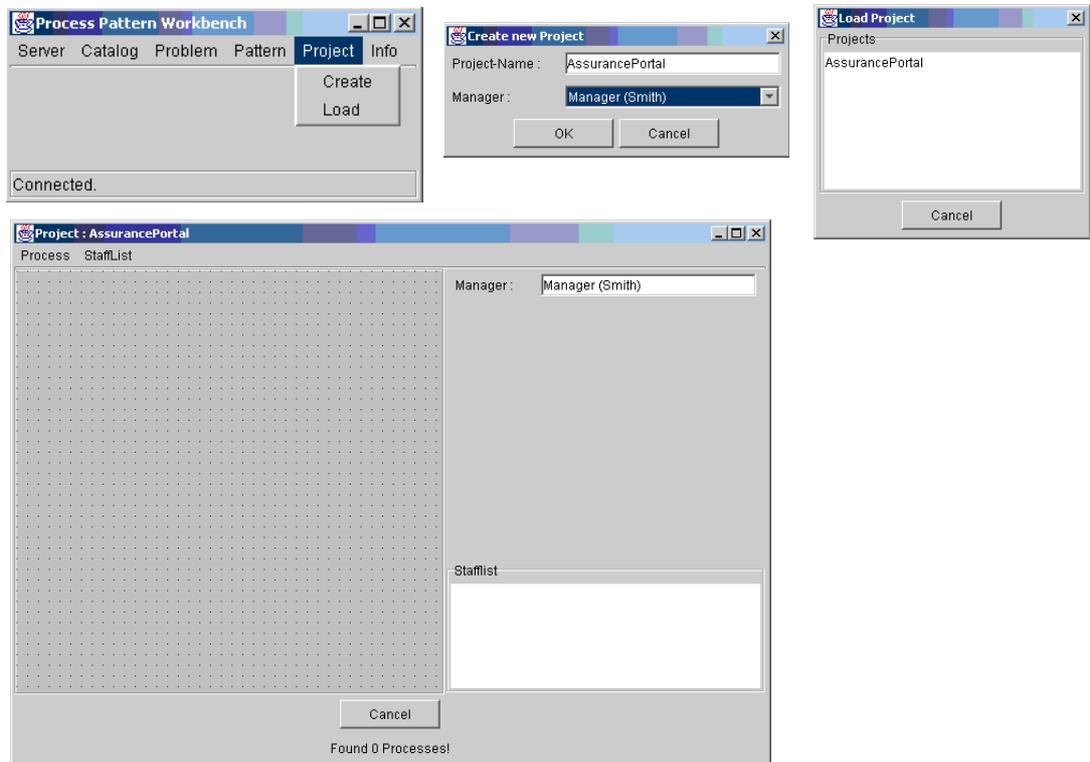


Abbildung 8-24: Erstellen (oben links, oben mittig und unten) und Öffnen von Workbenchprojekten (oben rechts)

Prozessmuster
auswählen

Über die Funktion „Create Initial Process“ wird das Prozessmuster, mit dem das Projekt begonnen werden soll, ausgewählt. Ferner wird ein Prozessverantwortlicher zugeordnet (Abbildung 8-25)

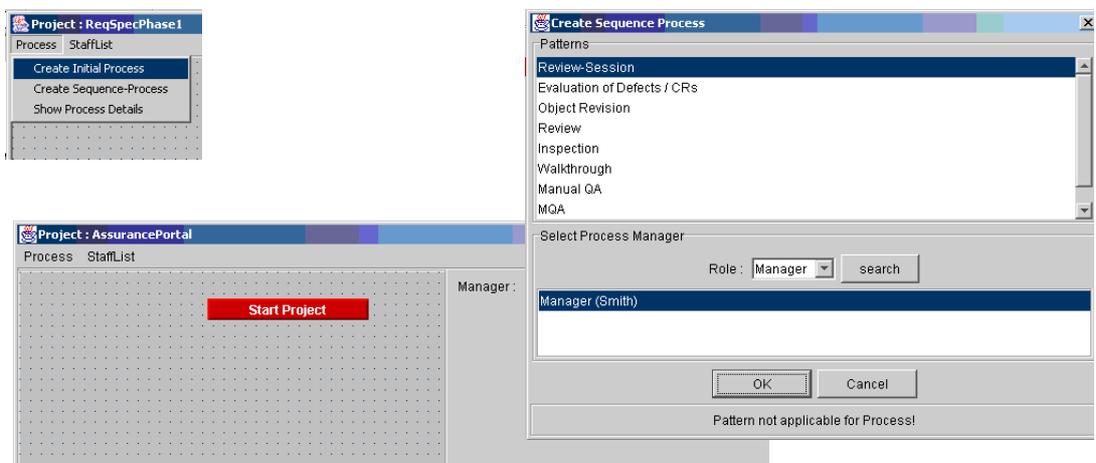


Abbildung 8-25: Auswahl des initialen Prozessmusters und Zuordnung eines Prozessverantwortlichen

In einem Workbenchprojekt wird zu einem schon ausgewählten Projekt ein Nachfolgemuster zugeordnet (Abbildung 8-26). Falls die syntaktischen Regeln diese Verknüpfung erlauben (oder durch eine Prozessmusterbeziehung schon festgelegt wurde), wird die Verknüpfung angelegt, andernfalls erscheint eine Fehlermeldung.

Nachfolgemuster zuordnen

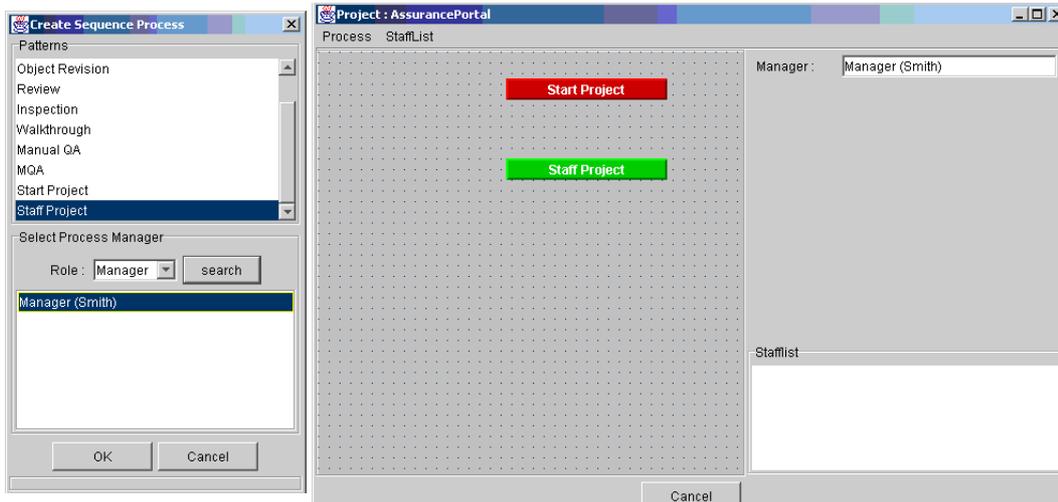


Abbildung 8-26: Festlegung von Sequenzen von Prozessmusterinstanzen

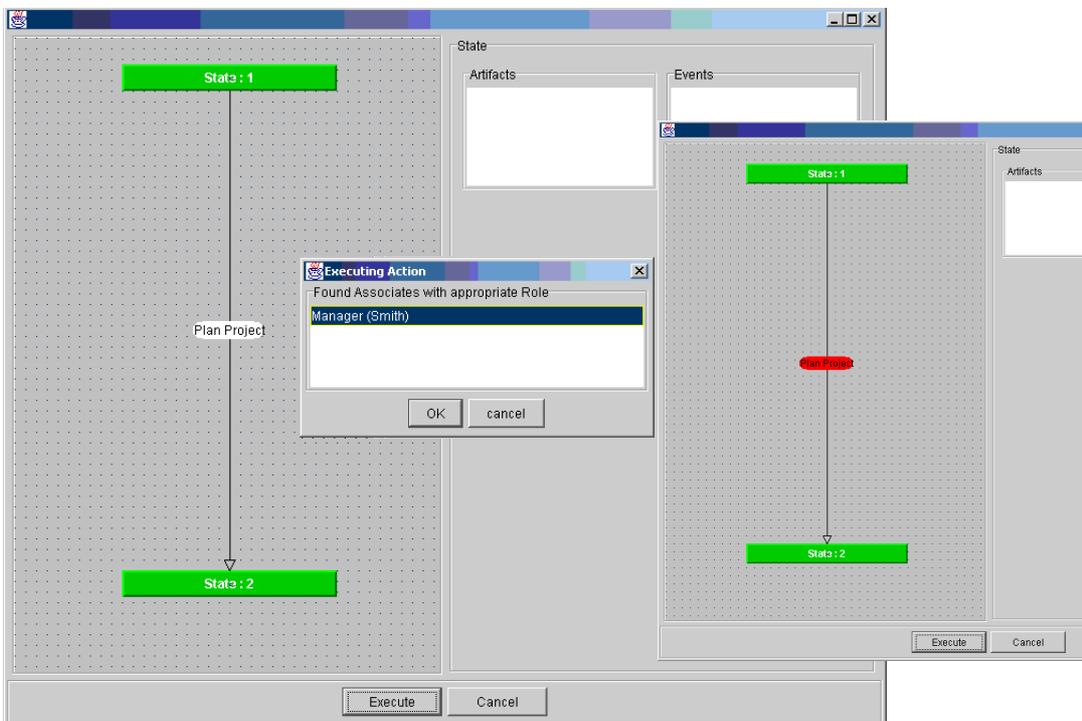


Abbildung 8-27: Dokumentation der Ausführung von Aktivitäten

Prozessmuster ausführen

Nachdem ein Prozessmuster ausgewählt wurde, kann es ausgeführt werden (Abbildung 8-27). Bei der Durchführung kann vom Akteur eine Aktion innerhalb eines Prozesses gewählt und ausgeführt werden. Bei der Durchführung einer Aktion werden im Zielzustand der Aktion die Artefakt- und Event-Instanzen erzeugt (d.h. sichtbar gemacht), deren Typen durch den Aktionstypen (=Aktivität) beschrieben wird. Für die Anwendung einer Aktion wird die Zuordnung einer Person, die die vorgegebene Rolle besitzt, verlangt. Die Anwendung der Aktion wird farblich markiert.

8.4 Erweiterungsmöglichkeiten

In diesem Abschnitt definieren wir eine Menge von Erweiterungsmöglichkeiten. Darunter fallen Erweiterungen der bereits bekannten Use Cases als auch neue Use Cases.

8.4.1 Benutzerverwaltung

Erweiterung UC 1 „Benutzer anlegen“	<p>Kurzbeschreibung: Der Use Case wird erweitert um die Angabe eines Passworts für den Benutzer.</p> <p>Primärer Akteur: Workbench Administrator</p>
Erweiterung UC 2 „Rolle anlegen“	<p>Kurzbeschreibung: Der Use Case wird erweitert um die Verknüpfung einer Menge von Rechten mit der Rolle. Hierzu muss Konzept zum Rechtemanagement (z.B. für die Verankerung der Rechte im System) ausgearbeitet werden.</p> <p>Primärer Akteur: Workbench Administrator</p>
Neuer Use-Case: Login/Logout	<p>Kurzbeschreibung: Der Akteur wählt den Menüpunkt Login aus, gibt Usernamen und Passwort an und erhält anschließend die seiner Rolle entsprechenden Rechte.</p> <p>Der Akteur wählt den Menüpunkt Logout aus, bestätigt und ist damit abgemeldet. Er hat keinerlei Zugriff auf die Workbench.</p> <p>Primärer Akteur: Workbench Administrator, Pattern Designer, Pattern User</p>
Neuer Use-Case: Benutzer löschen/ Rolle löschen	<p>Kurzbeschreibung: Nachdem der Akteur über die Suche einen Benutzer bzw. eine Rolle gefunden hat, löscht er die Rolle bzw. den Benutzer. Sind Benutzer einer Rolle zugeordnet, die gelöscht werden soll, so erscheint eine Fehlermeldung. Das Löschen der Rolle ist in diesem Fall nicht möglich. Die entsprechenden Benutzer müssen vorher einer anderen Rolle zugeordnet oder gelöscht werden.</p> <p>Primärer Akteur: Workbench Administrator</p>

8.4.2 Katalogverwaltung

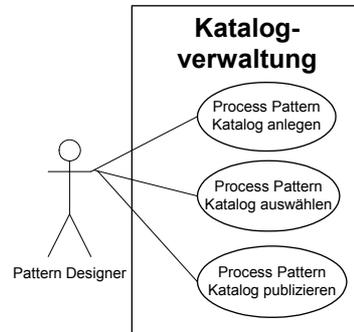


Abbildung 8-28: Use Cases der Katalogverwaltung

Kurzbeschreibung: Der Akteur legt einen neuen Prozessmusterkatalog an und weist diesem einen Namen und eine Domäne zu. Initial erhält der Katalog die Version 1.

Primärer Akteur: Pattern Designer

Neuer Use-Case:
Prozessmusterkatalog anlegen

Kurzbeschreibung: Der Akteur wählt zur Bearbeitung einen Prozessmusterkatalog aus.

Primärer Akteur: Pattern Designer

Neuer Use-Case:
Prozessmusterkatalog auswählen

Kurzbeschreibung: Wird ein Projekt angelegt, so bezieht sich dies auf eine bestimmte Version des Prozessmusterkatalogs. Der Akteur publiziert eine neue Version des Prozessmusterkatalogs. Diese Version ist ab sofort die Vorlage für alle neuen Projekte.

Primärer Akteur: Pattern Designer

Neuer Use-Case:
Prozessmusterkatalog publizieren

8.4.3 Problemdefinition

Kurzbeschreibung: Der Use Case wird erweitert um die Komposition von Objekten und Ereignissen.

Primärer Akteur: Pattern Designer

Erweiterung UC 5
„Objekt/Ereignis definieren“

Kurzbeschreibung: Der wird erweitert um die Verfeinerung von Problemen. Der Akteur kann ein Problem als Verfeinerung eines anderen Problems modellieren. Hierzu wählt der Akteur aus einer Dropdownliste das Superproblem aus.

Primärer Akteur: Pattern Designer

Erweiterung UC 6
„Problem definieren“

Neuer Use-Case:
Problem modifi-
zieren

- Kurzbeschreibung:** Der Akteur modifiziert Name, Beschreibung oder Problemkontext. Der Name oder die Beschreibung können geändert werden. Wird das Problem referenziert (z.B. in der Detailansicht Prozessmuster), wird stets der aktuelle Name angezeigt. Soll der Problemkontext modifiziert werden, so werden alle Probleme und Prozessmuster angezeigt, die von dieser Änderung betroffen sind. Ferner wird angezeigt, welcher Art diese Änderungen sind. Der Akteur kann die Änderungen übernehmen. Andernfalls kann der Akteur ein neues Problem definieren. Durch die Modifikation wird eine neue Version des Problems angelegt.
- Vorbedingungen:** Das Problem wurde modelliert und über die Suche gefunden.
- Nachbedingungen:** Der Pattern User kann nur jeweils die aktuelle Version eines Problems sehen.
- Primärer Akteur:** Pattern Designer

Erweiterung UC 7
„Problem suchen“

- Kurzbeschreibung:** Der Use Case wird erweitert um die Suche durch Angabe von Objekten und Ereignisse (unterteilt nach initialem und resultierendem Kontext). Auf diese Weise werden Probleme gesucht, die diese Elemente enthalten.
- Primärer Akteur:** Pattern Designer, Pattern User

Neuer Use Case:
Objekt löschen/
Ereignis löschen

- Kurzbeschreibung:** Der Akteur entfernt ein Objekt oder ein Ereignis aus dem Prozessmusterkatalog.
- Vorbedingungen:** Das Objekt bzw. das Ereignis darf keinem Problem oder Prozessmuster zugeordnet sein.
- Primärer Akteur:** Pattern Designer

8.4.4 Patterndefinition

Erweiterung UC 10
„Prozessmuster
definieren“

- Alternative Vorgehens-
gänge:** Der Akteur definiert ein Prozessmuster ohne vorherige Auswahl eines Problems. Beim Abspeichern des Problems wird der Akteur vom System gefragt, welches Problem dem Prozessmuster zugeordnet werden soll. Ist kein passendes Problem vorhanden, fragt das System, ob ein dem Musterkontext entsprechendes Problem angelegt werden soll. Verneint dies der Akteur, so kann das Prozessmuster nicht abgespeichert werden.
- Primärer Akteur:** Pattern Designer
- Erweiterungen:** Diese Use Case verwendet die Use Cases „Problem suchen“ und „Problem definieren“.

<p>Kurzbeschreibung: Der Akteur modifiziert den Prozess, indem er die Aktivitäten bearbeitet (d.h. modifiziert, hinzufügt, entfernt). Das Ändern des initialen und resultierenden Kontexts des Prozessmusters ist nicht erlaubt.</p> <p>Primärer Akteur: Pattern Designer</p>	<p>Neuer Use Case: Prozess modifizieren</p>
<p>Kurzbeschreibung: Nachdem der Akteur ein Prozessmuster erstellt hat, kann er festlegen, mit welchen anderen Prozessmustern es in Beziehung steht. Auf Anforderung des Akteurs ermittelt das System, mit welchen Prozessmustern das betrachtete Prozessmuster in Beziehung stehen könnte. Der Akteur muss aus den vorgeschlagenen Beziehungen diejenigen auswählen und bestätigen, die er für sinnvoll hält.</p> <p>Primärer Akteur: Pattern Designer</p>	<p>Erweiterung UC 15 „Musterbeziehungen festlegen“</p>

8.4.5 Reporting

Die Anwendungsfälle des Reportings dienen der statistischen Auswertung bzw. der Diskussion bestimmter Workbenchinhalte.



Abbildung 8-29: Anwendungsfälle des Reportings

<p>Kurzbeschreibung: Der Pattern-Anwendungsgrad definiert, wie oft ein Muster bisher in allen Entwicklungsprozessen angewendet worden ist. Der Anwendungsgrad ist definiert über den Mustertyp und berücksichtigt keine Musterversionen. Ferner können Mustersequenzen ermittelt werden. Mustersequenzen sind häufig angewendete Abfolgen von Prozessmustern.</p> <p>Primärer Akteur: Pattern Designer, Pattern User</p>	<p>Neuer Use Case: Musteranwendungsgrad ermitteln</p>
<p>Kurzbeschreibung: Der Akteur kann bzgl. eines Mustertyps an einer Diskussion teilnehmen und dort einen Diskussionsbeitrag in ein Diskussionsforum einstellen. Ein Diskussionsforum existiert für jeden Mustertyp und kann beliebig viele Einträge enthalten.</p>	<p>Neuer Use Case: Diskussion hinzufügen</p>

Primärer Akteur: Pattern Designer, Pattern User

Neuer Use Case:
Pattern-Historie
anzeigen

Kurzbeschreibung: Über die Pattern-Historie kann sich der Akteur alle Versionen eines Prozessmusters anzeigen lassen.

Primärer Akteur: Pattern Designer, Pattern User

8.4.6 Projektverwaltung

Neuer Use-Case:
Prozessmuster
entfernen

Kurzbeschreibung: Ein festgelegtes, aber noch nicht durchgeführtes Prozessvorgehen kann aus einem Gesamtprozess wieder gelöscht werden. Für den gelöschten Prozess gilt, dass

- alle Prozessbausteine einer Prozesssequenz ebenfalls gelöscht werden
- alle Prozessbausteine von Sub-Prozessen ebenfalls gelöscht werden

Primärer Akteur: Pattern User

Neuer Use-Case:
Prozessmuster
austauschen

Kurzbeschreibung: Innerhalb eines ausgewählten Prozessmusters kann ein Sub-Prozess gegen einen anderes Vorgehensmodell ausgetauscht werden. Bei Prozess-Sequenzen ist zu beachten, dass der resultierende Kontext des eingetauschten Vorgehensmodells den eingehende Kontext des folgenden Vorgehens erfüllt. Wird ein Sub-Prozess ausgetauscht, müssen ebenfalls die Kontexte wieder die Anwendbarkeits- und die Lösbarkeitsbedingung erfüllen.

Primärer Akteur: Pattern User

Neuer Use-Case:
Projektstatus
ermitteln

Kurzbeschreibung: Der Projektstatus kann aus den durchgeführten Prozessen im Vergleich zu den definierten Prozessen abgeleitet werden. Wenn z. B. innerhalb eines Projekts 20 Artefakte erstellt werden müssen, aber erst 10 Instanzen erstellt worden sind, ergibt sich ein Projektstatus von 50%.

Primärer Akteur: Pattern User

Neuer Use-Case:
Pattern Sequences
ermitteln

Kurzbeschreibung: Pattern Sequences definieren häufig verwendete Kombinationen von Prozessmustern. Sie dienen dazu, dem Benutzer wertvolle Hinweise auf sinnvolle Kombinationen von Prozessmustern zu liefern.

Primärer Akteur: Pattern User, Pattern Designer

8.5 Zusammenfassung

In diesem Kapitel haben wir gezeigt, wie Konzepte der Sprache PROPEL softwaretechnisch umgesetzt werden können. Mit der Process Pattern Workbench existiert eine Plattform, die Funktionalitäten zur Modellierung einzelner Prozessmuster und ihrer Beziehungen ermöglicht. Hierdurch wird die Handhabung von Prozessmustern vereinfacht. Bei der Process Pattern Workbench handelt es sich nichtsdestotrotz um eine prototypische Realisierung. Für eine optimale Unterstützung bei der Modellierung und Anwendung von Prozessmustern ist die Workbench um weitere Funktionalitäten wie in Abschnitt 8.4 vorgestellt zu ergänzen.

