Modellgetriebener Einsatz von Softwareentwurfsmustern

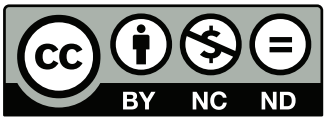
von
Dietrich Travkin

Diese Dissertation wurde am 20. Dezember 2017 bei der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn eingereicht und am 18. Juni 2018 vor der Promotionskommission verteidigt.

Promotionskommission

Vorsitzender:	Jun.-Prof. Dr.-Ing. Anthony Anjorin *Universität Paderborn*
Koreferat:	Prof. Dr. Albert Zündorf *Universität Kassel*
Beisitz:	Prof. Dr. Heike Wehrheim *Universität Paderborn*
Beisitz:	Dr. Matthias Meyer *Fraunhofer-Institut für Entwurfstechnik Mechatronik IEM*
Beisitz:	Dr. Stefan Sauer *Universität Paderborn, Software Quality Lab s-lab,* *Software Innovation Campus Paderborn SICP*

Das vorliegende Werk wurde über den Publikationsservice der Universitätsbibliothek Paderborn digital veröffentlicht und ist über folgende DOI (Digital Object Identifier) und URN (Uniform Resource Name) referenzierbar.
DOI: 10.17619/UNIPB/1-335 (http://dx.doi.org/10.17619/UNIPB/1-335)
URN: urn:nbn:de:hbz:466:2-30882 (http://nbn-resolving.de/urn:nbn:de:hbz:466:2-30882)



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut
Fachgebiete Softwaretechnik & Model-Driven Software Engineering
Zukunftsmeile 1
33102 Paderborn

Modellgetriebener Einsatz von Softwareentwurfsmustern

Dissertation

zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Dietrich Travkin

Betreuer:
Prof. Dr. Wilhelm Schäfer
und
Jun.-Prof. Dr.-Ing. Anthony Anjorin

Paderborn, Juni 2018

Tag der Einreichung: 20. Dezember 2017
Tag der Verteidigung: 18. Juni 2018

Zusammenfassung

Softwareentwurfsmuster haben sich in der Softwareentwicklung bewährt und sind weit verbreitet. Sie beschreiben unter anderem, wie Software erweitert oder angepasst werden kann, weswegen das Wissen über in einem Softwaresystem eingesetzte Entwurfsmuster vor Entwurfsänderungen besonders wichtig ist. Sind solche Musterverwendungen nicht geeignet dokumentiert oder bekannt, kann es nachweislich zu erhöhtem Wartungsaufwand, Entwurfsfehlern und schließlich zu Design-Erosion kommen.

Zur Reduzierung von Entwurfsfehlern und zur Unterstützung von Entwicklern beim Einsatz von Entwurfsmustern stelle ich in dieser Arbeit meinen modellgetriebenen Ansatz zur semiautomatischen Musteranwendung, Dokumentation und Validierung vor. Dabei werden Entwickler bei der Anwendung von Entwurfsmustern in einem Softwareentwurfsmodell geleitet. Nach Vorgaben der Entwickler wird eine Musterimplementierung im Entwurfsmodell generiert. Gleichzeitig wird die Anwendungsstelle (Musterrollen und ihre Implementierungen) automatisch in einem separaten Modell genau erfasst.

Durch ihre explizite Erfassung können Anwendungsstellen nicht mehr so einfach übersehen werden, weil sie bei der Darstellung des Entwurfs, z.B. in Klassendiagrammen, mit Hilfe dafür entwickelter Notationen jederzeit hervorgehoben oder separat dargestellt werden können (d.h. sie sind dokumentiert). Für die Darstellung der detailliert erfassten Anwendungsstellen werden verschiedene Detailgrade angeboten.

Wird das Entwurfsmodell geändert, kann seine Konformität zu den angewandten Mustern mit Hilfe der erfassten Musteranwendungsstellen automatisch geprüft und Abweichungen aufgezeigt werden. So können Entwickler einst angewandte Entwurfsmuster entweder bewusst durch andere Lösungen ersetzen oder die versehentliche Abweichung korrigieren.

Zu den wichtigsten Beiträgen dieser Arbeit gehören

(i) eine Musterspezifikationssprache, mit Hilfe derer objektorientierte Entwurfsmuster samt Struktur, Verhalten, Abhängigkeiten und Implementierungsvarianten kompakt beschrieben werden können,

(ii) ein semiautomatisches Verfahren zur Generierung einer Musterimplementierung aus einer Musterspezifikation und zur gleichzeitigen Erfassung der Anwendungsstelle in einem detaillierten Modell,

(iii) verschiedene Notationen zur Darstellung von Musteranwendungsstellen mit wählbarem Detailgrad,

(iv) ein Verfahren zur Prüfung von Musterimplementierungen auf Konsistenz zur Musterspezifikation,

(v) eine Evaluierung des Verfahrens anhand einer prototypischen Implementierung und diverser Anwendungsszenarien.

Abstract

Software design patterns are approved and well-known to software developers. Knowledge about the design patterns applied in a software system is crucial for preparing the system's design changes, since patterns often describe how software can be extended or adapted. If usages of design patterns are not appropriately documented or known, this can evidently lead to increased maintenance effort, design flaws, and eventually to design erosion.

In this thesis, in order to reduce the risk for design flaws and to support developers in applying design patterns, I introduce my model-driven approach for semi-automatically applying design patterns as well as automatically documenting and validating the resulting pattern implementations. Developers are led through the step of applying a design pattern in a software design model. According to the developer's choices, a pattern implementation is generated in the design model. In addition, the applied pattern (the pattern roles and their implementations) is accurately and automatically captured in a separate model.

Due to their explicit capturing, pattern implementations can be highlighted while viewing the software design, e.g. in class diagrams, or be presented separately (i.e. the pattern usages are documented). Hence, pattern implementations can hardly be missed during design changes. Different views and levels of detail are offered to visualise the precisely captured model of applied patterns.

After changing the software design model, its conformance to the design patterns in use can automatically be checked by means of the detailed model of all applied patterns. Violations are automatically revealed. This way, developers can consciously replace earlier pattern implementations with other design solutions or revise their accidental design deviations.

The main contributions of this thesis are

(i) a pattern specification language that can be used to compactly describe object-oriented design patterns, covering structure, behaviour, dependencies, and implementation variants,

(ii) a semi-automatic approach for generating a pattern implementation from a pattern specification and simultaneously and automatically capturing the applied pattern (the usage of the pattern) in a detailed model,

(iii) different notations with various levels of detail for the visualisation of pattern implementations and applied patterns,

(iv) an approach for validating the compliance of pattern implementations in regard to the corresponding pattern specifications,

(v) an evaluation of the presented concepts by means of an implemented prototype and several usage scenarios.

Danksagung

Während der Arbeit an meiner Dissertation wurde ich von vielen Menschen begleitet und unterstützt. Diesen Menschen möchte ich ganz herzlich danken.

Wilhelm Schäfer danke ich ganz besonders, denn er bot mir die Möglichkeit zur Promotion, eine sichere Stelle als wissenschaftlicher Mitarbeiter an seinem Softwaretechnik-Lehrstuhl sowie unzählige Gelegenheiten, an spannenden Themen praxisnah zu forschen, zu lernen und das Wissen weiterzugeben.

Anthony Anjorin danke ich ganz herzlich für seine Betreuung in der letzten Phase meines Promotionsvorhabens und insb. für sein hilfreiches, aufbauendes Feedback und zahlreiche, wertvolle Verbesserungsideen zu meinem Dissertationsentwurf.

Ich danke meiner Promotionskommission bestehend aus Anthony Anjorin, Albert Zündorf, Heike Wehrheim, Matthias Meyer und Stefan Sauer für ihre Bereitschaft, mich zu prüfen und für die angenehme Atmosphäre während meiner Prüfung. Ganz besonders danke ich Anthony Anjorin und Albert Zündorf für das Anfertigen der Gutachten zu meiner umfangreichen Dissertation.

Lothar Wendehals und Matthias Meyer danke ich dafür, dass sie mich während meines Informatikstudiums für den Softwaretechnik-Lehrstuhl begeistert und letztendlich vom Promovieren überzeugt haben. Als studentische Hilfskraft lernte ich von ihnen wie man (auch in großen, verteilten Teams) gute Software entwickelt. Ihre gute Betreuung bei meinen Abschlussarbeiten (Studien- und Diplomarbeit) hat mein Interesse für die praxisnahen Forschungsthemen des Lehrstuhls geweckt.

Meinen Kolleginnen und Kollegen aus der Zeit als wissenschaftlicher Mitarbeiter danke ich für ihre Hilfsbereitschaft, Ideen, Tipps, gute Zusammenarbeit und ein tolles Arbeitsklima. Besonderer Dank gilt Markus von Detten für die fruchtbaren Diskussionen zu meinem Dissertationsthema und seine Unterstützung bei der Projektgruppe (PG) POSE[1]. Die Zusammenarbeit mit Joel Greenyer, Jan Rieke, Christian Heinzemann, Markus von Detten, Marie Christin Platenius und Markus Fockel zu Themen wie Graphtransformationen und Reverse Engineering war sehr produktiv und wertvoll. Danke auch für die gemeinsamen Kaffeepausen, bei denen wir – insb. Jens Frieben, Oliver Sudmann, Christopher Brink und ich – uns zu verschiedensten Themen gerne ausgetauscht und voneinander gelernt haben. Ebenso danke ich allen Kolleginnen und Kollegen, mit denen ich Papiere geschrieben, Software entwickelt oder Lehrveranstaltungen vorbereitet habe wie das für Betreuer und Studenten unvergessliche Softwaretechnikpraktikum. Bei organisatorischen und technischen Angelegenheiten standen Jutta Haupt und Sammy a.k.a. Jürgen Maniera mir immer zu Seite, vielen Dank dafür. Namentlich möchte ich folgenden, bisher nicht genannten, damaligen Kolleginnen und Kollegen danken: Anas Anis, Björn Axenath, Matthias Becker, Steffen Becker, Christian Bimmermann,

[1] Pattern-Oriented Software Engineering

Christian Brenner, Stefan Dziwok, Tobias Eckardt, Matthias Gehrke, Christopher Gerking, Holger Giese, Stefan Henkler, Martin Hirsch, Jörg Holtmann, Nicola Danielzik, Ekkart Kindler, Sebastian Lehrig, Renate Löffler, Jan Meyer, Uwe Pohlmann, Claudia Priesterjahn, Vladimir Rubin, Daniela Schilling, David Schmelter, Florian Stallmann, Julian Suck, Matthias Tichy, Robert Wagner.

Den Mitgliedern der PG POSE danke ich für ihren Einsatz, ihre Ideen und erste Ergebnisse, welche von Andre Backofen in seiner Master-Arbeit vertieft und von mir schließlich weiterentwickelt wurden. Aljoscha Hark danke ich für seine Unterstützung als studentische Hilfskraft bei der Entwicklung eines Prototypen.

Ich danke den Fujaba-, Story-Diagramm-, TGG- und Graphtransformations-Communities für spannende Forschungsthemen und Projekte, tolle Zusammenarbeit und ansteckende Motivation sowie die Ideen und Software, auf denen mein Prototyp zum Teil aufbaut. Vor allem danke ich Andy Schürr, Albert Zündorf, Wilhelm Schäfer und ihren Lehrstühlen an der TU Darmstadt, Uni Kassel und Uni Paderborn. Sie haben diese Communities überhaupt erst ermöglicht.

Meinem Arbeitgeber, Deutsche Post Adress GmbH & Co. KG, danke ich für die Möglichkeit, meine Arbeitszeit temporär zu reduzieren.

Ein ganz besonderer Dank gilt meiner Familie. Meiner Frau, Kathrin Travkin (geb. Kürten), danke ich für ihre unglaubliche Geduld während der langen Zeit bis zur Fertigstellung meiner Dissertation. Sie hat mich stets unterstützt, viele Abstriche gemacht, mir den Rücken freigehalten und mit mir bis zum Schluss durchgehalten. Meinem jüngeren Bruder, Oleg Travkin, danke ich für seine vielseitige Unterstützung, sein offenes Ohr und die manchmal nötige Ablenkung oder Motivation. Vor allem danke ich meinen Eltern, die den Weg zu meiner Promotion ebneten, indem sie mich stets motiviert, gefördert, an mich geglaubt und enorm unterstützt haben. Zu Schulzeiten haben sie mir viel zugetraut und mich angespornt, von der Haupt- auf eine Realschule und später auf ein Gymnasium zu wechseln. Das unersättliche Interesse für Computer haben sie mit dem Kauf damals sündhaft teurer Hardware gefördert und waren stolz, als ein Informatikstudium und schließlich die Promotion nachkamen... und das bei beiden Söhnen! Danke! Ich bin genauso stolz auf euch wie ihr auf Oleg und mich seid!

Inhalt

1 Einleitung 1
- 1.1 Motivation . . . 2
  - 1.1.1 Beispiel . . . 4
  - 1.1.2 Bisherige Ansätze . . . 8
- 1.2 Ziele und Forschungsfragen . . . 9
- 1.3 Lösungsansatz . . . 11
- 1.4 Wissenschaftliche Beiträge und Ergebnisse der Arbeit . . . 15
- 1.5 Struktur der Arbeit . . . 16

2 Grundlagen 19
- 2.1 Softwareentwurfsmuster . . . 19
  - 2.1.1 Eigenschaften und Abgrenzung . . . 20
  - 2.1.2 Arten und Sammlungen von Softwareentwurfsmustern . . . 23
  - 2.1.3 Verwendete Begriffe . . . 25
- 2.2 Modellgetriebene Softwareentwicklung – MDSD . . . 27
- 2.3 Modelltransformationen . . . 29
  - 2.3.1 Modelle . . . 29
  - 2.3.2 Modelltransformationsansätze . . . 29
  - 2.3.3 Graphtransformationen . . . 31
- 2.4 Modellgetriebene Entwicklung mit Klassen- und Story-Diagrammen . . . 33
  - 2.4.1 Story Patterns . . . 34
  - 2.4.2 Story-Diagramme . . . 35
  - 2.4.3 Story-Driven Modeling . . . 38

3 Spezifikation von Entwurfsmustern 39
- 3.1 Anforderungen . . . 40
  - 3.1.1 Zweck der Musterspezifikationen . . . 41
  - 3.1.2 Anforderungen an die Musterspezifikationssprache . . . 43
- 3.2 Überblick . . . 45
- 3.3 Konzeption der Musterspezifikationssprache am Beispiel . . . 47
  - 3.3.1 Was lässt sich formal beschreiben? . . . 47
  - 3.3.2 Wahl der Notation . . . 50
  - 3.3.3 Repräsentation von Implementierungsvarianten . . . 53
  - 3.3.4 Beschreibung, Spezifikation und Implementierungsvarianten von Entwurfsmustern . . . 56
- 3.4 Design Abstraction Language – Repräsentation eines objektorientierten Softwareentwurfs . . . 58
  - 3.4.1 Klassenstrukturen . . . 58
  - 3.4.2 Interaktionen . . . 61

3.4.3 Subsysteme . . . 65
3.5 Pattern Specification Language – Repräsentation von Variabilität und Regeln . . . 66
3.5.1 Muster, Musterspezifikationen und Musterspezifikationskataloge 66
3.5.2 Set Fragments – Wiederkehrende Entwurfsstrukturen . . . 67
3.5.3 Bedingungen zu Abhängigkeiten und Kopplung . . . 72
3.5.4 Entwurfsaufgaben . . . 74
3.6 Erweiterbarkeit der Musterspezifikationssprache . . . 75
3.7 Einschränkungen . . . 76
3.8 Zusammenfassung und Ausblick . . . 77

4 Modellierung und Visualisierung von Musteranwendungsstellen 79
4.1 Überblick . . . 81
4.2 Pattern Application Language – Erfassung von Korrespondenzen . . . 84
4.3 Rollenzuordnungen in der Musteranwendungssicht . . . 87
4.4 Musterimplementierungen in Klassen- und Story-Diagrammen . . . 89
4.4.1 Entwurfssicht . . . 90
4.4.2 Musterrollen-im-Entwurf-Sicht . . . 93
4.4.3 Musterrollen-im-Entwurf-Ausschnittssicht . . . 93
4.5 Einschränkungen . . . 96
4.6 Zusammenfassung und Ausblick . . . 98

5 Synthese von Musterimplementierungen 99
5.1 Überblick . . . 101
5.2 Beispiel einer Musteranwendung . . . 105
5.3 Rollenzuordnung: Musteranwendung aus Benutzersicht . . . 108
5.3.1 Ablauf bei initialer Musteranwendung . . . 109
5.3.2 Nachträgliches Ändern einer Musteranwendung . . . 111
5.4 Auffaltung: Generieren des Anwendungsmodells . . . 112
5.4.1 Anwendungsmodell . . . 112
5.4.2 Ablauf der Rollenzuordnung und Auffaltung . . . 115
5.4.3 Erstellen einer Anwendungsstelle (eines Korrespondenzmodells) 117
5.4.4 Instanziieren eines Anwendungsmodells . . . 118
5.4.5 Ergänzen einer Set-Fragment-Instanz . . . 118
5.5 Übersetzung des Anwendungsmodells in den Entwurf . . . 121
5.5.1 Herausforderungen . . . 121
5.5.2 Wahl der Transformationssprache . . . 125
5.5.3 Aufbau der Transformation . . . 127
5.5.4 Transformationsregeln . . . 131
5.6 Verbleibende, nicht automatisierbare Schritte . . . 136
5.6.1 Nicht generierbare Elemente ergänzen . . . 137
5.6.2 Aufgaben abarbeiten . . . 138
5.7 Mögliche Anpassungen einer generierten Musterimplementierung . . . 139
5.7.1 Nachträglich zusätzliche Set-Fragment-Instanzen ergänzen . . . 139
5.7.2 Direkte durch indirekte Beziehungen ersetzen . . . 139
5.7.3 Methoden und Referenzen in Vererbungshierarchien verschieben 140
5.8 Einschränkungen . . . 141

5.9 Zusammenfassung und Ausblick 143

6 Validierung der Konsistenz von Musterimplementierungen 145
6.1 Überblick 146
6.2 Abweichungen von Musterspezifikationen aufdecken 147
6.2.1 Vollständigkeit der Rollenzuordnung 148
6.2.2 Einhalten spezifizierter Eigenschaften einer Rolle 149
6.2.3 Konsistenz zu spezifizierten Rollenbeziehungen 150
6.2.4 Konsistenz zum spezifizierten Verhalten 152
6.2.5 Einhalten von Kopplungsrestriktionen 153
6.3 Einschränkungen 154
6.4 Zusammenfassung und Ausblick 154

7 Prototypische Implementierung des Ansatzes 155
7.1 Architektur 156
7.2 Benutzungsschnittstelle am Beispiel einer Observer-Musteranwendung 158
7.3 Herausforderungen, Ergebnisse und Einschränkungen des Prototypen 166

8 Evaluation 169
8.1 Spezifikation von Entwurfsmustern 169
8.1.1 Spezifikation der GoF-Entwurfsmuster 170
8.1.2 Spezifikation anderer Muster 178
8.1.3 Fazit 181
8.2 Werkzeug-gestützte Musteranwendung 186
8.2.1 Eignung des Verfahrens für unterschiedliche Einsatzszenarien . 187
8.2.2 Beispiele für Musteranwendungen bei der Entwicklung eines grafischen Petrinetz-Editors 193
8.2.3 Modellgetriebene Entwicklung des JUnit-Frameworks durch inkrementelle Anwendung diverser Muster 200
8.2.4 Fazit 203
8.3 Modellierung und Visualisierung von Anwendungsstellen 205
8.4 Validierung von Anwendungsstellen 205
8.5 Zusammenfassung 206

9 Verwandte Arbeiten 207
9.1 Spezifikation von Entwurfsmustern 208
9.1.1 Mathematisch formale Musterspezifikationen für Beweise . . . 208
9.1.2 Musterspezifikation zwecks Werkzeug-Unterstützung im Forward Engineering 210
9.1.3 Musterspezifikation zwecks Werkzeug-Unterstützung im Reverse Engineering 217
9.2 Dokumentation von Musteranwendungsstellen 218
9.2.1 Modellierung von Anwendungsstellen 219
9.2.2 Visualisierung von Anwendungsstellen 221
9.3 Werkzeug-gestützte Musteranwendung 227
9.3.1 Musteranwendung im Quellcode 227

9.3.2 Musteranwendung in Entwurfsmodellen 230
9.4 Architektur- und Design-Konformitätsprüfung 235
9.4.1 Konsistenz von Musterimplementierungen im Quellcode . . . 235
9.4.2 Konsistenz von Musterimplementierungen in Entwurfsmodellen 236
9.4.3 Weitere Architektur- und Entwurfskonformitätsprüfungen . . . 238
9.5 Zusammenfassung und Fazit 240

10 Zusammenfassung und Ausblick 243
10.1 Zusammenfassung 243
10.2 Offene Fragen und Ausblick 246

Anhang

A Musterspezifikationsprache 249
A.1 Abstrakte Syntax 249
A.2 Konkrete Syntax und Sprachumfang 252
A.2.1 DAL: Klassenstrukturen 252
A.2.2 DAL: Interaktionen 258
A.2.3 DAL: Subsysteme 262
A.2.4 PSL: Set Fragments 262
A.2.5 PSL: Abhängigkeits- & Kopplungsregeln 263
A.2.6 PSL: Entwurfsaufgaben 264
A.3 Semantik 265
A.3.1 DAL: Abbildung der Struktur auf Ecore 266
A.3.2 DAL: Abbildung des Verhaltens auf Story-Diagramme 269
A.3.3 PSL: Formale Definition der Semantik von Set Fragments . . 280

B Musterspezifikationen 291
B.1 Spezifikation der Gang-of-Four-Muster 291
B.1.1 Erzeugungsmuster (Creational Patterns) 291
B.1.2 Strukturmuster (Structural Patterns) 292
B.1.3 Verhaltensmuster (Behavioral Patterns) 295
B.2 Spezifikation weiterer Entwurfsmuster, Architekturmuster und Idiome 300

C Von der Musterspezifikation zur Musterimplementierung 303
C.1 Korrespondenzmodell und Anwendungsstelle erstellen 303
C.2 Auffaltung 307
C.2.1 Erstellen eines initialen Anwendungsmodells (Instanziierung) . 307
C.2.2 Ergänzen einer Set-Fragment-Instanz (add-Operation) 310
C.2.3 Konformität zur formal definierten Semantik von Set Fragments 318
C.3 Übersetzung 321
C.3.1 Transformationsregeln bzw. Elementübersetzer 321
C.3.2 Übersetzung von Klassenstrukturen in ein Ecore-Modell . . . 323
C.3.3 Übersetzung von Aktionen in Story-Diagramme 328

D Schrittweise Musteranwendungen im JUnit-Framework 341
D.1 Anwendung des Command-Musters vorbereiten 341
D.2 Anwenden des Musters Template Method 342
D.3 Anwenden der Musters Collecting Parameter 344
D.4 Ergänzen einer Fehlerbehandlung 348
D.5 Anwenden des Adapter-Musters 351
D.6 Anwenden des Musters Pluggable Selector 352
D.7 Anwenden des Composite-Musters 354
D.8 Vervollständigen der Musteranwendungen durch Hinzunahme einer konkreten Testfallimplementierung 358
D.8.1 Composite 358
D.8.2 Pluggable Selector 359
D.8.3 Command 360
D.9 Fazit 362

E Implementierungsdetails 363

Glossar 365

Eigene Veröffentlichungen 369

Literatur 371

Abbildungen

1.1 Informelle Beschreibung des Observer-Musters 4
1.2 Dokumentierte Struktur des Observer-Musters 5
1.3 Gedachte Struktur des Observer-Musters 5
1.4 Eine Verwendung des Observer-Musters 6
1.5 Eine falsche Entwurfserweiterung, abweichend vom Observer-Muster . 7
1.6 Der komplette Ansatz im Überblick 11
1.7 Spezifikation des Observer-Entwurfsmusters 12
1.8 Zuordnung der Musterrollen zu Elementen im Entwurf 13
1.9 Anwendung und Validierung von Entwurfsmustern 14
1.10 Überblick zum vorgestellten Ansatz und den Kapiteln dieser Arbeit . 17

2.1 Ontologie der Softwareentwurfsmuster 26
2.2 Kernidee hinter modellgetriebener Softwareentwicklung (MDSD) . . . 27
2.3 Einordnung modellgetriebener Softwareentwicklung (MDSD) 28
2.4 Vom PIM über PSMs zum Code 28
2.5 Aufbau von Modelltransformationen nach Czarnecki & Helsen 30
2.6 Aufbau einer Graphtransformation 31
2.7 Beispiel für ein Modell als Objektdiagramm 32
2.8 Meta-Modell für das Modell aus Abb. 2.7 32
2.9 Attributierter Graph G 32
2.10 Typgraph T für Graphen G aus Abb. 2.9 32
2.11 Typisierter, attributierter Graph 32
2.12 Eine Graphtransformationsregel 33
2.13 Ein Startgraph für eine Graphtransformation 33
2.14 Ein Ergebnis nach einer Graphtransformation 33
2.15 Ein Story Pattern 34
2.16 Ein Story-Diagramm 36
2.17 Ein Story-Diagramm mit Methodenaufrufen und einer Schleife 37

3.1 Repräsentation eines Musters und diverser Musterimplementierungen 41
3.2 Musterformalisierung, Musteranwendung und Konsistenzprüfung . . . 42
3.3 Die Struktur des Musters Abstract Factory in UML-Notation 43
3.4 Aufbau der Musterspezifikationssprache 45
3.5 Repräsentation von Implementierungsvarianten eines Musters 46
3.6 Formalisierungsversuch für eine Observer-Entwurfslösung 49
3.7 Spezifikation des Observer-Musters, Variante 1 52
3.8 Spezifikation des Observer-Musters, Variante 2 53
3.9 Abstrakte Repräsentation einer Implementierungsvariante des Observer-Musters in der Design Abstraction Language (DAL) 54

3.10 Kanonische Implementierungsvariante des Observer-Musters modelliert in Klassen- und Story-Diagrammen . . . 55
3.11 Nicht-triviale Implementierungsvariante des Observer-Musters modelliert in Klassen- und Story-Diagrammen . . . 55
3.12 Abdeckung der Implementierungsvarianten zu einem Muster . . . 57
3.13 Konkreter Typ in DAL und Ecore . . . 59
3.14 Beliebiger Typ in DAL und Ecore . . . 59
3.15 Operation und Komposition in DAL und Ecore . . . 60
3.16 Unidirektionale Referenz in DAL und Ecore . . . 60
3.17 Bidirektionale Referenz in DAL und Ecore . . . 60
3.18 Spezialisierung bei Typen in DAL und Ecore . . . 61
3.19 Spezialisierung bei Operationen in DAL und Ecore . . . 61
3.20 Aktion in DAL und Ecore . . . 62
3.21 Kontrollfluss in DAL und Ecore . . . 62
3.22 Aktion call in DAL und Ecore (a) . . . 63
3.23 Aktion call in DAL und Ecore (b) . . . 63
3.24 Aktion create in DAL und Ecore . . . 64
3.25 Subsystem in DAL und Ecore . . . 65
3.26 Ausschnitt der Spezifikation des Observer-Musters . . . 68
3.27 Ausschnitt der Spezifikation des Strategy-Musters . . . 68
3.28 Zur Spezifikation aus Abb. 3.26 passende Entwurfsstruktur (a) . . . 68
3.29 Zur Spezifikation aus Abb. 3.26 passende Entwurfsstruktur (b) . . . 68
3.30 Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (a) . . . 68
3.31 Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (b) . . . 68
3.32 Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (c) . . . 68
3.33 Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (d) . . . 68
3.34 Abbildungen (bzw. Operationen) bei einer Auffaltung . . . 71
3.35 Instanziierung – Abbildung *instance* . . . 71
3.36 Hinzufügen einer Set-Fragment-Instanz – Abbildung *add* . . . 71
3.37 Projektion – Abbildung π . . . 71
3.38 Drei Beispiele für Kopplungsregeln . . . 72
3.39 Hierarchie der Kopplungsarten . . . 72
3.40 Spezifikation des MVC-Entwurfsparadigmas . . . 74
3.41 Spezifikation des Entwurfsmusters Facade . . . 74
3.42 Notation von Entwurfsaufgaben . . . 75

4.1 Zusammenhang einer Musterspezifikation und ihrer Implementierung . 81
4.2 Zusammenhang einer Musterspezifikation und ihrer Implementierung am Beispiel des Strategy-Musters . . . 82
4.3 Verknüpfung einer Musterimplementierung mit der Musterspezifikation 83
4.4 Ausschnitt eines Dekorationsmodells in abstrakter Syntax . . . 85
4.5 Abstrakte Syntax der Pattern Application Language (PAL) . . . 86
4.6 Musteranwendungssicht: Rollenzuordnungen und Musterspezifikation 88
4.7 Selektion einer Musterrolle und zugehörigem Entwurfselement . . . 89
4.8 Darstellung von offenen und erledigten Entwurfsaufgaben . . . 89
4.9 Klassendiagramm-Entwurfssicht: an Musterimplementierungen beteiligte Entwurfsteile sind markiert . . . 91

4.10 Klassendiagramm-Entwurfssicht mit Musterimplementierungsdetails eines selektierten Elements . . . 91
4.11 Klassendiagramm-Entwurfssicht mit Musterimplementierungsdetails einer selektierten Musterimplementierung . . . 91
4.12 Story-Diagramm-Entwurfssicht: an Musterimplementierungen beteiligte Entwurfsteile sind markiert . . . 92
4.13 Story-Diagramm-Entwurfssicht mit Musterimplementierungsdetails eines selektierten Elements . . . 92
4.14 Story-Diagramm-Entwurfssicht mit Musterimplementierungsdetails einer selektierten Musterimplementierung . . . 92
4.15 Musterrollen-im-Entwurf-Sicht (Klassendiagramm): Entwurf mit Markierungen und UML-Kollaborationen . . . 94
4.16 Musterrollen-im-Entwurf-Sicht (Klassendiagramm): Entwurf mit Markierungen, UML-Kollaborationen und selektierter Musterimplementierung 94
4.17 Musterrollen-im-Entwurf-Sicht (Story-Diagramm): Entwurf mit Markierungen und UML-Kollaborationen . . . 94
4.18 Entwurfssicht: Visualisierung von Anwendungsstellen, niedriger Detailgrad . . . 95
4.19 Musterrollen-im-Entwurf-Sicht: Visualisierung von Anwendungsstellen, hoher Detailgrad . . . 95
4.20 Entwurfssicht mit selektiertem Entwurfselement . . . 95
4.21 Musterrollen-im-Entwurf-Sicht mit selektierter Musterimplementierung 95
4.22 Isolierte Darstellung einer Musterimplementierung in einem Klassendiagramm . . . 97
4.23 Musterimplementierung mit Hervorhebung der Elemente in der Set-Fragment-Instanz strategies[2] . . . 97
4.24 Musterimplementierung mit Hervorhebung der Elemente in der Set-Fragment-Instanz operations[1] . . . 97

5.1 Synthese einer Musterimplementierung . . . 101
5.2 Sprachebenen und Granularität bei der Auffaltung und Übersetzung . 104
5.3 Erweiterbarkeit des Ansatzes auf andere Modellierungssprachen . . . 104
5.4 Schritte bei einer Musteranwendung am Beispiel Observer – Teil 1 . . 106
5.5 Schritte bei einer Musteranwendung am Beispiel Observer – Teil 2 . . 107
5.6 Zuordnung einer Klasse zu ihrer Musterrolle aus Benutzersicht . . . 109
5.7 Eine mögliche Rollenbelegung aus Benutzersicht . . . 110
5.8 Zusätzliche Set-Fragment-Instanz aus Benutzersicht . . . 111
5.9 Ein aus Benutzereingaben hergeleitetes Anwendungsmodell für das Muster Strategy . . . 114
5.10 Vorbereiten einer Musteranwendung und Auffaltung . . . 115
5.11 Instanziieren und Auffalten eines Anwendungsmodells . . . 119
5.12 Ein- und Ausgaben bei der Übersetzungstransformation . . . 122
5.13 Korrespondenzen und Startpunkt bei Modelltransformationen . . . 123
5.14 Kontextabhängige Übersetzung von Operationen . . . 124
5.15 Unterschiedliche Abstraktions- und Detaillevel bei der Übersetzung . 125
5.16 Typischer Aufbau einer exogenen Update-Transformation . . . 128
5.17 Aufbau meiner Übersetzungstransformation . . . 128

5.18 Verhalten des Transformationsmechanismus bei meiner Übersetzung . 129
5.19 Verhalten eines Elementübersetzers 130
5.20 Type2EClass 132
5.21 TypeInheritance2EClassInheritance 132
5.22 Kontrollfluss des Übersetzers Type2EClass 132
5.23 Kontrollfluss des Übersetzers TypeInheritance2EClassInheritance . . . 133
5.24 Operation2EOperation ohne Spezialisierung 134
5.25 Operation2EOperation mit Spezialisierung 134
5.26 OperationBehavior2StoryDiagram 136
5.27 Manuelles Ergänzen einer nicht eindeutig spezifizierbaren Referenz . . 137
5.28 Manuelle Aufgaben nach Observer-Musteranwendung 138
5.29 Ersetzen einer direkten durch eine indirekte Vererbungsbeziehung . . 140
5.30 Ersetzen einer direkten durch eine indirekte Kompositionsbeziehung . 141
5.31 Rollenzuordnung bei einer indirekten Kompositionsbeziehung 141

6.1 Notwendigkeit der Überprüfung von Musterimplementierungen 147
6.2 Validierung im Verhalten eines Elementübersetzers 149

7.1 Architekturüberblick 156
7.2 Komponentenarchitektur des Prototyps 157
7.3 Spezifikation des Observer-Musters im Pattern Specification Editor . 159
7.4 Auswahl des Observer-Musters aus einem Musterkatalog 160
7.5 Initiale Darstellung einer Observer-Muster-Anwendungsstelle 160
7.6 Zuordnung einer Rolle zu einer Klasse im Entwurf 161
7.7 Ergänzen einer zusätzlichen Set-Fragment-Instanz 162
7.8 Abgeschlossene Rollenzuordnung, Auslösen der Musteranwendung . . 163
7.9 Zustand nach automatischer Musteranwendung 163
7.10 Erzeugtes Verhaltensmodell (Story-Diagramm) für eine Operation . . 164
7.11 Rollenzuordnung nach manueller Ergänzung des Entwurfs 165
7.12 Klassendiagramm und Musteranwendungsansicht nach vollständiger Musteranwendung 165

8.1 Grobe Idee der Entwurfslösung hinter dem Observer-Muster 171
8.2 Spezifikation des GoF-Entwurfsmusters „Observer“ – Variante mit beliebig vielen Subjekten je Beobachter 171
8.3 Grobe Idee der Entwurfslösung hinter dem Strategy-Muster 174
8.4 Spezifikation des GoF-Entwurfsmusters „Strategy“ 174
8.5 Grobe Idee der Entwurfslösung hinter dem Abstract-Factory-Muster . 176
8.6 Spezifikation des GoF-Entwurfsmusters „Abstract Factory“ 176
8.7 Grobe Idee der Entwurfslösung hinter dem Facade-Muster 177
8.8 Spezifikation des GoF-Entwurfsmusters „Facade“ 177
8.9 Spezifikation des Architekturmusters „Model-View-Controller“ (MVC) 179
8.10 Spezifikation des „Layers“-Architekturstils nach Fowler 179
8.11 Spezifikation des Enterprise Application Architecture Patterns „Data Transfer Object“ (DTO) 179
8.12 Spezifikation des SmallTalk Best Practice Patterns „Collecting Parameter“ 181
8.13 Vollständige Generierung einer Observer-Musterimplementierung . . . 188

8.14 Vervollständigung einer Observer-Musterimplementierung – Fall 1 . . 189
8.15 Vervollständigung einer Observer-Musterimplementierung – Fall 2 . . 190
8.16 Vervollständigung einer Observer-Musterimplementierung – Fall 3 . . 190
8.17 Vervollständigung einer Observer-Musterimplementierung – Fall 4 . . 190
8.18 Nachträgliche Rollenzuordnung bei einer bereits existierenden Observer-Musterimplementierung 192
8.19 Problematischer Fall bei der Anwendung des Strategy-Musters 195
8.20 Anwendung des Strategy-Musters nach Anpassung der Spezifikation . 195
8.21 Manuelle Anpassung des generierten Story-Diagramms 195
8.22 Klassendiagramm vor Anwendung des Musters Abstract Factory . . . 196
8.23 Rollenzuordnungen vor Anwendung des Musters Abstract Factory . . 196
8.24 Modelle nach Anwendung des Musters Abstract Factory 197
8.25 Ansicht vor Anwendung des Musters Model-View-Controller 199
8.26 Ansicht nach Anwendung des Musters Model-View-Controller 199
8.27 Entwurf des JUnit-Frameworks mit eingesetzten Mustern 201
8.28 Nachempfundener Entwurf des JUnit-Frameworks mit eingesetzten Mustern 201