9 Process Pattern Management

Unter Process Pattern Management fassen wir alle mit Patterns verknüpften Tätigkeiten wie Pattern-Dokumentation, Problem-Identifikation und -Spezifikation, Patternsuche, Patternauswahl, Patternanwendung, Patterneinführung, -einsatz und -verbesserung und die Unterstützung durch Werkzeuge zusammen. Je nach Erfordernis können einzelne oder alle Tätigkeiten dieses Spektrums ausgeübt werden.

Ziele

Für ein funktionierendes Process Pattern Management muss zunächst die Infrastruktur entsprechend vorbereitet werden (Abschnitt 9.1). Ferner sind Überlegungen bzgl. der Aufbauorganisation anzustellen (Abschnitt 9.2). Desweiteren ist die Einführung und die Etablierung des Konzepts der Prozessmuster und gegebenenfalls schon verfügbarer Prozessmusterkataloge zu planen (Abschnitt 9.3). Anschließend kann der Einsatz von Prozessmustern beginnen (Abschnitt 9.4).

Aspekte

9.1 Infrastruktur

Ist in einem Unternehmen beschlossen worden, Prozessmuster als ein den Softwareentwicklungsprozess unterstützendes Konzept einzuführen und anzuwenden, muss entschieden werden, inwieweit die Infrastruktur des Unternehmens miteinbezogen wird. Mit Infrastruktur meinen wir die Unterstützung durch Werkzeuge.

Die einfachste Art der Werkzeugunterstützung ist die Sammlung und Präsentation von Prozessmustern in einem einzelnen Textdokument. Das Dokument sollte – für alle Mitarbeiter des Unternehmens gut einsehbar – an einer zentralen Stelle präsentiert werden, also z.B. auf einem für alle zugreifbaren Laufwerk oder im Unternehmensintranet. Das Dokument nimmt damit die Rolle eines Prozessmusterkatalogs ein. Dieser sollte von der für den Katalog verantwortlichen Rolle gepflegt werden, nämlich vom Pattern Designer (Abschnitt 9.2).

Textdokument

Die anspruchsvollere Art der Werkzeugunterstützung ist der Einsatz eines dedizierten Prozessmuster-Werkzeugs. Ein solches Werkzeug verkörpert die Process Pattern Workbench, die wir in Kapitel 8 vorgestellt haben. Die Process Pattern Workbench erlaubt über das Sammeln und Präsentieren von Prozessmustern hinaus syntaktische Überprüfungen von Prozessmustern und deren Beziehungen. Insbesondere die Suche von Problemen und Prozessmustern erleichtert und fördert den Einsatz von Prozessmustern. Für weitere Vorteile und Details der Process Pattern Workbench verweisen wir auf Kapitel 8.

Prozessmuster-Werkzeug

Unabhängig von der Werkzeugunterstützung ist zu entscheiden, mit welchem Schema Prozessmuster zu beschreiben sind. Ein Vorschlag für ein Prozessmusterschema findet sich in Abschnitt 3.4. Dieses Schema ist allen Projektmitarbeitern zur Verfügung zu stellen.

9.2 Aufbauorganisation

Dieser Abschnitt beschreibt die organisatorischen Aspekte, die zum erfolgreichen Einsatz von Prozessmustern notwendig sind. Hierunter fällt (vgl. mit [Kne98]):

- die Aufbauorganisation, z.B. durch Einrichtung einer Gruppe von Mitarbeitern, die die Prozessmuster betreuen und die
- Ablauforganisation, d.h. die Prozesse, die mit der Entwicklung und Pflege der Prozessmuster zusammenhängen, z.B. Publikation der Prozessmuster (Abschnitte 9.3 und 9.4).

Ein unternehmensspezifisches Vorgehensmodell wird meistens von einer Gruppe von Mitarbeitern erarbeitet und gepflegt [Kne98]. Die Größe der Gruppe hängt von der Größe des Unternehmens und dem Betreuungsumfang ab. Dies verhält sich ähnlich für die Erarbeitung und Pflege eines Prozessmusterkatalogs. Die Rolle, die für die Entwicklung und Pflege der Prozessmuster verantwortlich ist, nennen wir im Folgenden „Pattern Designer“. Die Rolle, die Prozessmuster auswählt und anwendet, nennen wir „Pattern User“.

Affinität zum
Knowledge
Management

Die Aufteilung in diese beiden Rollen weist auf die enge Verwandtschaft zwischen Pattern Management und Knowledge Management hin. Muster dienen dazu, erprobtes Wissen zu dokumentieren und dadurch wieder für andere verfügbar zu machen ([CNM95], [GHJ96]). Das Knowledge Management (s. [PR97] für eine Einführung) verfolgt die gleichen Ziele: Implizites (sog. „tacit“) Wissen, das in den Köpfen von Individuen ruht, soll externalisiert, d.h. dokumentiert werden, um für andere Menschen zum Zwecke der Arbeitserleichterung zur Verfügung zu stehen. Ist das Wissen einmal expliziert, muss dieses dann von Individuen internalisiert (d.h. gelernt und verstanden) werden.

Der Pattern Designer

Der Pattern Designer hat die Aufgabe, neue Patterns zu identifizieren und zu dokumentieren. Manns und Rising bezeichneten in [MR01] diese Rolle auch als „Evangelist“ (der die Vorteile von Patterns preist) oder „Dedicated Champion“ (mit expliziter auf Patterns bezogene Stellenbeschreibung).

Der Pattern User

Der Pattern User ist der Anwender von Prozessmustern und somit die Zielgruppe des Pattern Designers. Pattern User sind nach unserem Verständnis Mitarbeiter in Softwareentwicklungsprojekten. Der Pattern User sucht für ein bestimmtes Problem eine geeignete Lösung.

9.3 Einführung und Etablierung

Bevor Prozessmuster in einem Unternehmen oder einem Projekt identifiziert und angewendet werden können, müssen alle Mitarbeiter das Konzept der Prozessmuster kennen und akzeptieren. Die Aktivitäten, um diese Akzeptanz zu erreichen, beschreiben wir in diesem Abschnitt.

Workshops

In Workshops sind Mitarbeiter mit dem Konzept der Prozessmuster vertraut zu machen. Dies können alle Mitarbeiter eines Unternehmens sein oder nur jeweils die Mitarbeiter eines Projekts. Vor Beginn eines jeden Projekts sollte immer ein Auffrischungsworkshop stattfinden. Auf diese Weise können alle Mitarbeiter auf den gleichen Wissensstand gebracht werden und ihnen können neue oder modifizierte Prozessmuster präsentiert werden, die zwischenzeitlich (z.B. im Rahmen anderer Projekte) entstanden sind.

Die Einführung der Arbeit mit Prozessmustern sollte zunächst in einem Pilotprojekt erfolgen und dann nach und nach auf alle Projekte ausgeweitet werden. Hieraus ergeben sich zwei Vorteile: Der Pattern Designer kann Erfahrung mit der Einführung von Prozessmustern sammeln und diese nach und nach auf das Unternehmen und die Mitarbeiter abstimmen. Ferner kann der Prozessmusterkatalog nach und nach erweitert werden. Hierdurch ergibt sich der zweite Vorteil: Die Mitarbeiter können sich nach und nach mit dem Konzept der Prozessmuster auseinandersetzen. Ihnen wird nicht ein komplexes Vorgehensmodell präsentiert, sondern eine überschaubare Menge von Prozessfragmenten mit einem hohen Wiedererkennungswert, da sie ja aus dem eigenen Unternehmen stammen.

Pilotprojekt

Durch Einrichtung eines elektronischen Diskussionsforums können Mitarbeiter in die Entwicklung neuer und Modifikation bestehender Patterns miteinbezogen werden.

Diskussionsforum

Eine hohe Akzeptanz erreicht man auch dadurch, indem man geführte Writers Workshops durchführt, in denen die Mitarbeiter selbst Prozessmuster identifizieren und dokumentieren.

Writers Workshops

9.4 Einsatz

Beim Anwenden von Prozessmustern unterscheiden wir zwischen den Aktivitäten des Pattern Designers und denen des Pattern Users (Abbildung 9-1).

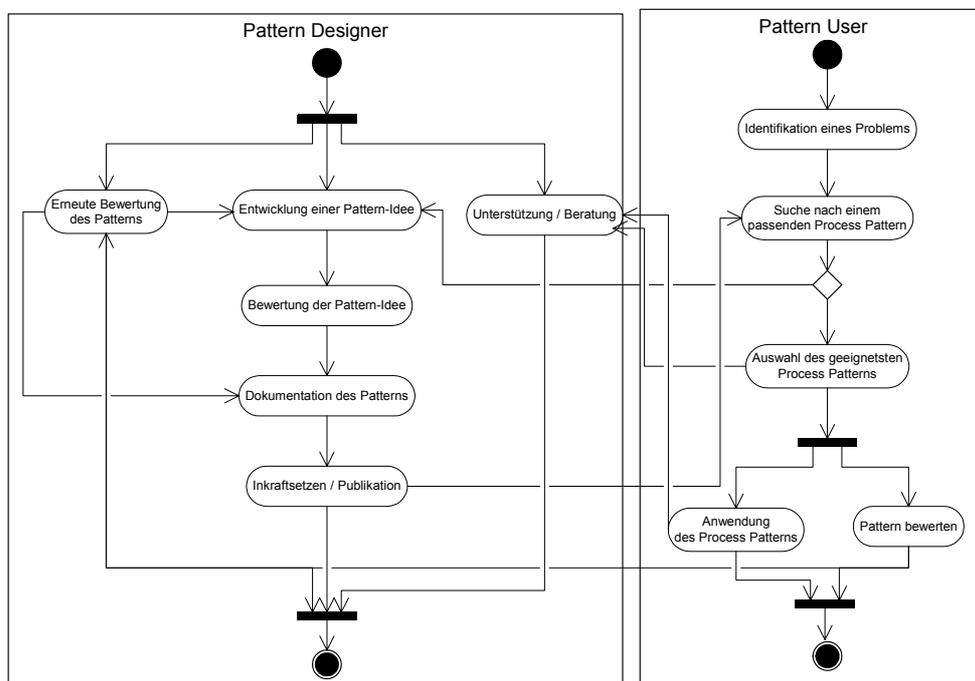


Abbildung 9-1: Aktivitäten bei Entwicklung und Einsatz von Prozessmustern

9.4.1 Aktivitäten des Pattern Designers

Der Prozess zur Entwicklung von Mustern wird im allgemeinen auch als „Pattern Mining“ bezeichnet. Er vollzieht sich angelehnt an [Czi01] (dort existieren nur die ersten drei Aktivitäten) in sechs Aktivitäten.

Entwicklung einer Pattern-Idee	Für das Entwickeln einer Pattern-Idee gibt es verschiedene Herangehensweisen (Abschnitt 2.1.9). Anstoß kann z.B. ein Problem sein, dass vom Pattern User identifiziert wurde, zu dem es jedoch noch kein lösendes Prozessmuster gibt. Eine andere Möglichkeit ist ein geführtes Brainstorming mit Know-How-Trägern, um Muster in deren Vorgehensweisen zu entdecken. Basis aller Techniken ist stets die Befragung von Experten, die implizites Wissen über Prozesse in sich tragen (s. auch [DL99]).
Bewertung der Pattern-Idee	Wurde ein mögliches Pattern identifiziert, muss dieses bzgl. seiner Bedeutung bewertet werden. Die zu stellende Frage lautet immer: „Ist dieses Prozessmuster wirklich ein Pattern?“ Als Daumenregel gilt hier die „Rule of Three“. Diese besagt, dass ein Pattern erst ein Pattern ist, wenn sein Vorkommen in der Praxis mindestens dreimal beobachtet werden konnte. Ein weiteres Kriterium ist, dass das Prozessmuster nur Lösungen mit positivem Ergebnis beschreibt. Ein Prozessmuster, das beschreibt, wie ein Projekt gegen die Wand gefahren wird, auch wenn dies mehrfach in der Praxis beobachtet werden konnte, ist sicherlich kein sinnvolles Prozessmuster. Alexander markierte z.B. zusätzlich seine Pattern mit zwei, einem oder keinem Sternchen ¹⁰ , um die Bedeutung eines Musters hervorzuheben [AIS77].
Dokumentation des Musters	Das Niederschreiben des Musters ist ebenfalls ein wichtiger Schritt im Dokumentationsprozess. Hieran scheitern viele Muster-Autoren, weil sie kein geeignetes Musterschema wählen oder der Schreibstil ungeeignet ist. Dies liegt sicherlich auch darin begründet, dass es eine Menge von Beschreibungsschemata gibt, von denen sich jedoch keines als Quasi-Standard erheben konnte ¹¹ . Aus diesen Gründen haben wir für Prozessmuster ein eigenes Beschreibungsschema definiert (Abschnitt 3.4).
Erneute Bewertung des Musters	Der letzte, von uns neu eingeführte Schritt betrifft die kontinuierliche Verbesserung der dokumentierten Muster. Zur Zeit ist zu beobachten, dass einmal dokumentierte Muster in ihrer Originalversion bestehen bleiben. Varianten oder Muster mit ähnlichen Inhalten werden nach und nach mit Hinweis auf das originäre Muster dokumentiert. Es erfolgt jedoch keine erneute Bewertung des Originalmusters mit der Konsequenz, dass das Originalmuster verbessert, (zugunsten anderer Muster) verworfen oder mit anderen Mustern zu einem neuen besseren Muster verschmolzen wird. Diese unterlassene Bewertung von Mustern führt letztendlich zu einer unübersichtlichen Menge von Mustern unterschiedlicher Qualität und Aktualität. Dies ist aus Sicht des Anwenders eine große Hürde. Die kontinuierliche Verbesserung ist deswegen eine wichtige Komponente im Pattern Management (s. hierzu auch den Deming-Kreis PDCA für kontinuierliche Verbesserung [Dem86]).
Inkraftsetzen/ Publikation	Nach Erstellung oder Überarbeitung eines Prozessmusters muss dieses Prozessmuster freigegeben und publiziert werden. Erst mit der Publikation eines Prozessmusters kann ein Pattern User dieses Prozessmuster (bzw. die neue Version) verwenden.

10. Ein **-Muster enthält eine Lösung, die so oder in ähnlicher Weise zur Lösung des Problems angewendet werden muss. Zu einem *-Muster gibt es gegebenenfalls noch Varianten. Ein *-Muster sollte deshalb nur als mögliche Lösungsvariante betrachtet werden. Ein Muster ohne Sternchen enthält gegebenenfalls noch nicht mal die beste Lösung, sondern nur eine beliebige Lösung.

11. Zwar gaben 65 % aller Befragten in einer Studie von [Czi01] die Nutzung des GoF-Schemas an, dieses ist jedoch nur für Entwurfsmuster geeignet.

Der Pattern Designer ist desweiteren für die Unterstützung und die Beratung des Pattern Users in allen Fragen der Pattern-Anwendung zuständig.

Unterstützung/
Beratung

9.4.2 Aktivitäten des Pattern Users

Der Prozess der Suche und der Auswahl eines geeigneten Musters vollzieht sich in Anlehnung an [Czi01] in vier Schritten (Abschnitt 2.1.9):

Im Rahmen eines Projekts weiß ein Pattern User nicht, wie er in einer bestimmten Situation vorgehen soll. Durch Identifikation der Ausgangssituation (Objekte und Ereignisse) und der Zielsituation (Objekte und Ereignisse) identifiziert er das Problem. Probleme sind stets der zentrale Ausgangspunkt bei der Suche und der Anwendung von Mustern. Die Häufigkeit des Auftretens eines Problems ist ein Anzeichen dafür, dass für dieses Problem ein Muster gefunden werden sollte.

Identifikation eines
Problems

Durch die Identifikation des Problems kann der Pattern User nun nach einem geeigneten Prozessmuster suchen. Wurde kein Prozessmuster gefunden, kann der Pattern User dem Pattern Designer das identifizierte Problem mitteilen. Der Pattern Designer kann das Problem daraufhin in den Prozessmusterkatalog aufnehmen und beginnen, lösende Prozessmuster für dieses Problem zu suchen.

Suche nach einem
passenden Prozess-
muster

Da die Suche nach Mustern immer mit der Identifikation eines Problems (Schritt 1) beginnt, haben wir Probleme als primäres Strukturierungskriterium in Musterkatalogen oder Sprachen eingeführt. Hierdurch wird die Effizienz und Übersichtlichkeit der Suche erhöht. Prozessmuster dienen dann erst als sekundäres Strukturierungskriterium (Schritt 2). Wurden beispielsweise einem Problem mehrere variante Muster zugeordnet, die das Problem zu lösen vermögen, kann der Anwender auf diese Varianten schnell zugreifen. Hierzu ist das geeignetste Muster auszuwählen (Schritt 3) und anzuwenden (Schritt 4).

Ein Prozessmuster beschreibt für ein Problem in einem Kontext eine erprobte Lösung. Da Prozessmuster als Prozessmodelle von der Realität abstrahieren, müssen Prozessmuster für die praktische Anwendung an die vorherrschenden Verhältnisse angepasst werden. Erfolgen diese Anpassungen in großem Umfang, steht der Pattern-Modellierer vor zwei Alternativen: Entweder er wirft das alte Prozessmuster weg und ersetzt es durch das neue Muster, da sich gezeigt hat, dass die neue Vorgehensweise besser ist als die alte. Die zweite Alternative ist dann gegeben, wenn beide Vorgehensweise erprobt sind und für den Pattern-Modellierer Bedeutung haben. In diesem Fall sollte der Pattern-Modellierer das neue Muster aufnehmen und als Variante des bestehenden Musters definieren.

Wurden mehrere lösende Prozessmuster (d.h. Prozessvarianten) gefunden, muss der Pattern User eine Auswahl zwischen diesen treffen. Durch Begutachtung des Prozesses und weiterer Details wie der „Diskussion“ (als Element des Prozessmusters) kann der Pattern User seine Entscheidung fundieren.

Auswahl des
geeignetsten
Prozessmusters

Im Rahmen der Anwendung wird ein Prozessmuster vom Anwender instanziiert. Die Instanzierung führt dazu, dass dem abstrakten Prozessmuster Details (z.B. weitere Prozessschritte, weitere Objekte, Ereignisse) vom Anwender hinzugefügt werden. Diese Details sind aber nur für den Anwender ersichtlich. Dies ist erlaubt, solange die Vorgaben des Prozessmusters nicht verletzt werden. Das Prozessmuster im Prozessmusterkatalog verbleibt in seiner abstrakten Form. Erweist sich die detaillierte Muster-Instanz jedoch ebenfalls als Musterlösung,

Anwendung des
Prozessmusters

kann dieses Prozessmuster als Verfeinerung des abstrakteren Musters (falls die Verfeinerungsbedingungen gelten) oder als völlig eigenständiges Prozessmuster in den Katalog aufgenommen werden.

Die Process Pattern Workbench erlaubt es zu kennzeichnen, welche Prozessmuster in welchen Projekten ausgewählt und angewendet wurden. Hierdurch kann beispielsweise einem Projektmitarbeiter 1 signalisiert werden, dass ein anderer Projektmitarbeiter 2 an einem bestimmten Problem arbeitet und bestimmte Objekte erstellt bzw. bestimmte Ereignisse auslösen wird. Der Projektmitarbeiter 1 kann seine Aktivitäten dann darauf abstimmen (z.B. warten, bis Projektmitarbeiter 2 mit der Anwendung seines Prozessmusters fertig ist).

Muster bewerten

Durch die Anwendung eines Prozessmusters ist der Pattern User in der Lage, ein Prozessmuster nach verschiedenen Kriterien zu beurteilen. Solche Kriterien sind z.B. die Verständlichkeit des Musters, korrekte Darstellung des Kontexts und des Prozesses, fehlende Beziehungen usw. Diese Erfahrungen kommuniziert der Pattern User an den Pattern Designer, der daraufhin entsprechende Änderungen an dem Prozessmuster vornehmen kann.

10 Fazit und Ausblick

In diesem Kapitel fassen wir unsere Ergebnisse zusammen. Anschließend überprüfen wir, inwieweit die an die Arbeit gestellten Anforderungen erfüllt wurden. Zum Abschluss geben wir einen Ausblick auf zukünftige Arbeiten.

10.1 Zusammenfassung

Zu Beginn der Arbeit beschrieben und diskutierten wir grob Vor- und Nachteile existierender Prozessmodellierungssprachen und Vorgehensmodelle. Wir stellten fest, dass traditionelle Vorgehensmodelle aufgrund ihrer Rigidität und Komplexität eine flexible und dynamische Anpassung der Prozesse oft nicht zulassen. Wir identifizierten Prozessmuster als mögliche Alternative zur Beschreibung von Prozessen und Vorgehensmodellen. Wir stellten ferner fest, dass Prozessmuster eine unzureichende formale Basis besitzen, die jedoch für die Beschreibung von Prozessen unerlässlich ist. Als Ziel unserer Arbeit formulierten wir die Entwicklung einer Sprache zur Modellierung von Prozessmustern und deren Beziehungen, die die dynamische Anpassung von Vorgehensmodellen in einem Projekt zulässt.

Kapitel 1

Für einen Überblick über den derzeitigen Stand der Technik erläuterten und bewerteten wir Arbeiten und Ansätze aus den Bereichen (Prozess-)Muster, Prozessmodellierungssprachen, Semantiken von UML-Aktivitätsdiagrammen und Vorgehensmodelle: (i): Im Bereich der (Prozess-)Muster wurde klar, dass keine einheitliche Terminologie existiert. Dies mindert den Einfluss von (Prozess-)Mustern als Kommunikationsmittel. (ii): UML sticht als Prozessmodellierungssprache hervor, die zwar nur semiformal ist, aber eine hohe Praxistauglichkeit, einen hohen Bekanntheits- und Akzeptanzgrad und einen eingebauten Mechanismus zur Erweiterung besitzt. (iii): Ansätze zur Formalisierung der Semantik von Aktivitätsdiagrammen basieren bis auf eine Ausnahme auf der Notation von UML-Aktivitätsdiagrammen, wodurch keine vollständige Semantikdefinition erreicht wird. (iv): Bei der Betrachtung verschiedener rigider und agiler Vorgehensmodelle kamen wir zu dem Ergebnis, dass die Beschreibung der Prozesse zumeist starr ist und nur unzureichend die Beziehungen zwischen den Prozessen angeben werden.

Kapitel 2

In den nachfolgenden Kapiteln wurde die Sprache PROPEL in vier Schritten (Konzepte, Syntax, Semantik und Notation) entwickelt. Im ersten Schritt führten wir den Leser in die Konzepte von PROPEL ein, indem wir die Konzepte – zunächst informal – definierten. Dies war notwendig, da noch keine einheitlichen, aufeinander abgestimmten Begriffsdefinitionen vorlagen. Hierzu zählten beispielsweise die Konzepte Prozessmuster, Ergebnismuster, Prozessmusterkatalog und die Prozessmusterbeziehungen. Auf Basis dieser Konzepte definierten wir Schemata für Prozessmuster und Problem.

Kapitel 3

Als nächstes definierten wir die Syntax von PROPEL formal durch Erweiterung der Sprache UML um neue Metaklassen, Metaassoziationen und -attribute und um OCL-Constraints. Die UML gilt als „Lingua Franca“ der Software-Engineering-Community, d.h. sie ist als Modellierungs- und Kommunikationsmittel in der Praxis akzeptiert. Diesen Vorteil nutzten wir für unsere Beschreibungssprache.

Kapitel 4

- Kapitel 5 Aufbauend auf der Syntax definierten wir die formale Semantik von PROPEL. Da die UML keine formale Semantik besitzt, ergänzten wir die Sprache PROPEL um eine formale Semantik durch Abbildung der Syntax auf die semantische Domäne der Petri-Netze. Durch Verwendung der Petri-Netz-Semantik konnten Eigenschaften und Verhalten von einzelnen und durch Beziehungen verknüpften Prozessmustern definiert werden.
- Kapitel 6 Im vierten und letzten Schritt definierten wir für neu eingeführte Elemente eine Notation und bildeten diese auf die Syntax von PROPEL ab. Die bestehende UML-Notation wurde übernommen.
- Kapitel 7 Mit der auf diese Weise entwickelten Sprache PROPEL erarbeiteten wir einen Prozessmusterkatalog, der dem Leser die Konzepte der Beschreibungssprache an einem komplexen, praxisnahen Beispiel präsentiert. Hierzu verwendeten wir einen Ausschnitt aus dem Rational Unified Process, den wir mit Hilfe von Prozessmustern beschrieben. Das Beispiel zeigte, dass PROPEL sehr gut dazu geeignet ist, Prozessmuster und deren Beziehungen untereinander zu modellieren. Der Anwender von Prozessmustern erhält syntaktisch und semantisch Hilfestellung, in welchen Kombinationen er Prozessmuster ausführen kann. Diese Hilfestellung ist in den bekannten Vorgehensmodellen wie z.B. RUP nicht vorhanden.
- Kapitel 8 Eine prototypische Realisierung der Konzepte von PROPEL führten wir mit der Process Pattern Workbench durch. Mit Hilfe der Process Pattern Workbench können Prozessmuster dokumentiert, präsentiert, gesucht, modifiziert und deren Anwendung aufgezeichnet werden. Die Workbench ist insbesondere dann ein Vorteil, wenn der Prozessmusterkatalog groß ist und damit die Gefahr der Unübersichtlichkeit wächst. Ein Teil des Prozessmusterkatalogs wurde mit Hilfe der Process Pattern Workbench umgesetzt.
- Kapitel 9 Neben Bereitstellung einer Sprache (PROPEL) und einem Werkzeug (Workbench) entwickelten wir eine Systematik zur Handhabung von Prozessmustern in der Praxis. Dieser Aspekt wird in den meisten Arbeiten zum Thema (Prozess-)Muster ignoriert. Beim Process Pattern Management geht es deshalb darum, zu klären, welche Maßnahmen ein Unternehmen ergreifen muss, wenn es Prozessmuster einsetzen will. Das Process Pattern Management beschreibt Aktivitäten zum Aufbau einer geeigneten Prozessmuster-Infrastruktur (z.B. ein elektronischer Prozessmusterkatalog), zur organisatorischen Einbettung, Einführung und Etablierung von Prozessmustern und dem letztendlichen Einsatz von Prozessmustern.

10.2 Bewertung

Tabelle 10-1 fasst zusammen, welche der von uns definierten Anforderungen erfüllt wurden und welche nicht. Anschließend bewerten wir ausführlich, inwieweit jede der Anforderungen erfüllt wurde.

Anforderung	erfüllt?
Anforderung 1: Leichte Handhabbarkeit und Intuitivität der Beschreibungssprache	eingeschränkt
Anforderung 2: Definition einer formalen Syntax und Semantik der Beschreibungssprache	ja
Anforderung 3: Definition eines Beschreibungsschemas für Prozessmuster	ja
Anforderung 4: Begriffsbildung	ja
Anforderung 5: Definition eines Klassifizierungsschemas für Prozessmuster	ja
Anforderung 6: Entwicklung eines Werkzeugs zur Handhabung von Prozessmustern	ja
Anforderung 7: Entwicklung einer Methodik zur Handhabung von Prozessmustern	ja

Tabelle 10-1: Bewertung der zu erfüllenden Anforderungen

Definition einer formalen Syntax und Semantik der Beschreibungssprache

Anforderung 1

PROPEL basiert auf der Sprache UML, die insbesondere durch ihre Praxistauglichkeit und der hohen Akzeptanz von Praktikern ausgezeichnet ist. Da PROPEL die UML um Syntax und Semantik ergänzt, aber nicht verändert, gelten also UML-Syntax und Semantik weiter. Der UML-kundige Leser muss sich daher nur noch die neuen PROPEL-Elemente aneignen. Syntax, informale Semantik und Notation haben wir analog zur UML-Spezifikation definiert, damit die Integration mit bestehenden Elementen eindeutig ist. Das Verstehen der formalen Semantik erfordert allerdings einen höheren Lernaufwand, da der Leser Syntax und Semantik von Petri-Netzen kennen bzw. lernen muss. Für Syntax und informale Semantik kann die Anforderung also als erfüllt gelten, für die formale Semantik nur eingeschränkt.

Definition einer formalen Syntax und Semantik der Beschreibungssprache

Anforderung 2

Die Syntax von PROPEL wurde formal durch das Metamodell und zusätzlich durch eine kontextfreie Grammatik angegeben. Die Semantik von PROPEL wurde durch Abbildung der Syntax auf die formale Semantik von Petri-Netzen formal definiert. Die Anforderung kann daher als erfüllt gelten.

Definition eines Beschreibungsschemas für Prozessmuster

Anforderung 3

Im Rahmen der Klärung der Konzepte von PROPEL haben wir definiert, welche Elemente zur Beschreibung von Prozessmustern notwendig sind. Hierzu gehören die Elemente Name, Version, Domäne, Phase, Aspekt, Katalog, Synonyme, Problem, Kontext, Prozess, Rolle, Beziehungen zu anderen Mustern, Beispiel und Diskussion. Darüber hinaus haben wir ein Schema zur Beschreibung von Problemen definiert. Dies umfasst die Elemente Name, Version, Kontext, Beschreibung und lösende Prozessmuster. Die Anforderung kann daher als erfüllt gelten.

Anforderung 4 Begriffsbildung

In Kapitel 2 haben wir die im Bereich der (Prozess-)Muster häufig verwendeten Begriffe erläutert und ihre Eindeutigkeit und Genauigkeit bewertet. Auf Basis dieser Ergebnisse haben wir Begriffe wie Prozessmuster, Prozessmusterkatalog und Prozessmusterbeziehung in Kapitel 3 neu definiert und in den Kapiteln 4 und 5 formal fundiert. Die Anforderung 4 gilt daher als erfüllt.

Anforderung 5 Definition eines Klassifizierungsschemas für Prozessmuster

Im Rahmen der Klärung der Konzepte von PROPEL haben wir definiert, welche Dimensionen das Klassifizierungsschema für Prozessmuster umfasst, nämlich Domäne, Phase, Aspekt und Katalog. Für jede dieser Dimensionen haben wir mögliche Ausprägungen angegeben. Die Anforderung kann daher als erfüllt gelten.

Anforderung 6 Entwicklung eines Werkzeugs zur Handhabung von Prozessmustern

Mit der Process Pattern Workbench haben wir eine prototypische Implementierung der Konzepte von PROPEL vorgenommen. Der Prototyp erleichtert die Modellierung und Anwendung von Prozessmustern. Die Anforderung kann daher als erfüllt gelten.

Anforderung 7 Entwicklung einer Methodik zur Handhabung von Prozessmustern

Das Process Pattern Management umfasst Mechanismen zum Aufbau einer geeigneten technischen Infrastruktur, einer geeigneten Aufbauorganisation und geeigneten Maßnahmen zu Einführung, Etablierung und Einsatz von Prozessmustern. Die Anforderung kann daher als erfüllt gelten.

10.3 Ausblick

Weiterführende Arbeiten im Bereich der Prozessmuster konzentrieren sich auf die Identifikation weiterer Prozessmusterbeziehungen, die Weiterentwicklung der Process Pattern Workbench, die Anpassung an die kommende UML 2.0, auf den Einsatz der Sprache und des Werkzeugs in Projekten und auf semantische Variationen.

Identifikation
weiterer Prozess-
musterbeziehungen

Auf Basis der in dieser Arbeit definierten Prozessmusterbeziehungen können weitere komplexere Prozessmusterbeziehungen identifiziert werden. Hierzu zählt z.B. die Definition einer Beziehung, die einen abwechselnden Kontrollfluss zwischen zwei Prozessmustern oder eine Synchronisation zweier Prozessmuster durch sogenannte Synchronisationspunkte zulässt ([Eck95], [WMC96]).

Weiterentwicklung
der Process Pattern
Workbench

Die Process Pattern Workbench kann wie in Abschnitt 8.4 dargestellt um einige sinnvolle Funktionalitäten erweitert werden. Hierzu gehören z.B. das Anlegen verschiedener Prozessmusterkataloge in einer Workbench und Reportingfunktionalitäten für die Auswertung der Nutzung der Workbench und ihrer Elemente. Hierdurch können z.B. häufige Abfolgen von Prozessmustern identifiziert werden, also Muster von Prozessmusterabfolgen. Diese Muster geben wiederum Hinweise auf sinnvolle Prozessmusterkombinationen.

Die Konzepte dieser Arbeit können an die kommende UML-Version 2.0 angepasst werden. Diese Anpassung betrifft insbesondere die Syntax und Semantik der Sprache. Da die UML 2.0 sich an der Petri-Netz-Semantik orientiert, liegen Syntax und Semantik näher beieinander. Ob dies zu einer Verbesserung der sprachlichen Konzepte führt, ist dann festzustellen.

Anpassung an UML 2.0

Zum Zwecke der praktischen Validierung können Sprache und gegebenenfalls die Workbench in verschieden großen Softwareentwicklungsprojekten eingesetzt werden. Hierdurch lassen sich gegebenenfalls weitere Entwicklungspotentiale der Sprache PROPEL identifizieren.

Einsatz von Sprache und Workbench in Projekten

Die Semantik kann um weitere Aspekte wie z.B. semantische Variationspunkte ergänzt werden. Semantische Variationspunkte identifizieren Semantiken, die bei der Instanziierung eines Prozessmusters variieren, d.h. verschiedene Ausprägungen haben können. Hierdurch kann die Flexibilität der Sprache weiter erhöht werden.

Definition von semantischen Variationspunkten

Anhang

A Ergänzungen zu den Grundlagen

In diesem Kapitel werden Details zu den Grundlagen aus Kapitel 2 angegeben.

A.1 Muster

A.1.1 Musterschemata

In diesem Abschnitt wird eine Auswahl von Musterschemata detailliert vorgestellt. Eine vergleichende Übersicht findet sich in Abschnitt 2.1.4.

Das Alexandersche Schema ist die ursprüngliche Form aller Patterns. Sie wurde von Alexander zur Beschreibung von wiederkehrenden Gebäude-Architekturen verwendet. Alexander beschreibt seine Patterns mit folgenden Elementen [AIS77]:

Alexandersches Schema

Name	Name des Architekturpatterns
Abbildung	Zeigt ein Patternbeispiel.
Einführung	Erläutert den Kontext und wie das Pattern mit anderen Patterns zu einem größeren Pattern zusammengesetzt werden kann.
Problem	Problemüberschrift Problembeschreibung
Lösung	Anweisungen, um das Problem zu lösen.
Diagramm	Darstellung der Lösung in Diagrammform.
Benutzte Patterns	Kleinere Patterns, die zur Ausführung dieses Patterns benutzt werden.

Tabelle A-1: Alexandersches Schema

Das Coplien-Schema ist dem Alexanderschen Schema entlehnt, dient aber zur Beschreibung von Design Pattern, Organizational Patterns oder Prozessmustern. Sie unterscheidet verschiedene Aspekte eines Patterns genauer als das Alexandersche Schema: Der Abschnitt „Einführung“ heißt hier „Kontext“. Um zu verdeutlichen, dass sich durch Anwendung eines Patterns der Kontext ändert, wird der Abschnitt „Resultierender Kontext“ eingeführt. Coplien unterscheidet wie Alexander auch zwischen „Problemüberschrift“ und „Problembeschreibung“, nennt diese Abschnitte jedoch „Problem“ und „Pro und Contra“ (im engl. „Forces“). Der Abschnitt „Grundprinzip“ (engl. „Rationale“) ist neu hinzugefügt.

Coplien-Schema

Name	Der Name eines Patterns spiegelt den Verwendungszweck des Patterns wider. Er sollte möglichst kurz sein. Häufig verwendete Patterns erhalten im Laufe der Zeit meistens synonyme Bezeichnungen. In diesem Fall kann ein zusätzliches Patternelement („Synonym“, „Auch bekannt als“ oder „Alias“) dem Pattern hinzugefügt werden.
Problem	Das Problem beschreibt die Problemstellung bzw. die Ziele, die mit dem Einsatz des Patterns gelöst bzw. erreicht werden sollen. Die Ziele stehen in Wechselwirkung mit Kontext und Pro und Contra des Patterns.
Kontext	Der Kontext beschreibt (Vor-)Bedingungen, unter denen das Problem auftritt und die Lösung angewendet werden kann. Der Kontext bedingt also die Anwendbarkeit des Patterns. Bezogen auf Design Patterns kann der Kontext als Konfiguration des Systems verstanden werden, die vorausgesetzt wird, um das Pattern anwenden zu können.
Pro und Contra	Der Abschnitt dient dazu, das Problem zu verdeutlichen und den Trade-off (z.B. in Form von Nebeneffekten) zu beschreiben, der mit Einsatz des Patterns erkaufte wird, d.h. Pro- und Contra-Argumente für bzw. gegen den Einsatz des Patterns. Die Auswahl des richtigen Patterns soll hiermit erleichtert werden. Die englische Bezeichnung „Forces“ rührt von der architektonischen Herkunft der Patterns her. „Forces of Gravity“ sind Schwerkraften, die bei Gebäuden ausbalanciert werden müssen.
Lösung	Die Lösungsbeschreibung kann Text, Diagramme oder Abbildungen enthalten.
Resultierenden Kontext	Die Anwendung eines Patterns verändert den Kontext und hinterlässt einen neuen, resultierenden Kontext. Der resultierende Kontext beschreibt daher die Konsequenzen, die aus der Anwendung eines Patterns resultieren: Nebeneffekte, (un)balancierte Kräfte („Forces“), neue Probleme. Der resultierende Kontext gibt auch Hinweise darauf, welche Patterns anschließend (durch Vergleich des Resultierenden Kontexts des Vorgänger-Patterns und des Kontexts des Nachfolger-Patterns) angewendet werden können. Coplien bezeichnete deswegen Kontexte auch als Mittel, um Patterns zu einer Pattern Language „zu weben“ [Cop96].
Grundprinzip	Das Grundprinzip liefert eine sinnhafte Begründung der Schritte und Regeln eines Patterns. Im Gegensatz zur Lösung, die die Außensicht auf ein System liefert, dient das Grundprinzip dazu, zu begründen, wie das Pattern funktioniert, warum es funktioniert und warum das Pattern gut ist.

Tabelle A-2: Coplien-Schema

GoF-Schema Das GoF-Schema wurde in [GHJ96] als Standard für Design Patterns etabliert. Ein etwas abgewandeltes Schema findet man in [BMR96]. Das der Lösung entsprechende Element heißt hier Struktur, da die Struktur des Entwurfs, meistens in Form von Klassendiagrammen, dargestellt wird. Desweiteren gibt es designspezifische Beschreibungselemente wie Teilnehmer, Interaktion, Implementierung und Beispielcode.

Mustername und Klassifizierung	Der Name eines Patterns spiegelt den Verwendungszweck des Patterns wider. Die Klassifizierung dient zur systematischen Einordnung des Patterns. Klassifikationskriterien sind im GoF-Katalog die Aufgabe und der Gültigkeitsbereich eines Patterns.
Zweck	Der Zweck beschreibt die Problemstellung bzw. die Ziele, die mit dem Einsatz des Patterns gelöst bzw. erreicht werden sollen.
Auch bekannt als	Synonyme des Patterns
Motivation	Die Motivation enthält ein Szenario, in dessen Rahmen das Entwurfsproblem geschildert und die Lösung durch das Pattern erläutert wird. Die Motivation dient zur Konkretisierung des Patterns.
Anwendbarkeit	Dieser Abschnitt beschreibt, in welchen Situationen das Design Pattern angewendet werden kann.
Struktur	Mit Struktur ist eine grafische Repräsentation der Klassen gemeint, die das Problem lösen sollen. Die Repräsentation kann beispielsweise durch UML-Diagramme erfolgen.
Teilnehmer	Teilnehmer sind die beteiligten Klassen und Objekte und ihre Zuständigkeiten.
Interaktionen	Dieser Abschnitt beschreibt, wie die Klassen und Objekte miteinander interagieren.
Konsequenzen	Dieser Abschnitt beschreibt, welche Resultate aus der Anwendung des Patterns zu erwarten sind. Dieser Abschnitt bildet das Gegenstück zum Abschnitt Anwendbarkeit.
Implementierung	Dieser Abschnitt beschreibt, wie das Pattern konkret, d.h. programmiertechnisch, umgesetzt werden kann.
Beispielcode	Entsprechend zum Abschnitt Implementierung können Codefragmente angegeben werden. Beispiele können der Praxis entnommen werden (siehe auch „Bekannte Verwendungen“).
Bekannte Verwendungen	Der Abschnitt liefert einen Beweis für die erprobte Anwendung eines Patterns in der Praxis. Der Abschnitt kann, weiter detailliert, auch als Beispiel (s. Beispielcode) dienen.
Verwandte Patterns	In diesem Abschnitt werden Beziehungen zu anderen Patterns aufgezeigt. Beziehungen können durch kompatible Kontexte entstehen (Vorgänger-Nachfolger-Beziehung), durch konfligierende Forces oder Solutions (Alternative-Beziehung) oder durch ergänzende Lösungen (Komposit-Beziehung).

Tabelle A-3: GoF-Schema

Das Störrle-Schema in [Stö00] ist neben dem ZEN-Schema eines der wenigen Schemata zur Beschreibung von Prozessmustern. Das Beschreibungselement Lösung heißt hier Prozess. Teilnehmer sind statt Objekte und Klassen (vgl. GoF-Schema) alle an einem Prozess teilnehmenden Entitäten. Das Störrle-Schema enthält folgende Beschreibungselemente:

Störrle-Schema

Titel	Name des Patterns
Klassifizierung	Klassifizierung des Pattern nach Phase, Zweck (engl. Purpose) und Anwendungsbereich (engl. Scope).
Verwandte Patterns	Beziehungen zu anderen Patterns.
Zweck	Problembeschreibung
Diskussion	Enthält Motivation und Konsequenzen des Patterns.
Teilnehmer	Rollen, Ressourcen, Techniken, Werkzeuge und Dokumenttypen, die für den beschriebenen Prozess benötigt werden.
Anwendbarkeit	Vorbedingungen für die Anwendung des Patterns.
Ergebnisse	Nachbedingungen bzw. geänderte oder erzeugte Artefakte des Patterns.
Prozess	Beschreibung eines Prozesses, der das Problem (s. „Zweck“) löst.

Tabelle A-4: Störrle-Schema

Die Elemente „Anwendbarkeit“ und „Ergebnisse“ entsprechen dem Element „Kontext“, also den Vor- und Nachbedingungen, die bei Anwendung des Patterns gelten. Weder diese beiden Elemente noch das Element „Prozess“ werden formal definiert.

ZEN-Schema

Das ZEN-Schema (von uns so vereinfacht genannt nach dem Forschungsprojekt ZEN, in dessen Rahmen dieses Schema entstanden ist) dient zur Beschreibung von Prozessmustern. Es enthält ein zusätzliches Beschreibungselement „Realisierte Aktivität“. Eine Aktivität kann durch mehrere Prozessmuster gelöst („realisiert“) werden. Eine solche Aktivität befindet sich also auf einer höheren Abstraktionsebene als das lösende Prozessmuster. Das ZEN-Schema enthält folgende Beschreibungselemente [GMP01b]:

Name	Name des Patterns
Autor	Autor des Patterns
Version	Versionsnummer des Patterns
Auch bekannt als	Synonyme
Keywords	Schlagworte zur Beschreibung von Kontext und Zweck des Patterns
Zweck	Motivation und Darstellung des Problems.
Problem	Problem, das durch das Pattern gelöst wird inkl. der dadurch adressierten Forces.