9.1 Übersicht der Themengebiete verwandter Arbeiten 207
9.2 Spezifikation des Observer-Musters nach Florijn et al. 211
9.3 LePUS3-Spezifikation des Musters Abstract Factory nach Eden et al. 212
9.4 Class-Z-Spezifikation des Musters Abstract Factory nach Eden et al. . 212
9.5 LePUS3-Spezifikation des Observer-Musters nach Eden et al. 213
9.6 Spezifikation des Musters Abstract Factory nach Maplesden et al. . . 213
9.7 Spezifikation des Observer-Musters nach Kim et al. 215
9.8 Visualisierung von Anwendungsstellen in der UML 222
9.9 Visualisierung von Anwendungsstellen nach Gamma 222
9.10 Visualisierung von Anwendungsstellen nach Schauer & Keller 222
9.11 Visualisierung von Anwendungsstellen nach Dong et al. 222
9.12 Visualisierung von Anwendungsstellen nach Eden et al. 224
9.13 Visualisierung von Anwendungsstellen nach Smith 225
9.14 Visualisierung der Komposition eines Musters aus Elementarmustern nach Smith 226
9.15 Visualisierung einer Musterimplementierung mit Klassendiagrammen nach Smith 226

A.1 Abstrakte Syntax der Musterspezifikationssprache (PSL) ohne DAL . 250
A.2 Abstrakte Syntax der Design Abstraction Language (DAL) 251
A.3 Klassenstruktur des Observer-Musters in der DAL 255
A.4 a) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung 255
A.5 b) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung 255
A.6 c) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung 255
A.7 Korrigierte Klassenstruktur des Observer-Musters in der DAL 256

A.8 a) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung . . . 256
A.9 b) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung . . . 256
A.10 c) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung . . . 256
A.11 Mit einer Aktion spezifizierte Umleitung im Observer-Muster . . . 261
A.12 a) Zur redirect-Aktion in Abb. A.11 passende Java-Implementierung . 261
A.13 b) Zur redirect-Aktion in Abb. A.11 passende Java-Implementierung . 261
A.14 c) Zur redirect-Aktion in Abb. A.11 passendes Story-Diagramm . . . 261
A.15 Abbilden von Typen auf Klassen . . . 267
A.16 Abbilden einer Vererbungsbeziehung . . . 267
A.17 Abbilden von Attributen . . . 267
A.18 Abbilden von Referenzen . . . 268
A.19 Abbilden von Operationen und ihren Signaturen . . . 268
A.20 Abbilden von Operationsverhalten auf ein Story-Diagramm . . . 269
A.21 Abbilden von call-Aktionen mit verschiedenen Zielobjekten . . . 271
A.22 Abbilden einer call-Aktion mit Argumenten . . . 273
A.23 Abbilden einer call-Aktion mit Aufrufergebnis . . . 274
A.24 Abbilden einer redirect-Aktion . . . 275
A.25 Abbilden einer delegate-Aktion . . . 276
A.26 Abbilden von create-Aktionen . . . 277
A.27 Abbilden einer produce-Aktion . . . 278
A.28 Abbilden von return-Aktionen mit verschiedenen Variablen . . . 279
A.29 Veranschaulichung der Instanziierung laut Definition A.10 . . . 284
A.30 Veranschaulichung der Projektion laut Definition A.11 . . . 285
A.31 Veranschaulichung der Abbildung $add(G_I, i)$ laut Definition A.12 . . . 287
A.32 Beispiele für Abbildungen $add(G_I, i)$ nach Definition A.12 . . . 288

B.1 Spezifikation des GoF-Entwurfsmusters „Abstract Factory“ . . . 291
B.2 Spezifikation des GoF-Entwurfsmusters „Builder“ . . . 291
B.3 Spezifikation des GoF-Entwurfsmusters „Factory Method“ . . . 292
B.4 Spezifikation des GoF-Entwurfsmusters „Prototype“ . . . 292
B.5 Spezifikation des GoF-Entwurfsmusters „Singleton“ . . . 292
B.6 Spezifikation des GoF-Entwurfsmusters „Object Adapter“ . . . 292
B.7 Spezifikation des GoF-Entwurfsmusters „Bridge“ . . . 293
B.8 Spezifikation des GoF-Entwurfsmusters „Composite“ mit separaten Schnittstellen für *Leaf* und *Composite* . . . 293
B.9 Spezifikation des GoF-Entwurfsmusters „Composite“ mit einheitlicher Schnittstelle für *Leaf* und *Composite* . . . 293
B.10 Spezifikation des GoF-Entwurfsmusters „Decorator“ . . . 293
B.11 Spezifikation des GoF-Entwurfsmusters „Facade“ . . . 294
B.12 Spezifikation des GoF-Entwurfsmusters „Flyweight“ . . . 294
B.13 Spezifikation des GoF-Entwurfsmusters „Proxy“ . . . 294
B.14 Spezifikation des GoF-Entwurfsmusters „Chain of Responsibility“ . . . 295
B.15 Spezifikation des GoF-Entwurfsmusters „Command“ . . . 295
B.16 Spezifikation des GoF-Entwurfsmusters „Interpreter“ . . . 295

B.17 Spezifikation des GoF-Entwurfsmusters „Iterator“ 296
B.18 Spezifikation des GoF-Entwurfsmusters „Mediator“ 296
B.19 Spezifikation des GoF-Entwurfsmusters „Memento“ 296
B.20 Spezifikation des GoF-Entwurfsmusters „Observer“ – pull-Variante mit beliebig vielen Subjekten je Beobachter 297
B.21 Spezifikation des GoF-Entwurfsmusters „Observer“ – pull-Variante mit nur einem Subjekt je Beobachter 297
B.22 Spezifikation des GoF-Entwurfsmusters „Observer“ – push-Variante mit beliebig vielen Subjekten je Beobachter 297
B.23 Spezifikation des GoF-Entwurfsmusters „State“ – Variante 1: Kontext setzt nächsten Zustand 298
B.24 Spezifikation des GoF-Entwurfsmusters „State“ – Variante 2: aktueller Zustand setzt nächsten Zustand 298
B.25 Spezifikation des GoF-Entwurfsmusters „Strategy“ 298
B.26 Spezifikation des GoF-Entwurfsmusters „Template Method“ 298
B.27 Spezifikation des GoF-Entwurfsmusters „Visitor“ 299
B.28 Spezifikation des Architekturmusters „Model-View-Controller (MVC)“ 300
B.29 Spezifikation des „Layers“-Architekturstils – Ausprägungen nach Fowler 300
B.30 Spezifikation des Enterprise Application Architecture Patterns „Layer Supertype“ 300
B.31 Spezifikation des Enterprise Application Architecture Patterns „Data Transfer Object“ 301
B.32 Spezifikation des SmallTalk Best Practice Patterns „Collecting Parameter“ 301
B.33 Spezifikation des SmallTalk Best Practice Patterns „Pluggable Selector“ 301

C.1 Erstellen einer Anwendungsstelle am Beispiel Strategy 304
C.2 Erstellen einer Anwendungsstelle im Dekorationsmodell 305
C.3 Rollzuordnungen im Korrespondenzmodell am Beispiel Strategy . . . 306
C.4 Erstellen eines initialen Anwendungsmodells am Beispiel Strategy . . 308
C.5 Initiales Erstellen des Anwendungsmodells (Instanziierung) 309
C.6 Ergänzen einer Set-Fragment-Instanz am Beispiel Strategy 311
C.7 Korrespondenz- und Anwendungsmodell vor einer Musteranwendung am Beispiel Strategy 319
C.8 Übersicht aller Elementübersetzer bzw. Transformationsregeln 322
C.9 Type2EClass 323
C.10 Kontrollfluss des Übersetzers Type2EClass 324
C.11 TypeInheritance2EClassInheritance 324
C.12 Kontrollfluss des Übersetzers TypeInheritance2EClassInheritance . . . 325
C.13 Attribute2EAttribute 326
C.14 Parameter2EParameter 326
C.15 Reference2EReference 326
C.16 Operation2EOperation ohne Spezialisierung 327
C.17 Operation2EOperation mit Spezialisierung 327
C.18 Kontrollfluss des Übersetzers Operation2EOperation 328
C.19 OperationBehavior2StoryDiagram 329
C.20 Kontrollfluss des Übersetzers OperationBehavior2StoryDiagram . . . 330

C.21 Kombinieren der Übersetzungen von Aktionen 331
C.22 Dekomposition der Übersetzung einer call-Aktion 333
C.23 CallAction2ControlFlow 333
C.24 Kontrollfluss des Übersetzers CallAction2ControlFlow 334
C.25 Zusammenführen der Teilausdrücke zu einem 335
C.26 Bilden einer Sequenz von Aktivitätenknoten 335
C.27 Bilden einer iterierten Sequenz von Aktivitätenknoten 336
C.28 CallTarget2Expression 336
C.29 Parameter2Expression 337
C.30 Result2Expression 337
C.31 Reference2Expression 337
C.32 Reference2BoundVariable 338
C.33 MultipleCallTargets2ExpressionAndLoop 339
C.34 CallArgument2ParameterBinding 339
C.35 CallResult2ResultBinding 340

D.1 Partielle Rollenzuordnung vor Anwendung des Command-Musters . . 342
D.2 Rollenzuordnung vor Anwendung des Musters Template Method . . . 343
D.3 Ergebnis nach Anwendung des Musters Template Method 343
D.4 Manuelle Vervollständigung des Verhaltens bei der Anwendung des Musters Template Method 343
D.5 Rollenzuordnung vor Anwendung des Musters Collecting Parameter . 344
D.6 Ergebnis nach Anwendung des Musters Collecting Parameter 346
D.7 Manuelle Entwurfsanpassungen nach Anwendung des Musters Collecting Parameter 347
D.8 Verhaltensmodell der Methode *TestResult::startTest(Test)* 348
D.9 Klassendiagramm nach manueller Ergänzung einiger Klassen 349
D.10 Story-Diagramm der Methode *TestCase::run(TestResult)* nach manueller Erweiterung 350
D.11 Verhalten der Methode *TestResult::addError(Test, Throwable)* 350
D.12 Rollenzuordnung vor Anwendung des Musters Pluggable Selector . . 353
D.13 Ergebnis nach Anwendung des Musters Pluggable Selector 353
D.14 Zustand nach Entfernen der unnötigerweise generierten Methode *TestCase::pluggableSelector()* 354
D.15 Angepasstes Verhalten der Methode *TestCase::runTest()* 354
D.16 Rollzuordnung vor Anwendung des Composite-Musters 356
D.17 Ergebnis nach Anwendung des Composite-Musters 357
D.18 Korrigierte Rollzuordnung in der Anwendung des Composite-Musters 359
D.19 Korrigierte Rollzuordnung in der Anwendung des Musters Pluggable Selector 359
D.20 Rollenzuordnung vor Anwendung des Command-Musters 360
D.21 Ergebnis nach Anwendung des Command-Musters 361
D.22 Angepasstes Verhalten der Methode *MoneyTest::runTest()* 361
D.23 Entwurf nach Anwendung aller Muster 362

Tabellen

A.1 Typen, Operationen und Attribute 253
A.2 Beziehungen in Klassenstrukturen 254
A.3 Interaktionen 259
A.4 Subsysteme 262
A.5 Set Fragments – Wiederholbare Entwurfsstrukturen 263
A.6 Kopplungsrestriktionen 264
A.7 Entwurfsaufgaben – nicht formal erfassbare Entwurfsinformationen . 264
A.8 Gültige Kombinationen von Kopplungsarten und Entwurfsteilen . . . 265

E.1 Repositories mit dem Quellcode des Prototypen 363
E.2 Implementierungsumfang in Lines of Code und Eclipse-Plug-ins . . . 364

1 Einleitung

Software wird während ihrer Lebensdauer aufgrund neuer Anforderungen oder entdeckter Probleme kontinuierlich angepasst oder erweitert [Par94, Leh96]. Dabei fällt ein Großteil der gesamten Entwicklungszeit eines Softwaresystems – man schätzt 85 % bis 90 % [Erl00, Som07] – auf Wartungsarbeiten, also Änderungen der Software nach ihrer initialen Entwicklung. Bei Reengineering-Aufgaben, wozu Verstehen, Ändern, Dokumentieren und Testen der Software gehört, werden aufgrund der Komplexität der Softwaresysteme und fehlender oder mangelhafter Dokumentation bis zu 50 % der Reengineering-Zeit für das Einarbeiten und Verstehen der Software aufgebracht [Fra92]. Durch Reduktion des Einarbeitungsaufwands vor Softwareänderungen lässt sich also auch der Wartungsaufwand einer Software signifikant reduzieren.

Hoher Wartungsaufwand

Aufgrund von Zeitdruck, schlechter oder gar fehlender Dokumentation oder fehlender Erfahrung der Entwickler kann der Entwurf der zu ändernden Software missverstanden oder fehlinterpretiert werden. Als Konsequenz daraus wird die Software auf unbeabsichtigte Art und Weise angepasst. Dabei können vorgesehene und durch den ursprünglichen Entwurf geförderte Softwareeigenschaften wie Erweiterbarkeit, Effizienz oder Sicherheit verloren gehen. Mit zunehmenden Änderungen weicht die Implementierung der Software immer mehr von dem ursprünglichen Entwurf und den geplanten Eigenschaften ab. Das Einarbeiten wird zunehmend erschwert und lässt den Wartungsaufwand steigen. In solchen Fällen spricht man von Design-Erosion [vGB02].

Design-Erosion

Softwareentwurfsmuster beschreiben bewährte Entwurfslösungen für wiederkehrende Entwurfsprobleme. Sie sind inklusive ihrer Intention, den Implementierungsmöglichkeiten und der daraus resultierenden Konsequenzen gut dokumentiert und aufgrund ihrer guten Eigenschaften weit verbreitet. Neben vielen anderen Qualitätsmerkmalen haben Entwurfsmuster häufig zum Ziel, die Erweiterbarkeit einer Software und damit ihre Wartbarkeit zu erhöhen.

Softwareentwurfsmuster

> „Whether intentional or not, all well-structured, software-intensive systems are full of patterns. [...] In my experience, patterns are one of the most important developments in software engineering in the past two decades.“
> Grady Booch [Smi12, Vorwort], IBM Fellow, über 20 Jahre leitender Wissenschaftler bei der Rational Software Corporation, Mitbegründer der UML

Entscheidet sich ein Entwickler oder Architekt für dein Einsatz eines bestimmten Entwurfsmusters, so entscheidet er sich nicht nur für eine bestimmte Lösung, sondern auch für die damit verbundenen Vor- und Nachteile und insbesondere für die daraus resultierenden Softwareeigenschaften. Damit diese Eigenschaften erhalten bleiben, muss die Intention und die Entscheidung für ein bestimmtes Entwurfsmuster bei späteren Softwareanpassungen berücksichtigt, d.h. erhalten oder

Entwurfsentscheidungen

bewusst überdacht werden. Zur Vermeidung von Design-Erosion müssen Entwickler vor jeder Änderung die bereits eingesetzten Entwurfsmuster kennen und die damit verbundene Intention verstehen. Ist der Entwurf einer Software inklusive der eingesetzten Entwurfsmuster gut dokumentiert, können der Aufwand beim Verstehen des Entwurfs vor Entwurfsänderungen und die Wahrscheinlichkeit für Entwurfsfehler nachweislich reduziert werden [PULPT02, Vok04].

1.1 Motivation

Entwurfsmuster vielseitig einsetzbar, aber vage beschrieben

Eine große Stärke von Entwurfsmustern ist ihre allgemeine Verwendbarkeit. Softwareentwurfsmuster können unabhängig von einem gewählten Softwareentwicklungsprozess (z.B. Wasserfall, Rational Unified Process, V-Modell XT, Model-Driven Architecture, Scrum), dem Zeitpunkt im Softwarelebenszyklus (initialer Entwurf oder Wartung) und der gewählten Modellierungs- und Programmiersprachen (z.B. MOF, UML, SysML, Java, C++, PHP) verwendet werden. Zu diesem Zweck werden Entwurfsmuster relativ vage, Prozess- und Sprachen-unabhängig beschrieben. Der Fokus wird insbesondere auf die Intention, die Lösungsidee und die Auswirkungen gelegt. Es wird nur eine Entwurfsskizze beschrieben, kein konkreter Entwurf. In vielen Fällen werden mehrere Implementierungsalternativen vorgestellt und diskutiert. Die Wahl der für einen bestimmten Anwendungsfall geeigneten Implementierung eines Entwurfsmusters wird absichtlich dem Anwender überlassen. So behält dieser alle Freiheiten bei der Konkretisierung der Lösungsidee und kann die allgemein beschriebene Lösung durch den Einsatz einer bestimmten Sprache, Technologie und Implementierungsvariante für seinen konkreten Fall anpassen (z.B. Teile des Musters abweichend von dem beschriebenen Entwurfsbeispiel durch den Einsatz von Dependency Injection oder Reflection implementieren).

Musteranwendung fehleranfällig

Unabhängig von der Situation, in welcher ein Entwurfsmuster eingesetzt werden soll, erfolgt die Anwendung eines Musters immer auf die gleiche Weise. Ist die Idee hinter einem Entwurfsmuster verstanden und wurde der Einsatz des Musters beschlossen, wird die allgemein beschriebene Lösung auf die konkrete Situation übertragen, konkretisiert, vervollständigt und angepasst. Für unkonkret beschriebene Teile der Entwurfslösung (z.B. für einen beobachtbaren Zustand im Observer-Muster [GHJV95]) wird eine konkrete Lösung für den vorliegenden Anwendungsfall bestimmt (z.B. ein Zustandsattribut oder eine Assoziation zu einer Zustandsklasse). Die beschriebene Entwurfslösung wird in eine bestimmte Modellierungs- oder Programmiersprache übersetzt. Bereits existierende oder zu ergänzende Teile der vorliegenden Software (z.B. Klassen, Methoden, Attribute oder Beziehungen) nehmen dabei bestimmte Rollen des angewandten Entwurfsmusters ein. In den meisten Fällen erfolgt eine Musteranwendung – die Herleitung einer anwendungsspezifischen Implementierung eines Musters – komplett manuell und ist dadurch fehleranfällig. Es ist schwer einzuschätzen, ob die in der Literatur beschriebene Lösung zu einem Muster korrekt und vollständig übertragen bzw. nachimplementiert wurde und damit die erwünschten Softwareeigenschaften erzielt wurden. Einzelne Implementierungsschritte können leicht vergessen werden.

Soll eine zum Teil oder vollständig entworfene oder implementierte Software

erweitert oder angepasst werden, stellt sich die Frage, was mit den bis dahin entstandenen Musterimplementierungen[1] geschieht. Viele Entwurfsmuster sehen Anpassungsstellen vor, z.B. Erweiterungsstellen für das Ergänzen zusätzlicher Strategien beim Strategy-Muster [GHJV95]. Ist eine durch ein Entwurfsmuster vorgesehene Anpassung gewünscht, kann sie auf einfache Weise vorgenommen werden, wenn die Musterimplementierung bekannt ist. Das Entwurfsmuster verrät wie die Anpassung vorzunehmen ist, die Musterimplementierung verrät wo. Leider sind Musterimplementierungen selten und nur spärlich dokumentiert. Dadurch sind sie leicht zu übersehen. Nutzt man nicht die vorgesehenen Anpassungsstellen, riskiert man unnötigen Implementierungsaufwand, erhöhte Softwarekomplexität und vor allem eine Einschränkung von mit der Musteranwendung verfolgten Softwareeigenschaften (z.B. zur Laufzeit austauschbare Strategien des Strategy-Musters).

Vorgesehene Anpassungsstellen nicht sichtbar

Gibt es keine geeignete, durch eine vorherige Musteranwendung vorgesehene Anpassungsstelle für eine durchzuführende Änderung, wird die Software auf andere Weise angepasst (z.B. durch Anwendung eines anderen Musters). Die Änderung kann eine Anpassung von Klassen und Beziehungen erfordern, welche an der Implementierung zuvor angewandter Muster beteiligt sind. In diesem Fall muss die Änderung die Intention und die Entwurfslösung der vorher angewandten Muster erhalten. Zum Beispiel darf die durch Anwendung eines Musters absichtlich erreichte Entkopplung bestimmter Klassen nicht durch Ergänzen neuer Beziehungen versehentlich aufgehoben werden. Auch in diesem Fall ist die Kenntnis über bereits angewandte Muster von hoher Bedeutung. Gelingt die Erhaltung der Intention angewandter Muster nicht, gehen erwünschte Softwareeigenschaften verloren. Es entsteht Design-Erosion.

Versehentliche Entwurfsabweichungen

Anforderungen können sich ändern. Als Folge daraus kann ein zunächst sinnvoll eingesetztes Muster zu einem späteren Zeitpunkt die Umsetzung neuer Anforderungen verkomplizieren (z.B. weil nun Flexibilität an einer Stelle benötigt wird, wo sie bisher unnötig war und die Stelle sich durch das eingesetzte Muster nur aufwändig anpassen lässt). In diesem Fall muss die gefällte Entscheidung für den Einsatz des Musters neu evaluiert und gegebenenfalls durch eine an die geänderten Anforderungen angepasste Entwurfsentscheidung ersetzt werden. Die vorliegende Musterimplementierung muss dann bewusst durch eine passendere Lösung ersetzt werden. Ist die Musterimplementierung nicht bekannt oder unzureichend dokumentiert, können Überbleibsel der Musterimplementierung die Softwarestruktur und zukünftige Änderungen verkomplizieren und mit zunehmenden Änderungen ebenfalls zu Design-Erosion führen.

Überbleibsel verworfener Entwurfslösungen

Die Kenntnis über Stellen, an denen Entwurfsmuster angewandt wurden, ist also für spätere Softwareanpassungen besonders wichtig, denn sie gibt Aufschluss über die (vermutlich[2]) verfolgte Intention der ursprünglichen Entwickler. Wo Entwurfsmuster bereits angewandt wurden, welche Klassen, Methoden und Beziehungen an Musterimplementierungen beteiligt sind und welche Rollen sie einnehmen, ist in der Praxis nur selten vollständig dokumentiert. Das führt zu hohen Einarbeitungsaufwänden und erhöht die Wahrscheinlichkeit für Entwurfsfehler.

Gewählte Entwurfslösung nicht erkennbar

Zusammenfassend stellt der Einsatz von Entwurfsmustern Entwickler vor fol-

[1] In meiner Arbeit unterscheide ich u.a. zwischen der Anwendung, der Implementierung und der Anwendungsstelle eines Musters (siehe Glossar S. 365 ff.).

[2] Annahme: Ein Entwurfsmuster wird stets entsprechend seinem Verwendungszweck angewandt.

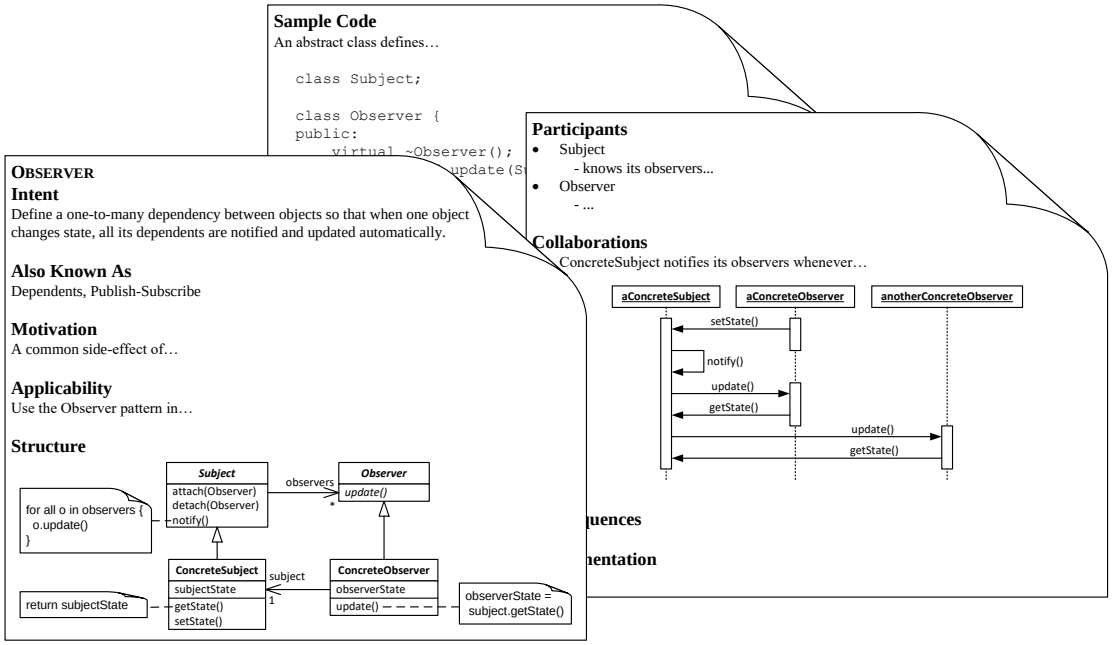


Abbildung 1.1: Informelle Beschreibung des Observer-Musters (11 Seiten) [GHJV95]. Die OMT-Notation wurde hier durch die UML-Notation ersetzt.

Herausforderungen

gende Herausforderungen:

(a) Fehleranfällige Musteranwendung durch ungenaue Musterbeschreibung und manuelle Anwendungsschritte

(b) Fehlende Sichtbarkeit von angewandten Entwurfsmustern

(c) Entwurfsfehler und Design-Erosion als Folge aus den Punkten (a) und (b)

1.1.1 Beispiel

Die Intention des Observer-Musters [GHJV95] ist es, mehreren Beobachtern zu ermöglichen, auf Änderungen eines Zustands zu reagieren. Dabei sind die Beobachter Objekte, die auf die Änderung eines Zustands eines anderen Objekts reagieren sollen. Das Muster schlägt eine Lösung vor, bei der Beobachter-Objekte zur Laufzeit ergänzt und entfernt werden können und diverse Typen von Beobachtern implementiert werden können, ohne die Implementierung der beobachteten Objekte zu ändern.

Die mit dem Muster vorgeschlagene Lösung wird von der „Gang of Four“ durch Skizzieren des Entwurfs in einem Klassen- und einem Sequenzdiagramm veranschaulicht (Abb. 1.1), durch konkrete Beispiele in Form von Klassendiagrammen und Code verdeutlicht und durch umfangreiche Erläuterungen auf insgesamt 11 Seiten beschrieben. Die dokumentierte Entwurfsstruktur ist in der Abb. 1.2 als UML-Klassendiagramm dargestellt.

Entwurfslösung durch Entwurfsbeispiel dargestellt

An dieser Abbildung wird deutlich, dass hier nur ein Beispiel für eine Implementierung des Entwurfsmusters präsentiert wird. Die Klassen- und Methodennamen können im konkreten Softwaresystem abweichen. Die Anzahl der ConcreteSubject- und ConcreteObserver-Klassen ist laut Musterbeschreibung beliebig, in der Skizze kommen sie nur ein Mal vor. Der zu beobachtende Zustand ist nicht näher

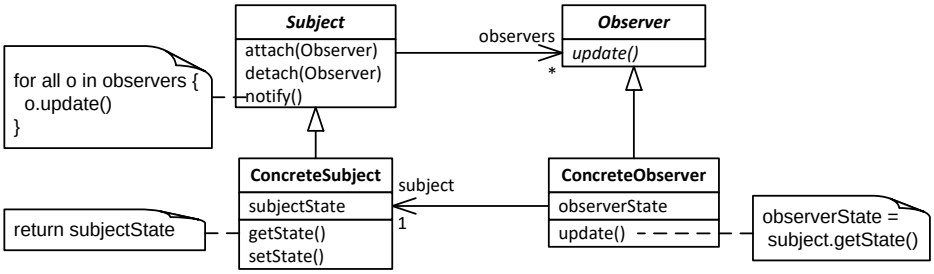


Abbildung 1.2: Dokumentierte Struktur des Observer-Musters (in UML- statt OMT-Notation) [GHJV95]

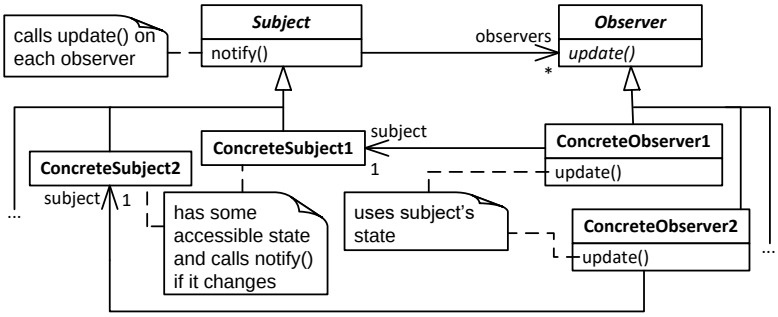


Abbildung 1.3: Gedachte Struktur des Observer-Musters

beschrieben. Dieser könnte ein Attributwert sein, eine Referenz auf ein anderes Objekt oder irgendeine andere beobachtbare Eigenschaft eines Objekts. In der dargestellten Entwurfsskizze wird der Zustand jedoch als Attributwert eines ConcreteSubject-Objekts angenommen. Ebenso ist das Verhalten der update-Methode nur eine mögliche Implementierung des Musters, in diesem Fall die Zuweisung des Attributwerts subjectState zu einem Attribut observerState der Klasse ConcreteObserver.

Korrekte Implementierungsvarianten nicht erkennbar

Aus der Darstellung des Musters in der Abb. 1.2 lässt sich nicht erkennen, in wieweit man beim Anwenden des Observer-Musters von der vorgeschlagenen Lösung abweichen kann, ohne vom Muster abzuweichen. Eine präzisere Beschreibung der Kernidee hinter dem Muster ist wünschenswert. Lässt man die für das Beispiel spezifischen Details wie die Attribute subjectState und observerState weg und berücksichtigt man eine beliebige Anzahl der Unterklassen von Subject und Observer, so lässt sich das Muster treffender[3] wie in der Abb. 1.3 skizzieren[4].

Beim Anwenden des Musters muss ein Entwickler entscheiden, wie er die vorgeschlagene Lösung für seine konkrete Situation anpasst, d.h. entscheiden, welche Klassen seines Systems welche Rollen spielen sollen, wie viele Beobachter es geben soll, was der zu beobachtende Zustand ist und wie er repräsentiert wird sowie was bei Änderung des Zustands geschehen soll.

Angenommen, das Muster soll bei der Implementierung eines Petrinetz-Editors verwendet werden, um die Darstellung eines Petrinetzes bei Änderung des Modells

[3] Die Multiplizität & Variabilität wird verdeutlicht. Erläuterung dazu in Abschnitt 3.3.1.

[4] In Abb. 1.3 habe ich u.a. die zur Implementierung von Assoziationen gehörenden Zugriffsmethoden getState, setState, attach, detach weggelassen.

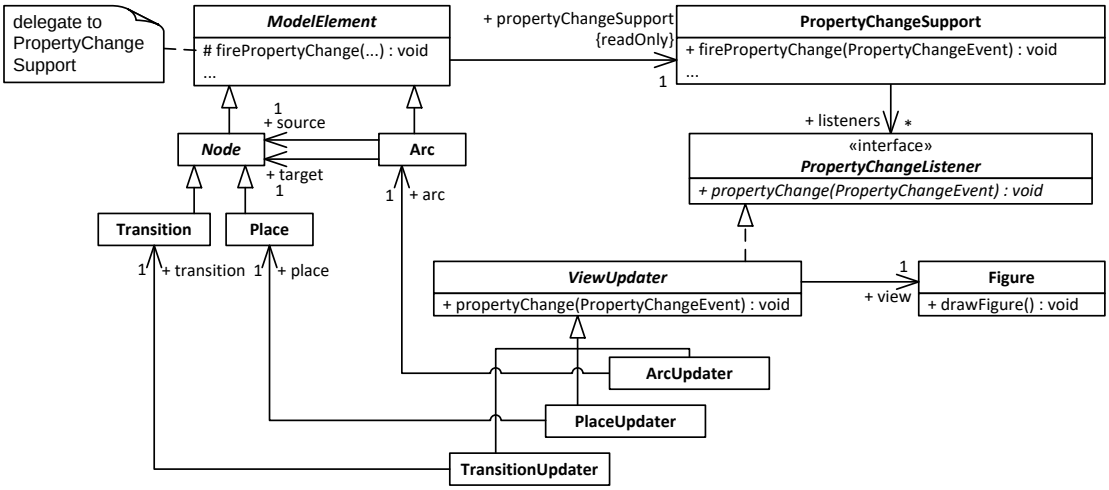


Abbildung 1.4: Eine Verwendung des Observer-Musters

zu aktualisieren. Dann könnte das Softwaresystem nach Anwendung des Musters aussehen wie in der Abb. 1.4 dargestellt. Die Rolle der Subject-Klasse übernimmt hier die Klasse ModelElement, die der Observer-Klasse die Schnittstelle PropertyChangeListener. Die ConcreteSubject-Rollen werden von den Klassen Transition, Place und Arc übernommen, während die ConcreteObserver-Rollen von den Klassen TransitionUpdater, PlaceUpdater und ArcUpdater gespielt werden. Ändert sich der zu beobachtende Zustand, so muss die Methode firePropertyChange der Klasse ModelElement aufgerufen werden.

Doch der Entwurf weicht von der Lösung in Abb. 1.3 etwas ab. Es gibt zusätzliche Klassen in den Klassenhierarchien wie Node und ViewUpdater. Außerdem ist die Referenz von Subject zu Observer durch eine Delegation von ModelElement über PropertyChangeSupport zu PropertyChangeListener realisert. Um die Benachrichtigung der Beobachter kümmert sich ein PropertyChangeSupport-Objekt. Die Rolle der update-Methode übernimmt die Methode propertyChange. Im Gegensatz zu update hat sie aber einen Parameter.

Fehleranfällige Musteranwendung

Hier wird eine semantische Lücke zwischen der Musterbeschreibung und seiner Implementierung deutlich. Die Struktur aus Abb. 1.3 weicht von der im vorliegenden Softwareentwurf in Abb. 1.4 deutlich ab. Solche Abweichungen können beabsichtigt oder das Ergebnis einer fehlerhaften Musteranwendung sein. Teile des Musters können falsch realisiert oder vergessen werden. Insbesondere können sich Fehler bei der Implementierung des zum Muster gehörenden Verhaltens einschleichen, z.B. wenn bei der Musteranwendung nur die Klassenstruktur betrachtet wird. Eine Unterscheidung zwischen korrekten und fehlerhaften Musterimplementierungen ist schwierig, weil dazu ein tiefgehendes Verständnis des Musters und des Softwaresystems benötigt wird.

> „And even though it is possible to bring together each program taking some role in a pattern, the editors surround the relevant elements with the clutter of other concerns in each of the program, making it (if you will pardon the expression) hard to see the pattern.“
> Bill Harrison [CHV00], IBM T.J. Watson Research, Informatik-Professor am Trinity College Dublin

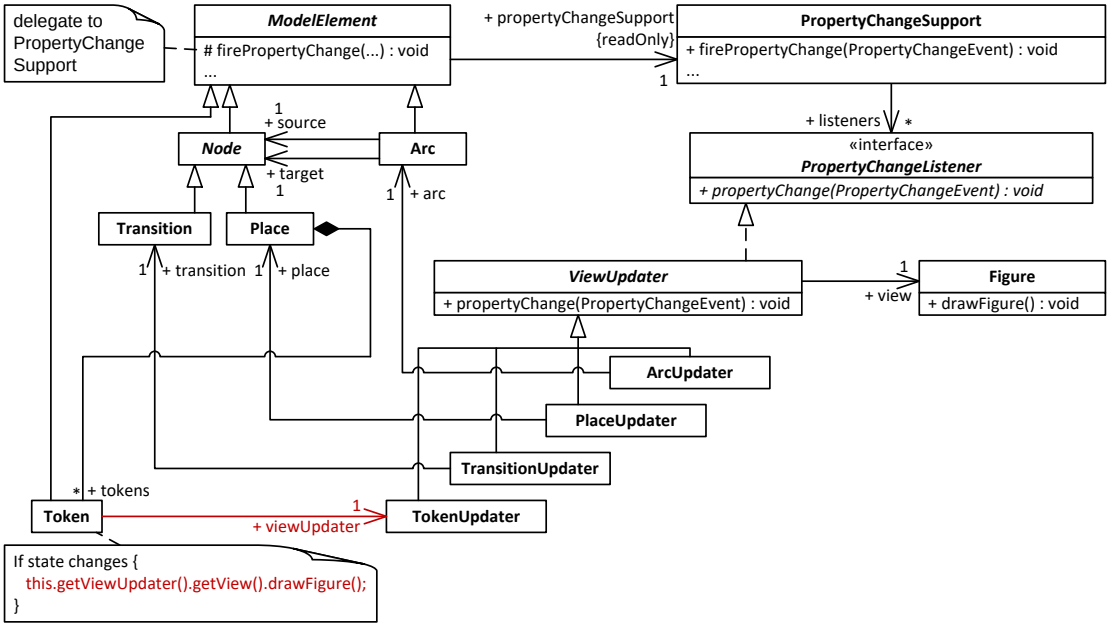


Abbildung 1.5: Eine falsche Entwurfserweiterung, abweichend vom Observer-Muster

Musterimplementierungen schwer zu erkennen

Ganz ohne oder mit unzureichender Dokumentation ist eine Implementierung eines Musters, zum Beispiel bei Wartungsarbeiten, aufgrund der Abweichungen vom vorgeschlagenen Entwurf und der vom verwendeten Muster ablenkenden zusätzlichen Entwurfsdetails (Klassen und Assoziationen, die nicht zum Muster gehören) schwer zu erkennen (siehe Harrison-Zitat). Ist die Implementierung eines Musters auf mehrere Sichten, Modelle, Dateien (z.B. Java-Dateien) oder andere Entwurfsartefakte verteilt, so verkompliziert das die Situation zusätzlich. Detailinformationen über das verwendete Muster und die Art, wie es angewendet wurde, verwischen meist nach der Musteranwendung. Wenn die Anwendung eines Musters dokumentiert wird, dann beschränkt sie sich meist allein auf die beteiligten Klassen, was zahlreiche Veröffentlichungen belegen (siehe Abschn. 9.2, z.B. Abb. 9.8 bis 9.10, S. 222). Welche Methoden und Assoziationen bestimmte Rollen des Musters einnehmen, wird üblicherweise nicht dokumentiert. Das liegt zum einen an Zeitmangel, zum anderen am Fehlen geeigneter Dokumentationsverfahren oder -werkzeuge.

Entwurfsfehler als Folge übersehener Musterimplementierungen

Übersehende Musterimplementierungen können zu Design-Erosion führen. Wird bei dem Beispiel aus Abb. 1.4 das angewandte Muster nicht erkannt, so kann das Muster bei Erweiterung des Entwurfs, z.B. um eine Token-Klasse, umgangen werden. So könnte die Token-Klasse wie in der Abb. 1.5 dargestellt über eine neue Assoziation direkt mit einem TokenUpdater-Objekt verbunden werden, um darüber die Darstellung des Tokens per Aufruf der drawFigure-Methode auf dem verknüpften Figure-Objekt zu aktualisieren. Diese Abweichung vom Muster macht den Entwurf noch schwieriger zu verstehen und erhöht damit den Wartungsaufwand. Außerdem geht die Intention des Musters verloren, weitere Beobachter ergänzen zu können, ohne die Implementierung der beobachteten Objekte zu ändern. Das Ergänzen weiterer Beobachter erfordert nun eine Anpassung der Token-Klasse, um auch auf die neuen Beobachter zu verweisen.

1.1.2 Bisherige Ansätze

Zu besseren Unterstützung von Entwicklern beim Einsatz von Entwurfsmustern wurden diverse Verfahren entwickelt. Ein Teil der Arbeiten legt den Fokus auf eine präzise Beschreibung der Entwurfslösung zu einem Muster. Andere Arbeiten bieten Werkzeuge, welche bei der Anwendung von Mustern oder bei der Visualisierung und Prüfung von Musterimplementierungen unterstützen sollen.

Codegenerierung

Erste Werkzeuge dieser Art generieren den Quellcode zur Implementierung eines Musters [BFYV96]. Das soll Fehler bei der Implementierung eines Musters reduzieren und bei der Auswahl einer von mehreren Implementierungsvarianten helfen. Der generierte Quellcode muss allerdings anschließend manuell angepasst und dokumentiert werden. Bei späteren Entwurfsänderungen ist die Implementierung eines Musters weiterhin schwer zu erkennen.

Prüfen von Code

In einer anderen, frühen Arbeit [FMvW97] wird ein Verfahren vorgestellt, bei dem die Implementierung eines Musters nicht nur generiert, sondern auch nach manuellen Anpassungen auf Konformität zum Muster überprüft wird. Allerdings sind die Prüfoperation relativ beschränkt und es müssen der zu generierende Code und die Prüfoperationen für jedes Muster manuell implementiert werden, was das Spezifizieren neuer Muster sehr aufwendig macht (Details in Abschn. 9.4.1).

Übertragen von UML-Entwurfsbeispielen

Mit der Verbreitung modellgetriebener Softwareentwicklung wurden auch Ansätze vorgestellt, welche Entwickler bei der Anwendung von Entwurfsmustern auf Modell-Ebene, insbesondere in UML-Modellen, unterstützen. Dazu zählen u.a. kommerzielle Werkzeuge [IBM, Spaa, Bor]. Muster werden dabei durch UML-Modelle wie UML-Klassen- und Sequenzdiagramme beschrieben. Die Modelle dienen als eine Art Prototyp, welcher bei der Anwendung eines Musters in einen existierenden Entwurf kopiert und angepasst wird. Dieses Vorgehen erlaubt nur das Übernehmen vordefinierter Entwurfsmodelle und schränkt damit die Flexibilität der Entwickler ein, z.B. durch eine vorbestimmte Anzahl von Observer-Klassen beim Observer-Muster. Außerdem ist ein Überprüfen von angepassten Musterimplementierungen in UML-Modellen mit diesen Werkzeugen nicht möglich oder die Prüfung bedarf der Implementierung eigener Prüfoperationen.

Konsistenzprüfung zwischen Muster und Entwurf

Einige Ansätze konzentrieren sich allein auf die Überprüfung von Musterimplementierungen im Code oder auf Modell-Ebene. Das Anwenden eines Musters wird nicht unterstützt. Eden et al. stellen eine formale Spezifikationssprache für objektorientierte Entwurfsmuster vor [NGEK09]. Die Sprache deckt viele Varianten einer Musterimplementierung u.a. mit Hilfe spezieller Relationen zwischen Klassen und Methoden ab. Für die Überprüfung eines Entwurfs muss ein Entwickler Klassen seines Softwaresystems den Rollen eines spezifizierten, angewandten Musters zuordnen. Diese Zuordnung wird bei Entwurfs- oder Implementierungsänderungen nicht aufrechterhalten und muss für jedes Muster einzeln vorgenommen werden. Im Gegensatz dazu versuchen Kim et al., eine solche Zuordnung automatisch zu finden [KS08], was jedoch mit den für Reverse-Engineering-Verfahren bekannten Ungenauigkeiten[5] bei der Suche verbunden ist und dadurch zu Fehleinschätzungen des Entwurfs führen kann.

[5] Es können False Positives und False Negatives auftreten. D.h., es können Stellen für Musterimplementierungen gehalten werden, die keine sind und existierende Musterimplementierungen übersehen werden.

Maplesden et al. beschreiben Muster durch UML-ähnliche Diagramme [MHG02, MHG07] und unterstützen Entwickler bei der Anwendung von Mustern. Jede Musterimplementierung wird manuell mit der zugehörigen Musterspezifikation verknüpft und kann somit später überprüft werden. Die Überprüfung beschränkt sich jedoch auf einfache Regeln wie das Erkennen von fehlenden, aber im Muster beschriebenen Klassen und Beziehungen. Verhalten wird hierbei nicht berücksichtigt.

Formalisierung von Mustern

Jede Art von Werkzeugunterstützung für den Einsatz von Entwurfsmustern erfordert irgendeine Art von Formalisierung der Entwurfsmuster oder einiger ihrer Eigenschaften. Aufgrund der vagen Beschreibung von Entwurfsmustern kommt es zu sehr unterschiedlichen Interpretationen der beschriebenen Entwurfslösung und damit zu unterschiedlichen Musterspezifikationen [Tai07a]. Insbesondere erweist es sich als sehr schwierig, wenn nicht unmöglich, die Essenz eines Musters und alle dazu passenden Implementierungsmöglichkeiten komplett in einer werkzeugverarbeitbaren Musterspezifikation zu vereinen. Die Konsequenz sind entweder sehr wenige erlaubte Musterimplementierungsvarianten von wenigen, sehr präzise beschriebenen Mustern oder sehr eingeschränkte Werkzeugunterstützung in Verbindung mit weniger strikten Musterspezifikationen.

Zusammenfassend bieten bisherige Arbeiten keinen Ansatz, bei dem objektorientierte Entwurfsmuster samt Verhalten und Implementierungsvarianten geeignet spezifiziert, voll- oder halbautomatisch zur Anwendung gebracht und zugehörige Anwendungsstellen automatisch sowie hinreichend präzise erfasst, visualisiert und validiert werden können.

1.2 Ziele und Forschungsfragen

Trotz ihrer guten Eigenschaften und Verbreitung ist die Verwendung von Entwurfsmustern nach wie vor schwierig und mit vielen, manuellen, fehleranfälligen Schritten verbunden (Details in Kap. 9). Es stellt sich also die Frage, ob Entwickler beim Einsatz von Entwurfsmustern besser unterstützt werden können.

Ziele

Um die Wahrscheinlichkeit für Entwurfsfehler und Design-Erosion zu reduzieren, verfolge ich mit meiner Arbeit insbesondere folgende Ziele:

(1) Entwickler durch die Anwendung eines Entwurfsmusters führen
(2) Musteranwendungsstellen erfassen und sichtbar machen
(3) Musteranwendungsstellen nach Entwurfsänderungen automatisch prüfen

Zur Vermeidung von Entwurfsfehlern sollen Entwickler bei der Anwendung von Mustern angeleitet werden. Bereits implementierte Muster im Entwurf sollen Entwicklern sichtbar gemacht werden, damit sie bei Entwurfsentscheidungen berücksichtigt werden können. Auf potentiell unbeabsichtigte Abweichungen von bereits eingesetzten Mustern sollen Entwickler aufmerksam gemacht werden, um ihre Entscheidungen zu überdenken und so Design-Erosion zu vermeiden.

wissenschaftliche Herausforderungen

Daraus ergeben sich einige Herausforderungen. Es müssen präzise Repräsentationen der absichtlich vage beschriebenen Entwurfsmuster bzw. der Entwurfslösungen gefunden werden. Dabei besteht die Schwierigkeit darin, möglichst alle Implementierungsvarianten präzise und vollständig zu erfassen und Entwicklern dennoch die Freiheit zu bieten, eine Entwurfslösung an eine konkrete Situation

anzupassen, Entwickler also nicht unnötig in ihren Entwurfsentscheidungen einzuschränken. Neben der Spezifikation der Entwurfslösung zu einem Muster ist zwecks Visualisierung und Prüfung auch eine detaillierte Erfassung von Musteranwendungsstellen gefordert. Das zu entwickelnde Verfahren soll Entwickler beim Einsatz von Entwurfsmustern möglichst entlasten.

Annahmen

In meiner Arbeit beschränke ich mich auf modellgetriebene Softwareentwicklungsverfahren wie die weit in der Praxis verbreiteten Ansätze MDSD [SV06] und MDA [OMG03]. Ich konzentriere mich vor allem auf objektorientierte Entwurfsmuster und nehme an, dass sowohl die Anwendung von Entwurfsmustern als auch sämtliche Entwurfsänderungen auf Modell-Ebene[6] und nicht im Code erfolgen sowie dass der Quellcode vollständig aus einem Entwurfsmodell generiert wird. Ebenso gehe ich davon aus, dass aufgrund der Korrektheit eines eingesetzten Codegenerators ein auf Modell-Ebene korrekt angewandtes Muster zu einer korrekten Implementierung des Musters im Code führt. Mit den von mir erarbeiteten Verfahren möchte ich Entwickler bei der Anwendung von Entwurfsmustern und bei dem Erhalt ihrer Intention unterstützen. Dabei gehe ich davon aus, dass sie mit den angewandten Entwurfsmustern gut vertraut sind, die Entwurfslösung verstanden haben und in der Lage sind, sie abgesehen von Flüchtigkeitsfehlern korrekt anzuwenden. Eine vollautomatische Anwendung von Mustern ist nicht beabsichtigt und meiner Meinung nach im Allgemeinen nicht möglich. Kein Werkzeug kann die Expertise der Entwickler vollständig ersetzen:

> „While mechanism can't subsume patterns, it can certainly support them. [...] We should continue to explore how tools can help all facets of the development process, pattern application included. [...] And no tool – no language – can compensate for its user's inexperience or incomplete understanding of the problem."
> John Vlissides [CHV00], IBM T.J. Watson Research, einer der vier Autoren des Buches „Design Patterns" [GHJV95], † Nov. 2005

Forschungsfragen

Aus den genannten Zielen leite ich folgende Forschungsfragen ab:

(i) Wie lässt sich die Entwurfslösung[7] eines Entwurfsmusters möglichst präzise, vollständig und Werkzeug-verarbeitbar unter Berücksichtigung möglichst aller Implementierungsvarianten spezifizieren?

(ii) Wie weit lässt sich die Anwendung von Entwurfsmustern automatisieren und wie können Entwickler bei der Anwendung geleitet werden?

(iii) Wie kann man Musteranwendungsstellen bzw. Musterimplementierungen in Entwurfsmodellen genau erfassen und zwecks Berücksichtigung bei Entwurfsänderungen geeignet visualisieren?

(iv) Wie lassen sich unbeabsichtigte Abweichungen von angewandten Entwurfsmustern automatisiert aufdecken?

[6] Mit Modell meine ich eine deutlich abstraktere und kompaktere Repräsentation als Quellcode, z.B. ein UML-Modell.

[7] Der (weitestgehend) formal erfassbare Teil des Musters. Siehe Abb. 2.1, S. 26 zur Klärung der verwendeten Begriffe.

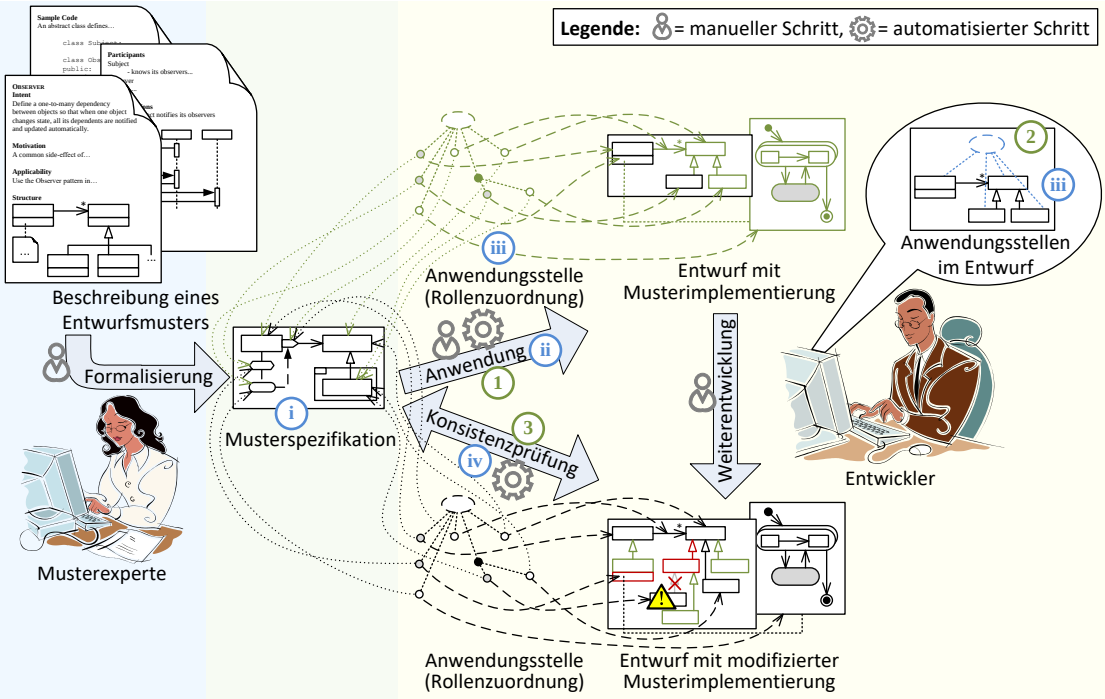


Abbildung 1.6: Der komplette Ansatz im Überblick

1.3 Lösungsansatz

Mein kompletter Lösungsansatz ist in der Abb. 1.6 skizziert. Hier sind auch die in Abschnitt 1.2 genannten Ziele (1) bis (3) und Forschungsfragen (i) bis (iv) angegeben.

Modell der Anwendungsstellen

Die Kernidee meiner Arbeit ist es, die Anwendungsstellen von Entwurfsmustern in einem speziellen Modell zu erfassen und dieses Modell zum einen zur Visualisierung und zum anderen zur Validierung von Musterimplementierungen zu verwenden (Ziele (2) & (3) und Forschungsfragen (iii) & (iv)). Eine Anwendungsstelle wird insbesondere durch eine Zuordnung der Rollen eines Entwurfsmusters zu den sie einnehmenden Elementen eines Softwareentwurfs charakterisiert.

Spezifikation der Entwurfslösung

Zur Erfassung der Rollen und der informell beschriebenen Entwurfslösung eines Entwurfsmusters modelliere ich diese mit Hilfe einer von mir entwickelten Musterspezifikationssprache (Forschungsfrage (i)).

Generierung der Modelle

Um den Aufwand für das Erstellen und Pflegen von Anwendungsstellenmodellen auf ein Minimum zu reduzieren und die Anwendung eines Musters weitestgehend zu vereinfachen, automatisiere ich einen Großteil der Anwendungsschritte (Forschungsfrage (ii)): Bei Anwendung eines Musters wird eine Anwendungsfallspezifische Musterimplementierung im Entwurfsmodell generiert. Gleichzeitig wird ein detailliertes Modell der Anwendungsstelle erzeugt. Letzteres wird zur Visualisierung und zur Validierung der Konsistenz zwischen einer Musterimplementierung und der zugehörigen Musterspezifikation genutzt.

Durch die Anwendung führen

Zur Vermeidung von Flüchtigkeitsfehlern setze ich ein semiautomatisches Verfahren zur Anwendung eines Entwurfsmusters ein, bei welchem Entwickler durch die Anwendung eines Entwurfsmusters geführt werden. (Ziel (1) und Forschungs-

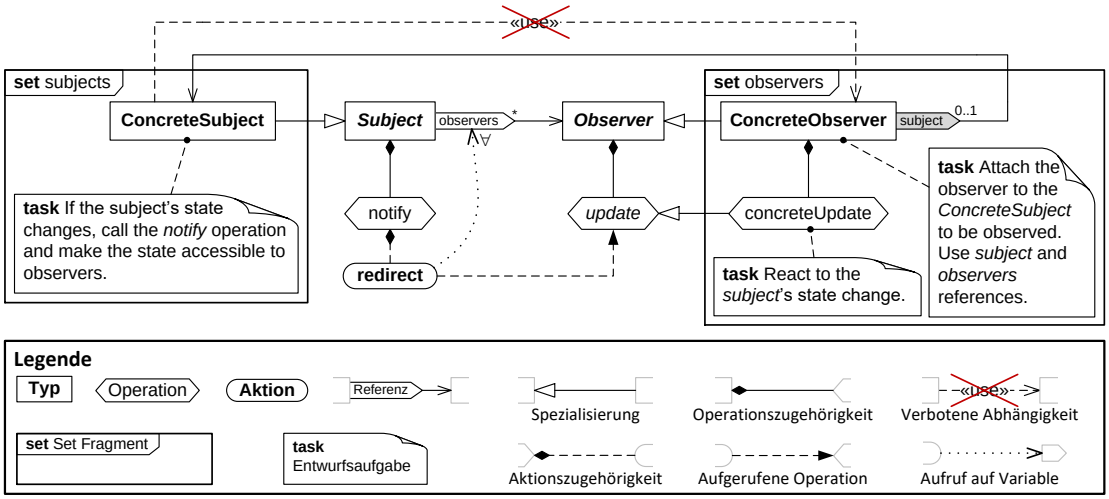


Abbildung 1.7: Spezifikation des Observer-Entwurfsmusters, eine weitere Spezifikationsvariante in Abb. B.21, S. 297

frage (ii)). Dazu wird visualisiert, welche Rollen des Musters bereits eingenommen wurden und welche Teilaufgaben der Musteranwendung noch offen geblieben sind.

objektorientierte Struktur und Verhalten

Das Konzept eines Entwurfsmusters umfasst mehrere Facetten, u.a. in welchen Situationen es angewandt werden kann, die Entwurfslösung, Lösungsvarianten, Konsequenzen und Implementierungsbeispiele. Für meinen Ansatz spezifiziere ich nur die Entwurfslösung, genauer: die objektorientierte Struktur und das Verhalten der beschriebenen Entwurfslösung.

Das Observer-Muster aus Abb. 1.3 spezifiziere ich z.B. wie in der Abb. 1.7 dargestellt. Zum einen wird die Struktur in Form von Typen, ihren Operationen, Attributen und Beziehungen beschrieben. Zum anderen wird auch Interaktionsverhalten durch den Knoten redirect angegeben. Hier drückt der Knoten aus, dass die Operation notify auf allen (∀) über die Assoziation observers verknüpften Observer-Objekten die Methode update aufruft und dabei sämtliche Argumente weiterreicht, falls vorhanden.

Spezifikation von Lösungsvarianten

Um möglichst viele Implementierungsvarianten eines Musters durch eine Musterspezifikation zu erfassen und Entwickler in ihren Entwurfsentscheidungen nicht unnötig einzuschränken, bietet meine Musterspezifikationssprache u.a. Konstrukte zur Beschreibung sich wiederholender Entwurfsstrukturen. So kann z.B. spezifiziert werden, dass bei der Anwendung des Observer-Musters beliebig viele ConcreteObserver mit je einer concreteUpdate-Methode und einer Referenz auf ein ConcreteSubject erstellt werden können (vgl. Abb. 1.3 auf S. 5). Ausgedrückt wird das durch sogenannte *Set Fragments* – in Abb. 1.7 durch ein Rechteck mit der Beschriftung **set** observers bzw. **set** subjects dargestellt.

Beziehungen wie Vererbung, Assoziationen und Delegationen werden so interpretiert, dass eine direkte Beziehung in der Spezifikation aufgrund ihrer Transitivität durch eine indirekte Beziehungen in der Implementierung realisiert werden kann. Dadurch ist es z.B. möglich, bei der Anwendung eines Musters zusätzliche Klassen in die Vererbungshierarchie einer Musterimplementierung einzufügen.

Neben der Struktur und dem Verhalten einer Entwurfslösung kann eine Muster-

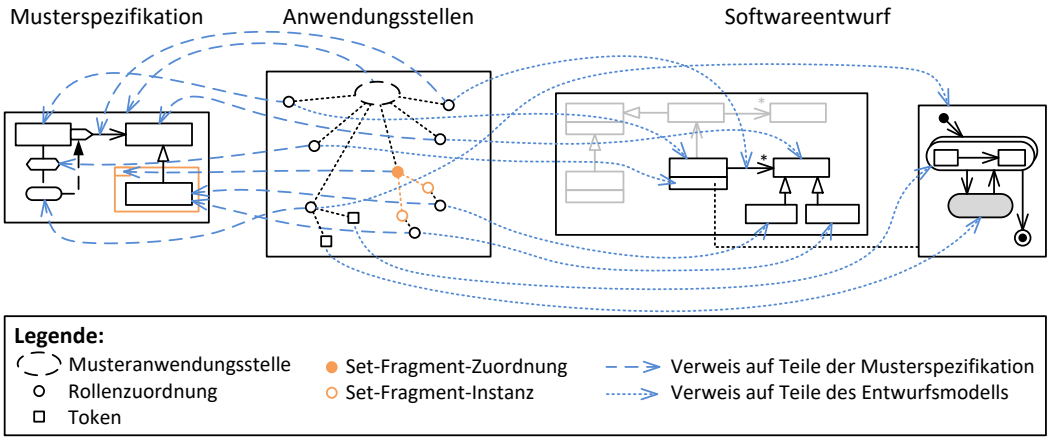


Abbildung 1.8: Zuordnung der Musterrollen zu Elementen im Entwurf

spezifikation Bedingungen enthalten, welche die Implementierungsmöglichkeiten einschränken. So kann z.B. beschrieben werden, dass die ConcreteSubject-Klasse keinerlei Abhängigkeit zu ConcreteObserver-Klassen haben darf – ausgedrückt durch die gestrichelte Kante mit dem durchgestrichenen Wort «use». So wird die Intention des Musters, Beobachtete von Beobachtern zu entkoppeln, in der Spezifikation erfasst. Nur so können Beobachter ergänzt oder entfernt werden, ohne die ConcreteSubject-Klassen ändern zu müssen.

Kopplungsrestriktionen

Nicht alle Teile der in einem Muster beschriebenen Entwurfslösung lassen sich formal beschreiben. Damit einzelne Schritte zur Anwendung eines Musters nicht vergessen werden, ergänze ich die Musterspezifikationen um in natürlicher Sprache formulierte Anweisungen an Entwickler.

Anleitung bei manuellen Schritten

Die Musterspezifikationssprache ist eine erweiterbare domänenspezifische Sprache für objektorientierte Entwurfsmuster. Sie lässt sich mit Hilfe geeigneter Transformationen in verschiedene Modellierungssprachen wie UML [OMG11a, OMG11b] oder EMOF[8] übersetzen. Eine solche Übersetzung stelle ich in dieser Arbeit exemplarisch vor. Dabei übersetze ich die Entwurfsstruktur in EMOF-Klassendiagramme und das Verhalten in Story-Diagramme, eine spezielle Form von Aktivitätendiagrammen (Details in Abschn. 2.4.2).

Eine Anwendungsstelle wird im Wesentlichen durch die Zuordnung sämtlicher Musterrollen zu den sie einnehmenden Elementen im Softwareentwurf modelliert (siehe Abb. 1.8). Alle Anwendungsstellen werden separat vom Entwurf in einem gemeinsamen Modell erfasst. Das Entwurfsmodell – in der Abbildung bestehend aus je einem Modell für Struktur und Verhalten – wird durch ein Modell der Anwendungsstellen nur ergänzt, aber nicht verändert.

Rollenzuordnung

Die Anwendung eines Entwurfsmusters erfolgt semiautomatisch (siehe Abb. 1.9). Vor Anwendung eines Musters existiert üblicherweise ein Teil des Softwareentwurfs, z.B. die Klassen Arc, ModelElement und Figure aus Abb. 1.4 (S. 6). Nach Zuordnung der Musterrollen zu existierenden Teilen des Entwurfs durch den Entwickler, z.B. der Klasse Arc zur Rolle ConcreteSubject, werden die Implementierung

Halbautomatische Musteranwendung

[8] EMOF: Essential MOF, eine Teilmenge der MOF [OMG11c]. ECore aus dem Eclipse Modeling Framework (EMF) [SBPM08] ist eine Implementierung der EMOF und eine weit verbreitete Modellierungssprache. EMF: http://www.eclipse.org/modeling/emf/

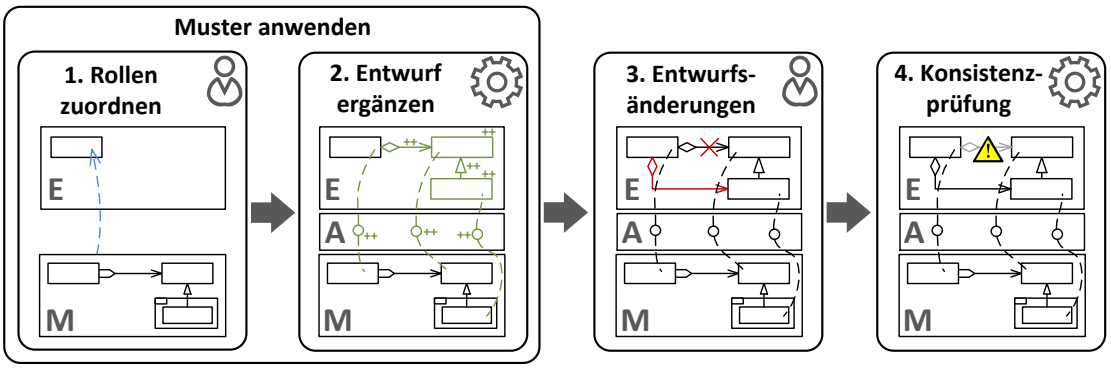


Abbildung 1.9: Anwendung und Validierung von Entwurfsmustern

des Musters im Entwurfsmodell und das Modell der Anwendungsstelle automatisch ergänzt (Schritt „Muster anwenden“). Dabei werden zu allen Rollen der Musterspezifikation ihnen entsprechende Elemente im Entwurfsmodell erstellt, sofern sie nicht schon existieren.

Nach Anwendung eines Musters bleibt der Softwareentwurf und die Implementierung des Musters selten unverändert. Entweder wird der noch unvollständige Entwurf weiterentwickelt und erweitert (Erstentwurf einer Software) oder er wird zu einem späteren Zeitpunkt an neue Anforderungen angepasst (Wartung). In jedem Fall können die Änderungen (Schritt „Entwurfsänderungen“ in Abb. 1.9) eine vorliegende Musterimplementierung negativ beeinflussen, z.B. Teile davon entfernen oder ungewollte Beziehungen einführen.

Automatische Konformitätsprüfung

Aus diesem Grund bietet mein Ansatz eine automatische Prüfung der Konformität einer Musterimplementierung zur zugehörigen Musterspezifikation (Schritt „Konsistenzprüfung“ in Abb. 1.9). Dabei wird – sofern das automatisch möglich ist – geprüft, ob eine Musterimplementierung auch nach Änderung des Entwurfs in Bezug auf die Musterspezifikation vollständig ist und keine ungewollten Abhängigkeiten eingeführt wurden.

Visualisierung während und nach der Musteranwendung

Damit bereits erfolgte Musteranwendungen bei späteren Entwurfsänderungen nicht übersehen werden, bietet mein Ansatz mehrere Arten der Visualisierung von Anwendungsstellen.

Während der Musteranwendung wird eine Visualisierung in Anlehnung an die Musterspezifikation verwendet. Dabei wird eine Musterspezifikation mit den Namen der die Musterrollen einnehmen Entwurfselemente angereichert. Noch nicht zugeordnete Rollen werden farblich hervorgehoben. Offene Teilaufgaben der Musteranwendung werden gekennzeichnet.

Nach erfolgter Musteranwendung werden die Musteranwendungsstellen bzw. Musterimplementierungen im Entwurfsmodell hervorgehoben. Hierbei werden alle an Musterimplementierungen beteiligten Elemente wie Klassen, Methoden und Assoziationen oder Teile eines Story-Diagramms markiert. Abhängig vom gewählten Detailgrad werden zusätzlich zu den Musternamen auch die Namen der eingenommenen Rollen dargestellt.

Die im Rahmen dieser Arbeit entwickelten Verfahren dienen als Ergänzung zu

bereits existierenden Vorgehensmodellen und sollen mit beliebigen modellgetriebenen Softwareentwicklungsprozessen eingesetzt werden. Die Anwendungsszenarien reichen von der initialen Anwendung eines Entwurfsmusters, über eine folgende Änderung des Entwurfs mit einer Anpassung oder Prüfung der Musterimplementierung, bis hin zu einer nachträglichen Dokumentation (Modellierung) zuvor manuell erstellter Musterimplementierungen. Selbst das Entfernen einer nicht mehr passenden Musterimplementierung wird unterstützt, indem alle beteiligten Entwurfsteile sichtbar gemacht werden.

Anwendungsszenarien

1.4 Wissenschaftliche Beiträge und Ergebnisse der Arbeit

Zu den vier Forschungsfragen aus Abschnitt 1.2 wurden Antworten gefunden. Es wurden Verfahren (i) zur Spezifikation von Entwurfslösungen, (ii) zur halbautomatischen Musteranwendung, (iii) zur Erfassung und Visualisierung von Anwendungsstellen und (iv) zur automatischen Prüfung von Anwendungsstellen auf Konformität zur Musterspezifikation entwickelt und anhand eines Prototypen und diverser Entwurfsmuster evaluiert.

Es wurde eine erweiterbare, menschen- und maschinenlesbare Musterspezifikationssprache zur Beschreibung objektorientierter Entwurfsmuster vorgestellt. Die Sprache umfasst neben der Entwurfsstruktur bis zu einem gewissen Grad auch Verhalten und bietet diverse Möglichkeiten zur Beschreibung zahlreicher Implementierungsvarianten und Kopplungsregeln in nur einer Spezifikation. Die Ausdrucksstärke der Musterspezifikationssprache wurde exemplarisch anhand der 23 GoF-Entwurfsmuster, einiger Architekturmuster und einiger Idiome eingeschätzt (Spezifikationen im Anhang B).

Ausdruckskraft der Spezifikationssprache

Für die Realisierung einer semiautomatischen Musteranwendung wurde eine spezielle Art einer exogenen Modelltransformation mit besonderen Herausforderungen entwickelt. Im Gegensatz zu herkömmlichen Übersetzungstransformationen beginnt die Übersetzung bei meinem Verfahren an einer vorgegebenen Stelle im Zielmodell (abhängig von der Rollenzuordnung vor Musteranwendung) und das Ergebnis der Übersetzung macht nur einen Teil des Zielmodells aus (eine Musterimplementierung ist nur ein Teil des Entwurfsmodells). Außerdem hängt das Ergebnis der Übersetzung von bereits existierenden Elementen im Zielmodell ab (z.B. kann die Signatur einer Methode bei der Übersetzung von einer anderen, im Entwurf schon existierenden Methode abhängen).

besondere Modelltransformation

Zur Bildung einer bzgl. Set Fragments gültigen Implementierungsvariante eines Musters wurde das Konzept der Auffaltung von Set Fragments entwickelt. Dabei wird der Graph, welcher die Entwurfsstruktur im Set Fragment beschreibt (z.B. eine ConcreteObserver-Klasse mit zugehöriger Operation und Referenz, siehe Abb. 1.7, S. 12), schrittweise repliziert und mit dem umgebenden Graphen verbunden. Das Konzept der Set Fragments und die zugehörige Auffaltungsoperation wurden formal definiert.

Set Fragments und ihre Auffaltung

Es wurde ein Verfahren zur detaillierten Erfassung einer Musteranwendungsstelle u.a. in Verhaltensmodellen entwickelt, welches sich sowohl zur Visualisierung sämtlicher Musterrollen, als auch zur Prüfung der Konformität mit einer Musterspezifikation eignet.

Modell der Anwendungsstellen

Notationen für Anwendungsstellen

Zur Visualisierung von Anwendungsstellen wurden diverse Notationen vorgestellt. Anwendungsstellen können sowohl einzeln in einem separaten Diagramm, als auch mit Hilfe von Markierungen mit ihrem Kontext direkt im Entwurf dargestellt werden. Es werden mehrere Detailgrade angeboten.

Konsistenzprüfung

Zusätzlich werden Wege zur Prüfung der Konformität einer Musterimplementierung aufgezeigt. Neben den prüfbaren Eigenschaften wird auch ein Verfahren zur Prüfung dieser Eigenschaften vorgestellt.

Prototyp und Evaluation

Die vorgestellten Konzepte und Verfahren wurden prototypisch implementiert und anhand diverser Anwendungsszenarien praktisch erprobt. Insbesondere wurde der Ansatz anhand des JUnit[9]-Frameworks praxisnah evaluiert. Dazu wurden Teile des Frameworks modellgetrieben neu entwickelt und dabei die von Gamma gut dokumentierten Entwurfsschritte und Musteranwendungen [Gam01] nachempfunden. Darüber hinausgehende empirische Untersuchungen (z.B. vergleichende Fallstudien, kontrollierte Experimente) wurden bisher nicht durchgeführt.

Der Prototyp wurde als Eclipse[10]-basierte Entwicklungsumgebung mit Hilfe des Eclipse Modeling Frameworks (EMF)[11] [SBPM08] implementiert. Die Klassenstruktur der zu entwickelnden Software wird in Ecore modelliert (Teil von EMF), während das Verhalten in Story-Diagrammen modelliert wird [NZJ13, vDHH+12, FNTZ00]. Die Anwendung von Entwurfsmustern erfolgt durch Übersetzen einer Musterspezifikation in Ecore- und Story-Diagramm-Modelle. Zur Musterspezifikation bietet der Prototyp einen grafischen Editor mit vollem Umfang der Musterspezifikationssprache.

Einschränkungen des Prototypen

Von den vorgestellten Visualisierungsmöglichkeiten für Anwendungsstellen wurde im Prototypen nur ein Teil implementiert. Eine vollständige Implementierung wäre zu aufwändig und ohne großen Mehrwert für die Evaluation, da der Nutzen der Visualisierungsmöglichkeiten bereits relativ gut anhand von Visualisierungsentwürfen für Anwendungsbeispiele abschätzbar ist. Aus ähnlichen Gründen habe ich auf die vollständige Implementierung der Konsistenzprüfung von Anwendungsstellen im Prototypen verzichtet.

1.5 Struktur der Arbeit

Grundlagen

In Kapitel 2 erläutere ich die Grundlagen für diese Arbeit. Dazu zählen Softwareentwurfsmuster und ihre Abgrenzung zu anderen Konzepten, modellgetriebene Softwareentwicklung, Modell- und Graphtransformationen sowie Story-Diagramme sowie in dieser Arbeit verwendete Begriffe.