Tabelle A-5: ZEN-Schema

Lösung	Beschreibt Aktivitäten zur Lösung des Problems. Gegebenenfalls wird eine Reihenfolge der Aktivitäten angegeben oder alternative Aktivitäten. Möglich sind auch die Angabe von zu verwendenden Methoden oder zu beachtenden Richtlinien.
Realisierte Aktivität	Aktivität, die durch das Prozessmuster beschrieben wird.
Initialer Kontext	Status des Projekts, d.h. Status der betreffenden Artefakte.
Resultieren-der Kontext	Status des Projekts, d.h. Status der betreffenden Artefakte nach Anwendung des Patterns.
Pro und Kontra	Diskussion der Ergebnisse, Konsequenzen und Trade-Offs des Patterns.
Beispiel	Illustriert Akzeptanz und Nützlichkeit des Patterns.
Verwandte Patterns	Liste von Patterns, die Alternativen zu diesem Pattern bilden und in Kombination mit diesem verwendet werden können.

Tabelle A-5: ZEN-Schema (Fortgesetzt)

A.1.2 Beziehungen zwischen Mustern

Nachfolgend werden die Definitionen der Musterbeziehungen von Zimmer und Noble im Detail vorgestellt.

Zimmer

Beziehung	Bedeutung
X uses Y in its solution	Ein Subproblem des Musters X ist dem Problem des Musters Y ähnlich. D.h. das Muster X benutzt Muster Y für seine Lösung. D.h. die Lösung von Y (Z.B. Klassen) ist ein Teil der Lösung von X.
X is similar to Y	Beide Muster adressieren ein ähnliches Problem, bieten aber unterschiedliche Lösungen. Die Ähnlichkeit manifestiert sich auch in der Klassifizierung der Muster.
X can be combined with Y	Zwei Muster werden zur Lösung ein und desselben Problems kombiniert. Die Beziehung darf nicht mit der Uses-Beziehung verwechselt werden, d.h. Muster X benutzt nicht Muster Y.

Tabelle A-6: Klassifikation von Entwurfsmuster-Beziehungen nach Zimmer

Noble

Beziehung	Bedeutung
Primär	
Uses	Die Uses-Beziehung bedeutet, dass ein Muster ein anderes Muster zur Lösung seines Problems nutzt. Hierdurch kann die Komposition eines Musters dargestellt werden. Mit der Beziehung geht auch oft einher, dass das nutzende Muster eher grobgranularer Natur und das genutzte Muster eher feingranularer Natur ist. Bei kompositen Mustern ist dies in der Regel der Fall. In Analogie zur objektorientierten Programmierung entspricht die Beziehung der Komposition.
Refines	Die Refines-Beziehung bedeutet, dass ein Prozessmuster ein anderes Prozessmuster verfeinert, d.h. ein Muster wird als Spezialisierung des anderen, allgemeineren Musters aufgefasst. In Analogie zur objektorientierten Programmierung entspricht diese Beziehung der Vererbung.
Conflicts	Zwei Muster stehen in einer Conflicts-Beziehung, wenn diese zu einem ähnlichen Problem alternative Lösungen anbieten, die sich gegenseitig ausschließen. Durch Abbildung der Conflicts-Beziehung in einer Muster Language wird dem Nutzer eine differenziertere Auswahl eines Musters ermöglicht. Durch Vergleich zweier Alternativen wird das Abwägen zwischen Für und Wider eines Musters erleichtert.
Sekundär	
Used by	Used by ist Inverse zu der Beziehungen Uses.
Refined by	Refined by ist Inverse zu der Beziehungen Refines.
Variants	<p>Stellt ein Muster eine Variante eines anderen Musters dar, so kann sich der variante Teil auf die Lösung (solution variants) oder auf das Problem (problem variants) beziehen. Das Auftreten von Lösungsvarianten ist häufiger als das von Problemvarianten. Dies rührt daher, dass es einfacher ist, zu einem Problem mehrere variante Lösungen als zu mehreren varianten Problemen eine Lösung finden zu können.</p> <p>Bei der Solution-Variants-Beziehung handelt es sich um die Kombination der Refines- mit der Conflicts-Beziehung. Bei der Problem-Variants-Beziehung handelt es sich um die Kombination der Uses- mit der Refines-Beziehung.</p>
Variant Uses	Die Variant-Uses-Beziehung beschreibt, dass von zwei Mustervarianten X und X' nur X' ein anderes Muster Y nutzt. Noble hat diese Beziehung zwar identifiziert, verwendet sie jedoch nicht für sein Ordnungsschema, da er sie für überflüssig hält. Anstatt dieser verwendet er die Uses-Beziehung.
Similar	Die Similarity-Beziehung fungiert als Auffangbecken für Muster-Beziehungen, die nicht durch die anderen Beziehungen abgedeckt werden.

Tabelle A-7: Klassifikation von Entwurfsmusterbeziehungen nach Noble

Beziehung	Bedeutung
Combines	Diese Beziehung beschreibt, dass zwei Muster kombiniert werden, um ein einzelnes Problem zu lösen, welches nicht durch irgendein anderes bestehendes Muster adressiert wird. Es gibt zwei Möglichkeiten, die Combines-Beziehung durch die primäre Uses-Beziehung abzubilden: In einfachen Fällen nimmt man ein grobgranulares Muster, welches das Gesamtproblem adressiert. Dieses grobgranulare Muster nutzt über die Uses-Beziehung dann ein feingranulares Muster zur Lösung eines Teilproblems. In komplexen Fällen sucht man ein Muster oder definiert ein neues Muster, das das Gesamtproblem adressiert und die Lösung skizziert und auf die Muster verweist, die in Kombination diese Lösung darstellen.
Require	Ein Muster X erfordert ein anderes Muster Y, wenn Y die Voraussetzung darstellt, um das von X adressierte Problem zu lösen. Diese Beziehung wird durch die Uses-Beziehung abgebildet. Der Unterschied zwischen der Uses- und der Requires-Beziehung liegt in der Reihenfolge der Anwendung: Erfordert Muster X das Muster Y, muss Y vor X angewendet werden. Im Falle der Uses-Beziehung kann das Muster X schon vor dem Muster Y angewendet werden, jedoch bis zur kompletten Abarbeitung der Lösung. Hierzu ist wiederum die Anwendung von Y vonnöten.
Tiling	Die Tiling-Beziehung drückt aus, dass ein Muster wiederholt angewendet wird, um ein einzelnes Problem zu lösen. Abgebildet wird diese Beziehung durch die Refines- und Uses-Beziehung.
Sequence of elaboration	Die Beziehung beschreibt eine Abfolge von Mustern, angefangen mit kleinen, einfachen Mustern, über Muster mit wachsender Komplexität bis hin zu großen Mustern, die große negative Auswirkungen auf die Komplexität des Systems haben.

Tabelle A-7: Klassifikation von Entwurfsmusterbeziehungen nach Noble (Fortgesetzt)

A.1.3 Gütekriterien

Lea definierte folgende Gütekriterien für Muster [Lea94]:

Jedes Muster kapselt ein Problem und dessen Lösung für eine bestimmte Domäne. Die Beschreibung eines Musters muss präzise genug sein, um über Anwendbarkeit und Ergebnis des Musters Aufschluss zu geben.	Kapselung
Ein Muster sollte abstrakt genug sein, um verschiedene Lösungen generieren zu können. Je nach Einsatzsituation wird ein Muster unterschiedlich instanziiert. Das Muster gibt lediglich vor, was bei der Lösung nicht vergessen werden darf.	Generativität
Die Problembeschreibung eines Musters enthält die „Forces“, d.h. bestimmte Bedingungen, die als Pro- und Contra-Argumente abzuwägen sind. Idealerweise werden diese „Forces“ durch Einsatz des Musters ausgeglichen.	Gleichgewicht
Muster repräsentieren eine Abstraktion von empirischen Erhebungen oder Wissen auf verschiedenen Granularitätsstufen.	Abstraktion
Ein Muster sollte seine Erweiterung durch weitere Muster zulassen. Diese Teilmuster können wiederum durch weitere Muster erweitert werden. Diese Eigenschaft bezeichnet Lea auch als „fraktal“.	Offenheit

Komponierbarkeit	Muster sind hierarchisch miteinander verknüpft. Z.B. können grobgranulare durch feingranularere Muster komponiert werden. Durch Beziehungen wird diese Komponierbarkeit angezeigt.
Bewertung	Bestehende Beschreibungsschemata sind allerdings nicht in der Lage, alle diese Eigenschaften zu erfüllen. Insbesondere die Offenheit und Komponierbarkeit von Mustern erfordert eine präzise Definition, wie Muster erweitert oder komponiert werden können. Derzeitige Beschreibungsschemata lassen eine solche Definition nicht zu. Durch Definition der Process Pattern Description Language PROPEL (Kapitel 3, 4 und 6) können Prozessmuster modelliert werden, die diese Gütekriterien erfüllen.

A.2 Grundbegriffe der Softwareprozessmodellierung

[DNR91], [FH92], [CFF92] sind Standardwerke zur Definition von Konzepten und Terminologie der Softwareprozessmodellierung. Insbesondere Lonchamp hat mit dem „universe of discourse“ ein konzeptuelles und terminologisches Rahmenwerk geschaffen [Lon93].

Software-Prozess	Ein Softwareprozess definiert, wie die Softwareentwicklung organisiert, gesteuert, gemessen, unterstützt und verbessert wird. Ein Softwareprozess (Synonym: Software-Entwicklungsprozess) ist eine Menge partiell geordneter Prozessschritte mit Bezug zu Mengen von Artefakten, Ressourcen, Organisationsstrukturen und Bedingungen und mit dem Ziel der Erstellung gewünschter Softwareergebnisse. Jedes Unternehmen oder Individuum, welches Software entwickelt, folgt einem Softwareprozess. Ein Softwareprozess kann implizit (d.h. es liegt keine Beschreibung vor) oder explizit sein, chaotisch oder strukturiert und werkzeugunterstützt sein.
Prozessschritt	Ein Prozessschritt ist ein Teil eines Prozesses. Prozessschritte können wiederum aus Prozessschritten bestehen.
Aktivität	Eine Aktivität ist ein elementarer Prozessschritt. Auf der Ebene der Abstraktion der Prozessbeschreibung besitzt er keine sichtbare Unterstruktur.
Agent	Ein Agent führt einen Prozess aus. Er kann ein Akteur (meist Personen) oder ein Werkzeug sein.
Artefakt	Ein Artefakt ist ein Produkt, das während eines Prozesses erzeugt oder modifiziert wird, entweder als Ergebnis oder als Zwischenergebnis.
Role	Eine Rolle umfasst eine Menge von Rechten und Pflichten, die zur Erreichung eines Ziels notwendig sind.
Ereignis	Ein Ereignis ist ein Signal, das durch die Erfüllung einer Bedingung ausgelöst wird. Ereignisse ermöglichen eine asynchrone Kommunikation, z.B. zwischen Aktivitäten und Aufgaben, und die Reaktion auf außergewöhnliche Umstände.
Softwareprozessmodell	Ein Softwareprozessmodell ist eine – mehr oder weniger formale – abstrakte Beschreibung eines Softwareprozesses. Ein Softwareprozessmodell beschreibt ein Abstraktionsniveau und eine bestimmte Sicht auf den Prozess.

Ein Prozessmetamodell ist ein konzeptuelles Rahmenwerk, das zur Beschreibung und Komposition von Softwareprozessmodellen dient. Es beschreibt Submodelle (z.B. Datenmodell, Aktivitätsmodell, Werkzeugmodell, Rollenmodell, Organisationsmodell) des Softwareprozessmodells, deren grundlegenden Konzepte und Zusammenhänge.

Prozessmetamodell

Eine Prozessmodellierungssprache ist ein Formalismus oder eine Sprache zur Darstellung von Softwareprozessmodellen. Sie ergänzt ein Prozessmetamodell um eine präzise Syntax und eine detaillierte Semantik.

Prozessmodellierungssprache

Softwareprozesse können textuell oder graphisch und informal, semiformal oder formal beschrieben werden. Textuelle Darstellungen beschreiben Prozesse mit natürlichsprachlichen Ausdrücken. Graphische Darstellungen beschreiben Prozesse mit graphischen Symbolen. Die formale Beschreibung eines Prozesses erfolgt durch Verwendung einer formalen Sprache. Eine formale Sprache wird durch eine Grammatik definiert, die angibt, welche Zeichenketten gültige Sätze der Sprache sind. Eine formale Sprache besitzt also eine formale Syntax und Semantik. Die informale Beschreibung eines Prozesses unterliegt hingegen keiner Grammatikregel. Semiformelle Beschreibungen sind stärker formalisiert als informale Beschreibungen. In der Regel besitzen sie eine formale Syntax, aber eine informale Semantik.

Beschreibung von Softwareprozessen

A.3 UML

Die Unified Modeling Language (UML) vereinigt drei objektorientierte Entwurfssprachen, nämlich die Object Modeling Technique (OMT) [RBP91], die Booch-Methode [Boo94] und Object Oriented Software Engineering (OOSE) [Jac92]. Mit der Version 1.3 wurde die UML 1999 von der Object Management Group (OMG) als Standard anerkannt. Zur Zeit ist die Version 1.5 verfügbar [UML1.5].

Mit Hilfe der UML kann ein Softwaresystem durch verschiedene Sichten beschrieben werden (Anhang A.3.1). Den Konzepten dieser Sichten liegt das UML-Metamodell zugrunde, welches durch Pakete strukturiert wird (Anhang A.3.2). Ein Beispiel zur Spezifikation von Syntax und Semantik des UML-Metamodells findet sich in (Anhang A.3.3). Anhang A.3.4 beleuchtet, welche Auswirkungen die kommende UML-Version 2.0 auf die Gestaltung von Aktivitätsdiagrammen hat.

A.3.1 UML-Sichten, -Diagramme und -Konzepte

Mit Hilfe der UML kann ein Softwaresystem durch verschiedene Sichten beschrieben werden (Tabelle A-8). Die Sichten sind struktureller oder dynamischer Natur, beschreiben Aspekte der Modellverwaltung oder der Modellerweiterung. Pro Sicht gibt es ein oder mehrere Diagrammtypen, die bestimmte Konzepte nutzen. Ein Beispiel für eine strukturelle, statische Sicht ist das Klassendiagramm, welches die Konzepte Klasse, Assoziation, Generalisierung, Abhängigkeit, Realisierung und Schnittstelle zur Modellierung verwendet. Eine ausführliche Beschreibung der UML-Sichten, -Diagramme und -Konzepte findet man in [RJB99].

Art	Sicht	Diagramm	Konzepte
Strukturell	statisch	Klassendiagramm	Klasse, Assoziation, Generalisierung, Abhängigkeit, Realisierung, Schnittstelle
	Use Case	Use Case Diagramm	Use Case, Akteur, Assoziation, Erweiterung, Inklusion, Use Case Generalisierung
	Implementierung	Komponentendiagramm	Komponente, Schnittstelle, Abhängigkeit, Realisierung
	Deployment	Deploymentdiagramm	Knoten, Komponente, Abhängigkeit, Lokation
Dynamisch	State Machine	State Chart Diagramm	Zustand, Ereignis, Transition, Aktion
	Aktivität	Aktivitätsdiagramm	Zustand, Aktivität, Completion Transition, Fork, Join
	Interaktion	Sequenzdiagramm	Interaktion, Objekt, Nachricht, Aktivierung
		Kollaborationsdiagramm	Kollaboration, Interaktion, Kollaborationsrolle, Nachricht
Modellmanagement	Modellmanagement	Klassendiagramm	Paket, Subsystem, Modell
Erweiterbarkeit	alle	alle	Constraint, Stereotyp, Tagged Values

Tabelle A-8: Sichten, Diagramme und Konzepte des UML-Metamodells

A.3.2 UML-Pakete

Die OMG-Spezifikation beschäftigt sich ausschließlich mit dem UML-Metamodell. Diese Schicht wird in mehrere logische Pakete zerlegt, die logisch zusammenhängende Sprachkonzepte enthalten: Foundation, Behavioural Elements und Model Management. Die Pakete werden wiederum in Subpakete zerlegt (Abbildung A-1).

Das Foundation-Paket spezifiziert in seiner Rolle als sprachliche Infrastruktur die statische Struktur der Modelle. Es enthält die folgenden Sub-Pakete:

- Das Core-Paket spezifiziert Basis-Konzepte und bildet das architektonische Fundament („skeleton“), zu dem weitere Sprachkonstrukte (Metaklassen, Metaassoziationen, Metaattribute) hinzugefügt werden können.
- Das Extensions-Mechanisms-Paket spezifiziert, wie Modellelemente semantisch angepasst und erweitert werden können. Voraussetzung für Erweiterungen ist, dass diese das Metamodell nur ergänzen, jedoch nicht verändernde oder widersprüchliche Konzepte einführen. Dies geschieht über die Definition von Stereotypen, Constraints, Tag Definitions und Tagged Values. Ein UML-Profil umfasst eine Menge solcher Elemente, die der Erweiterung der UML zu einem bestimmten Zweck dienen (z.B. UML Profile for Soft-

ware Development Processes in [UML1.5]). In Kapitel 4, 5 und 6 haben wir ein UML-Profil entwickelt, das die Erweiterung der UML zur Process Pattern Description Language PROPEL darstellt.

- Das Data-Types-Paket definiert elementare Datenstrukturen der Sprache.

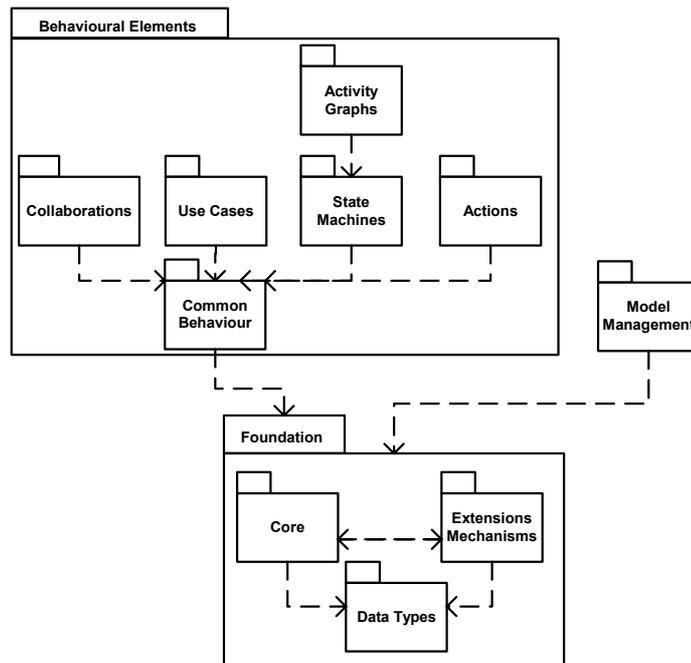


Abbildung A-1: Pakete und Subpakete des UML-Metamodells

Das Behavioural-Elements-Paket spezifiziert das dynamische Verhalten der Modelle. Es enthält die folgenden Pakete:

- Das Common-Behaviour-Paket spezifiziert Basis-Konzepte für Verhaltenselemente.
- Das Collaborations-Paket spezifiziert den Verhaltenskontext, der angibt, welche Modellelemente für welche Aufgaben zu nutzen sind.
- Das Use-Case-Paket definiert Verhalten mit Hilfe von Akteuren und Use-Cases.
- Das State-Machines-Paket definiert Verhalten von Systemen mit endlichen Zustandsübergängen.
- Das Activity-Graphs-Paket als Spezialfall des State-Machines-Paket definiert Prozesse.
- Das Actions-Paket definiert verschiedene Aktionsarten, die zu einer Prozedur zusammengesetzt werden.

Das Model Management-Paket definiert die Elemente Modell, Paket und Subsystem, welche als Ordnungseinheiten für andere Modellelemente fungieren.

A.3.3 Syntax und Semantik des UML-Metamodells

Das UML-Metamodell wird auf folgende Weise durch seine Syntax und Semantik beschrieben:

Abstrakte Syntax

Die abstrakte Syntax wird durch ein UML-Klassendiagramm repräsentiert, das Metaklassen und deren Beziehungen beschreibt. Durch Angabe der Multiplizität der Beziehungen wird gegebenenfalls auch schon die statische Semantik (in Form von OCL-Regeln, s. weiter unten) angegeben. Die Metaklassen werden zusätzlich durch eine natürlichsprachliche Beschreibung ergänzt, die neben einer einführenden Beschreibung alle Attribute und Assoziationen der Metaklasse auflistet und erläutert. Abbildung A-2 zeigt als Beispiel die abstrakte Syntax des Pakets Activity Graphs.