Hauptteil

In den folgenden Kapiteln stelle ich meinen Ansatz im Detail vor. Dieser ist in der Abb. 1.10 grob zusammengefasst. Die Zahlen verweisen auf die Kapitel, in welchen die jeweiligen Teile des Ansatzes beschrieben werden. Wie die Entwurfslösung zu einem Entwurfsmuster spezifiziert wird, stelle ich in Kapitel 3 vor. Die Modellierung und Visualisierung von Anwendungsstellen bespreche ich in Kapitel 4. Das semiautomatische Verfahren zur Anwendung von Entwurfsmustern

[9] JUnit-Test-Framework: http://junit.org
[10] Eclipse: http://www.eclipse.org
[11] Eclipse Modeling Framework (EMF): http://www.eclipse.org/modeling/emf/

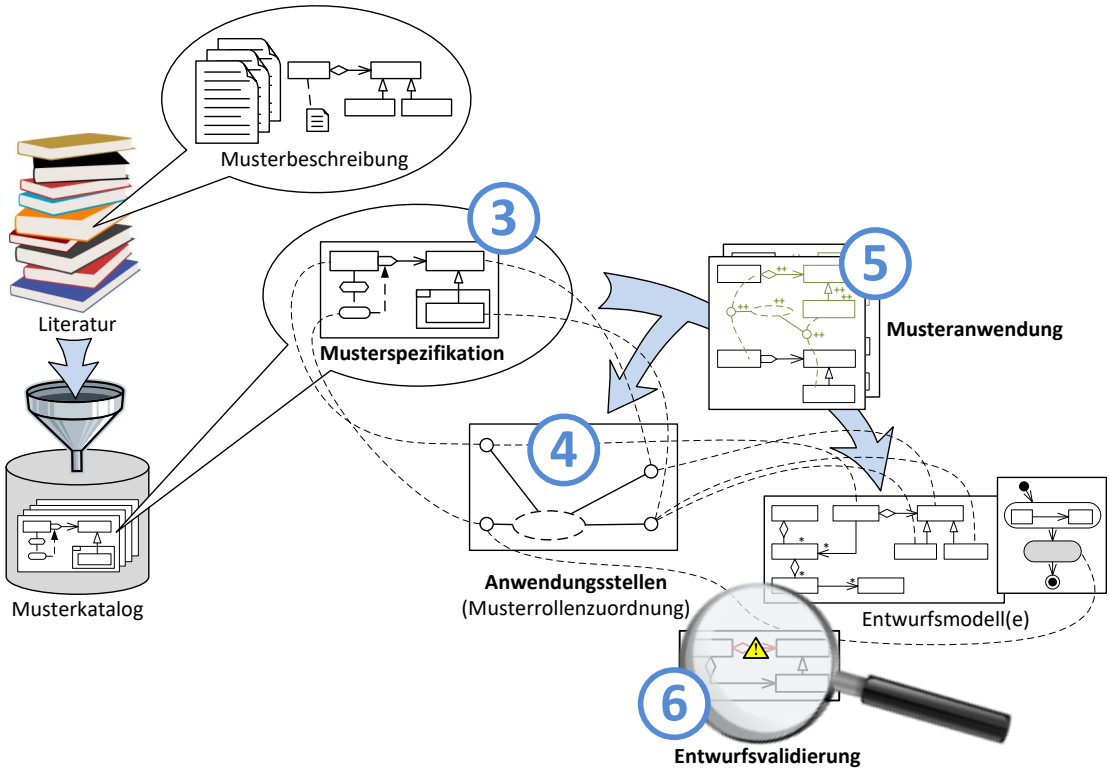


Abbildung 1.10: Überblick zum vorgestellten Ansatz und den Kapiteln dieser Arbeit

beschreibe ich in Kapitel 5. Auf das Konzept zur Validierung von Entwurfsmusterimplementierungen gehe ich in Kapitel 6 ein.

Evaluation, Abgrenzung, Schluss

Meine prototypische Implementierung der entwickelten Verfahren stelle ich in Kapitel 7 vor, während ich in Kapitel 8 die Evaluation der Verfahren bespreche. Verwandte Arbeiten diskutiere ich in Kapitel 9. Abschließend gebe ich in Kapitel 10 eine Zusammenfassung und einen Ausblick.

Anhang

Im Anhang biete ich einen detaillierteren, ergänzenden Einblick in meine Ergebnisse. Die Musterspezifikationssprache in vollem Umfang stelle ich im Anhang A vor. Dort dokumentiere ich neben der konkreten auch die abstrakte Syntax und definiere die Semantik. Sämtliche bisher erstellten Musterspezifikationen präsentiere ich im Anhang B. Alle Details einer Musteranwendung, der Überführung einer Musterspezifikation in eine Musterimplementierung, beschreibe ich im Anhang C. Die Schritte bei der Anwendung aller im JUnit-Framework eingesetzten Muster mit meinem Verfahren und meinen prototypischen Werkzeugen dokumentiere ich im Anhang D. Auf Details der prototypischen Implementierung gehe ich im Anhang E ein. Zum Nachschlagen verwendeter Begriffe biete ich neben dem Abschnitt 2.1.3 einen Glossar (S. 365 ff.).

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit nötigen Konzepte und Begriffe eingeführt. In Abschnitt 2.1 werden Softwareentwurfsmuster samt ihrer Intention, ihrer Eigenschaften und ihrer Arten beschrieben und von verwandten Konzepten abgegrenzt. Nachdem in Abschnitt 2.2 modellgetriebene Softwareentwicklung und Model-Driven Architecture besprochen werden, werden in Abschnitt 2.3 die Begriffe Modell, Modelltransformation und Graphtransformation erläutert. Anschließend wird in Abschnitt 2.4 vorgestellt wie modellgetriebene Softwareentwicklung mit Klassen- und Story-Diagrammen möglich ist und welche Rolle Story Patterns dabei spielen.

Ergänzend zu diesem Kapitel werden die wichtigsten der in dieser Arbeit verwendeten Begriffe im Glossar erläutert (S. 365 ff.).

2.1 Softwareentwurfsmuster

In der Softwareentwicklung stößt man immer wieder auf ähnliche Entwurfsprobleme. Um Zeit und Geld zu sparen, versucht man, bereits existierende Lösungen wiederzuverwenden. Neben dem Wiederverwenden existierender Architekturen, Frameworks, Softwarebibliotheken oder Algorithmen sowie der Nutzung diverser Konzepte wie der Objektorientierung und Polymorphie oder der komponentenbasierten Entwicklung bieten Softwareentwurfsmuster eine weitere Form der Wiederverwendung in der Informatik. Ein Softwareentwurfsmuster beschreibt ein Entwurfsproblem, die Intention der vorgeschlagenen Lösung, die Lösungsidee in Form eines beispielhaften Entwurfs und die daraus folgenden Konsequenzen. Die wohl bekannteste Sammlung von Entwurfsmustern, das Buch *Design Patterns – Elements of Reusable Object-Oriented Software*, stammt von der sogenannten „Gang of Four“ (kurz: GoF) [GHJV95].

Muster als Form der Wiederverwendung

Die Idee von wiederverwendbaren Lösungsmustern kommt ursprünglich aus dem Bereich der Gebäudearchitektur [AIS77] und wurde auf die Softwarearchitektur und den Softwareentwurf übertragen.

> „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“
> Christopher Alexander et al., 1977 [AIS77]

Seit der Veröffentlichung des Buches der „Gang of Four“ haben sich Entwurfsmuster in der Softwareentwicklung weit verbreitet und bewährt. Zusätzlich zu denen der „Gang of Four“ gibt es inzwischen unzählige Veröffentlichungen mit weiteren Entwurfsmustern (siehe Abschnitt 2.1.2).

Muster definieren Anpassungs- und Erweiterungsstellen

Alle diese Muster haben eine bestimmte Intention, z.B. etwas leicht austauschbar oder anpassbar zu machen und damit den Aufwand für Wartbarkeit und zukünftige Anpassungen gering zu halten. Die Muster beschreiben also in gewisser Weise wie man Software an bestimmten Stellen erweitern oder ihre Funktionalität ändern kann [GHJV95, S. 23–28].

Genau dieses Wissen ist essenziell für Entwickler, die die Software anpassen oder erweitern sollen, und muss in geeigneter Form festgehalten und verfügbar gemacht werden. Fehlt diese Information, müssen sich Entwickler aufwendig einarbeiten. Dabei ist die Gefahr hoch, dass verwendete Entwurfsmuster übersehen oder missinterpretiert werden. Das wiederum kann zu von der ursprünglichen Intention abweichenden Entwurfsänderungen führen und damit zu Design-Erosion [vGB02]. Die Dokumentation von Stellen, an denen Entwurfsmuster eingesetzt wurden, kann nachweislich das Verstehen des Entwurfs erleichtern und die Wahrscheinlichkeit für Entwurfsfehler reduzieren [PULPT02, Vok04]. Es ist also wichtig, eingesetzte Entwurfsmuster bei der Neu- und Weiterentwicklung von Software zu berücksichtigen.

2.1.1 Eigenschaften und Abgrenzung

Nach mehreren Jahren Forschung haben Buschmann et al. im letzten Band ihrer fünfteiligen Buchserie „Pattern-Oriented Software Architecture“ detailliert erklärt, was Softwareentwurfsmuster ausmacht und wie sie sich von anderen Konzepten unterscheiden [BHS07b, Kap. 1–3].

Eigenschaften eines Entwurfsmusters

Kontext, Problem, Lösung

Nach Buschmann et al. ist ein Softwareentwurfsmuster eine bewährte, in bestimmten Situationen (in einem bestimmten Kontext) anwendbare Entwurfslösung zu einem wiederkehrenden Entwurfsproblem. Zu dieser Lösung gehört immer auch das zugehörige Problem und der Kontext, in welchem die Lösung anwendbar ist. Diese drei Bestandteile allein machen allerdings noch kein Entwurfsmuster aus [BHS07b, S. 30 ff.].

Lösung ist praktisch bewährt

Eine weitere wichtige Eigenschaft von Entwurfsmustern ist die Qualität der Lösung. Die Lösung muss nachweislich Vorteile bringen (z.B. einen positiven Effekt auf Austauschbarkeit, Erweiterbarkeit, Performance oder Sicherheit haben) und vielfach erfolgreich angewandt worden sein, sich also praktisch bewährt haben. Als Gegenbeispiel wird das Singleton-Muster genannt [GHJV95], welches scheinbar mehr Probleme mit sich bringt als es löst[1] [BHS07b, S. 34, 35, 37].

allgemeine Lösung als Diagramm darstellbar

Die in einem Muster beschriebene Lösung ist allgemein und vielfach wiederverwendbar [BHS07b, S. 12]. Zum Beispiel sind Entwurfsmuster meist unabhängig von einer Programmiersprache und oft sogar unabhängig von einem Programmierparadigma, sind z.B. sowohl in prozeduralen als auch in objektorientierten Programmiersprachen anwendbar. Gleichzeitig ist die Lösung konkret genug, um

[1] Das Singleton-Muster führt eine zentrale Zugriffsstelle (statische Variable und Zugriffsmethode) auf die einzige Instanz einer Klasse und damit eine globale Variable ein, was eine starke Kopplung der Verwendungsstellen zu dieser Variable / Klasse mit sich bringt und als schlechte Praxis gilt.

anwendbar und sogar in Diagrammen darstellbar zu sein. Sie wird in Form einer Entwurfs- oder Implementierungsskizze beschrieben. Dabei kommt es nicht auf eine bestimmte Notation an. Ein (optionales) Diagramm dient zur Konkretisierung und Veranschaulichung eines Musters, repräsentiert aber nur eine von vielen im Muster beschriebenen Implementierungsvarianten und enthält nur beteiligte Rollen statt der konkreten Entwurfsteile.

mehrere, ggf. im Konflikt stehende Einflüsse

Zwecks Konkretisierung des zu lösenden Problems werden diverse Einflüsse (forces) – Anforderungen, Rahmenbedingungen und gewünschte Eigenschaften der Lösung – beschrieben, welche bei der Musteranwendung berücksichtigt werden müssen, um Fehlanwendungen des Musters zu vermieden. Solche Einflüsse können im Konflikt zueinander stehen, sodass eine geeignete Balance gefunden werden muss. Sie verdeutlichen die Intention des Musters, begründen die Lösung, schränken den Lösungsraum ein und präzisieren die Situationen, in denen ein Muster angewandt werden kann.

eindeutiger Name

Zwecks einfacher Kommunikation von Entwürfen und Entwurfsalternativen müssen Entwurfsmuster benannt und eindeutig identifiziert werden können, um nicht jedes Mal die gesamte Entwurfslösung erklären zu müssen. Darum haben Entwurfsmuster einen idealerweise einfachen, einprägsamen und eindeutigen Namen.

Dokumentation von Entwurfsmustern

Dokumentationsform

Entwurfsmuster sind in zahlreichen Veröffentlichungen dokumentiert worden. Die Musterbeschreibungen sind dabei unterschiedlich detailliert (eine halbe bis über 10 Seiten je Muster) und verwenden unterschiedliche Dokumentationsstrukturen (pattern form). Z.B. unterteilen Gamma et al. jede Musterbeschreibung in folgende Abschnitte [GHJV95, S. 6–7] (vgl. Abb. 1.1, S. 4): Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, Related Patterns. Alle Musterbeschreibungen umfassen laut Buschmann et al. jedoch mindestens die folgenden Aspekte eines Entwurfsmusters [BHS07b, S. 101 ff.]:

- Identification: Name and classification for identifying the pattern
- Context: Situation giving rise to a problem
- Problem: Set of forces repeatedly arising in the context
- Solution: Configuration to balance the forces
- Consequences: Consequences arising from application of the pattern

Sie können durch zusätzliche Informationen wie Anwendungsbeispiele, Entwurfsskizzen, Implementierungsbeispiele, Entwurfsvarianten und mehr ergänzt werden. Die Lösung kann neben der Struktur einer Entwurfslösung auch das zugehörige Laufzeitverhalten beschreiben.

Abgrenzung zu anderen, verwandten Konzepten

In den letzten über 20 Jahren haben Entwurfsmuster immer mehr an Beliebtheit und Verbreitung erfahren. Es haben sich jedoch auch einige Missverständnisse oder Fehlinterpretationen des Konzepts Entwurfsmuster eingeschlichen. Mit ihrer über

Jahre gesammelten Erfahrung auf diesem Gebiet gehen Buschmann et al. sehr detailliert auf zahlreiche solcher potentiellen Missverständnisse ein und grenzen Entwurfsmuster von anderen Konzepten ab [BHS07b, S. 8–23, 66–89]. Auf einige für diese Arbeit relevante Punkte gehe ich im Folgenden ein.

Entwurfsmängel als Muster: Anti-Patterns

Entwurfsmuster beschreiben gute, sich bewährte Entwurfslösungen. Neben diesen gibt es auch zahlreiche Beschreibungen von schlechten, jedoch verbreiteten „Entwurfslösungen“. Ihre Beschreibung ähnelt dabei der eines Entwurfsmusters. Sie wird häufig durch die Vorstellung eines Ansatzes zur Verbesserung des Entwurfs ergänzt. Bei diesen schlechten Entwurfslösungen handelt es sich nicht um Entwurfsmuster, sondern um verbreitete Entwurfsmängel (dysfunctional patterns), häufig auch Bad Patterns oder Anti-Patterns genannt [BHS07b, S. 31, 40–41].

Musterimplementierung

Ein Entwurfsmuster skizziert eine Lösungsidee, ein Konzept oder ein grobes Vorgehensschema. Fälschlicherweise wird eine konkrete Implementierung eines Musters häufig ebenfalls als Entwurfsmuster bezeichnet. Während ein Entwurfsmuster unzählige konkrete Implementierungen skizziert und Implementierungsdetails offen lässt, stellt eine Implementierung eines Musters nur genau eine, sehr spezielle, nicht wiederverwendbare Lösung dar. Sie ist abhängig von der Anwendungsstelle, der Programmiersprache, der umgebenden Entwurfsstruktur und vielem mehr. Das gleiche gilt für Musterimplementierungen in Frameworks (vollständig oder partiell) [BHS07b, S. 77–84]. Sie stellen nur eine konkrete, an das Framework angepasste Implementierung des Musters dar und sollen die Nutzung des Frameworks vereinfachen, ersetzen jedoch nicht das allgemeinere Konzept eines Entwurfsmusters. Aufgrund der nötigen Anpassungen an eine konkrete Situation kann es auch keine Referenz- oder Standard-Implementierung eines Musters geben [BHS07b, S. 88–89]. Es gibt nur Beispielimplementierungen eines Musters.

Schablonen und Beispielimplementierungen

Entwickler tendieren dazu, die in einer Musterbeschreibung als Diagramm dargestellte Entwurfsstruktur oder die in Code-Form bereitgestellten Implementierungsbeispiele für das Muster selbst zu halten [BHS07b, S. 14]. Das Muster wird als Schablone (Template) oder als eine sehr spezielle Konfiguration von Klassen angesehen und bei Musteranwendung nahezu unverändert in den Entwurf übernommen. Dieses Verständnis scheinen auch einige Werkzeugentwickler zu teilen, welche Quellcode-Schablonen oder exemplarische Entwurfsmodelle als „Muster“ speicherbar und durch Kopieren auf andere Stellen übertragbar machen (siehe Abschn. 9.3). Entwurfsmuster sind weder Code-Schablonen, noch bestimmte Klassenstrukturen. Ein Entwurfsmuster skizziert vielmehr beliebig viele Implementierungen oder Entwurfsstrukturen, deren Auswahl und Konkretisierung unter Berücksichtigung der vorliegenden Situation den Entwicklern überlassen wird.

> „A pattern is not a single destination with every detail finalized, ready to wear or ready for software. A pattern defines a space, not a point, a continuum of solutions, not a singularity. It should be seen as a sketch and inspiration rather than a blueprint or a requirement. Patterns define relationships between requirements and solutions. Without a good understanding of motivation, context, and problem forces, the solution aspect of a pattern makes little sense.“
> Frank Buschmann, Kevlin Henney, Douglas C. Schmidt [BHS07b, S. 85]

Aus diesem Grund ist es schwierig, eine geeignete Werkzeugunterstützung für Entwurfsmuster zu entwickeln. Es lassen sich zwar Klassen, Methodensignatu-

ren und Code-Schnipsel mit TODO-Kommentaren generieren, das umfasst jedoch nicht die gesamte Lösungsidee hinter einem Entwurfsmuster und bringt kaum Vorteile. Buschmann et al. formulieren es wie folgt (vgl. Zitat auf S. 10).

> „The use of patterns cannot be fully tool-based or automated.“
> Frank Buschmann, Kevlin Henney, Douglas C. Schmidt [BHS07b, S. 16]

> „Like code generators that just spit out class names, operation signatures, and matching curly braces, this approach roundly misses the point of both patterns and formalisms that aspire to genericity. Typing speed is not the bottleneck in software development – understanding is.“
> Frank Buschmann, Kevlin Henney, Douglas C. Schmidt [BHS07b, S. 86]

Komponenten

Da Entwurfsmuster nicht nur eine, sondern eine Vielzahl an konkreten Implementierungsvarianten beschreiben, lassen sie sich nicht wie Komponenten als Ganzes wiederverwenden. Sie haben keine klar definierten Schnittstellen, kapseln nicht ihre Implementierung und lassen sich nicht zu einem ganzen System zusammenstecken (siehe folgendes Zitat).

> „Patterns are not components: they are not usable out-of-the-box, commodity items that can just placed [sic] into a system in a preplanned way. Because many software patterns offer a diagram of code components in various roles, there is a temptation to view them as schemas or templates – just fill out the blanks and you are ready to run!“
> Frank Buschmann, Kevlin Henney, Douglas C. Schmidt [BHS07b, S. 17]

Entwurfsentscheidungen

Entwurfsmuster beschreiben gute Entwurfslösungen, weswegen sie häufig zur Dokumentation eines Systementwurfs herangezogen werden. Entwurfsmuster sind jedoch nach meinem Verständnis keine Entwurfsentscheidungen (design decisions). Ein Entwurfsmuster beschreibt eine Vielzahl möglicher Entwurfsentscheidungen. Das Fällen der Entscheidungen wird aber dem Anwender eines Musters überlassen. Dazu gehört z.B. die Auswahl eines bestimmten Entwurfsmusters und einer bestimmten Implementierungsvariante oder das Bevorzugen eines von mehreren in Konflikt stehenden Qualitätsmerkmalen (z.B. Wartbarkeit vor Effizienz). Die Dokumentation der getroffenen Entwurfsentscheidungen sowie der eingesetzten Entwurfsmuster samt zugehöriger Intention fördert das Verständnis des Softwaresystems, hilft bei der Weiterentwicklung [Rie11] und reduziert nachweislich das Risiko für Design-Erosion [PULPT02].

2.1.2 Arten und Sammlungen von Softwareentwurfsmustern

Mit zunehmender Beliebtheit und Verbreitung von Entwurfsmustern hat sich auch eine Fülle von Entwurfsmusterbeschreibungen und Sammlungen solcher Beschreibungen (pattern collections) entwickelt. Es gibt Sammlungen mit allgemein verwendbaren Entwurfsmustern wie die der Gang-of-Four [GHJV95], Muster für bestimmte Domänen oder Anwendungsgebiete wie die Entwurfsmuster für Enterprise-Architekturen [Fow02] und Sammlungen von Mustern mit einem bestimmten Abstraktions- oder Granularitätslevel wie Architekturmuster, Entwurfsmuster und

Idiome[2]. Henninger und Corrêa geben einen Überblick zu den entstandenen Mustersammlungen [HC07]. Buschmann et al. beleuchten die Eigenschaften solcher Sammlungen [BHS07b, Kap. 8].

Auf Konferenzen und in der Pattern Community veröffentlichte Entwurfsmuster wurden mit zunehmender Systematik in Büchern zusammengefasst und ggf. auf Basis neuer Erkenntnisse überarbeitet. So sind insb. die zwei 5-teiligen Buchreihen „Pattern Languages of Programm Design“ [CS95, VKC96, MRB97, HFR99, MVN06] und „Pattern-Oriented Software Architecture“ [BMR+96, SSRB00, KJ04, BHS07a, BHS07b] entstanden. Letztere schließt mit einem Buch über die bis dahin gewonnenen Erkenntnisse über Entwurfsmuster ab, zu welchen neben ihren Eigenschaften, ihrer Dokumentation und ihrer Verwendung auch Mustersprachen (Pattern Languages) gehören [BHS07b]. Die Buchreihe „Pattern-Oriented Software Architecture“ gehört außerdem zu einer größeren Buchserie, der *Wiley Series in Software Design Patterns*, zu welcher diverse weitere Sammlungen von Entwurfsmustern in Buchform gehören. In einem Entwurfsmuster-Almanach wurden über 700 Softwaremuster zusammengetragen [Ris00].

Zu den Anwendungsgebieten, zu denen Entwurfsmustersammlungen entstanden sind, gehören z.B. folgende:

- Enterprise-Anwendungen [ACM13, Fow02, HW03, AN04]
- Web Services [Dai11]
- Architektur, Modularität, Variabilität [Völ06, Völ09, Kno12]
- Ressourcen-Management [KJ04]
- Sicherheit (security) [Fer13]
- Nebenläufige / parallele Programmierung [Lea99, SSRB00, MSM04, OA10]
- Verteilte Systeme [BHS07a]
- Echtzeitsysteme [KC05]
- Fehler-tolerante Systeme [Han07]
- Eingebettete Systeme [KCC04, DBHT12, BBB+12]
- Kombination objektorientierter mit funktionaler Programmierung [Küh99]
- Objektorientierte Domänen-Analyse [Fow96]
- Domänen-spezifische Sprachen (DSLs) [Par10]
- Generative und modellgetriebene Softwareentwicklung [Völ03, VB04, AN04]

Zusätzlich zu den weit verbreiteten Entwurfs- und Architekturmustern werden auch Idiome beschrieben, eine Art Implementierungsmuster oder von einer Programmiersprache abhängige Entwurfsmuster, z.B. für SmallTalk oder Java [Bec96, GM05, Bec07] (manchmal handelt es sich eher um eine Programmierkonvention als um ein Entwurfsmuster mit einer Problembeschreibung, einem Kontext und einer Lösung [BHS07b, S. 215 ff.]).

[2] Inzwischen wird von einer disjunkten Unterscheidung von Architekturmustern, Entwurfsmustern und Idiomen abgeraten [BHS07b, Kap. 8.4]. *Entwurfsmuster* wird als Oberbegriff für alle diese Muster verwendet. Der Fokus eines Musters kann auf einem der genannten Level liegen. Ein Muster kann jedoch auch mehrere Abstraktions- / Granularitätslevel gleichzeitig umspannen.

Smith hat außerdem sogenannte Elementarmuster (elemental design patterns) definiert, eine Art atomare, objektorientierte Minimuster, auf deren Basis sich komplexere Entwurfsmuster wie die der Gang-of-Four spezifizieren lassen [Smi12].

Mit zunehmenden Erkenntnissen und Anwendungsgebieten entstehen stetig neue Entwurfsmuster. Vlissides, einer der Gang-of-Four, beschreibt wie ein Entwurfsmuster entsteht. Die Erläuterungen sollen Leser in die Lage versetzen, selbst Entwurfsmuster zu entdecken und sie zwecks Wiederverwendung für andere Entwickler und Architekten zu beschreiben [Vli98b].

2.1.3 Verwendete Begriffe

In meiner Arbeit verwende ich viele verwandte Begriffe wie Entwurfsmuster, Musteranwendung, Musterimplementierung oder Anwendungsstelle. Die Unterschiede und Zusammenhänge dieser Begriffe sind nicht offensichtlich. Aus diesem Grund stelle ich die für diese Arbeit wichtigen Begriffe rund um Softwareentwurfsmuster in einer Ontologie in Zusammenhang. Meine Interpretation der Begriffe ist in der Abb. 2.1 in Anlehnung an die Notation von UML-Klassendiagrammen dargestellt. Die wichtigsten Begriffe sind fett hervorgehoben.

Muster, Musterimplementierung, Musteranwendung

Ich unterscheide insbesondere ein Entwurfsmuster – das allgemeine Konzept – von einer Implementierung des Musters – konkrete, Anwendungsfall-spezifische Lösung im Code oder Modell. Eine Musteranwendung verstehe ich als die Tätigkeit, welche ein Entwurfsmuster in eine Musterimplementierung überführt.

Entwurfslösung

Ein Entwurfsmuster hat einen Namen, einen Anwendungsfall (d.h. eine Art von Situationen, in welchen das Muster anwendbar ist), eine Intention, also eine gewünschte, zu erhaltende Entwurfseigenschaft (z.B. Entkopplung der Klassen Observer und Subject des Observer-Musters oder die Erweiterbarkeit von Beobachtern, ohne die Subject-Implementierung ändern zu müssen, siehe Abschn. 1.1.1), Konsequenzen (positive wie negative) und (mindestens) eine Entwurfslösung. Nach meinem Verständnis besteht eine Entwurfslösung aus einer bestimmten Entwurfsstruktur mit zugehörigem Verhalten. Struktur und Verhalten werden mit Hilfe von Musterrollen beschrieben. Musterrollen sind benannte Platzhalter für Entwurfselemente einer Musterimplementierung. Struktur, Verhalten und Rollen werden insbesondere in Entwurfsskizzen verwendet (vgl. Abb. 1.1, S. 4).

Anwendungsstelle, Rollenzuordnung

Eine Anwendungsstelle definierte ich als alle Teile eines Softwareentwurfs, welche eine Musterrolle einnehmen oder einnehmen sollen. Vor Abschluss einer Musteranwendung beschreibt eine Anwendungsstelle den Ort einer geplanten Musteranwendung. Nach Musteranwendung beschreibt eine Anwendungsstelle alle wesentlichen Teilnehmer einer Musterimplementierung. Charakterisiert wird eine Anwendungsstelle durch die Zuordnung der Musterrollen eines Entwurfsmusters zu den sie einnehmenden Teilen eines objektorientierten Softwareentwurfs, also den Entwurfselementen wie z.B. Typen (Klassen), Operationen (Methoden), Assoziationen, etc. Eine Rollenzuordnung ordnet jeder Musterrolle beliebig viele Entwurfselemente zu. Dabei kann einer Entwurfselement auch mehrere Musterrollen (ggf. verschiedener Muster) einnehmen.

Implementierungsvariante

Eine Entwurfslösung ist so allgemein, dass sie mehrere Implementierungsvarianten repräsentiert. Eine Musterimplementierung entspricht genau einer dieser Implementierungsvarianten. Eine Musterimplementierung kann sich sowohl im Code

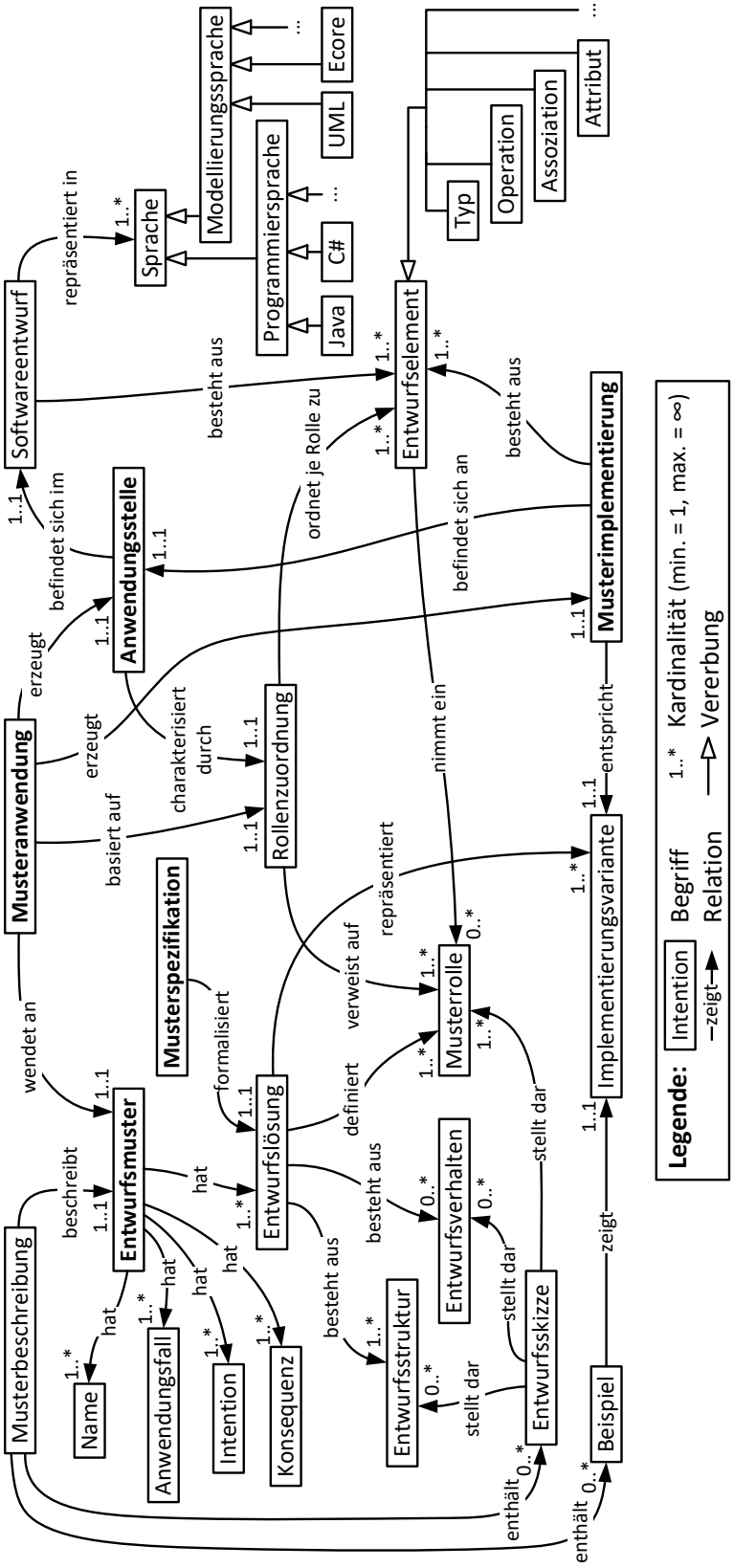


Abbildung 2.1: Ontologie der Softwareentwurfsmuster

als auch in einem Entwurfsmodell befinden, abhängig von der Sprache, in welcher ein Softwareentwurf vorliegt.

Musterrealisierung

Neben dem Begriff Musterimplementierung verwende ich den Begriff Musterrealisierung. Die Begriffe sind für mich synonym zueinander.

Musterspezifikation

In folgenden Kapiteln spreche ich u.a. von Musterspezifikationen (siehe Kap. 3). In meiner Arbeit beschränken sich Spezifikationen auf die Entwurfslösung, insbesondere auf die Struktur und das Verhalten der Entwurfslösung. Weitere, erst später in dieser Arbeit eingeführte Begriffe sind zusammen mit einer kurzen Erläuterung im Glossar zu finden (S. 365 ff.).

2.2 Modellgetriebene Softwareentwicklung – MDSD

Bei der modellgetriebenen Softwareentwicklung (model-driven software development, MDSD) handelt es sich um einen generativen Ansatz zur Softwareentwicklung. Ein Softwareprodukt wird durch Erstellen eines formalen, abstrakten Modells des Softwareprodukts entworfen. Anschließend wird aus diesem Softwareentwurfsmodell ausführbarer Programmcode generiert. Wenn nötig, wird der generierte durch selbst geschriebenen Quellcode ergänzt.

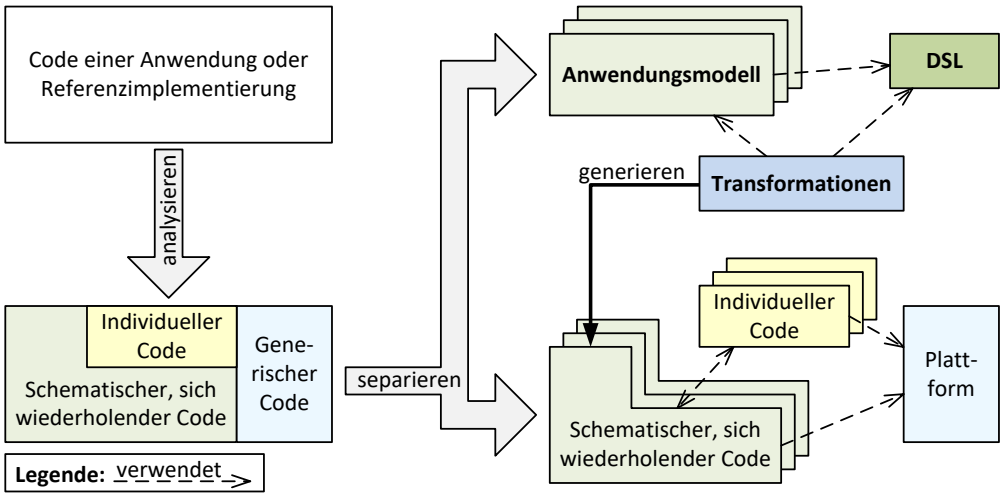


Abbildung 2.2: Kernidee hinter modellgetriebener Softwareentwicklung (MDSD) in Anlehnung an Stahl & Völter [SV06, S. 15]

Bei diesem Ansatz rückt das Modell eines zu entwickelnden Softwaresystems in den Mittelpunkt des Softwareentwicklungsprozesses und wird zum wichtigsten Artefakt im Entwicklungsprozess. Die wesentlichen Änderungen werden auf Modell-Ebene durchgeführt. Man spricht dann auch von modell-zentrierter (model-centric) Softwareentwicklung [Bro04] (entspricht dem Fall d) in Abb. 2.3). Im Gegensatz zu modellbasierter Softwareentwicklung, wo ein Softwareentwurfsmodell als Implementierungsvorgabe oder Dokumentation dient und Quellcode im Mittelpunkt des Entwicklungsprozesses liegt, spielt Quellcode bei der modellgetriebenen Softwareentwicklung eine untergeordnete Rolle. Er wird nach jeder Änderung erneut generiert. Manuell geschriebener Code – falls vorhanden – bleibt dabei erhalten (siehe Abb. 2.2).

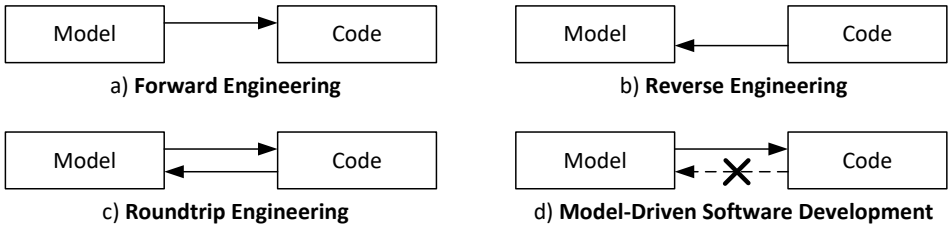


Abbildung 2.3: Einordnung modellgetriebener Softwareentwicklung (MDSD) laut Stahl & Völter [SV06, S. 74] (Begriffe von Chikofsky & Cross [CC90])

Durch das Generieren von Quellcode nach jeder Änderung wird ein Reverse Engineering[3] des Entwurfs aus ggf. manuell modifiziertem Quellcode unnötig (siehe Abb. 2.3). Das Softwareentwurfsmodell dient nicht nur als Vorgabe für den zu generierenden Code, sondern zusätzlich als abstrakte, kompakte und stets aktuelle Dokumentation des Softwareentwurfs.

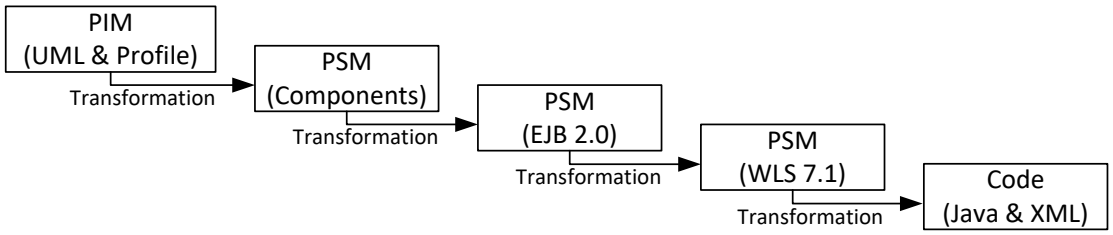


Abbildung 2.4: Von Plattform-unabhängigem Modell (PIM) über Plattform-spezifische Modelle (PSM) zum Code (Bsp. von Stahl & Völter [SV06, S. 17])

Die Object Management Group (OMG) definiert unter dem Begriff Model-Driven Architecture (MDA) einen standardisierten Prozess zur modellgetriebenen Entwicklung von Softwaresystemen [OMG03]. Dabei wird ein Plattform-unabhängiges Modell (platform-independent model, PIM) eines Softwaresystems schrittweise und meist mit Hilfe automatischer Modelltransformationen in ein Plattform-spezifisches Modell (platform-specific model, PSM) und dieses schließlich in Quellcode übersetzt (siehe Bsp. in Abb. 2.4). Der wesentliche Unterschied zum allgemeineren Konzept der modellgetriebenen Softwareentwicklung ist der Fokus auf Standards wie die Sprachen UML, OCL [OMG11a, OMG11b] und QVT [OMG11d]. Bei modellgetriebener Softwareentwicklung werden häufig auch andere Sprachen als die UML, insbesondere Domänen-spezifische Sprachen (domain-specific languages, DSLs) zur Modellierung eingesetzt.

[3] Chikofsky und Cross definieren den Begriff *Reverse Engineering* als den Prozess der Analyse eines Softwaresystems (a) zur Identifikation der Systemkomponenten und ihrer Beziehungen untereinander und (b) des Erzeugens einer Systemrepräsentation auf einem höheren als dem vorliegenden Abstraktionslevel (z.B. Klassendiagramm statt Java-Quellcode oder Komponentenarchitektur statt Klassendiagramme) [CC90].

2.3 Modelltransformationen

Bei der modellgetriebenen Softwareentwicklung werden Modelle mit Hilfe von Modelltransformationen in ausführbaren Code oder interpretierbare (detailliertere und ausführbare) Modelle übersetzt. Damit sind neben den Modellen auch Modelltransformationen wesentlicher Bestandteil der modellgetriebenen Softwareentwicklung (siehe Abb. 2.2 und 2.4).

2.3.1 Modelle

Zur Klärung des Konzepts hinter Modelltransformationen müssen wir zunächst den Modellbegriff präzisieren. Laut Stachowiak ist ein Modell eine partielle Repräsentation eines Originals zu einem bestimmten Zweck und wird durch folgende drei wesentliche Merkmale geprägt [Sta73, Kap. 2.1.1].

Abbildung / Repräsentation: „Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können."

Verkürzung (Reduktion): „Modelle erfassen im allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und / oder Modellbenutzern relevant erscheinen."

Pragmatismus: „Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte – erkennende und / oder handelnde, modellbenutzende – Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen."

Hesse und Mayr präzisieren den Modellbegriff im Kontext der Softwaretechnik noch weiter [HM08]. Sie sprechen von den Merkmalen Abbildung, Reduktion und Pragmatismus und gehen auf weitere Modelleigenschaften und Modellarten in der Softwaretechnik ein. Unter anderem gehen sie auf deskriptive und präskriptive Modelle ein (Modelle als Nach- oder Vorbilder). Zu den präskriptiven Modellen (Vorbildern) gehören insbesondere die Entwurfsmodelle in der modellgetriebenen Softwareentwicklung, welche zur Generierung detaillierterer Modelle oder von Quellcode verwendet werden.

Modelltransformationen lassen sich auf sehr unterschiedlichen Arten von Modellen anwenden. In der Softwaretechnik werden insbesondere objektorientierte Modelle eingesetzt. Diese können sowohl eine grafische als auch eine textuelle Repräsentation haben. Neben Allzweck-Modellierungssprachen wie der UML können auch Domänen-spezifischen Sprachen (DSLs) zur Modellierung verwendet werden. Auch Programmcode kann als Modell angesehen werden (es ist eine abstrakte Repräsentation von Maschinencode).

2.3.2 Modelltransformationsansätze

Im Allgemeinen behandeln Modelltransformationen Modelle (im Sinne von Repräsentationen auf einer höheren Abstraktionsebene als Programmcode) oder Mo-

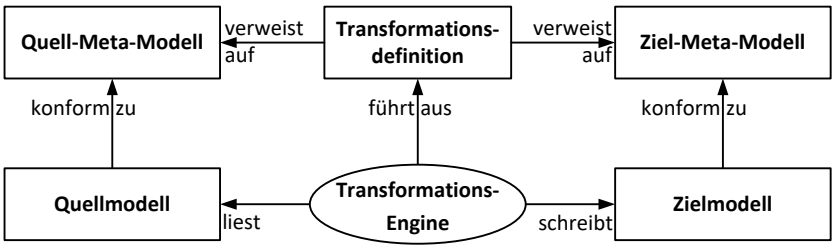


Abbildung 2.5: Allgemeiner Aufbau von Modelltransformationen nach Czarnecki & Helsen [CH06]

delle und Programme [CH06]. Gebräuchlich ist u.a. die Unterscheidung zwischen Modell-zu-Modell und Modell-zu-Text-Transformationen.

Den Grundaufbau von Modelltransformationen stellen Czarnecki und Helsen wie in der Abb. 2.5 dar [CH06]. Eine Transformation liest ein Quellmodell und erzeugt ein Zielmodell, wobei Quell- und Zielmodell identisch sein oder jeweils aus mehreren Modellen bestehen können. Quell- und Zielmodelle entsprechen ihren Meta-Modellen [Küh06b, Küh06a]. Ein Meta-Modell definiert die abstrakte Syntax (und statische Semantik) einer Modellierungssprache – die Grundbausteine und Struktur einer Sprache – unabhängig von ihrer Notation (der konkreten Syntax). Eine (Modellierungs-)Sprache besteht aus einem Meta-Modell, einer Notation und einer (dynamischen) Semantik [SV06, Kap. 4.1.1]. Während eine Transformation basierend auf den Meta-Modellen definiert wird, erfolgt ihre Ausführung mit Hilfe einer Transformations-Engine auf den konkreten Modellen.

Meta-Modelle & Sprachen

Czarnecki und Helsen geben einen detaillierten Überblick existierender Modelltransformationsansätze und stellen die Gemeinsamkeiten und Unterschiede anhand von Feature-Diagrammen dar [CH06]. Sie unterscheiden unter anderem zwischen folgenden Eigenschaften und Merkmalen, welche für meine Arbeit von besonderer Bedeutung sind:

Multi-Direktionalität: Einige Transformationen lassen sich in mehr als nur eine Richtung ausführen. Es werden unidirektionale und bidirektionale Transformationen unterschieden. Bidirektionale Transformationen können sowohl in Richtung des Zielmodells als auch in Richtung des Quellmodells ausgeführt werden (z.B. bei Hin- und Rückübersetzungen).

Domänensprache: Es wird zwischen Transformationen unterschieden, bei denen Quell- und Zielmodell auf demselben Meta-Modell basieren und solchen, bei denen Quell- und Ziel-Meta-Modell unterschiedlich sind. Im ersten Fall wird von endogenen Transformationen (endogenous transformations / rephrasings), im letzteren von exogenen Transformationen oder Übersetzungen (exogenous transformations / translations) gesprochen.

Beziehung zwischen Quell- und Zielmodell: Hier werden Transformationen unterschieden, welche (a) ein neues Zielmodell zu einem existierenden Quellmodell erzeugen (new target transformations), (b) ein existierendes Zielmodell anhand eines Quellmodells aktualisieren (update transformations) und (c) ein Quellmodell modifizieren (in-place transformations), wo das Quellmodell also dem Zielmodell entspricht. Der Fall (c) lässt sich mit (b) kombinieren.

2.3.3 Graphtransformationen

Objektorientierte Modelle können als Graphen bestehend aus Knoten und Kanten angesehen werden. Graphen lassen sich mathematisch präzise formalisieren und die Graphentheorie gibt Aufschluss über ihre Eigenschaften. Es liegt also nahe, Graphen, Graphtransformationssysteme und Graphgrammatiken zur Beschreibung von Modelltransformationen zu nutzen [Roz97, EEPT06].

Graphgrammatiken und Graphtransformationen beschreiben schrittweise Modifikationen von Graphen. Einzelne Modifikationsschritte – Graphtransformationsschritte – werden durch Graphtransformationsregeln (Graphproduktionen oder Produktionsregeln, im Englischen auch graph rewriting rules) beschrieben. Graphtransformationsregeln entsprechen prinzipiell den Ersetzungs- bzw. Produktionsregeln einer kontext-freien Grammatik [Sip97, Kap. 2], sind jedoch nicht auf Zeichenketten, sondern auf Graphen definiert. Mehrere Graphtransformationsregeln werden in einem Graphtransformationssystem zusammengefasst. Eine Graphgrammatik besteht aus einem Graphtransformationssystem und einem Startgraphen und definiert eine Sprache bestehend aus einer Menge von Graphen, welche sich durch Anwendung der Graphtransformationsregeln aus dem Startgraphen erzeugen lassen [EEPT06, S. 38, 183].

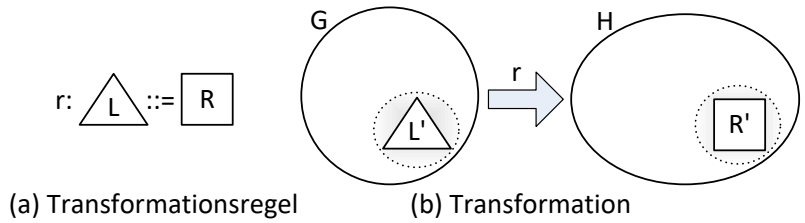


Abbildung 2.6: Aufbau einer Graphtransformation in Anlehnung an Ehrig et al. [EEPT06, S. 6]

Eine Graphtransformationregel r wird durch einen Graphen L und einen Graphen R beschrieben (siehe Abb. 2.6 (a)). Sie drückt aus, dass ein Graph L durch einen Graphen R ersetzt werden soll. Wird eine Regel r auf einen Graphen G angewandt (siehe Abb. 2.6 (b)), so wird ein zu L identischer Teilgraph L' in G bestimmt und durch einen zu R identischen Graphen R' ersetzt. Die Zuordnung aller Knoten und Kanten aus L zu den Knoten und Kanten in L' wird *Matching* genannt. Das Ergebnis der Graphtransformation ist ein neuer Graph H.

Bei einer Graphtransformation wird auch der Kontext der Graphen L' und R' berücksichtigt. Das sind Kanten in G, welche Knoten in L' mit Knoten in $G \setminus L'$ verbinden bzw. Kanten in H, welche Knoten in R' mit Knoten in $H \setminus R'$ verbinden (in Abb. 2.6 (b) durch grauen, gepunkteten Kreis angedeutet). Die algebraische Formalisierung von Graphtransformationsregeln[4] garantiert, dass nach Anwendung einer Regel r auf dem Graphen G in H keine losen Kanten (Kanten ohne Ziel- oder Quellknoten) entstehen.

[4]Eine Regelanwendung wird durch sogenannte Push-outs – Schritte zum „Verkleben“ der beteiligten Graphen – formalisiert. Es wird zwischen Single-Push-out- (SPO) und Double-Push-out-Ansätzen (DPO) unterschieden [Roz97, EEPT06], wobei diese Arbeit von SPO ausgeht.

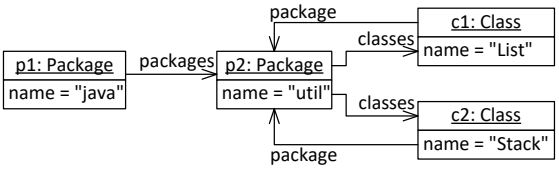


Abbildung 2.7: Beispiel für ein Modell als Objektdiagramm

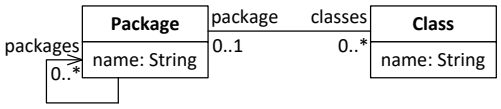


Abbildung 2.8: Meta-Modell für das Modell aus Abb. 2.7

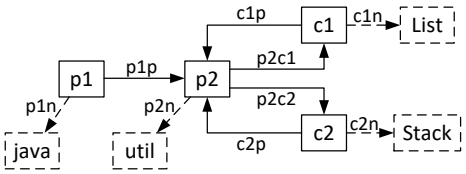


Abbildung 2.9: Attributierter Graph G

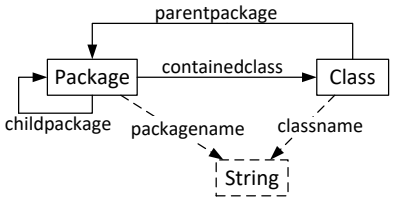


Abbildung 2.10: Typgraph T für Graphen G aus Abb. 2.9

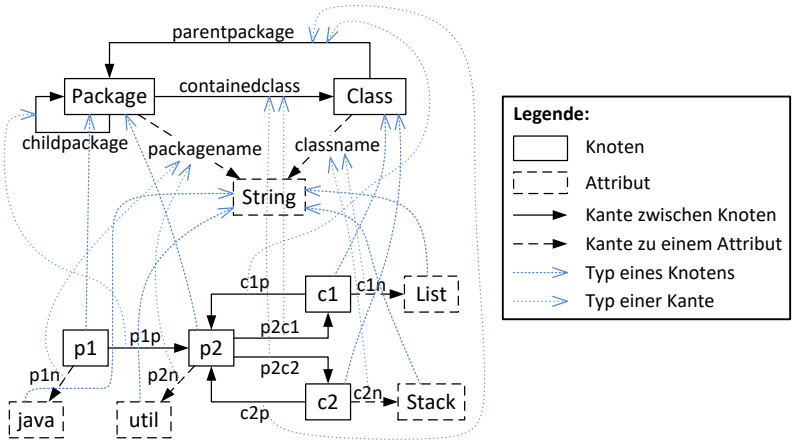


Abbildung 2.11: Typisierter, attributierter Graph

Zur Beschreibung objektorientierter Strukturen werden sogenannte typisierte (oder getypte), attributierte Graphen (typed, attributed graphs) mit Vererbung verwendet [EEPT06, Kap. 8, 13]. So ein Graph besteht aus zwei anderen Graphen, einem Graphen G und einem zugehörigen Typgraphen T. Jedem Knoten und jeder Kante in G wird ein Knoten bzw. eine Kante in T zugewiesen. Außerdem können die Knoten und Kanten in G und T beschriftet und mit Attributen versehen werden. In Abb. 2.9 und 2.10 sind zwei beschriftete, attributierte Graphen dargestellt. Durch Zuordnung von jedem Knoten in G zu einem Knoten in T und von jeder Kante in G zu einer Kante in T kann eine Typabbildung für G definiert werden. T wird damit zum Typgraphen für G. Zusammen mit der Typbeziehung bilden die beiden Graphen einen typisierten, attributierten Graphen wie in Abb. 2.11.

Während G eine Objektstruktur beschreibt (Objekte, ihre Eigenschaften und ihre Verknüpfungen untereinander), beschreibt T die zugehörigen Typen (Klassen, Attribute und Assoziationen). Zusammen mit der Typbeziehung entspricht G im Prinzip einer Objektstruktur in einem UML-Objektdiagramm (siehe Abb. 2.7), T einem zugehörigen Meta-Modell in einem UML-Klassendiagramm (siehe Abb. 2.8).

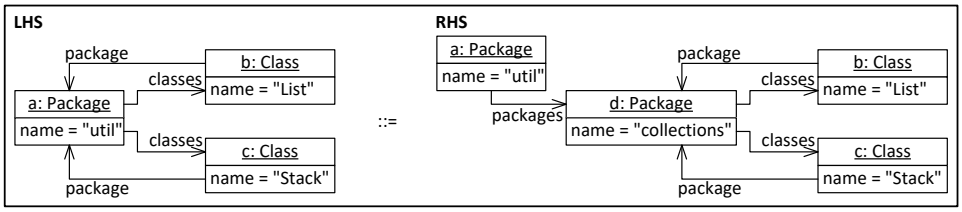


Abbildung 2.12: Eine Graphtransformationsregel

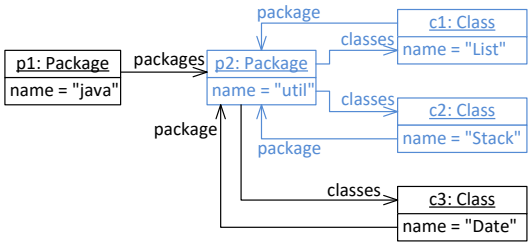


Abbildung 2.13: Ein Startgraph für eine Graphtransformation

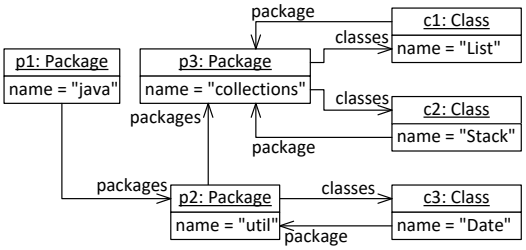


Abbildung 2.14: Ein Ergebnis nach einer Graphtransformation

Ein Beispiel für eine Graphtransformationsregel in vereinfachter an Objektdiagramme angelehnter Notation ist in der Abb. 2.12 dargestellt. Die Regel besteht aus einer linken und einer rechten Seite – der left-hand side (LHS) und der right-hand side (RHS). Die linke Regelseite beschreibt einen zu transformierenden Graphen L, die rechte einen aus L zu erzeugenden Graphen R (vgl. Abb. 2.6, S. 31). Der Typgraph zu dieser Regel ist in der Abb. 2.8 dargestellt.

Eine Graphtransformationsregel wird auf einem Wirtsgraphen (host graph) angewandt. Ein Beispiel für einen Wirtsgraphen ist in der Abb. 2.13 dargestellt. Die linke Seite L einer Graphtransformationsregel stellt eine Vorbedingung für die Regelanwendugn dar: Es muss ein Matching von L zu einem Teilgraphen L' des Wirtsgraphen geben, damit die Regel angewandt werden kann. So ein Teilgraph L' ist in der Abb. 2.13 blau hervorgehoben. Die rechte Seite R einer Graphtransformationsregel stellt eine Nachbedingung dar: Nach Anwendung der Regel muss der Teilgraph L' durch einen Teilgraphen R' ersetzt worden sein, sodass es ein Matching von R zu R' gibt. Das Ergebnis der Anwendung der Regel aus Abb. 2.12 auf den Wirtsgraphen aus Abb. 2.13 ist in der Abb. 2.14 dargestellt.

2.4 Modellgetriebene Entwicklung mit Klassen- und Story-Diagrammen

Bei der modellgetriebenen Softwareentwicklung mit Klassen- und Story-Diagrammen [FNT98, NNZ00, Zün01] werden UML-Klassendiagramme zur Modellierung der Klassen-Struktur eines zu entwickelnden Softwaresystems und sogenannte Story-Diagramme [FNTZ00, HRvD+11, vDHH+12, NZJ13] zur Modellierung des Verhaltens des Softwaresystems verwendet.

Story-Diagramme kombinieren eine imperative Spezifikation von Kontrollfluss basierend auf UML-Aktivitätendiagrammen mit der deklarativen Spezifikation

von Objektstrukturmodifikationen beschrieben durch Graphtransformationsregeln. Während UML-Aktivitätendiagramme in einem Aktivitätenknoten eine menschenlesbare, aber nicht ausführbare, textuelle Beschreibung einer Operation enthalten, enthält ein Aktivitätenknoten eines Story-Diagramms eine formal spezifizierte, ausführbare Graphtransformationsregel, ein sogenanntes Story Pattern. Story Patterns und Story-Diagramme besitzen eine wohldefinierte Ausführungssemantik [Zün01, vDHH+12]. Nach Czarnecki und Helsen [CH06] lassen sich Story-Diagramme als eine unidirektionale, endogene, destruktive (u.a. Löschungen erlaubende) In-Place-Update-Modelltransformationssprache klassifizieren (siehe Abschn. 2.3.2).

Aus den Klassen- und Story-Diagramm-Modellen eines modellierten Softwaresystems kann objektorientierter Code generiert werden, z.B. Java-Code [FNT98, GSR05, NZJ13]. Dabei werden nicht nur Methodendeklarationen, sondern auch vollständige Implementierungen von Methodenrümpfen generiert. Der generierte Code ist somit vollständig ausführbar. Alternativ dazu lassen sich die Modelle mit Hilfe eines Interpreters direkt ausführen [GHS09].

Die Kombination von Klassen- und Story-Diagrammen soll die modellgetriebene Softwareentwicklung unterstützen und wurde u.a. in der Entwicklungsumgebung Fujaba[5] [FNT98, NNZ00, Zün01, NSW+02] eingesetzt. Fujaba basiert auf Vorarbeiten an Graphtransformationen und ihrem Einsatz in Entwicklungsumgebungen [ELN+92, Sch86, Eng86], zu denen insb. die Entwicklung einer auf Graphgrammatiken basierenden Sprache und Entwicklungsumgebung namens PROGRESS (PROgrammed Graph REwriting SyStems) [SWZ95, Sch91] gehört.

2.4.1 Story Patterns

Story Patterns sind eine spezielle Form von typisierten, attributierten Graphtransformationsregeln. Ihr Typgraph wird als UML-Klassendiagramm angegeben. Ein Story Pattern fasst die linke und rechte Seite einer Graphtransformationsregel in nur einem Graphen zusammen und setzt Markierungen ein, um die linke und rechte Regelseite voneinander zu unterscheiden. Die Notation von Story Patterns ist an UML-Objektdiagramme angelehnt.

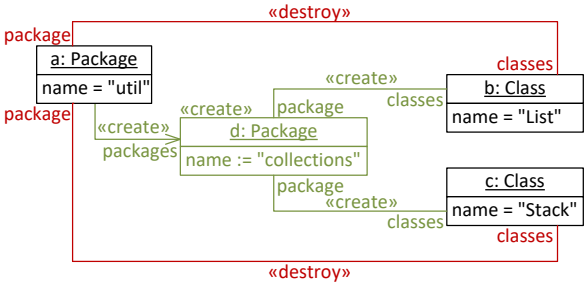


Abbildung 2.15: Ein Story Pattern

LHS und RHS in einem

Die Graphtransformationsregel aus der Abb. 2.12 lässt sich als Story Pattern wie in der Abb. 2.15 spezifizieren. Als Typgraph dient hierbei das Klassendiagramm aus Abb. 2.8 (S. 32). Bei Anwendung der Regel zu erzeugende Knoten

[5] Fujaba (From UML to Java and back again): http://www.fujaba.de

und Kanten werden mit dem Stereotyp «create» markiert und grün dargestellt, zu entfernende Knoten und Kanten werden mit dem Stereotyp «destroy» markiert und rot dargestellt. Alle anderen Knoten und Kanten bleiben bei Regelanwendung unverändert und werden schwarz dargestellt. Die linke Regelseite (LHS) besteht somit aus allen schwarz und rot dargestellten Knoten und Kanten, die rechte Regelseite (RHS) aus allen schwarz und grün dargestellten Knoten und Kanten (vgl. Abb. 2.12). Neben der Erzeugung von Knoten und Kanten gehören auch Zuweisungen zu der rechten Regelseite. Zum Beispiel wird der Variable name in Abb. 2.15 der Wert "collections" zugewiesen. Zuweisungen werden durch den Operator := gekennzeichnet und ebenfalls grün dargestellt.

Kanten werden bei Story Patterns über Assoziationen typisiert. Zum Beispiel entspricht der Typ der Verknüpfung zwischen den Knoten a und b in der Abb. 2.15 der Assoziation zwischen den Klassen Package und Class in der Abb. 2.8 (S. 32). Darum werden abweichend von Abb. 2.12 nicht zwei unidirektionale Verknüpfungen, sondern nur eine bidirektionale Verknüpfung zwischen a und b dargestellt.

Isomorphes Matching

Eine Besonderheit bei Story Patterns ist, dass sie ein isomorphes Matching der linken Regelseite auf einen Teilgraphen des Wirtsgraphen fordern. Mit speziellen Sprachkonstrukten (sogenannten maybe clauses / maybe links) kann diese Forderung gelockert werden [Zün01, vDHH+12]. Dazu kann für ein Paar von Knoten der Regel (oder für mehrere Paare) angegeben werden, dass sie auf denselben Knoten im Wirtsgraphen gematcht werden dürfen. Kann die linke Regelseite eines Story Patterns erfolgreich gematcht werden, erfolgt zuerst die Anwendung der Lösch- und anschließend der Erzeugungsoperationen[6]. Ist ein vollständiges, isomorphes Matching nicht möglich, wird die Regel nicht angewandt.

Negative Anwendungsbedingungen

Die Anwendung einer Regel kann noch weiter eingeschränkt werden, indem negative Anwendungsbedingungen (negative application conditions, NACs) spezifiziert werden. Eine negative Anwendungsbedingung ist ein Teilgraph der linken Regelseite. Eine Regel mit negativen Anwendungsbedingungen wird nur dann angewandt, wenn es für alle negativen Anwendungsbedingungen kein Matching gibt. Die Knoten und Kanten einer negativen Anwendungsbedingung werden durchgestrichen dargestellt [Zün01, vDHH+12].

Variablen

Die Knoten eines Story Patterns werden Objektvariablen (object variables) genannt, die Kanten werden Verknüpfungsvariablen (link variables) genannt. Beim Matching werden Objektvariablen konkreten Objekten eines Wirtsgraphen (eines zu transformierenden Modells) zugeordnet. Verknüpfungsvariablen repräsentieren Verknüpfungen zwischen Objekten im Wirtsgraphen. Beim Matching wird die Existenz der geforderten Verknüpfungen im Wirtsgraphen geprüft.

2.4.2 Story-Diagramme

Story-Diagramme sind eine spezielle Form von UML-Aktivitätendiagrammen. Sie betten in ihren Aktivitätenknoten Graphtransformationsregeln in Form von Story Patterns ein und sind dank ihrer formal definierten Semantik ausführbar [Zün01, FNTZ00, vDHH+12, NZJ13]. Mit Hilfe von Story-Diagrammen lassen sich uni-

[6] Die Anwendung der Regel erfolgt nach dem Single-Push-out-Ansatz [Zün01, Roz97, EEPT06].

direktionale, endogene[7], In-Place-Modelltransformationen spezifizieren.

Verhalten einer Methode vs. Transformationsregel

Story-Diagramme können für sich allein stehende Transformationsregeln spezifizieren oder als Teil eines Softwaremodells das Verhalten von in Klassen definierten Methoden modellieren [vDHH+12, Kap. 3.3.1]. Im letzteren Fall wird zu jeder in einem Klassenmodell enthaltenen Methode ein zugehöriges Story-Diagramm mit gleicher Signatur modelliert. Das resultierende Softwaremodell lässt sich ausführen, analysieren oder zur Generierung von lauffähigem Code in der modellgetriebenen Softwareentwicklung verwenden. Ein Story-Diagramm entspricht dann der Implementierung eines Methodenrumpfs.

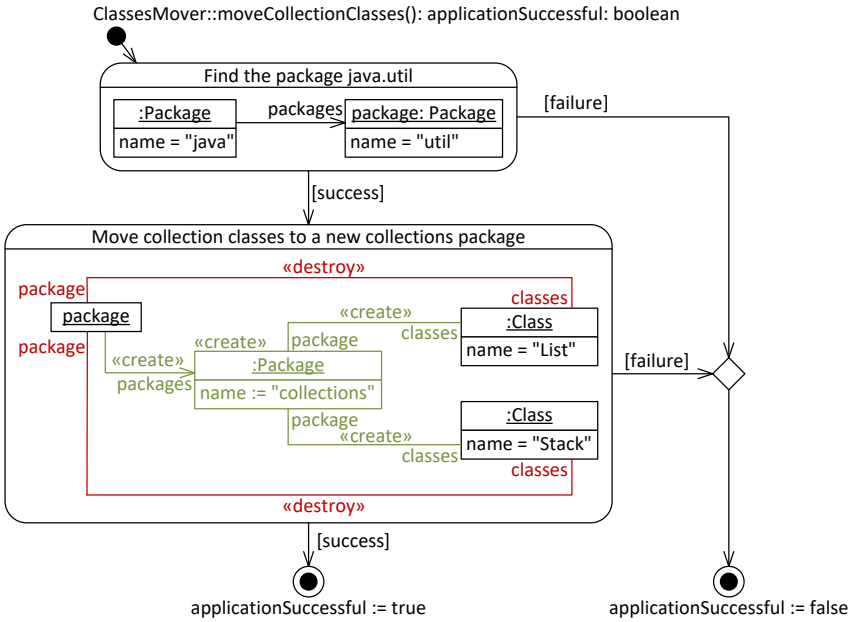


Abbildung 2.16: Ein Story-Diagramm

Die Notation für Kontrollfluss ist im Wesentlichen identisch zu der von UML-Aktivitätendiagrammen. In der Abb. 2.16 ist ein Beispiel für ein Story-Diagramm dargestellt. Hier findet man einen Startknoten, beschriftet mit der Signatur der durch das Story-Diagramm spezifizierten Operation moveCollectionClasses. Die Operation gehört zu der Klasse ClassesMover und hat einen Bool'schen Rückgabeparameter applicationSuccessful. In den beiden Endknoten wird der Rückgabeparameter mit dem Wert true oder false befüllt.

Zwei der Aktivitätenknoten enthalten je ein Story Pattern. Die ausgehenden Kanten sind mit Story-Diagramm-spezifischen Bedingungen (Guards) versehen, nämlich success bzw. failure. Wurde ein Story Pattern erfolgreich angewandt (Matching und anschließende Modifikation des Wirtsgraphen erfolgreich), so wird der umgebende Aktivitätenknoten entlang der Aktivitätenkante mit der Bedingung success verlassen. Andernfalls wird der Aktivitätenknoten entlang der Aktivitätenkante mit der Bedingung failure verlassen.

In diesem Beispiel[8] beschreibt das erste Story Pattern eine zu suchende Ob-

[7] Vereint man die (Meta-)Modelle von Quell-, Zielgraph und Korrespondenzgraph, so lassen sich mit endogenen Transformationen auch exogene Transformationen beschreiben.

[8] Das Bsp. ist etwas konstruiert. Man könnte auch mit nur einem Story Pattern auskommen.

jektstruktur bestehend aus zwei miteinander verknüpften Package-Objekten mit bestimmten Werten in ihren name-Attributen. Das zweite Story Pattern entspricht bis auf wenige Ausnahmen dem Story Pattern aus Abb. 2.15.

Gebundene Variablen

Während eine der Objektvariablen im ersten Story Pattern unbenannt ist (nur der Typ ist angegeben), trägt die andere Objektvariable den Namen package. Durch die Benennung einer Objektvariable lässt sich das Matching für diese Variable in folgenden Aktivitätenknoten und Story Patterns wiederverwenden. Die Objektvariable package wird im zweiten Story Pattern ohne Typ angegeben. Damit stellt sie eine gebundene (vorbelegte) Variable bzw. einen Cursor-Knoten dar [Zün01, Kap. A.6.1]. Ihr Wert wird aus dem Matching des vorhergehenden Story Patterns wiederverwendet. Beim Matching eines Story Patterns werden nur ungebundene Variablen den Knoten und Kanten eines Wirtsgraphen neu zugeordnet.

Das dargestellte Story-Diagramm beschreibt also eine Modelltransformation, die im ersten Schritt ein Paket namens „util“ in einem Paket „java“ sucht und bei Erfolg im zweiten Schritt ein neues Paket namens „collections“ in dem Paket „util“ erstellt und zwei Klassen namens „List“ und „Stack“ in das neue Paket verschiebt.

In diesem Beispiel spezifiziert das Story-Diagramm das Verhalten einer modellierten Methode aus einer ebenfalls modellierten Klasse ClassesMover (Klassendiagramm nicht dargestellt). Bei Aufruf der Methode wird das Story-Diagramm (bzw. der daraus generierte Code) ausgeführt.

Mit Hilfe weiterer Sprachkonstrukte lassen sich auch komplexere Abläufe beschreiben. So können z.B. Schleifen und Aufrufe anderer Story-Diagramme oder Methodenaufrufe modelliert werden.

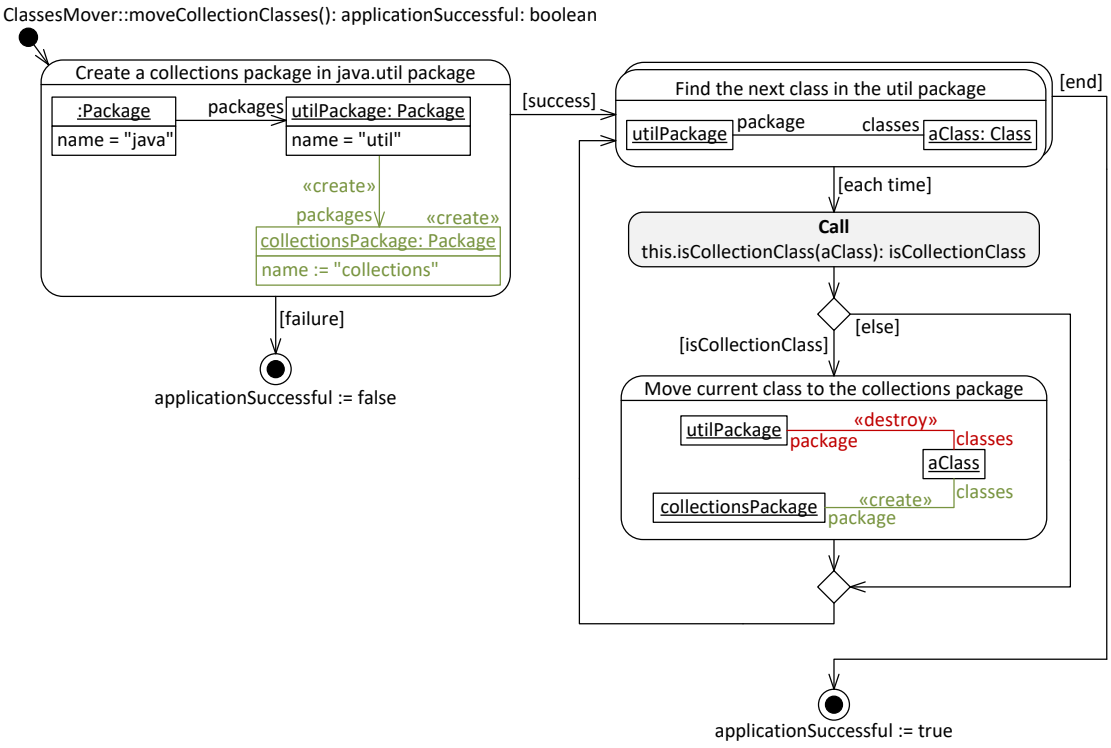


Abbildung 2.17: Ein Story-Diagramm mit Methodenaufrufen und einer Schleife

Möchte man das Verhalten aus der Abb. 2.16 verallgemeinern und es in kleinere Schritte zerlegen, so könnte man es wie in der Abb. 2.17 modellieren. Im ersten Story Pattern wird ein Paket „collections“ erstellt, falls ein Paket „util“ in einem Paket „java“ gefunden werden kann. Bei Misserfolg wird die gesamte Operation über die Kante mit dem Guard failure beendet, bei Erfolg wird das nächste Story Pattern ausgeführt. Dieses ist ein iteriertes Story Pattern – dargestellt durch einen kaskadierten Aktivitätenknoten (durch eine Art Schatten oder zusätzlichen Rahmen) – und beschreibt eine Schleife. Mit jedem erfolgreichen Matching dieses Story Patterns wird der gesamte über die Kante mit dem Guard each time erreichbare Kontrollfluss ausgeführt. Gibt es kein weiteres Matching, wird der Aktivitätenknoten über die Kante mit dem Guard end verlassen. In diesem Beispiel werden also alle Klassen in dem bereits zuvor gefundenen Paket „util“ nacheinander durchlaufen. Per Aufruf der Operation isCollectionClass[9] (modelliert durch einen speziellen Aktivitätenknoten, einen Call-Knoten) wird geprüft, ob es sich um eine Collection-Klasse[10] handelt. Das Ergebnis des Aufrufs wird in einer Variable isCollectionClass zwischengespeichert und in einem Bool'schen Guard geprüft. Bei Erfolg wird die Klasse vom Paket „util“ in das Paket „collections“ verschoben, indem die existierende Verknüpfung zum bisher enthaltenden Paket „util“ entfernt und eine Verknüpfung zum Paket „collections“ erstellt wird. Schließlich liefern die Endknoten in der Rückgabevariable applicationSuccessful den Erfolgsstatus der Ausführung des Story-Diagramms als Bool'schen Wert zurück.

Schleifen

2.4.3 Story-Driven Modeling

Wie im vorhergehenden Abschnitt beschrieben, lassen sich Story-Diagramme dazu nutzen, das Verhalten von Methoden zu modellieren. Zusammen mit Klassendiagrammen können Story-Diagramme in modellgetriebener Softwareentwicklung eingesetzt und zur Generierung von Code verwendet werden. Wann und in welchem Umfang Klassen und Story-Diagramme modelliert werden, bleibt dem Modellierer überlassen. Damit lassen sich Story-Diagramme auch in anderen Softwareentwicklungsprozessen einsetzen.

Zündorf et al. haben einen besonderen Softwareentwicklungsprozess entwickelt, bei welchem u.a. Story-Diagramme zum Einsatz kommen – das sogenannte Story-Driven Modeling [Zün01, NZJ13]. Dabei handelt es sich um eine agile, objektorientierte Softwareentwicklungsmethode. Die Konzeption eines Softwaresystems beginnt dabei mit konkreten Beispielsituationen bzw. Szenarien. Bestimmte Situationen in einem Szenario werden als Objektdiagramm modelliert. Verhalten bzw. Änderungen am Objektdiagramm wird durch sogenannte Story-Boards beschrieben. Ihre Notation orientiert sich stark an der von Story Patterns. Story-Boards stellen konkrete Objektstrukturen und ihre Veränderung in mehreren Schritten dar. Entfernte und hinzugefügte Objekte und Verknüpfungen werden mit «destroy» oder «create» markiert. Aus den exemplarischen Objektdiagrammen und Story-Boards werden schrittweise Klassendiagramme und Story-Diagramme abgeleitet. Ist das Softwaresystem komplett modelliert, kann schließlich ausführbarer Code generiert werden.