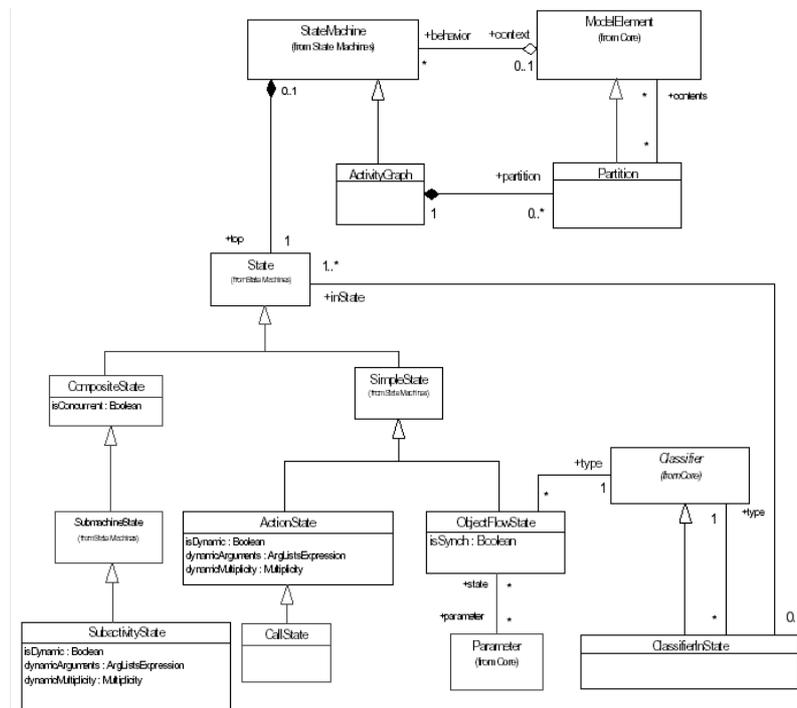
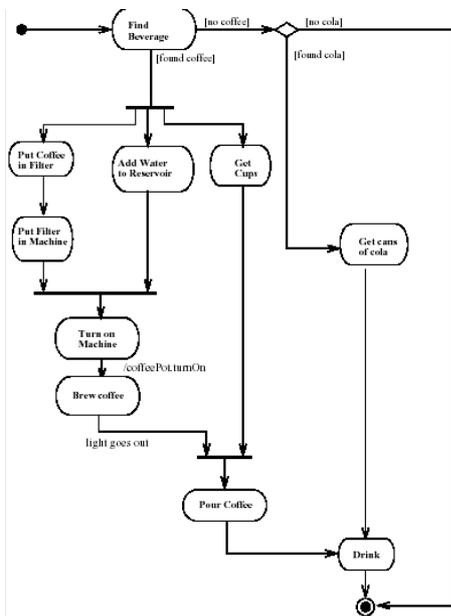


Abbildung A-2: Abstrakte Syntax des Pakets Activity Graphs (aus [UML1.5])

Konkrete Syntax

Die abstrakte Syntax ist notationsunabhängig; die konkrete Syntax ist definiert durch Abbildung einer Notation auf die abstrakte Syntax. Die konkrete Syntax wird durch natürliche Sprache und Beispieldiagramme dargestellt. Abbildung A-3 zeigt ein Beispiel für die konkrete Syntax des Pakets Activity Graphs.



„An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states. The entire activity diagram is attached (through the model) to a classifier, such as a use case, or to a package, or to the implementation of an operation. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). [...]“

Abbildung A-3: Konkrete Syntax des Pakets Activity Graphs – Beispiel Activity Diagram (aus [UML1.5])

Die statische Semantik einer Sprache definiert, welche Regeln bzw. Einschränkungen für die Konstruktioninstanzen gelten, die in der logischen Spezifikationssprache OCL (Object Constraint Language, s. [UML1.5]) formuliert sind. Diese OCL-Ausdrücke repräsentieren Constraints auf Attribute und Assoziationen. Der folgende Text zeigt beispielhaft die statische Semantik für die Klasse ActivityGraph aus dem Paket Activity Graphs:

Statische Semantik

[1] An ActivityGraph specifies the dynamics of

(i) a Package, or

(ii) a Classifier (including), or

(iii) a BehavioralFeature.

(self.context.oclIsTypeOf(Package) xor

self.context.oclIsKindOf(Classifier) xor

self.context.oclIsKindOf(BehavioralFeature))

Statische Semantik
der Klasse Activi-
tyGraph (aus
[UML1.5])

Die dynamische Semantik definiert in natürlicher Sprache die Bedeutung der vorab definierten Konstrukte. Sie wird in natürlicher Sprache und gegebenenfalls einigen Abbildungen präsentiert. Um hier ein Mindestmaß an Präzision zu gewährleisten, gibt die UML-Spezifikation eine Menge von Sprachkonventionen vor, etwa für die Benennung von Elementen. Trotzdem schränkt die Verwendung natürlicher Sprache die Präzision ein. Der folgende Text zeigt beispielhaft die dynamische Semantik für die Klasse ActivityGraph aus dem Paket Activity Graphs.

Dynamische
Semantik

Dynamische
Semantik der Klasse
ActivityGraph (aus
[UML1.5])

„The dynamic semantics of activity graphs can be expressed in terms of state machines. This means that the process structure of activities formally must be equivalent to orthogonal regions (in composite states). That is, transitions crossing between parallel paths (or threads) are not allowed, except for transitions used with synch states. As such, an activity specification that contains „unconstrained parallelism“ as is used in general activity graphs is considered „incomplete“ in terms of UML. [...]“.

A.3.4 UML 2.0

Die kommende UML-Version 2.0 beinhaltet grundlegende strukturelle und inhaltliche Änderungen.

Strukturelle
Änderungen

Strukturell betrachtet wurde die UML-Spezifikation, die bis dahin aus einem Dokument bestand, auf vier einzelne Dokumente aufgeteilt: Das Infrastructure-Dokument repräsentiert den sogenannten Sprachkern, der dazu verwendet werden kann, sogenannte Subsprachen zu spezifizieren. Eine solche Subsprache stellt die UML dar, die in dem Superstructure-Dokument spezifiziert wird. Zusätzlich gibt es ein OCL-Dokument zur Spezifikation der Constraints und schließlich das Diagram Interchange-Dokument, das den Austausch von Dokumenten bzw. Diagrammen zwischen Softwarewerkzeugen regelt. Die Neustrukturierung dient u.a. zur Entflechtung der Spezifikationsinhalte und zur Beseitigung von Redundanzen. Inwieweit diese Ziele erreicht wurden, ist noch zu beobachten. Anzumerken ist jedoch, dass die Spezifikation deutlich an Umfang zugenommen hat.

Inhaltliche
Änderungen

Inhaltlich betrachtet werden mit der neuen UML-Version verschiedene Ziele verfolgt. Das Infrastuktur-Dokument enthält Kernelemente, auf deren Basis Metamodelle wie die UML definiert werden können. Hierdurch wird die Wiederverwendung von Konzepten und die Trennung von spezifischen Elementen des Metamodells stärker deutlich gemacht. Ferner wurde eine gemeinsame Basis für einen verbesserten Austausch zwischen Werkzeugen geschaffen. Dem UML-Metamodell im Superstruktur-Dokument wurden darüber hinaus neue Konzepte hinzugefügt: Konzepte zur Darstellung kompositer Strukturen zur hierarchischen Beschreibung von Klassen, einem neuen Diagrammtyp, das sogenannte Interaction Frame Diagram, basierend auf Sequenz-Diagrammen, sowie einem neuen Konzept für Aktivitätsdiagramme.

Gründe für die
Neukonzeption der
Aktivitätsdiagramme

In den vorhergehenden UML-Versionen wurden Aktivitätsdiagramme als Spezialfall, d.h. als Subklasse von State Machines spezifiziert. Dieses Vorgehens hatte den Nachteil, dass alle Eigenschaften von State Machines an Aktivitätsdiagramme vererbt wurden, auch solche, die für Struktur und Verhalten von Aktivitätsdiagrammen nicht so geeignet waren. State Machines drücken Zustandsveränderungen aus, während Aktivitätsdiagramme eher Kontroll- und Datenflüsse ausdrücken. Aus diesem Grund werden mit der Version 2.0 Aktivitätsdiagramme von State Machines getrennt und erhalten ein eigenes, originäres Metamodell.

Konzepte von
Aktivitätsdiagrammen

Die wichtigste Neuerung ist die, dass die (informale) Semantik der Aktivitätsdiagramme sich an der Petrinetz-Semantik orientiert. Aktivitätsdiagramme heißen nun Aktivitäten, die durch einen Graph von Aktionsknoten, Kontrollknoten, Objektknoten und Kontroll- und Objektflüssen aufgespannt wird. Die graphische Notation sieht für diese Elemente der alten Notation noch recht ähnlich. Neu sind die Input Pins und Output Pins, welche Eingabe- und Ausgabeobjekte von Aktionen repräsentieren. Zusätzlich gibt es noch Aktivitätsparameterknoten, die Eingabe- und Ausgabeobjekte für Aktivitäten modellieren. Darüber hinaus gibt es Konzepte zum Objektstreaming (laufende(r) Erzeugung/Verbrauch von Objekten wäh-

rend der Ausführung einer Aktion), Ausnahmen, Objektmengen, lokale Vor- und Nachbedingungen, Finalknoten von Aktionen (zusätzlich zu den bislang auch schon bekannten Finalknoten von Aktivitäten) und Knoten zum Speichern von Objekten.

A.4 Vorgehensmodelle

A.4.1 Rational Unified Process

Nachfolgend zeigen wir einige der Original-RUP-Aktivitäten, von denen wir die Prozessmuster in Kapitel 7 abgeleitet haben.

Abbildung A-4 zeigt die RUP-Aktivität „Find Actors and Use Cases“, von der das Prozessmuster „Find Actors and Use Cases“ abgeleitet wurde.

Purpose <ul style="list-style-type: none"> ■ To outline the functionality of the system. ■ To define what will be handled by the system and what will be handled outside the system. ■ To define who and what will interact with the system. ■ Divide the model into packages with actors and use cases. ■ Create diagrams of the use-case model. ■ Develop a survey of the use-case model. 	
Steps <ul style="list-style-type: none"> ■ Find Actors ■ Find Use Cases ■ Describe How Actors and Use Cases Interact ■ Package Use Cases and Actors ■ Present the Use-Case Model in Use-Case Diagrams ■ Develop a Survey of the Use-Case Model ■ Evaluate Your Results 	
Input Artifacts: <ul style="list-style-type: none"> ■ Business Use-Case Model ■ Business Object Model ■ Use-Case Modeling Guidelines ■ Stakeholder Requests ■ Vision ■ Glossary 	Resulting Artifacts: <ul style="list-style-type: none"> ■ Use Case ■ Actor ■ Use-Case Model ■ Supplementary Specifications
Role: System Analyst	
Work Guidelines: <ul style="list-style-type: none"> ■ Use-Case Workshop ■ Storyboarding 	
Tool Mentors <ul style="list-style-type: none"> ■ Finding Actors and Use Cases Using Rational Rose ■ Finding Actors and Use Cases Using Rational Rose ■ Managing Use Cases Using Rational Rose and Rational RequisitePro ■ Managing Stakeholder Requests Using Rational ClearQuest and Rational RequisitePro 	
Workflow Details: <ul style="list-style-type: none"> ■ Requirements <ul style="list-style-type: none"> ■ Analyze the Problem ■ Define the System ■ Understand Stakeholder Needs 	

Abbildung A-4: RUP-Aktivität „Find Actors and Use Cases“ (aus [RUP])

Abbildung A-5 zeigt die RUP-Aktivität „Review Requirements“, von der das Prozessmuster „Review Requirements“ abgeleitet wurde.

Purpose <ul style="list-style-type: none"> ■ To formally verify that the results of Requirements conform to the customer's view of the system. 	
Steps <ul style="list-style-type: none"> ■ <u>Conduct Review Meetings</u> 	
Input Artifacts: <ul style="list-style-type: none"> ■ <u>User-Interface Prototype</u> ■ <u>Iteration Plan</u> ■ <u>Glossary</u> ■ <u>Vision</u> ■ <u>Use-Case Model</u> ■ <u>Use Case</u> ■ <u>Supplementary Specifications</u> ■ <u>Software Requirements Specification</u> ■ <u>Use-Case Modeling Guidelines</u> ■ <u>Change Request</u> ■ <u>Use-Case Package</u> 	Resulting Artifacts: <ul style="list-style-type: none"> ■ <u>Review Record</u>
Role: <u>Requirements Reviewer</u>	
Work Guidelines: <ul style="list-style-type: none"> ■ <u>Reviews</u> 	
Tool Mentor: <ul style="list-style-type: none"> ■ <u>Publishing Web-based Rational Rose Models Using Web Publisher</u> ■ <u>Publishing Web-based Rational Rose Models Using Web Publisher</u> ■ <u>Reviewing Requirements Using Rational RequisitePro</u> 	
Workflow Details: <ul style="list-style-type: none"> ■ <u>Requirements</u> <ul style="list-style-type: none"> ■ <u>Manage Changing Requirements</u> 	

Abbildung A-5: RUP-Aktivität „Review Requirements“ (aus [RUP])

Abbildung A-6 zeigt die RUP-Aktivität „Develop Vision“, von der das Prozessmuster „Develop Vision“ abgeleitet wurde.

Purpose <ul style="list-style-type: none"> ■ Gain agreement on what problems need to be solved. ■ Identify stakeholders to the system. ■ Define the boundaries of the system. ■ Describe primary features of the system. 	
Steps <ul style="list-style-type: none"> ■ <u>Gain agreement on the problem being solved</u> ■ <u>Identify stakeholders</u> ■ <u>Define the system boundaries</u> ■ <u>Identify constraints to be imposed on the system</u> ■ <u>Formulate problem statement</u> ■ <u>Define features of the system</u> ■ <u>Evaluate your results</u> 	
Input Artifacts: <ul style="list-style-type: none"> ■ <u>Stakeholder Requests</u> ■ <u>Business Rules</u> ■ <u>Business Use-Case Model</u> ■ <u>Business Object Model</u> ■ <u>Vision</u> 	Resulting Artifacts: <ul style="list-style-type: none"> ■ <u>Requirements Attributes</u> ■ <u>Supplementary Specifications</u> ■ <u>Vision</u>
Role: <u>System Analyst</u>	
Tool Mentors: <ul style="list-style-type: none"> ■ <u>Developing a Vision Using Rational RequisitePro</u> 	
Guidelines: <ul style="list-style-type: none"> ■ <u>Going from Business Models to Systems</u> ■ <u>Brainstorming and Idea Reduction</u> ■ <u>Fishbone Diagrams</u> ■ <u>Pareto Diagrams</u> 	
Workflow Details: <ul style="list-style-type: none"> ■ <u>Requirements</u> <ul style="list-style-type: none"> ■ <u>Analyze the Problem</u> ■ <u>Define the System</u> ■ <u>Manage the Scope of the System</u> ■ <u>Understand Stakeholder Needs</u> 	

Abbildung A-6: RUP-Aktivität „Develop Vision“ (aus [RUP])

A.4.2 Crystal Methodologies

Abbildung A-7 zeigt in einer Übersicht den Zusammenhang zwischen Anzahl der Mitarbeiter, Kritikalität und empfohlener Crystal-Methode. Für lebenskritische Systeme und Projekte mit mehr als 200 Mitarbeitern gibt es keine Empfehlungen.

		Anzahl Mitarbeiter						
		1-6	<20	<40	<100	<200	>500	
Kritikalität	Life (L)	L6	L20	L40	L100	L200	L500	?
	Essential Money (E)	E6	E20	E40	E100	E200	E500	
	Discretionary Money (D)	D6	D20	D40	D100	D200	D500	
	Comfort (C)	C6	C20	C40	C100	C200	C500	
		Crystal Clear	Crystal Yellow	Crystal Oran.	Crystal Red	?	?	

Abbildung A-7: Einordnung der Crystal Methodologies (adaptiert aus [CH02a])

B Details zur Semantik

Die Syntax der UML wird mit Klassendiagrammen und OCL-Constraints beschrieben. Da wir die semantische Abbildung nicht direkt auf dem PROPEL-Metamodell definieren können, müssen wir syntaktische Domänen definieren, die den Definitionsbereich der semantischen Funktionen repräsentieren und syntaktische Regeln, die die Struktur von PROPEL repräsentieren.

B.1 Syntaktische Domänen

Wir verwenden folgende in Syntaktische DomänenTabelle B-1 aufgelistete syntaktischen Domänen (alphabetisch sortiert). Für jede Metaklasse des PROPEL-Metamodells wird eine syntaktische Domäne abgeleitet. Diese Domänen sind Nichtterminale der kontextfreien Grammatik von PROPEL in der erweiterten Backus-Naur-Form (EBNF).

Domäne	Mengen von
ActionState	Aktivitäten
ActivityGraph	Aktivitätsdiagramme
ActivityProblemMapping	(Aktivität,Problem)-Paar
Aspect	Thematische Ausrichtung eines Prozessmusters, Problems, Objekts oder Ereignisses
CompositeState	Nicht-atomarer Zustand
Context	Kontext eines Prozessmusters
Classifier	Klassifizierer
ClassifierInState	Zustand des Klassifizierers
Element	Element
FinalState	Abschließender Zustand eines Zustandsautomaten/Aktivitätsdiagramms
ModelElement	Modellierungselement
ObjectFlowState	Objekt, Ereignis
OFSComposition	Komposition von Objekten bzw. Ereignisse
Parameter	Input-, Output-, oder Inoutparameter einer Aktivität
Problem	Problem eines Prozessmusters

Tabelle B-1: Syntaktische Domänen

Domäne	Mengen von
ProblemGraph	Problemdiagramm
ProcessPattern	Prozessmuster
ProcessPatternGraph	Prozessmusterdiagramm
ProcessPatternCatalog	Prozessmusterkatalog
ProcessPatternRelationship	Beziehungen zwischen Prozessmustern
Processvariance	Prozessmustervarianz
Refinement	Prozessmusterverfeinerung
RefineProblem	Problemverfeinerung
Role	Für eine Aktivität oder einen Prozess verantwortliche Rolle
Process	Prozess eines Prozessmusters
Sequence	Prozessmustersequenz
SimpleState	Atomarer Zustand
State	Zustand
StateMachine	Zustandsautomat
StateVertex	Knoten in einem Zustandsautomat
Transition	Transition
Use	Prozessmesternutzung
Tool	Für eine Aktivität oder einen Prozess eingesetztes Werkzeug

Tabelle B-1: Syntaktische Domänen (Fortgesetzt)

B.2 Semantische Funktionen

Wir definieren folgende in Syntaktische DomänenTabelle B-2 aufgelistete semantische Funktionen (alphabetisch sortiert). Handelt es sich um eine Familie von Funktionen (z.B. id), versehen wir diese mit einem Index.

Funktion	Definitionsbereich	Wertebereich
agraph:	ActivityGraph	eB/E-System
dec ₁ :	ObjectFlowState	Name ⁺
dec ₂ :	ObjectFlowState	Name
dec ₃ :	OFSComposition	Name ⁺

Tabelle B-2: Semantische Funktionen

dec ₄ :	Context	Name*
dec ₅ :	Context	Name*
dec ₆ :	ProcessPattern	eB/E-System
ic ₁ :	ProcessPattern	Name*
ic ₂ :	ProcessPattern	Name*
id ₁ :	ProcessPattern	Name
id ₂ :	PPR	(Name>Name)*
id ₃ :	SimpleState	Name
id ₄ :	FinalState	Name
id ₅ :	CompositeState	Name
id ₆ :	Z	Name
id ₇ :	Transition	Name
id ₈ :	ObjectFlowState	Name
id ₉ :	Context	Name*
id ₁₀ :	Sequence	(Name>Name)*
id ₁₁ :	Use	Name>Name
id ₁₂ :	Refinement	Name>Name
id ₁₃ :	Processvariance	Name>Name
id ₁₄ :	StateVertex	Name
init ₁ :	SimpleState	Name
init ₂ :	FinalState	Name
init ₃ :	CompositeState	Name*
init ₄ :	Z	Name*
pr ₁ :	Process	eB/E-System
pr ₂ :	ProcessPattern	eB/E-System
rc ₁ :	ProcessPattern	Name*
rc ₂ :	ProcessPattern	Name*
rel ₁ :	Sequence	eB/E-System
rel ₂ :	Use	eB/E-System
rel ₃ :	Refinement	eB/E-System
rel ₄ :	Processvariance	eB/E-System
rel ₅ :	Sequence	eB/E-System

Tabelle B-2: Semantische Funktionen (Fortgesetzt)

rel ₆ :	Use	eB/E-System
rel ₇ :	Refinement	eB/E-System
rel ₈ :	Processvariance	eB/E-System
structure:	ProcessPatternCatalog	Graph
sub ₁ :	SimpleState	Name×Name
sub ₂ :	FinalState	Name×Name
sub ₃ :	CompositeState	(Name×Name)*
sub ₄ :	Z	(Name×Name)*
terminal ₁ :	SimpleState	Name
terminal ₂ :	FinalState	Name
terminal ₃ :	CompositeState	Name*
terminal ₄ :	Z	Name*
trans:	Transition	T, T die Menge der Transitionen
type ₁ :	SimpleState	(Name->Type)
type ₂ :	FinalState	(Name->Type)
type ₃ :	CompositeState	(Name->Type)
type ₄ :	CompositeState	(Name->Type)
type ₅ :	Z	(Name->Type)

Tabelle B-2: Semantische Funktionen (Fortgesetzt)

C Implementierungsdetails

Bei der Entwicklung des Prototypen der Process Pattern Workbench wurde mit folgenden Werkzeugen gearbeitet:

Tätigkeit	Werkzeug
Modellierung des Analyse- und des anschliessenden Designmodells	TogetherJ 6.0
Implementierung der Serverkomponenten	IntelliJ 3.0
Visualisierung der Prozess- und Prozess-Instanz-Graphen	JGraph 2.0
EJB Applikationsserver	JBoss 3.0.4
Datenbankserver	IBM DB2 7.3g
Deployment	Jakarta Ant 1.5
Komponenten-Tests	JUnit 3.8.1
Programmierplattform	JDK 1.3.1 (Client) & J2EE 1.3.1 (Server)

Tabelle C-1: Eingesetzte Werkzeuge

Literatur

A

- [AA02] Agile Alliance: The Manifesto for Agile Software Development, <http://www.agilealliance.org>.
- [ABG93] Armenise, P.; Bandinelli, S.; Ghezzi, C. et al.: A Survey and Assessment of Software Process Representation Formalisms. In: Int. Journal on Software Engineering and Knowledge Engineering, vol. 3 (3), 1993, pp. 401 - 426.
- [ABT00] Aalst, W.; Barros, A.; ter Hofstede, A. et.al.: Advanced Workflow Patterns. Proceedings Seventh IFCIS International Conference on Cooperative Information Systems, September 2000. Also available at http://www.icis.qut.edu.au/~arthur/articles/-_apatterns.pdf.
- [ABW88] Alpert, S.; Brown, K.; Woolf, B.: The Design Patterns Smalltalk Companion. Addison-Wesley, 1988.
- [ACL96] Alencar, P.; Cowen, D.; Lucena, C.: A formal approach to architectural design patterns. In: Proceedings of the 3rd International Symposium of Formal Methods Europe, 1996, pp. 576-594.
- [AIS77] Alexander, C.; Ishikawa, S.; Silverstein, M. et al.: A Pattern Language. Oxford University Press, 1977.
- [AHK03] van der Aalst, W.; ter Hofstede, A.; Kiepuszewski, B. et al.: Workflow Patterns. Distributed and Parallel Databases, 14(3), pages 5-51, July 2003. Available at <http://tmitwww.tm.tue.nl/research/patterns/>.
- [Ale79] Alexander, C.: The Timeless Way of Building. Oxford University Press, 1979.
- [Amb98] Ambler, S. W.: Process Patterns, Cambridge University Press, 1998, Cambridge.
- [Amb99] Ambler, S. W.: More Process Patterns, Cambridge University Press, 1999, Cambridge.
- [App98] Appleton, B.: Patterns and Software: Essential Concepts and Terminology. <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>, 1998.
- [ARF97] Ambriola, V.; Conradi, R.; Fuggetta, A.: Assessing process-centered software engineering environments. In: ACM Transactions on Software Engineering and Methodology, 6,3, 1997, pp. 283-328.
- [ASL01] Apvrille, L.; de Saqui-Sannes, P.; Lohr, C. et al.: A new UML profile for real-time system formal design and validation. In: Proc. UML 2001, LNCS 2185, S. 287-301, Springer, 2001.
- [Aue98] Auer, K.: Philosophical Patterns In Software Development. In Proc. of PloP 98, http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P06.pdf, 1998.

B

- [Bac94] Bach, J.: The Immaturity of CMM. In: American Programmer, September 1994.

- [Bal98] Balzert, H.: Lehrbuch der Software-Technik. Spektrum, 1998.
- [Bau96] Baumgart, B.: Petri-Netze – Grundlagen und Anwendungen. Spektrum, 2. Auflage, 1996.
- [BCM98] Broy, M.; Coleman, D.; Maibaum, T. et al. (eds): Proceedings des PSMT Workshop on Precise Semantics for Modeling Techniques, Technische Universität München, TUMI9803, 1998.
- [BCR94] Basili, V.R.; Caldiera, G.; Rombach, D.: Experience Factory. In: Marciniak, J.J. (ed.), Encyclopedia of Software Engineering, vol 1, pp. 469 - 476; John Wiley & Sons; 1994.
- [BCR00] Börger, E.; Cavarra, A.; Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: Proc. International Conference on Algebraic Methodology and Software Technology (AMAST 2000), LNCS 1826, pages 293-308. Springer, 2000.
- [BD00] Bolton, C.; Davies, J.: Activity graphs and processes. In: Proc. Integrated Formal Methods (IFM 2000), LNCS 1945, pages 77-96, Springer, 2000.
- [Bec88] Beck, K.: Using a pattern language for programming. In: Workshop on Specification and Design. ACM SIGPLAN Notices 23, 5. Addendum to the Proceedings of OOPSLA 87, 1988.
- [Bec96] Beck, K.: Smalltalk Best Practice Patterns, Prentice Hall, 1996.
- [Bec00] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.
- [Ber97] Berten, A.: Entwurfsmuster und Software-Wiederverwendung. Studienarbeit, Gesamthochschule Siegen, 1997.
- [BFG94] Bandinelli, S.; Fuggetta, A.; Ghezzi, C. et al.: SPADE: An Environment for Software Process Analysis, Design, and Enactment. In: [FKN94], pp. 223-247, 1994.
- [BGJ99] Berner, S.; Glinz, M.; Joos, S.: A Classification of Stereotypes for Object-Oriented Modeling Languages. In: UML 1999, LNCS 1723, pp. 249-264, 1999.
- [BJ94] Beck, K.; Johnson, R.: Patterns Generate Architectures. ECOOP '94, LNCS 821, Conference Proceedings, pp. 139-149, Springer, 1994.
- [BLM97] Bicarregui, J.; Lano, K.; Maibaum, T.: Objects associations and subsystems. A hierarchical approach to encapsulation. In: Proceedings of ECOOP 97, LNCS 1489, Springer, 1997.
- [BM95] Buschmann, F., Meunier, R.: A System of Patterns. In: [CS95] .
- [BmBF00] Bundesministerium für Bildung und Forschung, in Zusammenarbeit mit GfK Marktforschung GmbH, Fraunhofer IESE und Fraunhofer ISI: „Analyse und Evaluation der Softwareentwicklung in Deutschland“, 2000.
- [BMR96] Buschmann, F.; Meunier, R.; Rohnert, H. et al. : Pattern-Oriented Software Architecture – A System of Patterns. Wiley and Sons, 1996.
- [Boc03a] Bock, C.: UML 2 Activity and Action Models. In: Journal of Object Technology, Volume 2, Number 4, July-August 2003, S. 43-53.
- [Boc03b] Bock, C.: UML 2 Activity and Action Models Part 2: Actions. In: Journal of Object Technology, Volume 2, Number 5, 2003, S. 41-56.

- [Boc03c] Bock, C.: UML 2 Activity and Action Models Part 3: Control Nodes. In: Journal of Object Technology, Volume 2, Number 6, 2003, 7-23.
- [Boo94] Booch, G.: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994.
- [Bos96] Bosch, J.: Language Support for Design Patterns. In: Proceedings TOOLS Europe, 1996.
- [Bos98] Bosch, J.: Design Patterns as Language Constructs. In: Journal of Object-Oriented Programming, May 1998, pp. 18-32.
- [Bos01] Bosch, J., Design Patterns & Frameworks: On the issue of Language Support. Internet: <http://www.ipd.hk-r.se/bosch/lsdforg/bosch.ps>.
- [BR02] Broy, M.; Rombach, D.: Software Engineering – Wurzeln, Stand und Perspektiven, Informatik Spektrum, Dezember 2002, S. 438-451.
- [Bre98] Bremer, G.: Genealogie von Entwicklungsschemata. In: [KMO98], S. 32-59.
- [Bro96] Brown, K.: Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Thesis, North Carolina State Technical Report TR-07-96, http://members.aol.com/kgb_1001001/Articles/THESIS/thesis.htm, 1996.
- [BRS98] Bergner, K.; Rausch, A.; Sihling, M.: A Component Methodology based on Process Patterns. TUM-I9823, Universität München, 1998.
- [Bün99] Bünnig, S.: Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines dazugehörigen Compilers. Universität Rostock, Fachbereich Informatik, 1999.
- [Coa92] Coad, P.: Object-oriented Patterns. Communications of the ACM, 35, 9, 1992.
- [Coa99] Coad, P.: Java Modeling in Color with UML, Prentice Hall, 1999.
- [CC02] Homepage of the SCRUM Methodology. <http://www.controlchaos.com/>.
- [CDG95] Cugola, G.; Di Nitto, E.; Ghezzi, C. et.al.: How to deal with deviations during process model enactment, Proceedings of the 17th international conference on Software engineering, p.265-273, 1995.
- [CFF92] Conradi, R.; Fernström, C.; Fugetta, A. et al.: Towards a Reference Framework for Process Concepts, Proceedings of the 2nd European Software Software Process Technology, J.-C. Derniame (ed.), Springer, 1992.
- [CH02a] Cockburn, A., Highsmith, J. (eds.): Agile Software Development. Addison-Wesley, 2002.
- [CH02b] Coplien, J., Harrison, N.: Organizational Patterns. To appear.
- [CHL94] Conradi, R.; Hagaseth, M.; Liu, C.: Planning support for cooperating transaction in EPOS. Proc. CAISE 94, pp. 2-13, 1994.
- [CLS00] Cass, A. G.; Staudt-Lerner, B.; Sutton, S.M. et.al.: Little-JIL/Juliette, Proceedings of the 22nd international conference on Software engineering, p.754-757, 2000.
- [CMII] CMII, www.cmii.de.

C

- [CNM95] Coad, P., North, D., Mayfield, M.: Object Models: Strategies, patterns, applications. Prentice-Hall, Englewood Cliffs, 1995.
- [Cop92] Coplien, J.: Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.
- [Cop94] Coplien, J.: A Development Process Generative Pattern Language. In: Proceedings of PLoP, 1994.
- [Cop96] Coplien, J. O.: Software Patterns. SIGS Book & Multimedia, 1996.
- [Crystal] Homepage of the Crystal Methodologies. <http://alistair.cockburn.us/crystal/index.html>
- [CS95] Coplien, J., & Schmidt, D. C. (Eds): Pattern Languages of Program Design. Reading, Massachusetts, Addison-Wesley, 1995.
- [Cun88] Cunningham, W.: Panel on Design Methodology. ACM SIGPLAN Notices 23, 5. Addendum to the Proceedings of OOPSLA87, 1988.
- [CWM03] OMG: Common Warehouse Metamodel. Final Adopted Specification. Verfügbar unter <http://www.omg.org/docs/ptc/03-06-05.pdf>.
- [CZ00] Coplien, J.O.; Zhao, L.: Symmetry and symmetry breaking in software patterns. In: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany, 2000. .
- [Czi01] Czichy, T.: Pattern-based Software Development: An Empirical Study – Summary of Results. University of Technology, Dresden, Department of Systems Engineering, 2001.

D

- [DeM96] DeMarco, Tom: The Role of Software Development Methodologies: Past, Present, and Future. In: 18th International Conference on Software Engineering. Berlin, Germany, 1996. Proceedings. IEEE Computer Society Press, 1996, pp. 2-4.
- [Dem86] Deming, W.: *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, MA, 1986.
- [DG94] Deiters, W.; Gruhn, V.: The FUNSOFT Net Approach to Software Process Management. International Journal of Software Engineering and Knowledge Engineering, 4(2), pp. 229- 256, 1994.
- [DGH02] Dittmann, T., Gruhn, V., Hagen, M.: Improved Support for the definition and usage of process patterns. 1st Workshop on Software Development Process Patterns at the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (SDPP02), Seattle, USA, November 04-08, 2002.
- [Dit02] Dittmann, T.: PDDL – Eine Beschreibungssprache für Process Patterns, Diplomarbeit, 2002, Universität Dortmund.
- [DKW99] Derniame, J.-C., Kaba, B., Wastell, D.: Software Process: Principles, Methodology and Technology, Springer, 1999.
- [DL99] DeMarco, T.; Lister, T.: Peopleware, Productive Projects and Teams. Dorset House Publishing Corporation, 1999.
- [DNR91] Dowson, M.; Nejme, B.; Riddle, W.: Fundamental Software Process Concepts, Proc. Of the 1st European Workshop on Software Process Modeling, AICA Press, 1991, Milan.

[DW99] D'Souza, D.F.; Wills, A.C.: *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.

E

[Eck95] Eckert, H.: Die Workflow Management Coalition, *Office Management* 6/1995, S. 26-32.

[Ede97] Eden, A.: Giving "The Quality" a Name. *Journal of Object Oriented Programming*, Mai, 1997.

[EDG99] Engels, G.; Dassen, J.H.M.; Groenewegen, L.P.J. et al.: A formalisation of SOCCA using Z; part 1: the type level concepts. Technical Report 1999-03, Leiden Institute of Advanced Computer Science, 1999.

[EDM98] Emam, K. E.; Drouin, J.-N.; Melo, W.: *SPICE – The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society, 1998.

[EFQM] The European Foundation for Quality Management (EFQM): *Selbstbewertung anhand des Europäischen Qualitätsmodells für Umfassendes Qualitätsmanagement (TQM): Richtlinien für die Identifizierung und Behandlung von Fragen zum Umfassenden Qualitätsmanagement*, Brüssel, 1994.

[EG94] Engels, G.; Groenewegen, L.: SOCCA: Specifications of Coordinated and Cooperative Activities. In: [FKN94], pp. 71-102, 1994.

[EG01] Eden, A.; Grogono, P.: *A Theory Of Object Oriented Software Architecture*, Department of Computer Science, Concordia University Montreal, 2001.

[EHL99] Eden, A.; Hirshfeld; Lundqvist, K.: *LePUS – Symbolic Logic Modeling of Object Oriented Architectures: A Case Study*. Second Nordic Workshop on Software Architecture (NOSA'99), Aug. 12-13, Ronneby, Sweden, 1999.

[EMC01] Ehrig, H.; Mahr, B.; Cornelius, F. et al.: *Mathematisch-strukturelle Grundlagen der Informatik*, 2. Auflage, Springer, 2001.

[ER02] Endres, A.; Rombach, D.: *Empirical Software and Systems Engineering – Observations, Laws & Theories*. Pearson, (to appear) 2002.

[Esh02] Eshuis, R.: *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. Dissertation, Universität Twente, 2002.

F

[FE03] Förster, A., Engels, G.: *Quality Ensuring Development of Software Processes*. EWSPT '03, Helsinki, Finland, 2003.

[FEL97] France, R.; Evans, A.; Lano. K.: *OOPSLA Workshop on OO Behavioral Semantics*. Technical Report TUM I9737, Technische Universität München, 1997.

[FH92] Feiler, P.H.; Humphrey, W.S.: *Software Process Development and Enactment: Concepts and Definitions*, CMU/SEI-92-TR-004, 1992, SEI.

[FKN94] Finkelstein, A.; Kramer, J.; Nuseibeh, B. (eds): *Software Process Modelling and Technology*, Research Studies Press, 1994.

[FO95] Foote, B.; Opdyke, W.F.: *Lifecycle and Refactoring Patterns that Support Evolution and Reuse*. In: [CS95], pages 239–258. Available at www.laputan.org/lifecycle/Lifecycle.html.

- [Fow96] Fowler, M.: Analysis Patterns: Reusable Object Models, Addison-Wesley, 1996.
- [FRF02] Fowler, M.; Rice, D.; Foemmel, M.: Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.
- [Fug00] Fuggetta, A.: Software Process: A Roadmap. In: Proceedings of the 22nd Conference on Software Engineering, The Future of Software Engineering, Anthony Finkelstein (ed.), ACM Press, 2000.
- [FY95] Foote, B.; Yoder, J.: Evolution, Architecture, and Metamorphosis. In [CS95]. Available via <http://st-www.cs.uiuc.edu/~plop/>.

G

- [GGW98] Gehrke, T.; Goltz, U.; Wehrheim, H.: The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process, 11/98, Universität Hildesheim, 1998.
- [GHJ93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design patterns: Abstraction and reuse of object-oriented design. In: European Conference on Object-Oriented Programming Proceedings (ECOOP'93), LNCS 707. Springer-Verlag, July 1993.
- [GHJ96] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, V.: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996.
- [GMP01a] Gnatz, M.; Marschall, F.; Popp, G. et al.: Towards a Living Software Development Process Based on Process Patterns. In: Software Process Technology, LNCS 2077, Springer, 2001, pp. 182-202.
- [GMP01b] Gnatz, M.; Marschall, F.; Popp, G.; et al.: Modular Process Patterns supporting an Evolutionary Software Development Process. In: Proceedings of the 3rd Int. Conference on Product Focused Software Development, 2001.
- [Gra97] Grady, Robert B.: Successful Software Process Improvement. Prentice Hall, 1997.
- [GW98] Gruhn, V.; Wellen, U.: Beschreibung von Vorgehensmodellen mit FUN-SOFT-Netzen. In: [KMO98], Kapitel V, S. 95-109.

H

- [Hag02] Hagen, M.: Support for the definition and usage of process patterns. Position Paper, EuroPloP 2002, http://www.haase-consulting.com/workshops/FgEuroplop02/position_papers.html.
- [Hag04a] Hagen, M.; Gruhn V.: PROPEL – A Language for the Description of Process Patterns (in German). In: Proc. of Modellierung 2004, LNI, Vol. P-45, Köllen, 2004, pp.203-218.
- [Hag04b] Hagen, M.; Gruhn V.: Process Patterns – a Means to Describe Processes in a Flexible Way. In: Proc. of ProSim 2004, 5th International Workshop on Software Process Simulation and Modeling (ProSim 2004), Edinburgh, United Kingdom, 2004.
- [Hag04c] Hagen, M.; Gruhn, V.: Towards flexible Software Processes by using Process Patterns. 8th IASTED International Conference on Software Engineering and Applications (SEA2004), Cambridge, USA, 2004.

- [Har95] Harrison, N. B.. Organizational Patterns for Teams. In: [CS95]. Available via <http://st-www.cs.uiuc.edu/~plop/>.
- [Hermes] Hermes 2003, www.hermes.admin.ch.
- [Hig99] Highsmith, J.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House Publishing, 1999
- [Hillside] The Hillside Group Web Pages. www.hillside.net.
- [HHG90] Helm, R.; Holland, I.M.; Gangopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. OOPSLA '90, SIGPLAN Notices 25, 10, 1990, pp. 169-180.
- [HK89] Humphrey, W. S.; Kellner, M. I.: Software process modeling, Proceedings of the 11th international conference on Software engineering, p.331-342, 1989.
- [HL89] Huff, K. E.; Lesser, V. R.: A plan-based intelligent assistant that supports the software development, ACM SIGPLAN Notices, v.24 n.2, p.97-106, Feb. 1989.
- [Hoa72] Hoare, C.: Notes on Data Structuring. In Dijkstra, E.; Hoare, C.; Dahl, O.-J.: Structured Programming, Academic Press, 1972.
- I
- [ISO9000] DIN Deutsches Institut für Normung e. V.: DIN EN ISO 9000:2000, QM-Systeme – Grundlagen und Begriffe, Beuth-Verlag, 2000.
- [ISO9001] DIN Deutsches Institut für Normung e. V.: DIN EN ISO 9001:2000, QM-Systeme – Anforderungen, Beuth-Verlag, 2000.
- [ISO9004] DIN Deutsches Institut für Normung e. V.: DIN EN ISO 9004:2000, QM-Systeme – Leitfaden zur Leistungsverbesserung, Beuth-Verlag, 2000.
- [ISO12207] ISO/IEC: ISO12207: Information technology – Software lifecycle processes, 1995.
- [IST89] Inoue, K., Seki, H., Taniguchi, K. et.al.: Compiling and Optimizing Methods for the functional Language ASL/F, Science of Computer Programming, 7, 11, pp.297-312, 1986.
- [ITIL] IT Infrastructure Library, www.itil.org.
- J
- [Jac92] Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.
- [JBR99] Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999.
- Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1998.
- [JHA01] Jeffries, R.; Hendrickson, C.; Anderson, A.: Extreme Programming Installed. Addison-Wesley, 2001.
- [JLC92] Jaccheri, L.; Larsen, J.-O.; Conradi, R.: Software Process Modeling and Evolution in EPOS, Proc. Fourth International Conference on Software Engineering and Knowledge Engineering (SEKE), 1992.

- [JLP98] Jaccheri, L.; Lago, P.; Picco, G.P.: Eliciting Software Process Models with the E3 Language, *ACM Transactions on Software Engineering and Methodology*, 7(4), 1998, pp. 368 - 410.
- [JSW99] Jäger, D.; Schleicher, A.; Westfechtel, B.: Using UML for Software Process Modeling. In: *Proceedings of the 7th European Software Engineering Conference, ESEC/FSE '99, LNCS 1687, Springer, 1999.*

K

- [Kat89] Katayama, T.: A hierarchical and functional software process description and its enactment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 343-352, 1989.
- [KB97] Kim, H.; Boldyreff, C.: Formalising Design Patterns and Frameworks: A Survey Report, http://www.dur.ac.uk/liz.burd/tr_02-97.ps, 1997.
- [Ker95] Kerth; N.: Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design. In [CS95], pages 293–320. Available at www1.bell-labs.com/user/cope/Patterns/Process/index.html.
- [KFP88] Kaiser, G. E.; Feiler, P. H.; Popovich, S. S.: Intelligent Assistance for Software Development and Maintenance. In: *IEEE Software*, May 1988, p. 40-49.
- [KMO98] Kneuper, R.; Müller-Luschnat, G.; Oberweis, A. (eds.): *Vorgehensmodelle für die betriebliche Anwendungsentwicklung*, Teubner, Serie Wirtschaftsinformatik, 1998.
- [Kne98] Kneuper, R.: Organisatorische Gestaltung des Einsatzes von Vorgehensmodellen. In: [KMO98], S. 228-248.
- [Koc93] Koch, G.: Process Assessment: the BOOTSTRAP approach. In: *Information and Software Technology*. 6/7, 1993, S. 387-404.
- [KR90] Kellner, M.; Rombach, D.: Comparison of Software Process Description. *IEEE*, 1990, pp. 7-18.
- [Kru00] Kruchten, P.: *The Rational Unified Process*, Addison-Wesley Professional, 2000.
- [KSR99] Keller, R.; Schauer, R., Robitaille, S. et al.: Pattern-Based Reverse Engineering of Design Components. In: *ACM Proceedings of the International Conference on Software Engineering, 1999*, pp. 226-235.