[9] Das kann ein anderes Story-Diagramm oder eine in Java implementierte Methode sein.
[10] Bei Java gehören zu den Collection-Typen u.a. die Typen *List, Set, Queue, Stack, Map.*

3 Spezifikation von Entwurfsmustern

Zweck der Musterspezifikation

Mit meiner Arbeit verfolge ich das Ziel, Entwurfsfehler und Design-Erosion zu reduzieren, indem ich Anwendungsstellen von Entwurfsmustern für Entwickler nachvollziehbar präsentiere und nach jeder Entwurfsänderung überprüfbar mache. Zusätzlich dazu unterstütze ich Entwickler bei der Anwendung von Entwurfsmustern, indem ich eine zuvor modellierte Entwurfslösung halbautomatisch in den Entwurf übertrage und zu erledigende Aufgaben aufzeige. Für alle diese Szenarien (siehe Ziele in Abschn. 1.2) ist eine Repräsentation der Entwurfslösung nötig.

Herausforderungen

Diese muss detailliert genug sein, um sowohl daraus eine Musterimplementierung abzuleiten als auch eine Musterimplementierung nachträglich auf Korrektheit zu prüfen. Gleichzeitig muss die Repräsentation allgemein genug sein, um möglichst viele Implementierungsvarianten zu erfassen und Entwicklern größtmöglichen Freiraum bei der Musteranwendung zu geben.

Grenzen verwandter Arbeiten

Die zahlreichen Ansätze zur Spezifikation von Entwurfsmustern (siehe Abschn. 9.1) eignen sich nicht oder nur eingeschränkt für die genannten Szenarien. Zum Beispiel wird eine Entwurfslösung von Eden et al. so abstrakt modelliert, dass eine Prüfung einer Musterimplementierung anhand einer kompakten Entwurfsvorlage möglich ist, jedoch das Erzeugen einer Musterimplementierung aus dieser Vorlage aufgrund fehlender Details[1] nicht möglich ist [NGEK09, EN11] (S. 211 ff.). Andere Arbeiten bieten nur beschränkte Möglichkeiten zur Erfassung von Implementierungsvarianten zu einer Entwurfslösung. Das zu einer Entwurfslösung gehörende Verhalten sowie zu vermeidende Abhängigkeiten in einer Musterimplementierung werden ebenfalls, wenn überhaupt, nur eingeschränkt berücksichtigt (Details in Abschn. 9.1).

Lösungsansatz

Bei meinem Ansatz erfasse ich die Entwurfslösung eines Musters durch Bilden eines abstrakten Modells der Implementierungsvarianten des Musters und stelle dafür eine erweiterbare Musterspezifikationssprache (eine DSL[2]) bereit. Ein Modell einer Entwurfslösung repräsentiert die Struktur und zum Teil auch das Verhalten einer Entwurfslösung, indem objektorientierte Konzepte wie Typen, Operationen, Vererbung, Assoziationen, aber auch Verhalten (Interaktionen) wie das Delegieren eines Methodenaufrufs oder das Instanziieren einer Klasse spezifiziert werden. Diese Repräsentation ist detailliert genug, um in objektorientierte Modellierungs- oder Programmiersprachen übersetzt zu werden. Ergänzend dazu können Kopplungsrestriktionen spezifiziert werden (zu vermeidende Abhängigkeiten), um nach der Implementierung eines Musters überprüft werden zu können.

Eine Spezifikation repräsentiert mehrere Implementierungsvarianten. Zum einen erlaubt die Spezifikation der Entwurfslösung bestimmte Abweichungen von der spezifizierten Struktur (Verfeinerungen) wie z.B. das Einfügen von zusätzlichen

[1] Z.B. werden mehrere Klassen einer Entwurfslösung nicht einzeln, sondern als unbestimmte Menge von Klassen in einer Klassenhierarchie modelliert (siehe Abb. 9.3, S. 212).

[2] DSL: Domain-specific Language

Klassen in eine Vererbungshierarchie oder das Verschieben von Operationen nach oben entlang einer Vererbungshierarchie. Zum anderen können mit Hilfe spezieller Sprachkonstrukte, sogenannter *Set Fragments*, in ihrer Anzahl variierende, zusammengehörige Teile der Entwurfslösung spezifiziert werden wie die zur Implementierung zusätzlicher ConcreteObserver nötigen Typen, Operationen und Beziehungen des Observer-Musters [GHJV95] (vgl. Abb. 1.3, S. 5 und Abb. 1.7, S. 12).

Zur Erfassung von in Entwurfsmustern häufig vorkommenden, vage beschriebenen Teilen einer Entwurfslösung wie eines nicht näher beschriebenen, situationsabhängigen, beobachtbaren Objektzustands im Observer-Muster [GHJV95] (siehe Abb. 1.3, S. 5) werden Aufgaben modelliert (task in Abb. 1.7, S. 12). Diese lassen sich weder automatisch übersetzen, noch automatisch überprüfen, vervollständigen jedoch die Spezifikation einer Entwurfslösung.

Basierend auf der Spezifikation einer Entwurfslösung werden insb. Anwendungsstellen eines Musters modelliert, indem Teile des Entwurfs den eingenommenen Musterrollen zugeordnet werden. Das Modell der Anwendungsstellen wird sowohl für Musteranwendungen (Kap. 5) als auch zur Visualisierung (Kap. 4) und Prüfung (Kap. 6) von Musterimplementierungen genutzt.

Wissenschaftliche Beiträge

Mit diesem Kapitel gebe ich Antworten auf die erste der vier Forschungsfragen aus Kapitel 1 (siehe S. 10) und zeige wie die Entwurfslösung zu einem Entwurfsmuster trotz ihrer absichtlich vagen Beschreibung in der Literatur präzise, vollständig, kompakt und inklusive zahlreicher Varianten modelliert werden kann. Zu den wesentlichen wissenschaftlichen Beiträgen dieses Kapitels gehören:

- eine erweiterbare Musterspezifikationssprache zur Modellierung der Entwurfslösung objektorientierter Entwurfsmuster, welche neben der Klassenstruktur auch Verhalten umfasst und sich sowohl zur Herleitung von Musterimplementierungen als auch zur Prüfung solcher geeignet ist
- ein Konzept zur kompakten Spezifikation von Implementierungsvarianten durch Verfeinerungsregeln und formal definierte Set Fragments
- ein Konzept zur Spezifikation von zu vermeidenden Abhängigkeiten in Musterimplementierungen
- eine Prüfung der Ausdruckskraft der Sprache durch Spezifikation der 23 GoF-Muster sowie einiger Architekturmuster und Idiome

Kapitelstruktur

Im Folgenden formuliere ich zunächst Anforderungen an eine Musterspezifikationssprache. Anschließend gebe ich einen Überblick zum Aufbau der entwickelten Sprache, führe das Konzept hinter der Sprache anhand eines Beispiels ein und stelle schließlich die Sprache selbst vor. Abschließend zeige ich die Grenzen des Ansatzes auf und fasse zusammen. Den vollen Umfang der entwickelten Musterspezifkationssprache beschreibe ich im Anhang A. Alle zur Einschätzung der Ausdruckskraft erstellten Musterspezifikationen präsentiere ich im Anhang B.

3.1 Anforderungen

Für die genannten Einsatzszenarien muss eine Musterspezifikation mehrere Aufgaben erfüllen. Sie muss eine Entwurfslösung so detailliert und präzise beschreiben,

dass daraus zum einen weitestgehend automatisch eine Musterimplementierung abgeleitet werden kann und zum anderen möglichst jede, auch manuell abgewandelte Musterimplementierung auf Konsistenz zur Musterspezifikation (zur Entwurfslösung) geprüft werden kann. Ergänzend dazu sollen die Teile einer Musterimplementierung den Teilen einer Musterspezifikation (den Musterrollen) zugeordnet werden können, damit eine Repräsentation / Dokumentation der Musterimplementierungen möglich ist.

3.1.1 Zweck der Musterspezifikationen

Eine Musterspezifikation soll ein in der Literatur informell beschriebenes Entwurfsmuster – genauer: eine Entwurfslösung (siehe Abb. 2.1, S. 26) – möglichst vollständig, präzise und eindeutig modellieren, um maschinell verarbeitet werden zu können. Gleichzeitig soll eine Musterspezifikation als Vorlage für Musterimplementierungen in einem Entwurfsmodell dienen (siehe Abb. 3.1). Greift man die Terminologie von Hesse und Mayr auf [HM08], so lässt sich eine Musterspezifikation als deskriptives Modell (Nachbild) eines Musters und präskriptives Modell (Vorbild) für Musterimplementierungen einstufen.

deskriptiv & präskriptiv

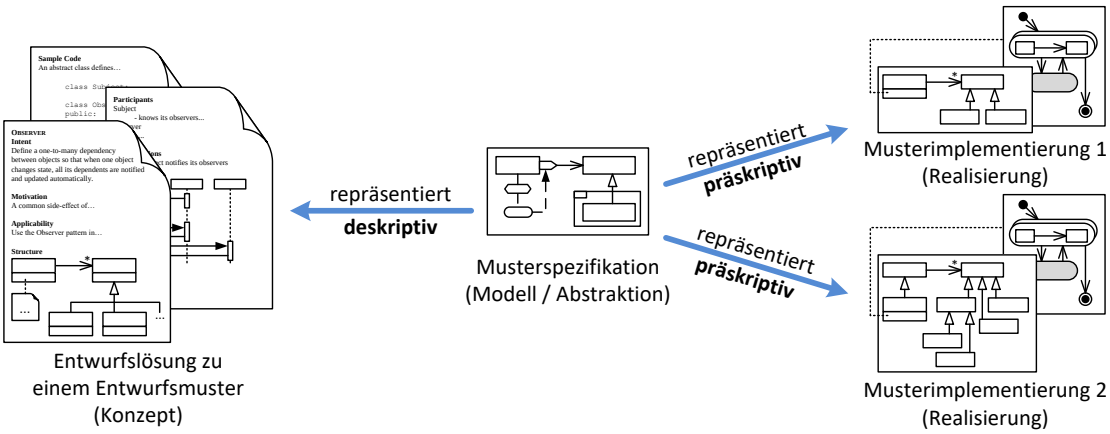


Abbildung 3.1: Repräsentation eines Musters und diverser Musterimplementierungen

Wie jedes Modell muss auch eine Musterspezifikation die drei von Stachowiak geprägten Merkmale Abbildung, Reduktion und Pragmatismus aufweisen[3] [Sta73, HM08]. Neben der Eigenschaft, eine Entwurfslösung und Musterimplementierungen zu repräsentieren bedeutet das insbesondere, dass eine Musterspezifikation sich auf das Wesentliche des Repräsentierten beschränken muss, um einem bestimmten Ziel zu dienen.

Ziele

In meiner Arbeit dient eine Musterspezifikation mehreren Zielen. Zum einen als menschenlesbare, kompakte Darstellung einer Entwurfslösung (ohne der erläuternden Details einer Musterbeschreibung), zum anderen als maschinenverarbeitbares in ein Entwurfsmodell übertragbares und auf Konsistenz mit einer Musterimplementierung überprüfbares Modell einer Entwurfslösung. Die Spezifikation soll also als Vorlage bei einer werkzeuggestützten Musteranwendung und gleichzeitig als

[3] In diesem Zusammenhang werden häufig auch die Begriffe Homomorphismus, Abstraktion und Pragmatismus verwendet.

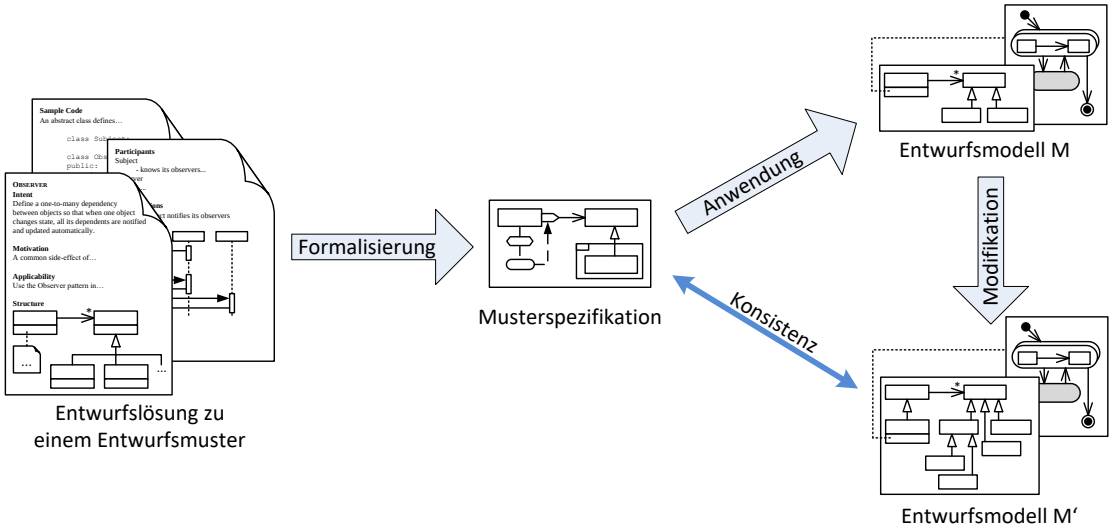


Abbildung 3.2: Musterformalisierung, Musteranwendung und Konsistenzprüfung

Sammlung von einzuhaltenden Bedingungen dienen, um Musterimplementierungen insb. nach manuellen Entwurfsänderungen automatisch auf Konsistenz zur Spezifikation zu prüfen (siehe Abb. 3.2).

> „A formal description that claims to be *the* definitive description of a pattern simply cannot be: it must always be restricted to a subset of possibilities.“
> Frank Buschmann et al., [BHS07b, S. 86]

das Wesentliche

Ich beschränkte mich auf die Spezifikation objektorientierter Entwurfsmuster und orientiere mich insbesondere an den Entwurfsmustern der Gang of Four [GHJV95]. Unter dem Begriff Entwurfslösung verstehe ich insbesondere den vorgeschlagenen Entwurf. So gehören zur Entwurfslösung eines GoF-Musters die Klassenstruktur und ggf. vorgesehenes Interaktionsverhalten zwischen den Klasseninstanzen (siehe Abb. 2.1, S. 26). Zu einem Entwurfsmuster gehört u.a. die damit verfolgte Intention, z.B. etwas austauschbar zu machen oder bestimmte Teile des Entwurfs zu entkoppeln. Die Intention lässt sich im Allgemeinen kaum formalisieren, jedoch spezifiziere ich zumindest unerwünschte Abhängigkeiten im Entwurf, um diese in Musterimplementierungen aufzeigen zu können. Ich verzichte auf die Spezifikation sonstiger Bestandteile einer Intention sowie anderer, nur schwer formalisierbarer Teile einer Musterbeschreibung, zu denen insb. das Einsatzgebiet (Anwendungsfälle), die Konsequenzen, Alternativen einer Entwurfslösung und Implementierungsbeispiele gehören. Ich nehme an, dass die Nutzer der Musterspezifikationen mit den Entwurfsmustern bereits vertraut sind (Einsatzgebiet, Intention, Konsequenzen und Alternativen kennen).

Reverse Engineering nicht im Fokus

Im Gegensatz zu einigen anderen Ansätzen zur Spezifikation von Entwurfsmustern [NSW+02, Wen07, Wen05b, BBF+11] dienen meine Musterspezifikationen (bisher) nicht zur automatischen Auffindung von Musterimplementierungen in Reverse-Engineering-Verfahren, was mit zusätzlichen Anforderungen verbunden wäre.

3.1.2 Anforderungen an die Musterspezifikationssprache

Repräsentation von Klassenstruktur und Interaktionsverhalten Es soll sowohl die Struktur als auch das Verhalten von objektorientierten Entwurfsmustern beschrieben werden können. Dabei soll die Essenz der beschriebenen Entwurfslösung repräsentiert werden. Die Struktur soll auf Klassenebene in Form von Typen, Beziehungen und Operationen ausgedrückt werden, während bei dem Verhalten die Interaktionen zwischen den Beteiligten festgehalten werden sollen. Der Detaillierungsgrad soll so gewählt werden, dass aus einer Musterspezifikation eine konkrete Musterimplementierung abgeleitet werden kann. Die Ausdrucksmächtigkeit soll sich an typischen, wiederkehrenden Elementen in Beschreibungen von Entwurfsmustern, insb. der Gang of Four [GHJV95], orientieren und bei Bedarf erweitert werden können.

Repräsentation von Implementierungsvarianten Damit ein Muster in verschiedensten Situationen angewendet und seine Implementierung an die konkrete Situation angepasst werden kann, soll eine Musterspezifikation möglichst viele Implementierungsvarianten eines Musters repräsentieren.

Von Details wie Klassennamen und den Signaturen von Operationen soll weitestgehend abstrahiert werden. Auch der Implementierungsort einer Operation in einer Klassenhierarchie soll nach Möglichkeit frei wählbar sein. Diese Details werden typischerweise, sofern nicht für die Entwurfslösung relevant, erst bei der Musteranwendung situationsabhängig bestimmt.

Variable Anteile einer Entwurfslösung sollen frei bestimmbar sein. Zum Beispiel soll die Anzahl der ConcreteObserver-Klassen bei der Implementierung des Observer-Musters (Abb. 1.3, S. 5) frei bestimmt werden können. Bei dem Muster Abstract Factory (Abb. 3.3) gibt es sogar mehrere voneinander abhängige, variable Größen. Es kann beliebig viele (konkrete) Fabriken und Produkte geben, allerdings muss jede Fabrik zu jedem Produkt eine passende create-Operation besitzen, wobei jede Fabrik nur Instanzen einer bestimmten Produktfamilie erzeugen soll. Diese Art von Varianz bei der Implementierung eines Musters soll in den Musterspezifikationen eindeutig und kompakt ausgedrückt werden können.

Beziehungen zwischen Klassen werden in Musterbeschreibungen in ihrer einfachsten Form präsentiert. Bei Musteranwendung sind oftmals geringfügige

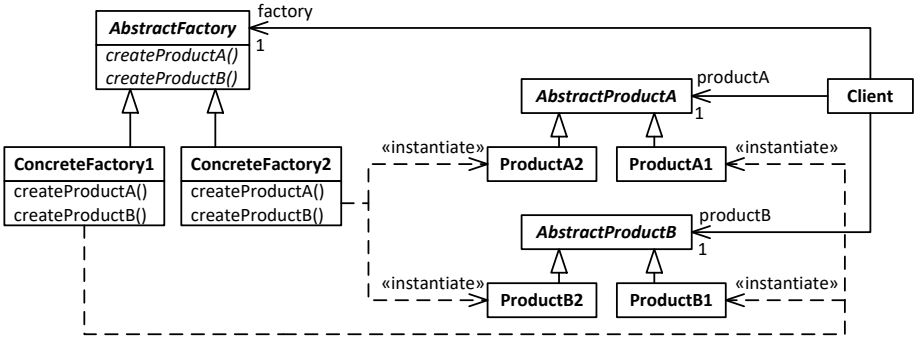


Abbildung 3.3: Die Struktur des Musters Abstract Factory in UML-Notation [GHJV95]

Abweichungen sinnvoll, z.B. können zusätzliche Klassen in eine Vererbungshierarchie eingefügt werden. Die Musterspezifikationssprache soll auch solche Abweichungen erlauben und bei transitiven Beziehungen das Ersetzen von direkten durch indirekte Beziehungen ermöglichen.

Kopplungsrestriktionen Viele Entwurfsmuster haben zum Ziel, bestimmte Teile des Entwurfs voneinander zu entkoppeln, z.B. soll bei dem Facade-Muster (siehe Abb. 8.7, S. 177) ein Teil eines Softwaresystems über eine Fassade vom übrigen System entkoppelt werden [GHJV95, S. 187]. Das bedeutet insb. dass auf das betroffene Teilsystem ausschließlich über die Fassade zugegriffen werden soll. Außerdem sollen die Klassen in dem Teilsystem keine Kenntnis von der Fassade haben, d.h. sie sollen darauf keine Referenz haben. Solche Bedingungen bzgl. der Kopplung von Teilen des Entwurfs sollen, soweit möglich, in der Musterspezifikationssprache beschreibbar sein.

Natürlichsprachliche Hinweise Damit sich eine Musterspezifikation als Zusammenfassung einer Entwurfslösung eignet, soll die Entwurfslösung möglichst vollständig erfasst werden können. Dazu gehören u.a. nicht näher beschriebene Teile des Entwurfs wie der Zustand eines ConcreteSubject-Objekts beim Observer-Muster (Abb. 1.3, S. 5). In solchen Fällen lässt sich der Entwurf nicht vollständig formalisieren. Darum sollen solche Entwurfsbeschreibungen natürlichsprachlich in Form von Hinweisen in Musterspezifikationen aufgenommen werden, den formalisierten Teil des Entwurfs ergänzen und Entwicklern bei der Musteranwendung als Erinnerung dienen. Solche Hinweise sollen nicht die Musterbeschreibung in der Literatur ersetzen und dienen nicht der automatischen Prüfung von Musterimplementierungen. Sie spiegeln lediglich die wichtigsten, nicht formalisierbaren, zu erledigenden Aufgaben bei der Implementierung eines Musters wider.

Übertragbarkeit auf verschiedene Modellierungssprachen Neben der weit verbreiteten UML [OMG11a, OMG11b], die in diversen Varianten und Teilmengen in Entwurfsumgebungen unterstützt wird, gibt es weitere Modellierungssprachen, die im modellgetriebenen Softwareentwurf eingesetzt werden können. So hat sich EMOF[4] im Umfeld der Entwicklungsumgebung Eclipse[5] und dem EMF-Framework[6] [SBPM08] zu einer Alternative zu UML-Werkzeugen beim modellgetriebenen Softwareentwurf entwickelt. Außerdem gibt es diverse, auf bestimmte Domänen spezialisierte Erweiterungen der UML, die ebenfalls bei der modellgetriebenen Softwareentwicklung Anwendung finden. Dazu zählt z.B. UWE (UML-based Web Engineering), ein auf Web Engineering spezialisiertes UML-Profil [KKZB08]. Die Musterspezifikationssprache soll Muster auf einer geeigneten Abstraktionsebene, weitestgehend unabhängig von solchen Modellierungssprachen beschreiben. Dadurch sollen einmal spezifizierte Muster mit den in dieser Arbeit entwickelten Verfahren nicht nur in einer, sondern in verschiedenen Modellierungssprachen und modellgetriebenen Entwurfsprozessen eingesetzt werden können.

[4] Essential MOF, eine Teilmenge der MOF [OMG11c].
[5] http://www.eclipse.org/
[6] http://www.eclipse.org/modeling/emf/

Notation Die Notation (die konkrete Syntax) der Musterspezifikationssprache soll möglichst einfache, kompakte und gut verständliche Musterspezifikationen ermöglichen. Die Notation soll eine hohe kognitive Effektivität aufweisen und im Fall einer grafischen Notation insb. die folgenden von Moody beschriebenen Prinzipien für effektive grafische Notationen einhalten [Moo09]: semiotische Klarheit (eindeutige Zuordnung von semantischen Konstrukten zu grafischen Symbolen), perzeptuelle Unterscheidbarkeit (klar voneinander unterscheidbare Symbole), visuelle Ausdrucksfähigkeit (hohe Ausnutzung von visuellen Variablen wie Position, Form, Größe, Farbe, Helligkeit, Ausrichtung und Textur). Neben diesen Prinzipien sollen wegen des Wiedererkennungswerts auch die für Softwareentwickler bekannten Konventionen berücksichtigt werden und die Sprache dazu, sofern sinnvoll, an bekannte Notationen wie die UML angelehnt werden.

3.2 Überblick

Die wesentliche Idee hinter meinem Ansatz zur Spezifikation der zu einem Muster beschriebenen Entwurfslösung ist eine zweigeteilte Sprache (siehe Abb. 3.4), die zum einen dazu dient, Teile eines objektorientierten Softwareentwurfs kompakt zu repräsentieren, und zum anderen Variabilität und Bedingungen bzgl. der Entwurfseigenschaften zu formulieren.

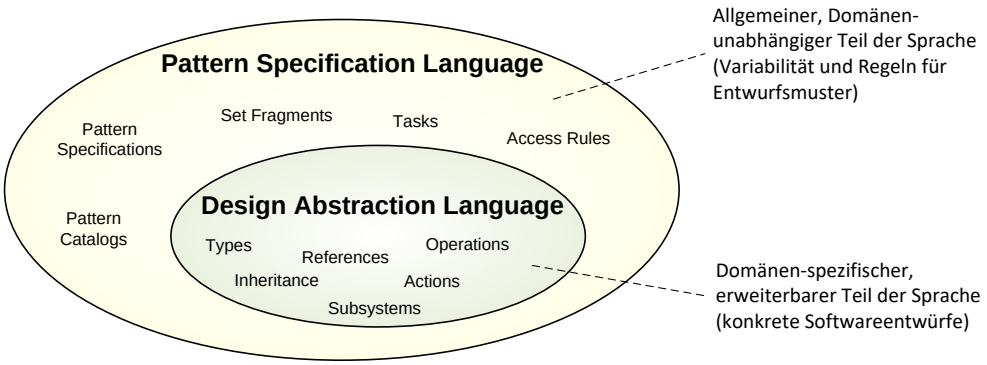


Abbildung 3.4: Aufbau der Musterspezifikationssprache als Euler-Diagramm

Einen Teil der Musterspezifikationssprache, der *Pattern Specification Language* (PSL), bildet die sogenannte *Design Abstraction Language* (DAL). Die DAL bietet Sprachkonstrukte zur Spezifikation von Softwareentwurfsstrukturen auf Basis von für objektorientierte Softwareentwurfsmodelle typischer Konzepte. Zu diesen zählen insbesondere Typen, Operationen, Vererbungsbeziehungen und Assoziationen. Solche Elemente finden sich typischerweise in allen Entwurfsmodellen wieder, welche die Klassenstruktur eines zu entwickelnden Systems beschreiben, so z.B. auch in UML-Klassen- und Ecore-Modellen (zu Ecore siehe Fußnote auf S. 13).

PSL & DAL

OO-Struktur

Zur Erfassung von Verhalten werden zusätzliche Sprachkonstrukte, sogenannte Aktionen, geboten. Diese repräsentieren für objektorientierte Entwurfsmuster typische Interaktionen wie die Delegation eines Methodenaufrufs an ein anderes Objekt oder die Instanziierung einer Klasse durch ein Objekt. Im Gegensatz

Verhalten

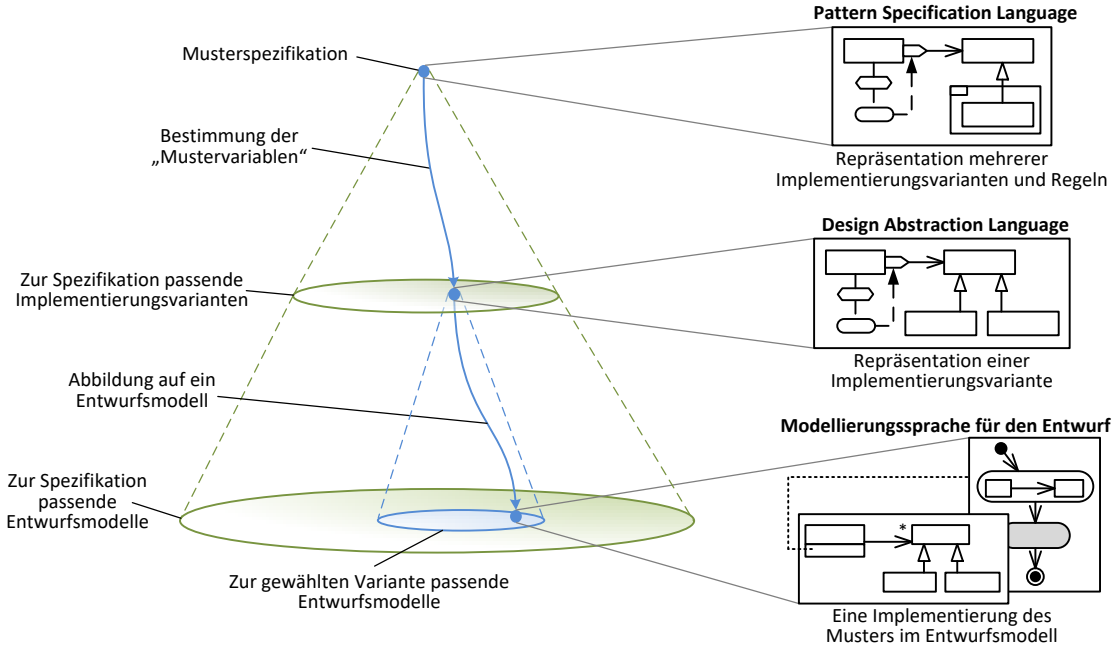


Abbildung 3.5: Repräsentation von Implementierungsvarianten eines Musters

zur Verhaltensmodellierung in UML-Aktivitätendiagrammen, State Charts oder Story-Diagrammen bietet mein Ansatz nur eingeschränkte Möglichkeiten zur Beschreibung von Kontrollfluss. Dafür bieten die wenigen vordefinierten Aktionen (z.B. redirect in Abb. 1.7, S. 12) eine äußerst kompakte Form der Verhaltensbeschreibung. Bei Bedarf könnte die DAL um Konzepte wie Softwarekomponenten, Ports und entsprechende Konnektoren oder andere bisher fehlende Sprachkonstrukte erweitert werden. Die DAL wird in Abschnitt 3.4 im Detail vorgestellt.

Zum einen macht die DAL Musterspezifikationen unabhängig von einer für den Softwareentwurf verwendeten Modellierungssprache wie der UML oder Ecore (EMOF). Zum anderen abstrahiert sie von den Details einer Musterimplementierung, sodass mehrere Implementierungsvarianten in einem Entwurfsmodell zur selben Repräsentation in der DAL konform sind. Letzteres wird in der Abb. 3.5 veranschaulicht. Links wird der gesamte Lösungsraum aller durch eine Spezifikation erfassbaren Implementierungsvarianten zu einem Muster skizziert, während rechts verschiedene Repräsentationen bestimmter Varianten angedeutet sind.

Implementierungsvarianten

Unten rechts im Bild wird eine Musterimplementierung in einem Klassen- und Story-Diagramm-Modell angedeutet (repräsentiert durch einen Punkt links im Bild). Neben dieser gibt es mehrere weitere Implementierungsvarianten desselben Musters im Entwurfsmodell (grüne Fläche unten links im Bild).

In der Mitte rechts ist eine DAL-Repräsentation einer Musterimplementierung skizziert. Sie abstrahiert von einigen Implementierungsdetails und fasst damit mehrere zur DAL-Repräsentation konforme Implementierungsvarianten zusammen (blaue Fläche unten links im Bild). Die DAL nutzt u.a. die Transitivität bestimmter Beziehungen (z.B. eine Klasse ist Unterklasse einer anderen oder einer Klasse steht eine bestimmte Operation zur Verfügung) zur Abdeckung von Implementierungsvarianten aus. Neben den direkten Beziehungen, wie sie in der DAL dargestellt werden, werden auch indirekte Beziehungen (z.B. indirekte Verer-

Transitivität

bungsbeziehungen) als konform zur DAL-Repräsentation angesehen. Ebenso werden bestimmte Abweichungen in Methodensignaturen akzeptiert.

Signaturen

Die Musterspezifikationssprache (PSL) ergänzt die DAL um zusätzliche, verallgemeinernde Sprachkonstrukte zur kompakten Repräsentation von Implementierungsvarianten. Mit Hilfe sogenannter Set Fragments ist es möglich, Teile einer DAL-Repräsentation zu kennzeichnen, welche in einer Musterimplementierung als Ganzes mehrfach vorkommen können. Z.B. lässt sich damit ausdrücken, dass die ConcreteObserver-Klassen des Observer-Musters samt zugehöriger Operationen und Beziehungen in einer Musterimplementierung beliebig häufig vorkommen können (ausgedrückt durch das Set Fragment observers in Abb. 1.7, S. 12). Auf die gleiche Weise lassen sich die voneinander abhängigen, aber in der Anzahl unbeschränkten Fabriken und Produkte des Musters Abstract Factory (siehe Abb. 3.3, S. 43) samt ihrer Operationen und Beziehungen zueinander modellieren.

Set Fragments

Oben rechts in Abb. 3.5 ist eine Musterspezifikation mit einem Set Fragment skizziert. Sie repräsentiert neben den durch die DAL beschreibbaren Implementierungsvarianten weitere, aus Set Fragments abgeleitete Varianten (grüne Flächen links im Bild). Diese Implementierungsvarianten unterscheiden sich im Wesentlichen in der Anzahl der Vorkommen der in einem Set Fragment definierten Entwurfsstruktur. Insofern kann die zu bestimmende Anzahl der Vorkommen (z.B. die Anzahl der ConcreteObserver-Klassen im Observer-Muster) als Variable der Spezifikation angesehen werden. Legt man sich auf einen Wert für diese Variable fest, landet man bei einer der DAL-Repräsentationen (Mitte links im Bild).

Neben den Implementierungsvarianten lassen sich in der PSL auch Bedingungen formulieren, welche die Kopplung der an einer Musterimplementierung beteiligten Entwurfsteile einschränken. Bei dem Facade-Muster [GHJV95] (siehe Abb. 8.7, S. 177), z.B., kann spezifiziert werden, dass das hinter einer Fassade verborgene Teilsystem keine Verweise auf die Fassade besitzen und das übrige System nur über die Fassade auf das Teilsystem zugreifen soll. Ergänzend dazu lassen sich Entwurfsaufgaben natürlichsprachlich formulieren. Mehrere Musterspezifikationen werden in Musterkatalogen zusammengefasst und in Kategorien organisiert.

Kopplungsrestriktionen

Entwurfsaufgaben

3.3 Konzeption der Musterspezifikationssprache am Beispiel

Im Folgenden erläutere ich anhand des Observer-Musters den Weg zu der von mir entwickelten Musterspezifikationssprache. Ich diskutiere, welche Teile einer Musterbeschreibung sich formalisieren lassen und motiviere die gewählte Notation. Abschließend gehe ich genauer auf die durch eine Musterspezifikation erfassbaren Implementierungsvarianten ein.

3.3.1 Was lässt sich formal beschreiben?

Entwurfsmuster beschreiben eine meist grobe Idee zur Lösung eines wiederkehrenden Entwurfsproblems. Erklärender Text wird häufig mit Entwurfsskizzen in Form von Diagrammen und Quellcode angereichert. Eine Anwendung des Musters kann zu einem von der Beschreibung abweichenden Entwurf führen und dennoch die beschriebene Idee verwirklichen, was die Entwurfsidee schwer greifbar macht.

Entwurfslösung schwer greifbar

Im Rahmen dieser Arbeit habe ich mir zum Ziel gesetzt, die wesentlichen Teile einer Entwurfsidee formal zu erfassen, jedoch nicht das Muster samt seines Kontextes, des behandelten Problems, der Intention und der Konsequenzen zu spezifizieren. Die Frage ist also: Was ist die Kernidee hinter einer in einem Muster beschriebenen Entwurfslösung und wie lässt sich diese spezifizieren?

Vielfalt an Implementierungsmöglichkeiten

Betrachtet man als Beispiel die Beschreibung des Observer-Musters durch die Gang of Four [GHJV95], so stellt man fest, dass sie zum größten Teil aus erklärendem Text besteht. Der Text wird zur Verdeutlichung durch Diagramme und Quellcode ergänzt, die exemplarisch den beschriebenen Entwurf skizzieren (siehe Abb. 1.1, S. 4). Die Diagramme stellen allerdings nicht die generelle Entwurfsidee dar, sondern nur eine mögliche Implementierung des beschriebenen Musters. Das Klassendiagramm (Abb. 1.2, S. 5) und das Sequenzdiagramm in dem Beispiel beschreiben eine Struktur mit nur einer ConcreteObserver-Klasse und ein mögliches Szenario für Interaktionen mit nur zwei Observer-Objekten. Das beschriebene Muster erlaubt jedoch eine beliebige Anzahl von ConcreteObserver- und ConcreteSubject-Klassen sowie eine beliebige, sich ggf. zur Laufzeit ändernde Anzahl an Observer-Objekten. Das beschriebene Szenario ist nur eines unter vielen möglichen. Das gleiche gilt für die exemplarische Implementierung, die ebenso in anderen Programmiersprachen erfolgen könnte.

Variabilität im Entwurf

Abstrahiert man von den Details der exemplarischen Beschreibung der Entwurfslösung in Abb. 1.1 (S. 4) bzw. Abb. 1.2 (S. 5) und berücksichtigt die Variabilität in der Anzahl der Klassen, so lässt sich die wesentliche Idee hinter dem Muster etwas treffender wie in Abb. 1.3 (S. 5) skizzieren. Aber auch diese Darstellung ersetzt keinesfalls den beschreibenden Text. Sie stellt lediglich die vorgeschlagene Struktur des Entwurfs dar und deutet an, wie der Entwurf von Situation zu Situation variieren kann. Da der Zustand eines beobachteten Subjekts nicht zwingend ein Attribut der Klasse ConcreteSubject sein muss, werden dieser und seine Verwendung durch die Beobachter nur natürlichsprachlich beschrieben.