L

- [Lang] Langenscheidt Online-Fremdwörterbuch. <http://www.langenscheidt.de/fremdwb/>.
- [Lea94] Lea, D.: Christopher Alexander: An Introduction for Object-Oriented Designers. In: *ACM Software Engineering Notes*, Januar 1994.
- [LEW96] Loeckx, J.; Ehrich, H.-D.; Wolf, M.: *Specification of abstract data types*, Wiley, 1996.
- [LH89] Liu, L. C.; Horowitz, E.: A formal model for Software Project Management, *IEEE Transactions on Software Engineering*, Oct. 1989.
- [LK98] Lauder, A.; Kent, S.: Precise Visual Specification of Design Patterns, *Proceedings of the 12th European Conference on Object-Oriented Programming*, pp. 114 - 134, 1998.

- [Lon93] Lonchamp, J.: A Structured Conceptual and Terminological Framework for Software Process Engineering. In: Proceedings of the 2nd International Conference on the Software Process – Continuous Software Process Improvement, 1993, Berlin, Germany.
- [Lor97] Lorenz, D.: Tiling design patterns – a case study. In: ECOOP Proceedings, 1997.
- [LRS00] Lesney, C.; Rumpe, B.; Schwerin, W.: Prozessmuster und Produktmodell, 2000.
- [LSX94] Lieberherr, K.J.; Silva-Lepe, I.; Xiao, C.: Adaptive Object-Oriented Programming. Communications of the ACM 37, 5, 1994, pp. 94-101.

M

- [MA94] Montangero, C.; Ambriola, V.: OIKOS: Constructing Process-Centered Environment. In: [FKN94], p. 131-152, 1994.
- [MD98] Meszaros, G.; Doble, J.: A pattern language for pattern writing. In: Pattern Languages of Program Design, Addison-Wesley, 1998.
- [Mei96] Meijers, M.: Tool Support for Object-Oriented Design Patterns. Department of Computer Science, Utrecht University, 1996.
- [MHS98] Mellis, W.; Herzwurm, G.; Stelzer, D.: TQM der Softwareentwicklung, Vieweg, 1998.
- [MOF1.4] OMG: Meta Object Facility. Verfügbar unter <http://www.omg.org/docs/formal/02-04-03.pdf>.
- [MR01] Manns, M.; Rising, L.: Introducing Patterns (or any new idea) into an Organization, 2002.
- [MS99] Mellis, W.; Stelzer, D.: Das Rätsel des prozeßorientierten Softwarequalitätsmanagements. In: Wirtschaftsinformatik, 1, Februar 1999, S. 31-39.

N

- [NB02] Noble, J.; Biddle, R.: Patterns as Signs. In: Proceedings of ECOOP 2002, LNCS 2374, Springer, pp. 368-391, 2002.
- [NN99] Nielsen, H.R.; Nielsen, F.: Semantics with Applications – A formal Introduction. Revised edition. <http://www.daimi.au.dk/>, 1999.
- [Nob98a] Noble, J.: Classifying Relationships Between Object-Oriented Design Patterns. Australian Software Engineering Conference, 1998.
- [Nob98b] Noble, J.: Towards a Pattern Language for Object Oriented Design, <http://www.mcs.vuw.ac.nz/~kix/papers/towards.pdf>, 1998.
- [Non91] Nonaka, I.: *The Knowledge-Creating Company*. In: Harvard Business Review, November, December 1991, pp. 96-104.

O

- [OP] Organizational Patterns Homepage. <http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?OldOrgPatterns>

P

- [PCC93] Paulk, M. C.; Curtis, B.; Chrissis, M. B. et.al.: Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-024, 1993.

- [Ped] Paedagogical Patterns Project, <http://www.pedagogicalpatterns.org/>.
- [Pin01] Pinheiro da Silva, P.: A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, 2001.
- [Ple95] Plexousakis, D.: Simulation and Analysis of Business Processes Using GOLOG. In Proc. of the Conference on Organizational Computing Systems (COOCS'95), pp. 311-323, 1995.
- [PMP] Project Management Patterns, <http://c2.com/cgi/wiki?ProjectManagement-Patterns>.
- [Portland] The Portland Pattern Repository, <http://c2.com/ppr/>.
- [PPP] The Paedagogical Patterns Project, <http://www.pedagogicalpatterns.org/>.
- [PR97] Probst, G.; Romhardt, K.: Bausteine des Wissensmanagements – ein praxisorientierter Ansatz. In: <http://www.cck.uni-kl.de/wmk/papers/public/Bausteine>, 1997.
- [Prince2] Prince2, <http://www.prince2.ch>.
- [PS92] Peuschel, B.; Schäfer, W.: Concepts and implementation of a rule-based process engine, Proceedings of the 14th International Conference on Software Engineering, p.262-279, 1992.
- [pUML] Precise UML Homepage. www.puml.org.

R

- [Rei86] W. Reisig. Petrinetze. Springer, 1986.
- [Rie96] Riehle, D.: Patterns for Encapsulating Class Trees. In: Pattern Languages of Program Design, Volume 2. Vlissides, J., Kerth, N., & Coplien, J. (ed.), Addison-Wesley, 1996.
- [Rie97a] Riehle, D.: Composite design patterns. In: ECOOP Proceedings, 1997.
- [Rie97b] Riehle, D.: A role based design pattern catalog of atomic and composite patterns structured by pattern purpose. Technical Report 97-1-1, UbiLabs, 1997.
- [Ris99] Rising, L.: Patterns Mining. In: Handbook of Object Technology, CRC Press, 1999, Kapitel 38.
- [Ris00] Rising, L.: The Pattern Almanac 2000. Vlissides, J. (ed.), Addison-Wesley, Software Pattern Series, 2000.
- [RBP91] Rumbaugh, J.; Blaha, M.; Premerlani, W. et al.: Object-Oriented Modeling and Design. Prentice-Hall, 1991.
- [RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G.: The unified modeling language reference manual. Addison-Wesley, 1999.
- [RP01] Rolland, C.; Prakash, N.: A multi-Model View of Process Modeling. Requirements Engineering Journal, to appear.
- [RSS97] Reimer, W.; Schäfer, W.; Schmal, T.: Towards a dedicated object oriented software process modelling language. In Object-Oriented Technology – ECOOP 97 Workshop Reader, LNCS 1357, pages 299-302, 1997.
- [RUP] Rational Unified Process, Rational Software Corporation, Version 2005.02.01.01.

- [RZ96] Riehle, D.; Züllighoven, H.: Understanding and Using Patterns in Software Development. In: Theory and Practice of Object Systems, 1996, pp. 3-13.
- [Sch95] Schmidt, D.: Using Design Patterns to Develop Reusable Object-Oriented Communications Software. Communications of the ACM 38, 10, pp. 65-74, 1995.
- [Sch01] Scheer, A.-W.: ARIS – Modellierungsmethoden, Metamodelle, Anwendungen, Institut für Wirtschaftsinformatik der Universität des Saarlands, 2001.
- [Sch03] Schröder, J.: Die Process Pattern Workbench, Diplomarbeit, Universität Dortmund, 2003.
- [SHO95] Sutton, S. M.; Heimbigner, D.; Osterweil, L. J.: APPL/A, ACM Transactions on Software Engineering and Methodology (TOSEM), v.4 n.3, p.221-286, 1995.
- [SK88] Shinoda, Y.; Katayama, T.: Attribute grammar based programming and its environment. Proceedings of the 1st Hawaii International Conference on System Sciences, Software Track, pages 612-620, 1988.
- [SK95] Slonneger, K., Kurtz, B.: Formal Syntax and Semantics of Programming Languages – A Laboratory Based Approach. Addison-Wesley, 1995.
- [SKE00] Selic, B.; Kent, S.; Evans, A. (editors.): Proc. of 3rd Intl. Conf. UML 2000-Advancing the Standard, LNCS 1939, Springer, 2000.
- [SO97] Sutton, S.; Osterweil, L.: The Design of a Next-Generation Process Language. LNCS 1301, Springer, 1997, S. 142-158.
- [SP98] Stevens, P.; Pooley, R.: Systems Reengineering Patterns. In: *Proceedings of the Joint 6th European Software Engineering Conference and the 6th ACM SIGSOFT Intl. Symp. Foundations of Software Engineering (ESEC/FSE'99)*, LNCS 1687, 1998. Available at www.dcs.ed.ac.uk/home/pxs/talksEtc.html.
- [Sta97] Stapleton, J.: Dynamic Systems Development: The Method in Practice, Addison Wesley, 1997
- [STO95] Sutton, S.; Tarr, P.; Osterweil, L.: An Analysis of Process Languages. CMPSCI TR 95-78, Laboratory for Advanced Software Engineering Research, University of Massachusetts, 1995.
- [Stö00] Störrle, H.: Models of Software Architecture. Design and Analysis with UML. Dissertation, Universität München, 2000.
- [Stö01a] Störrle, H.: Describing Process Patterns with UML. LNCS 2077 Software Process Technology, Springer, 2001, pp. 173-181.
- [Stö01b] Störrle, H.: Describing Fractal Processes with UML. Technical Report, Universität München, 2001.
- [SW97] Schürr, A.; Winter, A.J.: Formal Definition and Refinement of UML's Module/Package Concept. In: ECOOP 97 Workshop Reader, LNCS 1357, pp. 211-215, Springer, 1997.
- [SWG01] Sa, J; Warboys, B.; Greenwood, M. et al.: Modeling a Support Framework for Dynamic Organizations as Process Pattern Using UML. In: Software Process Technology, LNCS 2077, Springer, 2001, pp. 203-216.

- T
- [Tic97] Tichy, W.: A catalogue of general-purpose software design patterns. In TOOLS USA 1997, 1997.
- U
- [UML1.5] Object Management Group: OMG Unified Modeling Language Specification, March 2003, Version 1.5, formal/03-03-01. Available at <http://www.omg.org/docs/formal/03-03-01.pdf>.
- V
- [Ver98a] Verlage, M.: Vorgehensmodelle und ihre Formalisierung. In: [KMO98], Kapitel III, S. 60-75.
- [Ver98b] Verlage, M.: Modellierungssprachen für Vorgehensmodelle. In: [KMO98], Kapitel IV, S. 76-94.
- [Vli97a] Vlissides, J.: Multicast. *C++ Report*, SIGS Publications, September 1997.
- [Vli97b] Vlissides, J.: Multicast – Observer = Typed Message. *C++ Report*, SIGS Publications, November-December 1997.
- [VM97] V-Modell '97: Entwicklungsstandard für IT-Systeme des Bundes. BWB IT 15, 1997.
- W
- [WB98] Wieringa, R.; Broersen, J.: Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams. Technical Report TUMI9803, Technische Universität München, 1998.
- [WEIT] WEIT-Forum, <http://www4.informatik.tu-muenchen.de/~gnatzm/nuke/>.
- [Wel95] Wells, T.D.: A Pattern Language for Dynamic Project Management. In [CS95]. Available via <http://st-www.cs.uiuc.edu/~plop/>.
- [Wie01] Wiemers, M.: Kombination verschiedener Vorgehensweisen (Prozessarten). In: Leichte Vorgehensmodelle, Shaker-Verlag, 2001, S. 143-150.
- [WMC96] Workflow Management Coalition, Workflow Standard – Interoperability, Abstract Specification, Document Number WfMC TC-1012, Version 1.0, 1996.
- [Whi95] Whitenack, B.: RAPPeL: A Requirements-Analysis-Process Pattern Language. In [CS95], pages 259–292. Available at www1.bell-labs.com/user/cope/Patterns/Process/index.html.
- [Wikipedia] Wikipedia – die freie Enzyklopädie, <http://de.wikipedia.org/wiki/Hauptseite>.
- X
- [Xie01] Xie, T.: A Linguistic Study of Process Modeling Languages, Course Project Paper, University of Washington, Seattle, 2001.
- [XP] Extreme Programming Homepage, <http://extremeprogramming.org>.
- Z
- [Zim95] Zimmer, W: Relationships Between Design Patterns. In: [CS95], 1995.

Abbildungsverzeichnis

Abbildung 1-1: Aufbau der Arbeit	7
Abbildung 2-1: Prozessmuster-Beziehungen bei Störle	14
Abbildung 2-2: Klassifizierung von Mustern nach Phasen	16
Abbildung 2-3: Klassifizierung von Mustern nach Czichy	17
Abbildung 2-4: Beziehungen zwischen GoF-Entwurfsmustern	21
Abbildung 2-5: Klassifikation von Entwurfsmusterbeziehungen nach Zimmer	22
Abbildung 2-6: Use-Beziehung von Noble	22
Abbildung 2-7: Nutzungsbeziehung bei Ambler	23
Abbildung 2-8: Prozessmusterbeziehungen bei Gnatz et al.	24
Abbildung 2-9: Erzeugnisstruktur bei Bergner et al.	24
Abbildung 2-10: Microprocess for Patterns	26
Abbildung 2-11: Das UML-Metamodell und seine Einbettung in die Modellhierarchie ...	32
Abbildung 2-12: Gegenüberstellung von leichten und schweren Vorgehensmodellen (aus [CH02a])	35
Abbildung 2-13: Produktfluss im V-Modell 97	37
Abbildung 2-14: Disziplin „Requirements“ (aus [RUP])	38
Abbildung 2-15: Workflow Detail „Analyze the Problem“ (aus [RUP])	39
Abbildung 2-16: RUP-Aktivität „Capture a Common Vocabulary“ (aus [RUP])	40
Abbildung 3-1: Skizzenhafte Darstellung eines Prozessmusterkatalogs	47
Abbildung 3-2: Elemente eines Prozessmusters	48
Abbildung 3-3: (De-)komposition eines Objekts	49
Abbildung 3-4: Beispiel: Kontext des Problems „How to perform Reviews?“	51
Abbildung 3-5: Beispiel: Kontext des Prozessmusters „Review“	51
Abbildung 3-6: Beispiel: Prozess des Prozessmusters „Review“	53
Abbildung 3-7: Beispiele für die Sequence-Beziehung	56
Abbildung 3-8: Veranschaulichung der Use-Beziehung	57
Abbildung 3-9: Kompositmuster und Komponentenmuster	58
Abbildung 3-10: Zusammenhang zwischen Aktivität, Problem und Prozessmuster	58
Abbildung 3-11: Beispiel: Prozess des Prozessmusters „Review-Session“	59
Abbildung 3-12: Supermuster „Manual QA“ und Submuster „Review“	60
Abbildung 3-13: Verfeinerung von Prozessmuster und Problem	61
Abbildung 3-14: Prozessvariante Prozessmuster	62
Abbildung 3-15: Komposition einer Softwarespezifikation aus einzelnen Objekten	62
Abbildung 3-16: Sequence-Beziehung ermöglicht durch Objektkomposition	63
Abbildung 3-17: Use-Beziehung ermöglicht durch Objektkomposition	63
Abbildung 3-18: Refinement-Beziehung ermöglicht durch Objektkomposition	64
Abbildung 3-19: Processvariance-Beziehung ermöglicht durch Objektkomposition	65
Abbildung 4-1: Die Modellierungsebenen von PROPEL	69
Abbildung 4-2: Integration der PROPEL-Metaklassen in das UML-Metamodell	71
Abbildung 4-3: Abstrakte Syntax von PROPEL – Teil 1	72
Abbildung 4-4: Abstrakte Syntax von PROPEL – Teil 2	73
Abbildung 4-5: Modellierung von Inout-Objekten	76
Abbildung 4-6: Abstrakte Syntax des Pakets Process Pattern Relationship	85
Abbildung 4-7: Abstrakte Syntax des Pakets Process Pattern Graphs – Graphs	95

Abbildung 5-1:	Aufteilung einer Sprache in Syntax und Semantik	99
Abbildung 5-2:	Beispiel: Abstrakte Syntax eines Aktivitätsdiagramms	102
Abbildung 5-3:	Verknüpfung von B/E-Systemen	106
Abbildung 5-4:	Verfeinerung/Inklusion von B/E-Systemen	108
Abbildung 5-5:	Zwei Systeme (Y und Y'), die zu einem System Y'' mit konfligierenden Transitionen ineinandergefügt werden	110
Abbildung 5-6:	Abbildung von Aktivitätsdiagrammen auf Elemente eines Petri-Netzes	116
Abbildung 5-7:	Abbildung des Prozesses eines Prozessmusters auf ein etikettiertes B/E-System	121
Abbildung 5-8:	Beispiel für eine Sequence-Beziehung	122
Abbildung 5-9:	Aktivität b mit leerem resultierenden Kontext	123
Abbildung 5-10:	Mehrfache Verwendung des Objekts o_4 im initialen Kontext des Nachfolgermusters	123
Abbildung 5-11:	Beispiel für eine Use-Beziehung	125
Abbildung 5-12:	Beispiel für eine Refinement-Beziehung	126
Abbildung 5-13:	Beispiel für eine Processvariance-Beziehung	127
Abbildung 5-14:	Dekomposition bei der Sequence-Beziehung	128
Abbildung 5-15:	Dekomposition bei der Use-Beziehung	129
Abbildung 5-16:	Dekomposition bei der Refinement-Beziehung	130
Abbildung 5-17:	Dekomposition bei der Processvariance-Beziehung	131
Abbildung 6-1:	Notationsbeispiel Prozessmuster	134
Abbildung 6-2:	Notationsbeispiel Problem	135
Abbildung 6-3:	Notationsbeispiele für Objekte (links), Ereignisse (Mitte) und Aggregation (rechts)	135
Abbildung 6-4:	Notationsbeispiel für Objekt- und Ereignisspezialisierung	136
Abbildung 6-5:	Notationsbeispiel für ein Problemdiagramm	136
Abbildung 6-6:	Notationsbeispiel für Kontext	137
Abbildung 6-7:	Notationsbeispiele für Action States (links), Action States mit zugehörigem Problem (mitte) und Action States mit zugehörigem Problem und Prozessmuster (rechts)	138
Abbildung 6-8:	Notationsbeispiele für Rolle	138
Abbildung 6-9:	Notationsbeispiele für Werkzeuge	139
Abbildung 6-10:	Notationsbeispiel für ein Prozessdiagramm	139
Abbildung 6-11:	Notationsbeispiele: Einfache Sequence-Notation (A), mit vereinigten Pfeilenden (B) oder alternativ mit Kasten (C), abgekürzte Schreibweise für mehrere Sequence-Beziehungen (D), für alternative Vorgängermuster (E) und für die Darstellung eines Vorgängermusters aus ein Menge von Vorgängermustern (E)	141
Abbildung 6-12:	Notationsbeispiel Use	141
Abbildung 6-13:	Notationsbeispiel für Problemverfeinerung	142
Abbildung 6-14:	Notationsbeispiel für Refinement	143
Abbildung 6-15:	Notationsbeispiel für Prozessvarianten	143
Abbildung 6-16:	Notationsbeispiel für ein Beziehungsdiagramm	144
Abbildung 6-17:	Notationsbeispiel für ein Prozessmusterdiagramm	145
Abbildung 6-18:	Notationsbeispiel für ein Katalogdiagramm – Use-Sicht	146
Abbildung 7-1:	Aufbau der RUP-Disziplin „Requirements“	148
Abbildung 7-2:	Katalogdiagramm – Sicht „Use-Beziehung“	149
Abbildung 7-3:	Katalogdiagramm – Sicht „Sequence-Beziehung“	151
Abbildung 7-4:	Katalogdiagramm – Sicht „Refinement-Beziehung“	152

Abbildung 7-5:	Katalogdiagramm – Sicht „Processvariance-Beziehung“	152
Abbildung 7-6:	Objekt- und Ereignisstruktur – Aggregation	153
Abbildung 7-7:	Objekt- und Ereignisstruktur – Generalisierung	154
Abbildung 7-8:	Problemdiagramm „How to elicit the Requirements for an existing system?“	155
Abbildung 7-9:	Problemverfeinerungsdiagramm „How to analyse the Requirements?“	155
Abbildung 7-10:	Prozessmusterdiagramm „Reanalyse the Requirements“	156
Abbildung 7-11:	Problem „How to refine the Problem“	157
Abbildung 7-12:	Prozessmusterdiagramm „Refine the Problem“	158
Abbildung 7-13:	Problem „How to manage the requirements?“	159
Abbildung 7-14:	Prozessmusterdiagramm „Develop Requirements Management Plan“	159
Abbildung 7-15:	Problem „How can Actors and Use-Cases be found?“	160
Abbildung 7-16:	Verfeinerung des Problems „How to model the system behaviour?“	160
Abbildung 7-17:	Prozessmusterdiagramm „Find Actors and Use Cases“	161
Abbildung 7-18:	Problem „How can a Glossary be produced?“	162
Abbildung 7-19:	Prozessmusterdiagramm „Capture a Common Vocabulary“	162
Abbildung 7-20:	Problem „How can a Glossary be produced (detailed)?“	163
Abbildung 7-21:	Verfeinerung des Problems „How can a Glossary be produced?“	163
Abbildung 7-22:	Prozessmusterdiagramm „Capture Vocabulary by System Analyst“	164
Abbildung 7-23:	Prozessmusterdiagramm „Capture Vocabulary by Project Team“	165
Abbildung 7-24:	Problem „How to envision the system?“	166
Abbildung 7-25:	Prozessmusterdiagramm „Develop Vision“	166
Abbildung 7-26:	Problem „How to review Requirements?“	167
Abbildung 7-27:	Prozessmusterdiagramm „Review Requirements“	168
Abbildung 7-28:	Prozessmusterdiagramm „Walkthrough Requirements“	169
Abbildung 8-1:	Use Cases der Benutzerverwaltung	174
Abbildung 8-2:	Use Cases der Problemdefinition	175
Abbildung 8-3:	Patterndefinition	177
Abbildung 8-4:	Anwendungsfälle der Projektverwaltung	179
Abbildung 8-5:	Architektur der Process Pattern Workbench	181
Abbildung 8-6:	Aufteilung der Funktionsschicht auf dem EJB-Server	182
Abbildung 8-7:	Navigation der Process Pattern Workbench	182
Abbildung 8-8:	Menüleiste der Process Pattern Workbench	183
Abbildung 8-9:	Benutzer anlegen (links, mittig) und Benutzer suchen (mittig, rechts)	183
Abbildung 8-10:	Rolle anlegen (oben) und Rolle suchen (unten)	184
Abbildung 8-11:	Benutzer-Rollen-Verknüpfung (links oben, unten) und Benutzer-Rollen-Verknüpfungen suchen (oben rechts)	184
Abbildung 8-12:	Anlegen eines Objekts (links, Mitte) und Suchen eines Objekts (rechts)	185
Abbildung 8-13:	Definition eines Problems	185
Abbildung 8-14:	Suche und Detailansicht eines Problems	186
Abbildung 8-15:	Entfernen eines Problems	186
Abbildung 8-16:	Erzeugen eines Prozessmusters (links oben) über die Auswahl eines Problems (rechts oben) und Vergabe eines Namens (unten)	187
Abbildung 8-17:	Modellierung eines Prozesses durch Auswahl der Funktion „Edit Solution“, (links), Definition von Aktivitäten und Kontexten (rechts)	188
Abbildung 8-18:	Dialog zur Modellierung einer einzelnen Aktivität	188
Abbildung 8-19:	Betrachtung eines finalisierten Prozesses und Zuordnung von Problemen zu Aktivitäten (oben), Ansicht der Aktivitäts-Problem-Paare (unten)	189
Abbildung 8-20:	Suche nach Prozessmustern (links), Ergebnis der Suche (rechts)	189

Abbildung 8-21: Entfernen eines Prozessmusters aus dem Prozessmusterkatalog	190
Abbildung 8-22: Definition von Prozessmusterbeziehungen – Teil 1	190
Abbildung 8-23: Definition von Prozessmusterbeziehungen – Teil 1	191
Abbildung 8-24: Erstellen (oben links, oben mittig und unten) und Öffnen von Workbenchprojekten (oben rechts) 192	
Abbildung 8-25: Auswahl des initialen Prozessmusters und Zuordnung eines Prozessverantwortlichen 192	
Abbildung 8-26: Festlegung von Sequenzen von Prozessmusterinstanzen	193
Abbildung 8-27: Dokumentation der Ausführung von Aktivitäten	193
Abbildung 8-28: Use Cases der Katalogverwaltung	195
Abbildung 8-29: Anwendungsfälle des Reportings	197
Abbildung 9-1: Aktivitäten bei Entwicklung und Einsatz von Prozessmustern	203
Abbildung A-1: Pakete und Subpakete des UML-Metamodells	225
Abbildung A-2: Abstrakte Syntax des Pakets Activity Graphs (aus [UML1.5])	226
Abbildung A-3: Konkrete Syntax des Pakets Activity Graphs – Beispiel Activity Diagram (aus [UML1.5])227	
Abbildung A-4: RUP-Aktivität „Find Actors and Use Cases“ (aus [RUP])	229
Abbildung A-5: RUP-Aktivität „Review Requirements“ (aus [RUP])	230
Abbildung A-6: RUP-Aktivität „Develop Vision“ (aus [RUP])	231
Abbildung A-7: Einordnung der Crystal Methodologies (adaptiert aus [CH02a])	232

Tabellenverzeichnis

Tabelle 2-1:	Vergleich verschiedener Musterschemata.....	13
Tabelle 2-2:	Prozessmodellierungssprachen im Vergleich.....	30
Tabelle 2-3:	Ansätze zur Formalisierung von UML-Aktivitätsdiagrammen.....	33
Tabelle 2-4:	Wertesystem der Agile Alliance.....	36
Tabelle 2-5:	Vergleich verschiedener Vorgehensmodelle bzw. Methoden.....	43
Tabelle 3-1:	Klassifizierungsbeispiele.....	47
Tabelle 3-2:	Prozessmusterschema.....	66
Tabelle 3-3:	Problemschema.....	67
Tabelle 4-1:	Zuordnung der Konzepte zu den Syntax-Elementen von PROPEL.....	70
Tabelle 5-1:	Syntaxregeln.....	100
Tabelle 6-1:	Abbildung von Notationslementen auf Metamodellelemente.....	133
Tabelle 10-1:	Bewertung der zu erfüllenden Anforderungen.....	209
Tabelle A-1:	Alexandersches Schema.....	215
Tabelle A-2:	Coplien-Schema.....	216
Tabelle A-3:	GoF-Schema.....	217
Tabelle A-4:	Störrle-Schema.....	218
Tabelle A-5:	ZEN-Schema.....	218
Tabelle A-6:	Klassifikation von Entwurfsmuster-Beziehungen nach Zimmer.....	219
Tabelle A-7:	Klassifikation von Entwurfsmusterbeziehungen nach Noble.....	220
Tabelle A-8:	Sichten, Diagramme und Konzepte des UML-Metamodells.....	224
Tabelle B-1:	Syntaktische Domänen.....	233
Tabelle B-2:	Semantische Funktionen.....	234
Tabelle C-1:	Eingesetzte Werkzeuge.....	237

Glossar

		A
Agilität	Wird gerne mit dem Motto „light but sufficient“ erläutert: Das Vorgehensmodell soll so wenig wie möglich vorgeben (light), dabei aber das Projekt noch ausreichend unterstützen (sufficient).	
Aktivität	Repräsentiert die Ausführung einer atomaren Aktion. Besitzt wie Prozesse auch einen initialen und einen resultierenden Kontext. Basiert auf UML-Action States.	
Aktivitätsdiagramm	UML-Konzept zur Beschreibung des dynamischen Verhaltens in der Regel von Systemen.	
Aspekt	Gibt Aufschluss über die thematische Ausrichtung des Prozessmusters.	
		B
B/E-System	Bedingungs-/Ereignis-System, spezielle Art der Petri-Netze.	
		D
Domäne	Bezeichnet den fachlichen Einsatzbereich von Prozessmustern.	
		E
eB/E-System	Etikettierte Bedingungs-/Ereignis-Systems, spezielle Art der Petri-Netze.	
Entwurfsmuster	Beschreibt in der Regel einen softwaretechnischen Entwurf für ein softwaretechnisches Problem.	
Ereignis	Bezeichnet ein Geschehen, das einen Prozess auslöst oder während eines Prozesses ausgelöst wird. Ereignisse sind Teil des Prozessmuskontexts.	
Ergebnismuster	Beschreibt in seiner Lösung ein Prozessresultat, welches ein bestimmtes Problem in einem bestimmten Kontext löst. Der Prozess selbst bleibt bei Ergebnismustern stets unberücksichtigt, d.h. im Vordergrund steht das Ergebnis.	
Extreme Programming	Bekanntester Vertreter der agilen Methoden. Geprägt durch kleine Iterationszyklen und intensive Kommunikation.	
		G
Gang-of-Four	Die Herren Gamma, Helm, Johnson, und Vlissides, die das berühmte „Design Patterns“-Buch [GHJ96] geschrieben haben	

I		
	Inoutparameter	Objekte oder Ereigniss, das von einer Aktivität konsumiert und produziert wird.
	Inputparameter	Objekte oder Ereigniss, das von einer Aktivität konsumiert wird.
K		
	Katalogdiagramm	Stellt alle Prozessmuster, die in einer bestimmten Beziehung stehen, graphisch dar. Für jeden Beziehungstypen gibt es ein Katalogdiagramm.
	Kontext	Der Kontext eines Prozessmusters definiert die Bedingungen, die vor und nach Anwendung des Prozessmusters erfüllt sein müssen.
L		
	Leichtgewichtiges Vorgehensmodell	Vorgehensmodell, bei dem Fertigkeiten, die Disziplin und das Verständnis von hoher Bedeutung sind. Gegensatz zu schwergewichtigen Vorgehensmodellen.
M		
	Metamodell	Modell, das beschreibt, wie Modelle gebaut werden.
	Methode	Methoden sind planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen. [Bal98]
	Musterhandbuch	Ein Musterhandbuch fasst die Konzepte einer Domäne in Gestalt von Mustern zusammen.
	Musterkatalog	Darstellung von Mustern und – in eingeschränkter Form – deren Beziehungen. Bekanntester Katalog: [GHJ96].
	Musterklassifizierung	Die Klassifizierung eines Musters dient dazu, ein Muster anhand verschiedener Kriterien einordnen zu können.
	Musterschema	Gibt die Struktur von Mustern eines bestimmten Typs vor.
	Mustersequenz	Mustersequenzen sind häufig angewendete Abfolgen von Prozessmustern.
	Mustersprache	Mustersprachen gelten als Mengen miteinander verknüpfter Muster. Durch die enge Verknüpfung der Muster sind mehr und komplexere Probleme lösbar als bei Musterkatalogen und Mustersystemen.
	Mustersystem	Vergleichbar mit einem Musterkatalog, jedoch strukturierter. Ein Mustersystem ist organisiert in Gruppen und Untergruppen, die zueinander in Beziehung stehen und die auf verschiedenen Granularitätsstufen existieren.
N		
	Notation	Regelsatz zur Schreibweise und Auswertereihenfolge mathematischer Ausdrücke. Im Kontext der UML graphische Darstellung von UML-Konzepten.

O

Objekt	Repräsentiert ein bestimmtes Artefakt, das während eines Prozesses konsumiert, erstellt oder modifiziert wird. Objekte sind Teil des Kontexts von Prozessmustern.
OCL-Regel	Object Constraint Language; Regeln dieser Sprache schränken die abstrakte Syntax der UML in Form von Constraints ein.
Outputparameter	Objekte oder Ereigniss, das von einer Aktivität produziert wird.

P

Pattern Management	Alle mit Mustern verknüpfte Tätigkeiten wie Muster-Dokumentation, Problem-Identifikation und -Spezifikation, Mustersuche, Musterauswahl, Musternanwendung und Mustereinführung und -einsatz.
Pattern Mining	Prozess zur Identifikation und Dokumentation von Wissen in Form von Mustern.
Petri-Netz	Ist eine mathematische Repräsentation von verteilten Systemen. Besteht aus Systemzuständen (Stellen), Übergängen (Transitionen) und gerichteten Kanten zwischen Stellen und Transitionen, die diese miteinander verbinden.
Phase	Beschreibt, in welchem Projekt- oder Arbeitsabschnitt ein Prozessmuster eingesetzt werden kann.
Problem	Probleme sind eingebettet in einen Rahmen von Ereignissen und Objekten, die dieses Problem bedingen bzw. die durch Lösung des Problems ausgelöst werden.
Problemdiagramm	Stellt ein PROPEL-Problem, dessen Kontext und lösende Prozessmuster graphisch dar.
Problemschema	Definiert, mit welchen Elementen ein Problem zu beschreiben ist.
Process Pattern Workbench	Prototyp zur softwaretechnischen Realisierung der PROPEL-Konzepte.
Processvariance-Beziehung	Verknüpft zwei Prozessmuster, die das gleiche Problem lösen.
PROPEL	Process Pattern Description Language; dient zur Beschreibung von Prozessmustern.
Prozess	Der Prozess repräsentiert eine Lösung für das durch das Prozessmuster zugeordnete Problem. Basiert auf UML-Aktivitätsdiagrammen.
Prozessdiagramm	Stellt einen PROPEL-Prozess graphisch dar.
Prozessmuster	Prozessmuster beschreiben in der Regel Lösungen für Probleme in der Softwareentwicklung in Gestalt von Prozessen.
Prozessmusterdiagramm	Stellt ein PROPEL-Prozessmuster, dessen Problem, Beziehungen zu anderen Prozessmustern und den Prozess graphisch dar.

Prozessmusterschema	Definiert, mit welchen Elementen ein Prozessmuster zu beschreiben ist.
Prozessmusterkatalog	Repräsentiert eine Menge von Prozessmustern und eine Menge von Prozessmusterbeziehungen, die die Prozessmuster miteinander verknüpfen.
R	
Rational Unified Process	Der Rational Unified Process (RUP) ist ein käuflich zu erwerbendes Produkt von Rational.
Refinement-Beziehung zwischen Problemen	Verknüpft ein abstrakteres Problem (sogenanntes Superproblem) und ein detaillierteres Problem (sogenanntes Subproblem).
Refinement-Beziehung zwischen Prozessmustern	Verknüpft ein abstrakteres Prozessmuster (sogenanntes Supermuster) und ein detaillierteres Prozessmuster (sogenanntes Submuster).
Rolle	Beschreibt Aufgaben, Verantwortlichkeiten und Rechte einer Gruppe von Individuen zur Erreichung eines Ziels. Wird Prozessen und Aktivitäten zugeordnet.
S	
Schwergewichtiges Vorgehensmodell	Vorgehensmodell, bei dem Prozesse, Formailität und Dokumentation von hoher Bedeutung sind.
Semantik	Definiert die Bedeutung von Worten, Ausdrücken und Sätzen.
Semantische Domäne	Menge mathematischer Objekte.
Semantische Funktion	Bildet syntaktische Domänen auf semantische Domänen ab.
Semantische Gleichung	Spezifizieren die semantischen Funktionen für einzelne syntaktische und semantische Objekte.
Sequence-Beziehung	Bei der Sequence-Beziehung werden Prozessmuster sequenziell miteinander verknüpft.
Stereotyp	Zum Zweck der UML-Erweiterung definierte Subklasse einer UML-Klasse.
Syntaktische Domäne	Menge von syntaktischen Objekten.
Syntaktische Regel	Gibt an, wie Objekte der syntaktischen Domänen strukturiert und miteinander verknüpft werden.
Syntax	Grammatische Regeln, die definieren, wie Symbole einer Sprache zu Worten und Sätzen kombiniert werden können.
U	
Unified Modeling Language (UML)	Eine durch die OMG standardisierte graphische Sprache zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Artefakten software-intensiver Systeme.
Use-Beziehung	Die Use-Beziehung bedeutet, dass ein Prozessmuster (sogenanntes Kompositmuster) ein anderes Prozessmuster (sogenanntes Komponentenmuster) nutzt.

Use Case	Anwendungsfall, der ein bestimmtes funktionales Verhalten beschreibt.	
V-Modell des Bundes	Wurde ursprünglich für den Bereich der deutschen Bundeswehr und der Bundesverwaltung geschaffen, wird mittlerweile aber auch von vielen Unternehmen als Vorgehensmodell-Standard für die Softwareentwicklung eingesetzt.	V
Werkzeug	Repräsentiert meistens ein Softwaresystem. Unterstützt die Ausführung einer Aktivität.	W
Workflowmuster	Workflowmuster dienen dazu, Workflowmanagementsysteme vergleichen und bewerten zu können.	

Sachindex

A

Adaptive Software Development 36
Agile Alliance 35
Agile Software Development Manifesto 35
Agilität 35
Aktivierte Transition 105
Aktivität 53
Aspekt 46

B

B/E-System (Bedingungs/Ereignis-System) 104

C

CMII 34
Common Warehouse Metamodel (CWM) 32
compound transitions 84
Connect-Operator 107
Crystal Family 41

D

deadlockfrei 105
Deming-Kreis 204
Denotationelle Semantik 99
Domäne 46
Dynamic System Development (DSDM) 36

E

Ereignis 49
Erweiterte Backus-Naur-Form 233
Extreme Programming 41

F

Feature Driven Development (FDD) 36
Folgemarkierungsrelation 105
Forces 10
From-Within-Strategie 170

G

Gang of Four (GoF) 9

H

HERMES 34

I

ITIL 34

K

Kante 104
Komposition von Objekten und Ereignissen 49
Konflikt 105
Kontext 51

L

Language for Pattern Uniform Specification (LePUS) 15
Leichtgewichtige Vorgehensmodelle 35

M

Markierung 105
Meta Object Facility (MOF) 31
Microprocess for Patterns 26
Muster
 Organisation
 Katalog 18
 Musterhandbuch 19
 Mustersprache 19
 Mustersystem 18
Musterdefinitionen 10
Musterschema 12
 Alexandersches Schema 12
 Coplien-Schema 12
 Enterprise Application Architecture 12
 GoF-Schema 12
 Objektorientierte Analyse 12
 Portland-Schema 12
 Projektmanagement-Schema 12
 Riehle-Schema 12
 SmallTalk 12
 Störle-Schema 12
 ZEN-Schema 13
Mustersequenz 197
Mustertyp

- Architekturmuster 11
- Entwurfsmuster 11
- Idiom 11
- Organisationsmuster 11
- Pädagogikmuster 11
- Philosophiemuster 11
- Projektmanagementmuster 11
- Mustertypen 11

- N**
- Nachbereich 104

- O**
- Objekt 49

- P**
- Pattern Designer 202
- Pattern Mining 26, 204
- pattern selection problem 27
- Pattern User 202
- Phase 46
- Prince2 34
- Problem 50
 - How can a Glossary be produced (detailed)? 163
 - How can a Glossary be produced? 161
 - How can Actors and Use-Cases be found? 159
 - How to elicit the Requirements for an existing system? 154
 - How to envision the system? 165
 - How to manage the requirements? 158
 - How to refine the Problem? 157
 - How to review Requirements? 167
- Problems
 - How to model the system behaviour? 160
- Problemschema 67
- Process Pattern Management 201
 - Aufbauorganisation 202
 - Einführung und Etablierung 202
 - Einsatz 203
 - Infrastruktur 201
- Process Pattern Package 72
- Process Pattern Workbench **173**
- Processvariance-Beziehung 61
- PROPEL-Metaklasse
 - ActivityProblemMapping 80
 - Aspect 94
 - Context 77
 - OFSComposition 83
 - Problem 76, 85
 - ProblemGraph 95
 - Process 78
 - ProcessPattern 73, 86
 - ProcessPatternCatalog 94
 - ProcessPatternGraph 96
 - ProcessPatternRelationship 86
 - Processvariance 92
 - Refinement 91
 - RefineProblem 90
 - Role 80
 - Sequence 86
 - Tool 81
 - Use 87
- PROPEL-Metamodell 69
- Prozess 52
- Prozessmuster 11
 - Capture a Common Vocabulary 162
 - Capture Vocabulary by Project Team 164
 - Capture Vocabulary by System Analyst 163
 - Develop Requirements Management Plan 159
 - Develop Vision 166
 - Find Actors and Use Cases 160
 - Reanalyse the Requirements 156
 - Refine the Problem 158
 - Review Requirements 167
 - Walkthrough Requirements 168
- Prozessmusterkatalog 47
- Prozessmusterschema 65
- Pseudozustand 84
- pUML 32

- Q**
- quality without a name 14

- R**
- Rational Unified Process 1, 38
- Refinement-Beziehung 59
- RefineProblem-Beziehung 60
- Relativer Rand 104
- Rolle 54
- Rolle (Workbench)
 - Pattern Designer 173
 - Pattern User 173

Workbench Administrator 173
Workbench Developer 173
Run-to-completion-Annahme 120

S

Schwergewichtige Vorgehensmodelle 34
SCRUM 36
Semantische Funktion 111
Semantische Funktionen 234
Semantische Gleichung 111
Sequence-Beziehung 55
Softwareprozessmodellierungssprache 28
SPML 1
Stelle 104
Stellenberandet 104
syntactic sugar 84
Syntaktische Domäne 233

T

The Hillside Group 9
Top-Down-Strategie 169
Transition 104

U

UML-Metaklasse
 ActionState 82
 ObjectFlowState 82, 85
 PseudoState 84
UML-Metamodell 31
UML-Profil 70
Use Case 173
 Benutzer anlegen 174
 Benutzer mit Rolle verknüpfen 174
 Login/Logout 194
 Rolle / Benutzer suchen 175
 Rolle anlegen 174
Use-Beziehung 56

V

V-Modell 97 1, 37
Vorbereich 104

W

Werkzeug 54
Workflowmuster 11

Autorenindex

A

Alexander, Christopher 9, 10
Ambler, Scott 22

B

Beck, Kent 12, 41
Bergner, Klaus 16, 24
Berten, André 16
Börger 33
Buschmann, Frank 15, 19

C

Cockburn, Alistair 12, 41
Coplien, Jim 16, 19
Czichy, Thoralf 17, 20, 26

E

Eden, Amnon H. 14
Eshuis, Henrik 33

F

Förster, Alexander 25
Fowler, Martin 12

G

Gamma, Erich 10
Gang of Four (GoF) 10
Gehrke, Thomas 33, 115
Gnatz, Michael 10, 23
GoF 15, 18, 20, 46

M

Manns, Mary-Linn 202

N

Noble, James 22, 27

O

Osterweil, Leon 4

P

Pinheiro da Silva, Paulo 33

R

Riehle, Dirk 15, 17, 19
Rising, Linda 17, 26, 202

S

Störrle, Harald 16, 23, 45

T

Tichy 15

V

van der Aalst 11
Verlage, Martin 1, 28

W

Wiemers, M. 1

Z

Zimmer, Walter 21
Züllighoven, Heinz 17, 19

Wissenschaftlicher Werdegang

Beruflicher Werdegang

Oktober 2003 - gegenwärtig Lehrstuhl für angewandte Telematik/e-Business, Universität Leipzig
Wissenschaftliche Mitarbeiterin

April 2000 - September 2003 adesso AG, Competence Center Processes, Dortmund
Consultant

März 1998 - März 2000 Fraunhofer-Institut für Software- und Systemtechnik (ISST), Abteilung Qualitätsmanagement, Dortmund
Wissenschaftliche Mitarbeiterin und Consultant

Ausbildung

1991 - 1998 Studium der Informatik, RWTH Aachen
Abschluss: Dipl.-Inform.

1993 - 1997 Studium der Betriebswirtschaftslehre, RWTH Aachen
Abschluss: Vordiplom

Selbständigkeits- erklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtlich Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialien oder erbrachten Dienstleistungen als solche gekennzeichnet.

Leipzig, 8. Oktober 2004

(Ort, Datum)

.....

(Unterschrift)