Kernidee

Die wesentlichen Eigenschaften des Musters und damit die Essenz oder Kernidee einer der beschriebenen Entwurfslösungen lassen sich wie folgt zusammenfassen:

- Es gibt abstrakte Typen Subject und Observer
- Ein Subject-Objekt referenziert eine beliebige Anzahl von Observer-Objekten
- Der Typ Observer definiert eine update-Methode
- Der Typ Subject definiert eine notify-Methode, die auf allen referenzierten Observer-Objekten die update-Methode aufruft
- Es kann beliebig viele Unterklassen von Observer und Subject geben
- Jedes Subject-Objekt hat einen Zustand, bei dessen Änderung die notify-Methode aufgerufen wird
- Jedes Observer-Objekt referenziert ein Subject-Objekt, auf dessen Zustandsänderung es hört und bei Aufruf der update-Methode auf den Zustand zugreift (Eine alternative Entwurfslösung[7] weicht in diesem Punkt ab.)
- Kein Subject-Objekt referenziert ein Observer-Objekt anders als über die Observer-Schnittstelle

[7] Ein Observer kann per Event-Objekt beim notify-Aufruf über die Zustandsänderung informiert und auf die subject-Referenz verzichtet werden (siehe Spezifikation in Abb. B.22, S. 297).

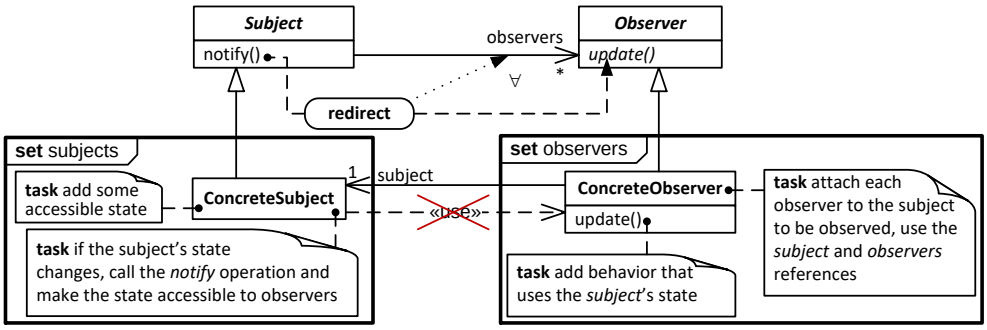


Abbildung 3.6: Formalisierungsversuch für eine Observer-Entwurfslösung

Der letzte Punkt ergänzt die Entwurfsskizze aus Abb. 1.3 (S. 5) um eine Einschränkung bzgl. der Kopplung zwischen beteiligten Subject- und Observer-Klassen.

Welche der aufgelisteten Eigenschaften lassen sich formal erfassen? Da die Entwurfslösung – wie bei vielen anderen Entwurfsmustern auch – in erster Linie aus der Beschreibung einer Klassenstruktur besteht, liegt es nahe, die Entwurfslösung in Anlehnung an UML-Klassendiagramme durch Beschreiben der Klassenstruktur zu formalisieren. Klassendiagramme beschreiben jedoch nur ganz konkrete Klassenstrukturen und bieten nicht die nötige Flexibilität zur Beschreibung der möglichen Varianten bei der Implementierung eines Musters. Hinzu kommt die Notwendigkeit, auch das bisher informell beschriebene Verhalten zu formalisieren.

Klassenstruktur

Ergänzt man die UML-Klassendiagramm-Notation um (a) *Set Fragments* zur Beschreibung von Entwurfsstrukturen, die beliebig häufig auftreten können, um (b) *Aktionen* zur Beschreibung von für Entwurfsmuster typischem Interaktionsverhalten und um (c) einige Sprachkonstrukte zur Beschreibung von Kopplungsrestriktionen, so erhält man eine Darstellung wie in Abb. 3.6. Zusätzlich zu den abstrakten Klassen Subject und Observer mit ihren Methoden notify und update sowie der Assoziation observers werden hier zwei Set Fragments subjects und observers (dargestellt als Rahmen mit dem Label set) verwendet, um zu beschreiben, dass es beliebig viele ConcreteSubject- und ConcreteObserver-Klassen mit den beschriebenen Eigenschaften geben soll. Insb. soll jede ConcreteSubject-Klasse von der Subject-Klasse erben, was analog für die Vererbungsbeziehung zwischen ConcreteObserver und Observer gilt (zur Semantik von Set Fragments später mehr). Die hier verwendete Notation ermöglicht es, die beliebig häufig auftretende Struktur nur ein Mal zu spezifizieren. Dabei kann so eine Struktur auch aus mehreren miteinander verbundenen Klassen und ihren Eigenschaften bestehen.

weitere Bestandteile der Entwurfslösung

Der Aufruf der update-Methode wird durch die eingeführte redirect-Aktion beschrieben – eine Art Delegation. Delegationen kommen in Entwurfsmustern häufig vor, was ein eigenes Sprachkonstrukt dafür rechtfertigt. Bei der verwendeten Spezifikation einer Delegation wird neben der aufgerufenen Methode auch spezifiziert, auf welchen Objekten die Methode aufzurufen ist. Im dargestellten Beispiel wird durch den gepunkteten Pfeil definiert, dass die Methode auf allen über die Assoziation observers referenzierten Observer-Objekten aufgerufen werden soll.

Delegation

Die Entkopplung der ConcreteSubject- und ConcreteObserver-Klassen voneinander wird durch die durchgestrichene use-Kante definiert. Das beschreibt, dass die Implementierung der ConcreteSubject-Klassen gänzlich unabhängig von den

Entkopplung

ConcreteObserver-Klassen erfolgen muss. Das bedeutet u.a., dass ConcreteSubject-Klassen keine Assoziationen auf eine der ConcreteObserver-Klassen haben dürfen und die ConcreteObserver-Klassen nicht als Parameter- oder Rückgabetyp in Methoden der ConcreteSubject-Klassen verwendet werden dürfen.

Durch diese Form der Musterspezifikation sind 6 der 8 zuvor aufgelisteten Eigenschaften der Observer-Entwurfslösung erfasst. Die vorletzten zwei Punkte lassen sich nicht formal beschreiben, da der Zustand eines Subjekts und was bei seiner Änderung geschehen soll sehr von dem Anwendungsfall abhängen. Diese im Muster absichtlich nicht näher definierten Eigenschaften werden hier nur informell, in natürlicher Sprache erfasst. Damit die Spezifikation des Musters weitestgehend vollständig ist und Entwicklern als Zusammenfassung dient, formuliere ich solche Eigenschaften in Form von Entwicklungsaufgaben (Anmerkungen mit dem Label task), die bei Musteranwendung erledigt werden müssen. Zu solchen Aufgaben zählt bei dieser Entwurfslösung das Ergänzen folgender Merkmale: der Zustand eines Subject-Objekts, der Aufruf der notify-Methode bei dessen Änderung, das Auslesen des Zustands sowie das Registrieren und Abmelden der Observer.

Hinweise bzw. Aufgaben

Die in der Abb. 3.6 verwendete, UML-nahe Notation hat einige gravierende Nachteile, weswegen ich mich bei meinem Ansatz für eine davon abgeleitete, aber abweichende Notation entschieden habe. Diese beschreibe ich als Nächstes.

3.3.2 Wahl der Notation

Da die Musterspezifikationssprache primär von Softwareentwicklern genutzt werden soll und diese insbesondere mit UML-Klassendiagrammen vertraut sind, liegt es nahe, die UML-Notation aufzugreifen. So ist der erwartete Wiedererkennungswert hoch und das Einarbeiten in die Musterspezifikationssprache einfacher, als bei gänzlich neuen Notationen.

Wiedererkennung

Die UML ist allerdings eine sehr umfangreiche und detaillierte Sprache. Für den in dieser Arbeit beschriebenen Ansatz ist die UML zu umfangreich und gerade beim automatisierten Anwenden von Mustern sowie bei der Prüfung von Anwendungsstellen kaum handhabbar. Abgesehen davon lässt sich mit einem UML-Modell nur eine Variante einer Musterimplementierung modellieren und nicht eine Vielzahl von Varianten in nur einer Repräsentation. Darum habe ich mich für eine eigene, kompaktere Sprache entschieden, mich aber zwecks Wiedererkennung an der UML orientiert.

Sprachumfang

Varianten

Die UML-Notation hat einige in der Literatur beschriebene Mängel bzgl. der kognitiven Effektivität der Darstellung [MvH09]. So gibt es in der UML z.B. eine Vielzahl von ähnlichen Notationen (Rechtecke und gestrichelte oder durchgezogene Linien mit ähnlichen Pfeilspitzen) für verschiedene Konzepte wie Klassen und ihre Beziehungen. Man spricht in solchen Fällen von knapper Verwendung von visuellen Variablen (Position, Form, Größe, Farbe, Helligkeit, Ausrichtung und Textur) [Moo09], wodurch sich die verschiedenen Symbole der Notation optisch schwer voneinander unterscheiden lassen. Außerdem werden in der Sprache viele Informationen textuell anstatt grafisch kodiert, z.B. werden Eigenschaften von Klassen, Assoziationen, Operationen, etc. in textueller Form angegeben, z.B. durch «interface», {unique}, translate(text: String): String. Dadurch muss ein Diagramm erst aufwändig *gelesen* werden, anstatt schnell *als Bild erfasst* zu werden.

kognitive Effektivität

Neben diesen Nachteilen kommt hinzu, dass die Notation in Kombination mit den eingeführten Set Fragments schwer zu lesen und nicht eindeutig ist. Das wird am Beispiel der Spezifikation des Observer-Musters in Abb. 3.6 deutlich. Hier wird eine redirect-Aktion durch Linien mit zwei Methoden und einem Assoziationsende verbunden. Die Methoden und das Assoziationsende sind nicht als Knoten dargestellt, wodurch nicht immer klar erkennbar ist, was genau durch die Linien verbunden wird.

Eindeutigkeit

Die Darstellung von Assoziationen, Operationen und Attributen in der Abb. 3.6 ist im Zusammenhang mit Set Fragments irreführend. Hier werden zwei Klassen ConcreteSubject und ConcreteObserver spezifiziert, die sich in je einem Set Fragment befinden. Somit dürfen diese Klassen bei Musterimplementierungen beliebig häufig und nicht zwingend in gleicher Anzahl vorkommen. Dadurch ist jedoch bei Assoziationen wie der subject-Referenz nicht klar, welchem Set Fragment diese zugeordnet werden und somit auch nicht wie häufig diese vorkommen dürfen (oder welche der ConcreteSubject- und ConcreteObserver-Klassen sie verbinden sollen). Außerdem soll bei Mustern wie Abstract Factory (siehe Abb. 3.3, S. 43) die Anzahl von Methoden unabhängig von der Anzahl der Klassen variieren. Da Set Fragments Vielfache von darin enthaltenen Teilgraphen beschreiben, ließen sich derartige Zusammenhänge einfacher spezifizieren, wenn Methoden als eigenständige Knoten dargestellt und damit eindeutig einem bestimmten Set Fragment zugeordnet werden könnten, ggf. sogar unabhängig von der zugehörigen Klasse.

Vielfache von Referenzen und Methoden

Um zumindest einige der genannten Mängel zu reduzieren, habe ich im Rahmen dieser Arbeit eine von der UML abweichende Notation für die Spezifikation von Entwurfsmustern entwickelt. Diese orientiert sich an der Notation für UML-Klassendiagramme, löst aber die Probleme bzgl. der Eindeutigkeit im Zusammenhang mit Set Fragments und setzt im Vergleich zur UML die visuelle Variable „Form“ mehr ein, was die visuelle Ausdruckskraft steigert[8] [MvH09]. Assoziationsenden, Operationen und Attribute werden in meiner Notation als Knoten visualisiert. Die verschiedenen Knoten werden durch unterschiedliche Formen voneinander abgehoben. Einige Merkmale werden durch unterschiedliche Schattierungen gekennzeichnet.

Anpassung der Notation

Die Spezifikation des Observer-Musters aus Abb. 3.6 wird in meiner Notation wie in Abb. 3.7 visualisiert. Klassen und Vererbungsbeziehungen behalten ihre UML-Notation. Set Fragments werden wie in Abb. 3.6 dargestellt.

Assoziationsenden wie observers und subject werden durch Rechtecke mit einer spitzen Seite in Richtung der referenzierten Klasse dargestellt, mit dem Rollennamen gefüllt und mit der Kardinalität (hier 1 bzw. *) annotiert. Eine Linie mit Pfeilspitze verweist auf die referenzierte Klasse, während das Assoziationsende im Gegensatz zur UML-Notation nicht auf Seite der referenzierten, sondern auf Seite der referenzierenden Klasse positioniert wird. So ist die Eigenschaft der Klasse, die eine andere referenzieren soll, räumlich nah an der Klasse dargestellt, welche diese Eigenschaft besitzen soll. Das vereinfacht zugleich die Zuordnung einer Referenz (einer unidirektionalen Assoziation) zu einem Set Fragment wie bei der subject-

Referenzen

[8] Helligkeit und Farbe ließen sich ebenfalls dazu nutzen, die Ausdruckskraft zu steigern. Bis auf einige Ausnahmen verzichte ich jedoch weitestgehend darauf, damit ich sie bei der Visualisierung von Anwendungsstellen einsetzen kann, wo ich eine den Musterspezifikationen sehr ähnliche Notation verwende. Details dazu werden in Abschnitt 4.3 (S. 87 ff.) beschrieben.

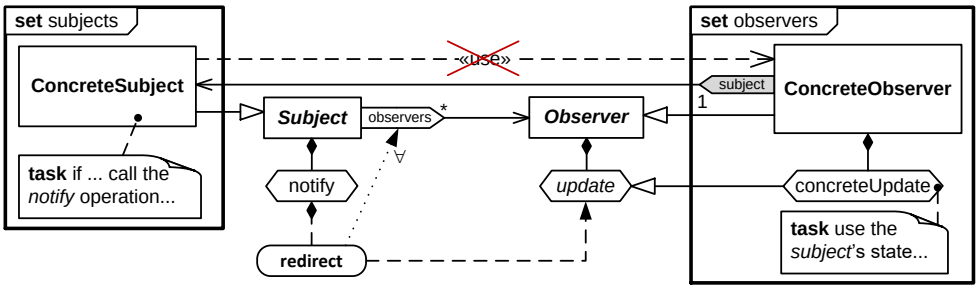


Abbildung 3.7: Spezifikation des Observer-Musters, Variante 1 (vereinfachte Darstellung, vollständige Spezifikation in Abb. B.20, S. 297)

Referenz der ConcreteObserver-Klasse, die hier eindeutig zu dem Set Fragment observers zugeordnet wird. Außerdem ist bei der gewählten Notation für Assoziationsenden als Knoten die Verknüpfung der redirect-Aktion aus Abb. 3.7 mit der observers-Referenz eindeutig dargestellt.

Operationen

Operationen wie notify und update werden als Sechsecke (Hexagone) dargestellt und durch eine Linie mit der zugehörigen Klasse verbunden. Die Linie wird dabei wie eine UML-Kompositionsbeziehung dargestellt (gefüllte Raute auf der Seite des enthaltenden Elements), um die Zugehörigkeit der Operation zur Klasse anzudeuten. Durch die Darstellung von Operationen als eigenständige Knoten lässt sich durch Set Fragments z.B. spezifizieren, dass eine Klasse eine frei wählbare Anzahl von Operationen enthalten soll. Solche Spezifikationen sind z.B. bei dem Muster Abstract Factory notwendig, wo eine Factory-Klasse eine create-Operation pro Produkt bereitstellt (vgl. Entwurfslösung in Abb. 8.5 und Spezifikation in Abb. 8.6, S. 176).

verbliebene Unklarheiten

Trotz der Darstellung von Operationen und Assoziationsenden als Knoten bleibt in der Abb. 3.7 unklar, mit welchen der beliebig vielen ConcreteSubject-Klassen eine ConcreteObserver-Klasse durch eine subject-Referenz verbunden werden soll. Bei dieser Spezifikation wird diese Entscheidung zugunsten der Flexibilität dem Anwender des Musters überlassen. So kann er entscheiden, ob er einen Beobachter pro Subjekt (und genau eine subject-Referenz pro ConcreteObserver) einsetzt oder einen Beobachter mehrere Subjekte beobachten lässt (und dafür mehrere subject-Referenzen in einem ConcreteObserver vorsieht). Dennoch gehört die Information, dass eine solche Referenz existieren soll, zur Entwurfslösung. Durch die Unklarheit, welche Klassen per Referenz verbunden werden sollen, kann in diesem Fall aus der Spezifikation keine vollständige Implementierung des Musters abgeleitet werden. Ebenso sind die Möglichkeiten, eine Musterimplementierung zur validieren, beschränkt. Zur Verdeutlichung dieser Unklarheit wird die Referenz grau hinterlegt. So dargestellte Klassen, Operationen, Attribute, Referenzen und Aktionen können nicht automatisch in den Entwurf übersetzt werden und müssen daher manuell implementiert werden.

Zur Vermeidung der beschriebenen Unklarheit, aber auf Kosten einiger Implementierungsvarianten kann die Entwurfslösung wie in Abb. 3.8 spezifiziert werden. Hier wird spezifiziert, dass zu jeder ConcreteSubject-Klasse beliebig viele ConcreteObserver-Klassen mit je einer eindeutigen Referenz auf das Subjekt existieren. Außerdem illustriert dieses Beispiel eine Schachtelung von Set Fragments.

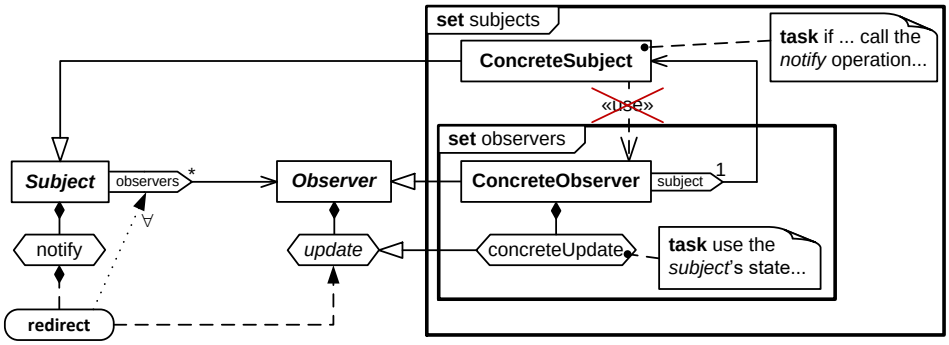


Abbildung 3.8: Spezifikation des Observer-Musters, Variante 2 (vereinfachte Darstellung, vollständige Spezifikation in Abb. B.21, S. 297)

Da in Entwurfsmusterbeschreibungen zwar von Operationen, aber meist nicht von ihren Signaturen gesprochen wird, ist bei einer Vererbungsbeziehung wie zwischen den Klassen Observer und ConcreteObserver aus Abb. 3.7 und 3.8 bei den angegebenen Operationen update und concreteUpdate selbst bei gleicher Benennung nicht klar, ob die Operationen einander überschreiben (bzw. implementieren) oder überladen sollen. Um diese beiden Fälle klar voneinander zu unterscheiden, setze ich im Fall des Überschreibens eine Vererbungskante zwischen den Operationen ein. In Abb. 3.7 und 3.8 drückt die Kante zwischen den Operationen update und concreteUpdate aus, dass concreteUpdate eine Spezialisierung von update ist und damit stets dieselbe Signatur haben soll wie update. Beim Überladen entfällt so eine Kante.

Spezialisierung bei Operationen

Abhängigkeiten zwischen Entwurfsteilen werden durch gestrichelte Pfeile dargestellt. Durch den durchgestrichenen use-Pfeil wird angegeben, dass keine ConcreteSubject-Klasse zu einer ConcreteObserver-Klasse eine Abhängigkeit haben darf. use ist die allgemeinste von mehreren Abhängigkeitsarten in meiner Musterspezifikationssprache.

Kopplung

Die nicht formal erfassbaren, aber wichtigen Aspekte einer Entwurfslösung werden wie in den vorhergehenden Beispielen als Aufgaben formuliert und wie UML-Kommentare in einem Notizsymbol dargestellt. So wird ihr nicht-formaler Charakter visualisiert.

Entwurfsaufgaben

3.3.3 Repräsentation von Implementierungsvarianten

Wie in Abschnitt 3.2 erläutert, ist die Musterspezifikationssprache zweigeteilt. Auf der einen Seite werden Klassen, Operationen, Vererbungsbeziehungen und Assoziationen beschrieben, auf der anderen Seite wird angegeben, wie viel Flexibilität Entwickler bei der Implementierung eines spezifizierten Musters haben und welche Bedingungen sie dabei einzuhalten haben. Es gibt also zahlreiche Implementierungsvarianten zu einer Musterspezifikation (vgl. Abb. 3.5, S. 46).

Mit den DAL-Sprachkonstrukten zur Beschreibung von Klassen, Operationen sowie der zugehörigen Beziehungen und Interaktion lässt sich kompakt und auf einem hohen Abstraktionslevel ein Teil eines Entwurfs beschreiben, der ein spezifiziertes Muster implementiert, also eine Implementierungsvariante des Musters. Die übrigen Sprachkonstrukte der Musterspezifikationssprache dienen dazu, einen

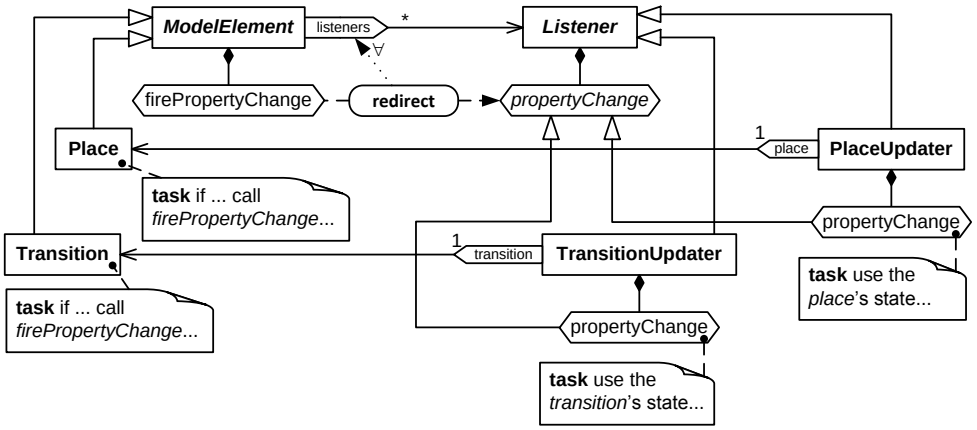


Abbildung 3.9: Abstrakte Repräsentation einer Implementierungsvariante des Observer-Musters in der Design Abstraction Language (DAL)

exemplarischen Entwurf, der in der DAL beschreibbar ist, zu verallgemeinern und zusätzliche Bedingungen zu spezifizieren.

Bei Anwendung des in Abb. 3.8 spezifizierten Observer-Musters legt ein Entwickler fest, welche Klassen, Methoden und Assoziationen seines ggf. zum Teil schon vorhandenen Entwurfs die in der Musterspezifikation definierten Rollen (z.B. Observer, ConcreteObserver, update und observers) spielen sollen oder wie die zu ergänzenden Entwurfsteile zu benennen sind. Dabei legt der Entwickler u.a. fest, wie häufig die in Set Fragments angegebenen Entwurfsstrukturen bei der Musterimplementierung vorkommen sollen. Sind alle diese Informationen vorhanden,

Implementierungsvariante in der DAL

lässt sich eine Implementierungsvariante des Musters mit der DAL wie in Abb. 3.9 beschreiben. Die ConcreteObserver-Klasse samt der concreteUpdate-Operation und der subject-Referenz (oder genauer: die entsprechenden Rollen) aus der Musterspezifikation in Abb. 3.8 kommen in diesem Beispiel zwei Mal in Form der Klassen PlaceUpdater und TransitionUpdater sowie zugehöriger Operationen und Referenzen vor. Das gleiche gilt für die Implementierung der Rolle ConcreteSubject, die ebenfalls von zwei Klassen im Entwurf eingenommen wird.

von der DAL zum Entwurfsmodell

Bis auf die redirect-Aktion lassen sich alle Elemente aus Abb. 3.9 mehr oder weniger 1-zu-1 in ein Klassenmodell (modelliert z.B. in der UML oder Ecore) übersetzen. Das durch die redirect-Aktion spezifizierte Verhalten gehört in ein separates Verhaltensmodell. Z.B. kann das Klassenmodell durch ein Story-Diagramm-Modell ergänzt werden, wo das Verhalten der Operation firePropertyChange spezifiziert wird (siehe Abschn. 2.4). Beim Verhalten ist keine 1-zu-1-Übersetzung möglich. In Verhaltensmodellen wie Story-Diagrammen, State Charts oder Aktivitätendiagrammen fehlen Sprachkonstrukte für Aktionen wie redirect.

kanonische Abbildung

Das Entwurfsmodell in Abb. 3.10, bestehend aus einem Klassen- und einem Story-Diagramm, ist eine von mehreren möglichen Implementierungen des in Abb. 3.9 definierten Entwurfs. Der Entwurf in Abb. 3.10 kann als kanonische Abbildung der DAL-Entwurfsbeschreibung aus Abb. 3.9 auf ein Entwurfsmodell ausgedrückt in Klassen- und Story-Diagrammen angesehen werden, weil sich der Entwurf von seiner DAL-Beschreibung strukturell nicht unterscheidet.

Aus der DAL-Entwurfsbeschreibung in Abb. 3.9 lassen sich jedoch auch an-

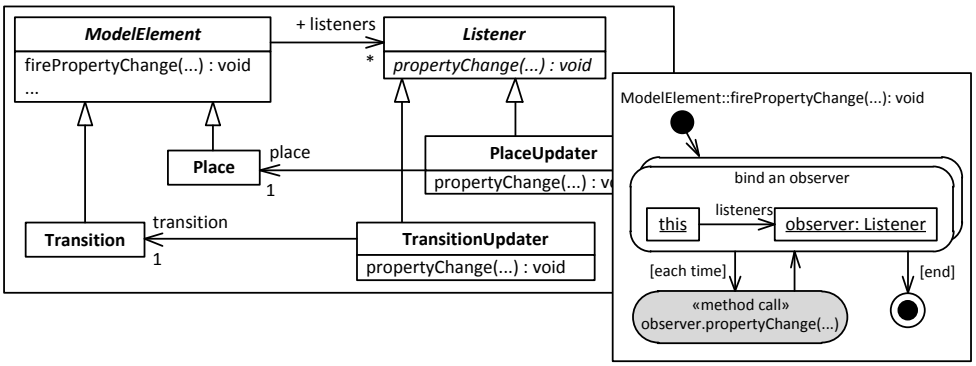


Abbildung 3.10: Kanonische Implementierungsvariante des Observer-Musters modelliert in Klassen- und Story-Diagrammen

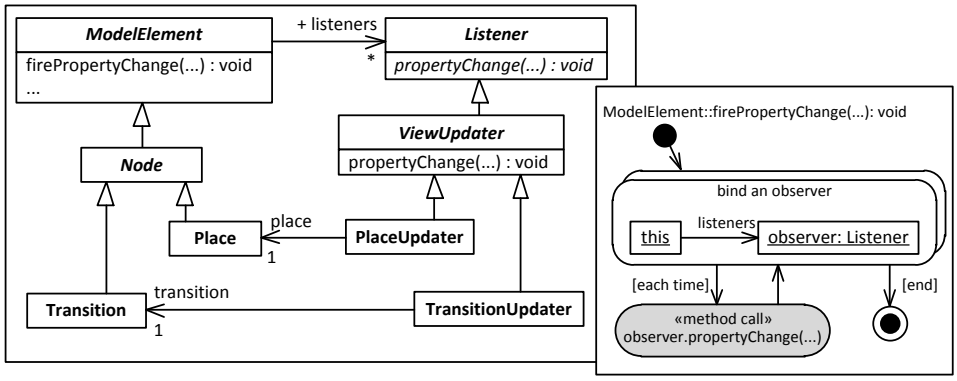


Abbildung 3.11: Nicht-triviale Implementierungsvariante des Observer-Musters modelliert in Klassen- und Story-Diagrammen

weitete zur DAL konforme Varianten

dere Varianten eines Entwurfsmodells ableiten, z.B. die Variante in Abb. 3.11. Hier kommen zusätzliche Klassen Node und ViewUpdater in den Klassenhierarchien vor, die zwar zu einer anderen Struktur des Klassenmodells führen, aber nicht von der mit dem Observer-Muster beschriebenen Entwurfslösung abweichen. Die Vererbungsbeziehungen in der DAL-Entwurfsbeschreibung stellen transitive Beziehungen dar[9]. Sie erlauben es, direkte Beziehungen in der Spezifikation durch indirekte in der Implementierung zu ersetzen. Zum Beispiel ist es nicht nötig, dass die Klassen Transition und Place direkt von ModelElement erben. Darum stellt das Entwurfsmodell in Abb. 3.11 eine gültige Implementierung der DAL-Entwurfsbeschreibung aus Abb. 3.9 dar.

Ähnlich verhält es sich bei Operationen. Eine Kompositionskante zwischen einer Operation und einer Klasse in der DAL besagt nur, dass in der Implementierung eine entsprechende Operation in einer entsprechenden Klasse zur Verfügung stehen soll (aufrufbar sein soll). Die Operation muss also nicht direkt in derselben Klasse implementiert werden, sondern kann stattdessen in einer der Oberklassen implementiert werden. Bei dem Beispiel in Abb. 3.11 wird die Operation propertyChange nicht wie in der DAL-Beschreibung angegeben direkt von den Klassen

[9]Die Transitivität der Vererbungsbeziehung sagt aus: erbt eine Klasse C von B und B von A, so erbt C auch von A (ist Unterklasse von A).

PlaceUpdater und TransitionUpdater implementiert, sondern von ihrer Oberklasse ViewUpdater.

Auf diese Weise stellt eine DAL-Entwurfsbeschreibung eine Implementierungsvariante eines Musters so abstrakt dar, dass bei der Abbildung auf ein Entwurfsmodell noch genügend Spielraum für Entwickler bleibt, die kanonische Implementierungsvariante an die vorliegende Situation individuell anzupassen und dennoch konform zur Musterspezifikation bzw. zur DAL-Entwurfsbeschreibung zu bleiben. Eine DAL-Entwurfsbeschreibung repräsentiert somit eine Vielzahl von dazu konformen Implementierungsvarianten in einem Entwurfsmodell.

3.3.4 Beschreibung, Spezifikation und Implementierungsvarianten von Entwurfsmustern

In der Abb. 3.12 sind die verschiedenen Repräsentationen eines Entwurfsmusters und ihr Zusammenhang zu unterschiedlichen Implementierungsvarianten in verschiedenen Sprachen (Notationen) veranschaulicht (Erweiterung der Abb. 3.5, S. 46). Die Kreise stellen Mengen von verschiedenen Modellen oder Musterrepräsentationen dar, während die Punkte für einzelne Modelle aus diesen Mengen stehen. Die Abbildung ist in vier Ebenen unterteilt. Jeder Ebene ist eine Sprache zugeordnet (links im Bild), in der eine Musterbeschreibung, Musterspezifikation oder Musterimplementierungsvariante ausgedrückt wird.

Musterbeschreibung

In der obersten, allgemeinsten Ebene ist eine informelle, natürlichsprachliche Musterbeschreibung wie sie in der Literatur vorkommt (vgl. Abb. 1.1, S. 4) als Punkt ⓐ dargestellt. Durch die sehr allgemeine Beschreibung des Musters kann es auf verschiedenste Weisen in einem Entwurfsmodell implementiert werden. Alle möglichen Entwurfsmodelle, auch ohne Musterimplementierung, werden in der untersten Ebene durch den größten Kreis in Abb. 3.12 repräsentiert. Alle Implementierungsvarianten zu einer Entwurfsmusterbeschreibung ⓐ sind durch den Projektionskegel und den orangenen Kreis in der untersten Ebene angedeutet.

Musterspezifikation

In der zweiten Ebene von oben ist eine Musterspezifikation ⓑ eingeordnet, welche die Entwurfslösung aus der Musterbeschreibung ⓐ formal in der PSL ausdrückt. Die Spezifikation könnte z.B. die aus Abb. 3.8 (S. 53) sein. Durch die Formalisierung legt man sich natürlich auf einen eingeschränkten Sprachumfang fest, wodurch nur ein Teil der Musterimplementierungsvarianten ausgedrückt werden kann, die von der informellen Musterbeschreibung ⓐ abgedeckt sind. Diese Einschränkung wird durch den kleineren Projektionskegel zur Musterspezifikation ⓑ und den kleineren, grünen Kreis in der untersten Ebene in Abb. 3.12 dargestellt.

DAL-Repräsentation

Bei Anwendung eines Musters legt ein Entwickler fest, welche Teile seines zu entwickelnden Softwaresystems welche Rollen der Musterspezifikation spielen sollen und bestimmt damit implizit mit, wie häufig die in Set Fragments vorkommenden Entwurfsstrukturen im Systementwurf vorkommen sollen. Z.B. wird so festgelegt, wie viele ConcreteObserver-Klassen bei der Implementierung des Observer-Musters an einer bestimmten Stelle im Entwurf vorkommen und wie diese heißen sollen. Durch diese Festlegung kommt man zu einer Implementierungsvariante ⓒ des spezifizierten Musters, die abstrakt in der DAL ausgedrückt werden kann. So eine Implementierungsvariante wird in der dritten Ebene in Abb. 3.12 als Punkt dargestellt und könnte wie in Abb. 3.9 (S. 54) aussehen. Andere Implementie-

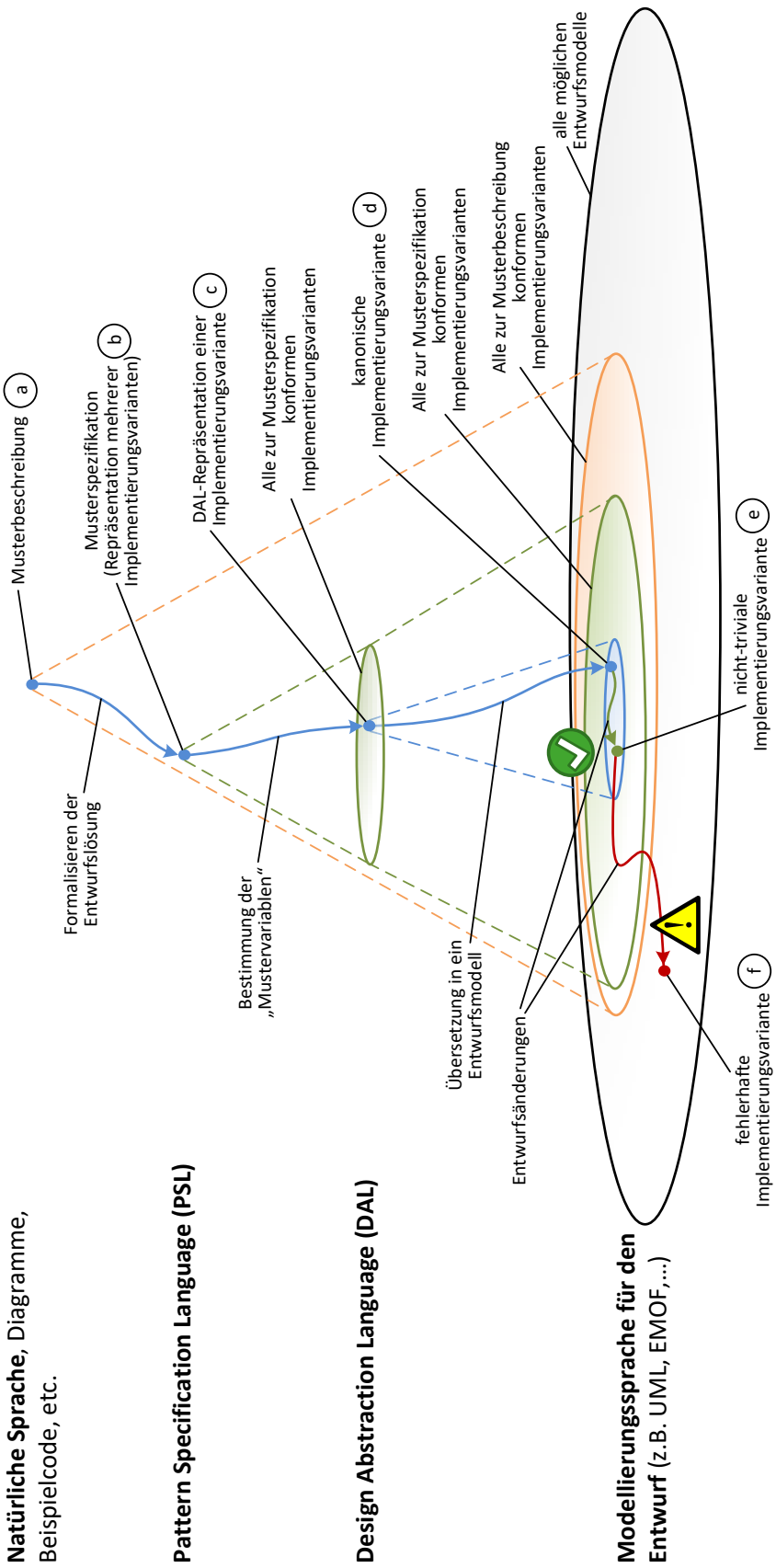


Abbildung 3.12: Abdeckung der Implementierungsvarianten zu einem Muster

rungsvarianten (dargestellt durch den grünen Kreis in dieser Ebene) erhält man durch Festlegen anderer Häufigkeiten für die in Set Fragments spezifizierten Entwurfsstrukturen sowie andere Namen für die Entwurfsteile.

Implementierung im Entwurfsmodell

Übersetzt man die in der DAL beschriebene Implementierungsvariante ⓒ in eine Modellierungssprache für den Systementwurf, so erhält man konkrete Musterimplementierungen in einem Entwurfsmodell wie in den Abb. 3.10 und 3.11 (S. 55). Diese Implementierungsvarianten werden durch die Punkte ⓓ und ⓔ in dem blauen Kreis in der untersten Ebene der Abb. 3.12 repräsentiert.

Konformität zur Spezifikation

Bei einer automatisierten Anwendung des Musters (Details in Kap. 5) wird bei meinem Ansatz nur die kanonische, also die naheliegendste, besonders einfache Implementierungsvariante ⓓ des Musters erzeugt (z.B. die in Abb. 3.10, S. 55). Entwickler haben danach die Möglichkeit, die Musterimplementierung in einem gewissen Rahmen ihren Wünschen und der vorliegenden Situation entsprechend anzupassen, solange die Implementierung noch zur Musterspezifikation konform bleibt. Das geänderte Entwurfsmodell ⓔ könnte wie in Abb. 3.11 (S. 55) aussehen. Solche geringfügig von der kanonischen Implementierungsvariante abweichenden Implementierungen des Musters werden durch den Validierungsmechanismus (Details in Kap. 6) als korrekt angesehen (in Abb. 3.12 durch den Haken bei dem Punkt ⓔ angedeutet). Entwurfsmodelle, in denen die in der Musterspezifikation beschriebene Entwurfsstruktur nicht mehr vorkommt, werden von dem Validierungsmechanismus als fehlerhaft angesehen (in der Abb. 3.12 durch das Warnsymbol bei dem Punkt ⓕ angedeutet). Weitere in Abb. 3.12 nicht dargestellte Implementierungsvarianten entstehen bei der Übersetzung eines Entwurfsmodells (z.B. ⓔ) in Quellcode, wobei die Variantenvielfalt erneut zunimmt.

3.4 Design Abstraction Language – Repräsentation eines objektorientierten Softwareentwurfs

Wie in Abschnitt 3.2 beschrieben ist die Musterspezifikationssprache zweigeteilt. Sie besteht u.a. aus der Design Abstraction Language (kurz: DAL) zur kompakten Repräsentation von Teilen eines objektorientierten Softwareentwurfs unabhängig von der für den Entwurf eingesetzten Modellierungssprache. In diesem Abschnitt stelle ich zunächst die wichtigsten Sprachkonstrukte der DAL vor. Dazu erkläre ich wie Klassenstrukturen und Objektinteraktionen wie sie in objektorientierten Entwurfsmustern wie denen von Gamma et al. [GHJV95] vorkommen in der DAL beschrieben werden. Außerdem stelle ich das Konzept der Subsysteme vor, welches z.B. die Spezifikation des Facade-Musters oder des MVC-Paradigmas ermöglicht. Die DAL-Semantik definiere ich durch eine exemplarische Abbildung der DAL auf Ecore und Story-Diagramme. Den vollen Umfang der DAL inklusive der abstrakten Syntax dokumentiere ich im Anhang A.

3.4.1 Klassenstrukturen

Objektorientierte Sprachen bieten Konzepte wie Klassen oder Typen, Operationen, Attribute, Vererbung, teilweise auch Assoziationen zwischen Klassen. Zur Beschreibung der Struktur objektorientierter Softwareentwürfe enthält die DAL

Abbildung 3.13: Konkreter Typ in DAL und Ecore

Abbildung 3.14: Beliebiger Typ in DAL und Ecore

für die wichtigsten dieser Konzepte entsprechende Sprachkonstrukte. Dabei habe ich mich vor allem auf solche Konzepte fokussiert, die in Entwurfsmustern der Gang of Four vorkommen [GHJV95].

Fokus auf Konzepte der GoF-Muster

Die DAL-Sprachkonstrukte sind möglichst einfach und abstrakt gehalten, damit eine in der DAL beschriebene Entwurfsstruktur möglichst kompakt ist und in möglichst viele objektorientierte Sprachen (z.B. UML oder Ecore) übersetzt werden kann. Die DAL bildet im Prinzip den kleinsten gemeinsamen Nenner. Es wird von vielen zur Beschreibung von Entwurfsmustern irrelevanten Details wie sie in objektorientierten Modellierungs- und Programmiersprachen vorkommen abstrahiert. Zum Beispiel werden Assoziationseigenschaften wie *subset*, *redefine* oder *ordered*, die Navigierbarkeit von Assoziationsenden und Profile bzw. Annotationen aus Sprachen wie der UML oder Ecore in der DAL nicht mitmodelliert. Ebenso werden aus objektorientierten Programmiersprachen wie Java, Scala, Kotlin, C++, C#, Eiffel oder Smalltalk bekannte Details wie statische bzw. Klassenoperationen und -attribute, Zeiger, Sichtbarkeiten (*public*, *protected*, *private*, *friend*), Konstanten, generische Typen bzw. Templates und Ausnahmen (Exceptions) weggelassen.

Reduktion auf das Wesentliche

Zur Beschreibung von Klassenstrukturen bietet die DAL Sprachkonstrukte für Typen (Klassen), primitive Datentypen (z.B. boolean), Operationen inklusive Parametern und Rückgabetyp, Attribute, Assoziationen und Vererbung. Damit deckt die Sprache einen Teil der in UML- und Ecore-Klassendiagrammen vorkommenden Sprachkonstrukte ab. Einen Teil dieses Sprachumfangs stelle ich im Folgenden vor. Die Semantik der eingeführten Sprachkonstrukte definiere ich durch ihre Abbildung auf Ecore. Alle Sprachkonstrukte und ihre Semantik sind im Anhang A.2 und A.3 beschrieben.

Typen werden auf Ecore-Klassen abgebildet (siehe Abb. 3.13) und repräsentieren eine Klasse oder eine Schnittstelle (wie z.B. in Java). Typen können konkret oder abstrakt sein (letztere werden analog zur UML kursiv dargestellt). Sollte es irrelevant sein, ob ein Typ konkret oder abstrakt ist, wird es als beliebig modelliert und gestrichelt dargestellt (siehe Abb. 3.14). In diesem Fall ist die Abbildung auf eine Ecore-Klasse nicht mehr eindeutig: Ein Typ kann auf eine konkrete oder auf eine abstrakte Ecore-Klasse abgebildet (übersetzt) werden. In beiden Fällen wäre die Konsistenz zur DAL bzw. einer Musterspezifikation gewahrt.

Typen

Die DAL bietet vier primitive Datentypen für Attribute und Signaturen von Operationen, nämlich boolean, integer, real und string. Sie werden auf die Ecore-Typen EBoolan, EInt, EDouble und EString abgebildet.

Aufgrund der in Abschnitt 3.3.2 genannten Mehrdeutigkeiten bei der Spezifikation von sich wiederholenden Klassenstrukturen mit Hilfe von Set Fragments stelle ich Operationen und Attribute anders als in UML-Klassendiagrammen in eigenen Knoten und nicht zusammen mit der zugehörigen Klasse im selben Rechteck dar.

Operationen

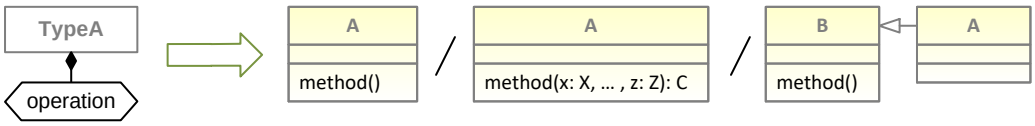


Abbildung 3.15: Operation und Komposition in DAL und Ecore

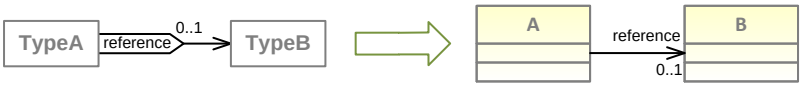


Abbildung 3.16: Unidirektionale Referenz in DAL und Ecore

Abbildung 3.17: Bidirektionale Referenz in DAL und Ecore

Komposition

Um dennoch die Zugehörigkeit einer Operation oder eines Attributs zu einem Typ visuell ausdrücken zu können, habe ich eine Kompositionskante eingeführt (Notation wie in der UML, siehe Abb. 3.15). Zur besseren Unterscheidung von Klassen werden Operationen als Sechseck dargestellt[10].

Signatur

Eine DAL-Operation lässt offen, welche Signatur eine Musterimplementierung haben soll. Dadurch lässt die Übersetzung einer DAL-Operation in eine Ecore-Methode viele Möglichkeiten offen, wovon einige in der Abb. 3.15 dargestellt sind. Sind keine Parameter in der DAL-Operation spezifiziert, können sie frei gewählt werden. Das Gleiche gilt für den Rückgabetyp. Sind Parameter in der DAL definiert (Notation laut Tab. A.1, S. 253), können in der Implementierung weitere ergänzt und die Reihenfolge geändert werden. Die Kompositionsbeziehung zwischen einem DAL-Typ TypeA und einer DAL-Operation fordert nicht zwingend, dass die zu TypeA gehörende Ecore-Klasse A eine der DAL-Operation entsprechende Methode in der Klasse A enthält. Die Methode kann auch in einer der Oberklassen liegen (3. Fall in Abb. 3.15).

Referenzen

Um Mehrdeutigkeiten im Zusammenhang mit Assoziationen und Set Fragments zu vermeiden, stelle ich die Rollen einer Referenz (analog zu Operationen) explizit in eigenen Knoten dar (siehe Abb. 3.16). Im Gegensatz zur UML werden in der DAL Rollen nicht auf der Seite der referenzierten Klasse, sondern auf der Seite der referenzierenden Klasse platziert. Das ermöglicht das Platzieren der Referenzen und der sie besitzenden Klassen im selben Set Fragment. Es werden sowohl unidirektionale als auch bidirektionale Beziehungen unterstützt (Abb. 3.16 und 3.17). DAL-Refernzen werden auf entsprechende Ecore-Referenzen abgebildet.

Spezialisierung

Die DAL bietet Vererbungsbeziehungen analog zur UML, wobei die Notation identisch ist. Bei der Abbildung einer DAL-Vererbungsbeziehung auf eine Vererbungsbeziehung in Ecore werden jedoch sowohl direkte als auch indirekte Vererbungsbeziehungen erlaubt (siehe Abb. 3.18).

Analog zu Spezialisierungsbeziehungen zwischen Klassen (den Vererbungsbeziehungen) bietet die DAL auch Spezialisierungsbeziehungen zwischen Operationen.

[10] Das erhöht die perzeptuelle Unterscheidbarkeit (siehe Anforderung *Notation* auf S. 45).

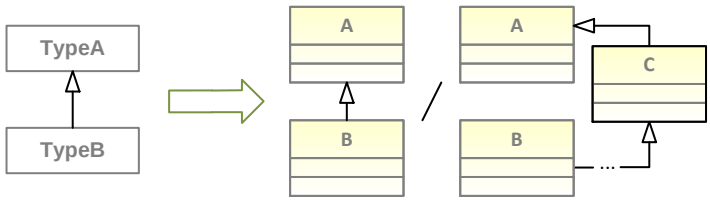


Abbildung 3.18: Spezialisierung bei Typen in DAL und Ecore

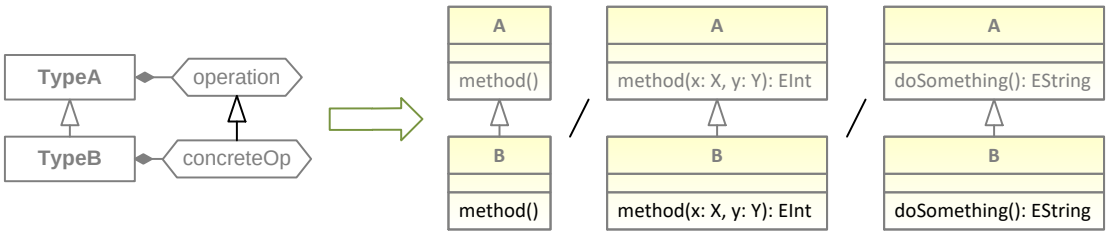


Abbildung 3.19: Spezialisierung bei Operationen in DAL und Ecore

So eine Beziehung drückt aus, dass eine Operation concreteOp eine andere Operation operation (aus einer Oberklasse) überschreibt (siehe Abb. 3.19) oder eine abstrakte Operation implementiert. Diese Art von Beziehungen wird benötigt, um auszudrücken, dass die in der DAL unvollständig spezifizierte Signatur einer Operation bei der Implementierung (Abbildung auf Ecore) von der Signatur einer bestimmten anderen Operation abhängt. So bestimmt z.B. in der Abb. 3.19 die zur Rolle operation gehörende Methode in der Klasse A die Signatur der zur Rolle concreteOp gehörenden Methode in der Klasse B. Weitere Beispiele hierzu sind im Anhang ab S. 254 beschrieben (vgl. Abb. A.3 bis A.6 auf S. 255 und Abb. A.7 bis A.10 auf S. 256).

Transitivität

Das Ausnutzen der Transitivität[11] von Spezialisierungs- und Kompositionsbeziehungen in der DAL erlaubt eine große Vielfalt an zur DAL-Spezifikation konformen Implementierungen in Ecore (Details dazu im Anhang ab S. 257).

3.4.2 Interaktionen

Entwurfsmuster wie die der Gang-of-Four [GHJV95] beschreiben neben der vorgeschlagenen Klassenstruktur auch das Verhalten der beteiligten Objekte. Dabei werden insbesondere die Interaktionen zwischen den Objekten beschrieben, also z.B. an wen Informationen oder Aufgaben weitergereicht oder durch wen Objekte erzeugt werden. Das tatsächliche Verhalten – z.B. anwendungsfallabhängige Berechnungen – wird in Mustern nicht beschrieben oder nur angedeutet.

Bei dem Observer-Muster z.B. (siehe Abb. 1.1, S. 4 oder Abb. 1.2, S. 5) wird beschrieben, dass ein Subject-Objekt bei Änderung seines beobachteten Zustands durch Aufruf einer update-Methode auf allen registrierten Observer-Objekten diese über die Änderung benachrichtigt. Was die einzelnen Observer-Objekte bei Aufruf der update-Methode tun, wird nur angedeutet. Sie reagieren nämlich auf die Änderung, z.B. aktualisieren sie die Darstellung des beobachteten Zustands

[11] Eine Relation $r \subseteq \Sigma \times \Sigma$ ist transitiv, wenn für alle $a, b, c \in \Sigma$ gilt: $r(a,b) \wedge r(b,c) \Rightarrow r(a,c)$.

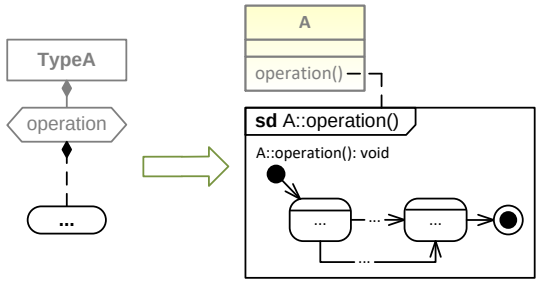


Abbildung 3.20: Aktion in DAL und Ecore

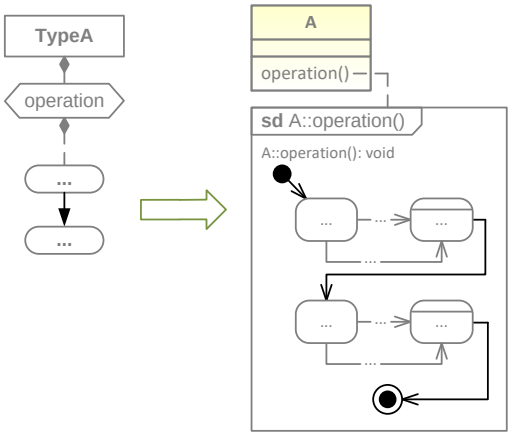


Abbildung 3.21: Kontrollfluss in DAL und Ecore

des Subjekts. Auch bei anderen Entwurfsmustern werden Objektinteraktionen in ähnlicher Weise beschrieben.

Um die Kernidee eines Entwurfsmusters zu erfassen, ist es also notwendig, auch diese für das Muster relevanten Interaktionen zwischen den Objekten – den Instanzen der im Entwurfsmuster beschriebenen Klassen – zu spezifizieren.

Aktionen

Zur kompakten Beschreibung des Verhaltens von Operationen habe ich das Sprachkonstrukt *Aktion* eingeführt. Eine Aktion beschreibt einen Schritt einer Operation, z.B. eine Interaktion zwischen zwei Objekten in Form eines Operationsaufrufs.

Bei objektorientierter Software gibt es eine Reihe von allgegenwärtigen Interaktionsarten zwischen Objekten. Im Wesentlichen zählen zu diesen Interaktionen das Erzeugen von Objekten, das Aufrufen von Operationen und das Lesen und Schreiben von Attributen bzw. das Erstellen und Entfernen von Verknüpfungen zwischen Objekten. Damit lässt sich bereits ein Großteil des in objektorientierten Entwurfsmustern beschriebenen Verhaltens ausdrücken. In meiner Musterspezifikationssprache habe ich Aktionen für genau diese Art von Interaktionen definiert. Die Auswahl hat jedoch nicht den Anspruch, vollständig zu sein, und kann in Zukunft erweitern werden, um z.B. die Kommunikation in nebenläufigen, verteilten Systemen zu beschreiben wie sie in anderen Entwurfsmustern vorkommen [SSRB00, BHS07a].

Eine DAL-Aktion wird als Rechteck mit abgerundeten Ecken und einem Label dargestellt (z.B. call in Abb. 3.22, S. 63 oder linkes in der Abb. 3.20). Durch Abstraktion von Details wie bestimmten Anweisungen und zugehörigem Kontrollfluss werden sehr kompakte Musterspezifikationen ermöglicht. Leider schränkt das gleichzeitig die Ausdrucksmächtigkeit der DAL auf die vordefinierten Aktionsarten ein. Bisher habe ich in der DAL neun Arten von Aktionen definiert (Details dazu später).

Komposition

Eine Aktion wird per gestrichelter Kompositionskante einer Operation zugeordnet (siehe Abb. 3.20). Das komplette Verhalten einer Operation, bestehend aus einer oder ggf. mehreren Aktionen, wird auf ein Story-Diagramm abgebildet, welches das Verhalten einer Ecore-Methode modelliert (Abb. 3.20 und 3.21). Dabei

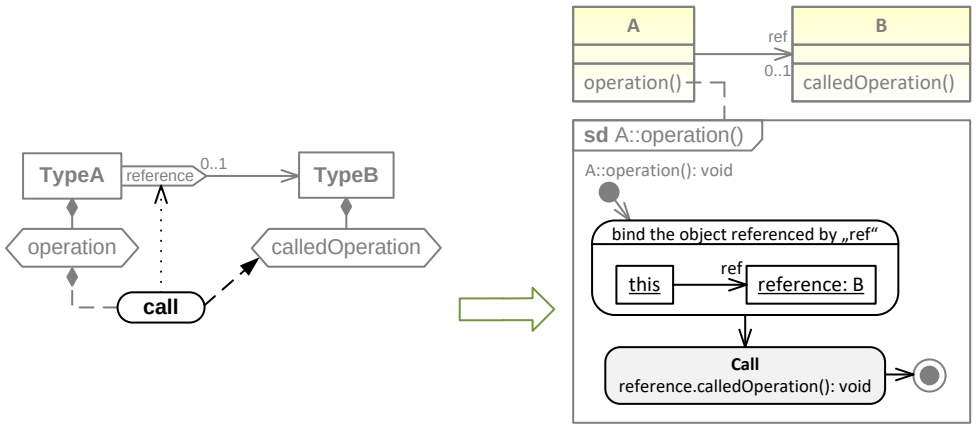


Abbildung 3.22: Aktion call in DAL und Ecore (a)

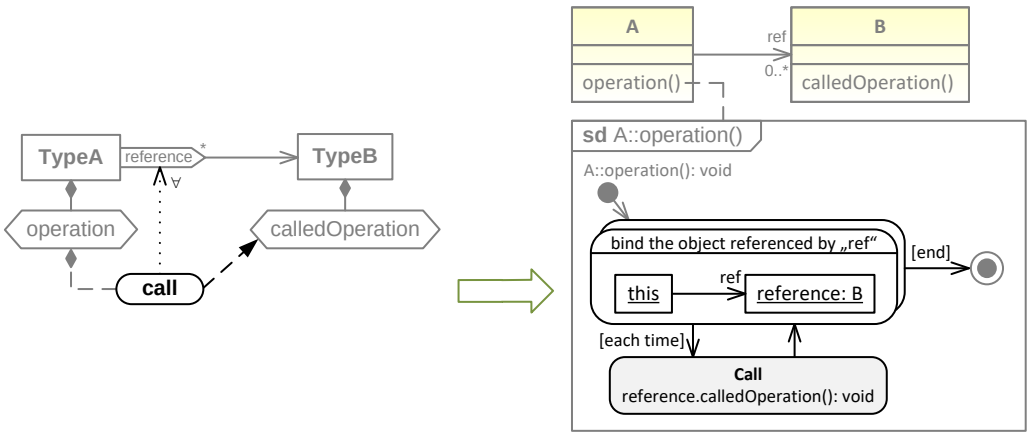


Abbildung 3.23: Aktion call in DAL und Ecore (b)

repräsentiert jede Aktion einen Teil des in einem Story-Diagramm implementierten Kontrollflusses. Eine DAL-Kontrollflusskante beschreibt die Ausführungsreihenfolge von Aktionen (Abb. 3.21). Komplexere Kontrollflussbeschreibungen waren bei den bisher betrachteten Entwurfsmustern (siehe Anhang B)) nicht nötig. Datenfluss wird bis auf Parameterübergaben und Variablenzugriffe (Details im Anhang A.2.2) nicht betrachtet.

Kontrollfluss

Neben dem Namen einer Aktion werden abhängig von der Art der Aktion zusätzliche Informationen benötigt. Zum Beispiel gibt ein gestrichelter Pfeil in der Abb. 3.22 bei einer call-Aktion die aufzurufende Operation an. Ein gepunkteter Pfeil gibt durch Verweis auf eine Referenz an, auf welchem Objekt die Operation aufzurufen ist. Bei einer create-Aktion wird der zu instanziierende Typ per gestricheltem Pfeil angegeben (siehe Abb. 3.24). Ergänzend dazu kann eine Variable spezifiziert werden (hier: result) , wo das erzeugte Objekt abgelegt wird, um in einer späteren Aktion weiterverwendet zu werden. Alle derartigen Parameter von Aktionen werden im Anhang A.2.2 beschrieben.

Parameter von Aktionen

Die kompakte Repräsentation eines Operationsaufrufs aus Abb. 3.22 wird in ein Story-Diagramm mit zwei Schritten (Story Nodes[12]) übersetzt. Im ersten Schritt

Semantik

[12] Aktivitätenknoten mit einem Story Pattern (siehe Abschnitt 2.4.2)

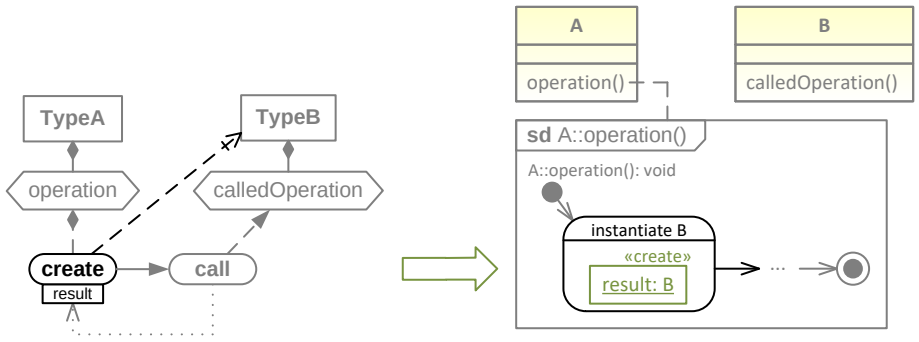


Abbildung 3.24: Aktion create in DAL und Ecore

wird per Graph Matching das referenzierte B-Objekt ermittelt und in der Objektvariable reference abgelegt. Im zweiten Schritt erfolgt auf diesem Objekt der Aufruf der Methode calledOperation. Hat die Referenz eine Multiplizität * oder 1..*, so werden diese zwei Schritte für jedes referenzierte Objekt wiederholt, im Story-Diagramm wird also eine Schleife wie in Abb. 3.23 modelliert (die Notation von Story-Diagrammen wird in Abschnitt 2.4.2 beschrieben).

Die create-Aktion aus Abb. 3.24 wird in ein Story-Diagramm übersetzt, wo eine Instanz der Klasse B erzeugt und in einer Objektvariable result abgelegt wird.

Arten von Aktionen

Insgesamt habe ich in der DAL folgende Aktionen vordefiniert. Dabei unterscheide ich in Anlehnung an die *Core Elemental Design Patterns* von Smith [Smi12] drei Arten von Operationsaufrufen: call, redirect und delegate.

call Aufruf einer Operation auf einem oder mehreren Objekten. Neben der aufrufenden und der aufgerufenen Operation gibt eine call-Aktion durch Verweis auf eine Variable auch an, auf welchen Objekten die Operation aufgerufen wird. So eine Variable kann ein Parameter der aufrufenden Operation sein, eine Referenz des aufrufenden Objekts oder das Ergebnis eines vorhergehenden Aufrufs.

redirect Eine Umleitung eines Aufrufs an eine andere Operation. Die redirect-Aktion ist eine spezielle call-Aktion, bei der alle ggf. vorhandenen Argumente der aufrufenden Operation an die aufgerufene Operation als Argumente übergeben werden.

delegate Eine Delegation eines Aufrufs an eine andere Operation. Die delegate-Aktion ist eine spezielle redirect-Aktion, die zusätzlich zu den Eigenschaften von redirect auch das Ergebnis des Aufrufs – sofern vorhanden – als eigenes Ergebnis zurückgibt. Eine Delegation erfolgt immer an höchstens ein Objekt.

create Klasseninstanziierung. Erzeugt ein neues Objekt vom angegebenen Typ.

produce Eine spezielle Variante der Klasseninstanziierung. Wie bei create wird auch hier ein Typ instanziiert, also ein Objekt erzeugt. Dieses Objekt wird jedoch auch als Ergebnis der ausführenden Operation zurückgegeben (semantisch äquivalent zu einer Hintereinanderausführung der Aktionen create und return, wobei die return-Aktion das Ergebnis der create-Aktion zurückgibt).

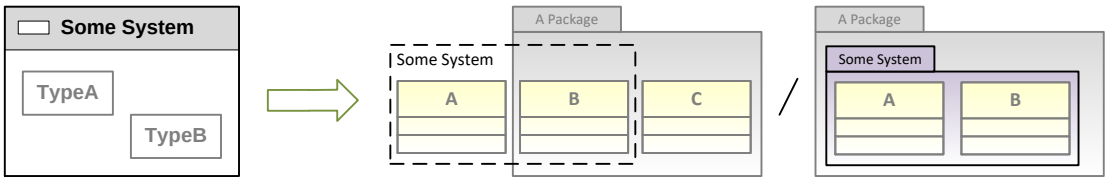


Abbildung 3.25: Subsystem in DAL und Ecore

read Lesezugriff. Auf eine Variable wie eine Referenz oder ein Attribut eines Objekts wird lesend zugegriffen.

return Rückgabe einer Variable. Verwenden einer Variable als Ergebnis eines Operationsaufrufs und somit ein spezieller Lesezugriff.

write Schreibzugriff. Auf eine Variable wie eine Referenz oder ein Attribut eines Objekts wird schreibend zugegriffen.

delete Löschen eines Variablenwerts. Ein spezieller Schreibzugriff, bei dem eine Referenz entfernt oder eine andere Variable auf null gesetzt wird.

Aufgrund der Vielfalt von Aktionen, ihren Parametern und ihrer vom Kontext[13] abhängigen Abbildung auf Story-Diagramme entsteht eine gewisse Komplexität bei der Übersetzung in Story-Diagramme. Auf die vielfältigen Notationsmöglichkeiten gehe ich im Anhang A.2.2 ein (siehe Tab. A.3, S. 259). Zu allen Aktionen definiere ich im Anhang A.3.2 eine Abbildung auf Story-Diagramme und beschreibe ab S. 260 wie eine Spezifikation unterschiedliche Verhaltensimplementierungen repräsentiert.

3.4.3 Subsysteme

Neben den für objektorientierte Entwurfsmuster typischen Klassenstrukturen werden bei einigen Mustern u.a. Teile eines Systems mit ihren Abhängigkeiten sowie Restriktionen bzgl. ihrer Kopplung beschrieben. Bei dem Facade-Muster zum Beispiel (siehe Abb. 8.7, S. 177) [GHJV95] wird ein hinter einer Fassade verborgenes Teilsystem bestehend aus zugehörigen Klassen beschrieben. Dieses Teilsystem wird vom übrigen System entkoppelt, sodass (bis auf wenige Ausnahmen) ausschließlich die Fassade darauf zugreift. Bei dem MVC-Paradigma [BMR+96, Fow02] werden die Teilsysteme Model, View und Controller beschrieben, die voneinander weitestgehend entkoppelt sind. Damit ich in meiner Musterspezifikationssprache auch solche Muster erfassen kann, habe ich Sprachkonstrukte für Subsysteme und ihre Abhängigkeiten ergänzt.

Gruppe von Typen

Als Subsystem verstehe ich schlicht einen Ausschnitt eines Softwaresystems. Dieser Ausschnitt wird durch einen Namen identifiziert und durch eine Gruppe von darin enthaltenen Typen oder anderen enthaltenen Subsystemen bestimmt. Ein Subsystem wird als Rechteck mit einem grauen Label dargestellt. Die darin enthaltenen Subsysteme und Typen werden innerhalb des Rechtecks dargestellt (links in Abb. 3.25 oder Tabelle A.4, S. 262).

[13] z.B. von unterschiedlicher Kardinalität beteiligter Assoziationen (vgl. Abb. 3.22 und 3.23)

Die ggf. hierarchieschen Subsysteme werden auf eine beliebige Auswahl von Ecore-Klassen abgebildet. Ein Subsystem kann logisch zusammenhängende Klassen gruppieren (1. Fall in Abb. 3.25) oder eine physikalische Entsprechung in der Implementierung haben, z.B. ein Paket (2. Fall in Abb. 3.25).

Abgrenzung zu Komponenten

Subsysteme ähneln hierarchischen Komponenten ([Szy98, OMG11b]). Allerdings sind für die Teilsysteme Model, View und Controller des MVC-Paradigmas anders als bei Komponenten keine Schnittstellen definiert und das hinter einer Fassade verborgene Subsystem wird lediglich durch eine Gruppe von Klassen bestimmt. Zwecks breiterer Anwendbarkeit habe ich mich für das allgemeinere Konzept der Subsysteme entschieden. Durch dieses sehr allgemeine Verständnis von einem Subsystem lassen sich die durch Subsysteme beschriebenen hierarchischen Strukturen auf eine Vielzahl von Konzepten abbilden, z.B. auf hierarchische Module, Komponenten, Plug-ins, Bibliotheken, Architekturschichten, Paket- oder Verzeichnisstrukturen.

Diese Art der Abstraktion erlaubt es mir, Abhängigkeiten und Kopplungsrestriktionen für beliebige solcher Subsysteme (zumindest sehr abstrakt) zu beschreiben. Die Spezifikation solcher Restriktionen wird im Abschnitt 3.5.3 ab Seite 72 beschrieben.

3.5 Pattern Specification Language – Repräsentation von Variabilität und Regeln

Im vorhergehenden Abschnitt habe ich meinen Ansatz zur kompakten Spezifikation von objektorientierten Entwurfslösungen durch Modellieren von abstrakten Implementierungsvarianten eines Musters vorgestellt. Im Folgenden stelle ich vor wie sich ein so beschriebener Softwareentwurf mit Hilfe zusätzlicher Sprachkonstrukte der Pattern Specification Language (kurz: PSL) zu einer Musterspezifikation erweitern lässt.

Zur Repräsentation von Variabilität führe ich Set Fragments ein, mit denen sich mehrere Implementierungsvarianten in nur einer Spezifikation kompakt beschreiben lassen. Ich stelle Kopplungsrestriktionen vor – Sprachkonstrukte zur Definition von erwarteten und verbotenen Abhängigkeiten in Musterimplementierungen. Ergänzend dazu erläutere ich wie nicht formalisierbare Teile einer Entwurfslösung, insb. manuell durchzuführende Aufgaben, in Musterspezifikationen erfasst werden.

3.5.1 Muster, Musterspezifikationen und Musterspezifikationskataloge

Neben den Sprachkonstrukten zur Beschreibung der in einem Entwurfsmuster vorgeschlagenen Lösung habe ich einen Weg zur Repräsentation der Entwurfsmuster und ihrer Spezifikationen ergänzt. Diese werden jedoch ausschließlich in abstrakter Syntax erfasst (siehe Anhang A.1, Abb. A.1, S. 250).

Entwurfsmuster

Ein Entwurfsmuster wird explizit modelliert (DesignPattern), erhält einen eindeutigen Namen und eine informelle Beschreibung, wo sich z.B. ein Verweis auf die Literatur mit der ursprünglichen Musterbeschreibung befinden kann.

Die Entwurfslösung eines Musters wird mit Hilfe der DAL beschrieben und in einer Musterspezifikation (PatternSpecification) zusammengefasst. Mit den DAL-

Musterspezifikation

Sprachkonstrukten (siehe Abschnitt 3.4) lassen sich bereits mehrere ähnliche Implementierungsvarianten eines Musters mit nur einer DAL-Struktur repräsentieren. Es kommt allerdings vor, dass es mehr als eine Variante eines Musters gibt, die sich strukturell und evtl. auch in ihrem Verhalten deutlich unterscheiden. Zum Beispiel gibt es zwei Lösungsvarianten bei dem Muster Composite [GHJV95]: mit und ohne einer Component-Klasse. Jede dieser Varianten wird bei meinem Ansatz durch eine eigene Spezifikation repräsentiert. Zu einem Entwurfsmuster kann es also mehrere Musterspezifikationen geben. Jede Musterspezifikation hat einen Namen, eine informelle, textuelle Erläuterung der modellierten Lösungsvariante sowie die ausmodellierte Entwurfsstruktur samt zugehörigem Verhalten.

Musterkatalog

Mehrere Entwurfsmustermodelle und ihre Musterspezifikationen fasse ich in Musterkatalogen zusammen. Solche Kataloge lassen sich durch Kategorien strukturieren, in welche die Muster eingeordnet werden. So lassen sich Muster in die Kategorien *Creational*, *Structural* und *Behavioral* der Gang-of-Four [GHJV95] einordnen oder anderen Kategorien zuordnen, z.B. Kommunikation, Ressourcenmanagement, verteilte Systeme, parallele Ausführung und ähnliche. Ein Muster kann mehreren Kategorien gleichzeitig zugehören.

3.5.2 Set Fragments – Wiederkehrende Entwurfsstrukturen

Bei vielen Entwurfsmustern dürfen sich Teile der beschriebenen Entwurfslösung in den Musterimplementierungen beliebig häufig wiederholen. Zum Beispiel kann es beliebig viele konkrete Unterklassen der Observer-Klasse geben, die alle eine Referenz auf das beobachtete Subjekt und eine Implementierung der update-Operation besitzen (siehe Abb. 1.3, S. 5). Ebenso gibt es bei dem Muster Abstract Factory (siehe Abb. 3.3, S. 43) beliebig viele Produktfamilien mit je einer zugehörigen Fabrik zur Erzeugung der Produkte dieser Familie und beliebig viele Produkte. Damit man nicht für jede beliebige Anzahl solcher Strukturen eine eigene Musterspezifikation erstellen muss, habe ich *Set Fragments* eingeführt.

Variantenvielfalt

Ein Set Fragment kennzeichnet einen zusammenhängenden Ausschnitt einer Musterspezifikation und definiert somit einen bestimmten Ausschnitt einer Entwurfslösung, der in jeder zugehörigen Musterimplementierung mindestens ein Mal (0 Vorkommen waren bei keinem der betrachteten Muster nötig), aber beliebig oft als Ganzes vorkommt. Somit lassen sich alle Implementierungsvarianten, die sich nur in der Anzahl des Auftretens des Entwurfslösungsausschnitts unterscheiden, in nur einer Musterspezifikation zusammenfassen.

Notation

Jedes Set Fragment wird als Rechteck dargestellt, erhält zur Identifizierung einen eindeutigen Namen und wird mit einem Label mit dem Schlüsselwort set und seinem Namen versehen (siehe Abb. 3.26 und 3.27). Ein Set Fragment umschließt die als Knoten dargestellten Teile des Entwurfs[14].

Konsistenz zur Spezifikation

Bei Musteranwendung können die innerhalb eines Set Fragments spezifizierten Entwurfsteile *als Ganzes* vervielfältigt werden. Sie dürfen also nur zusammen auftreten und müssen wie spezifiziert miteinander in Beziehung stehen. Sind wie bei dem Beispiel aus Abb. 3.26 Set-Fragment-übergreifende Beziehungen Teil der Spezifikation (z.B. beginnt die subject-Referenz innerhalb des Set Fragments observers

[14] Z.B. Typen, Operationen, Referenzen, Attribute, Aktionen, Aufgaben.

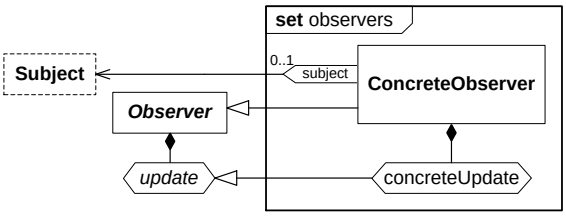


Abbildung 3.26: Ausschnitt der Spezifikation des Observer-Musters

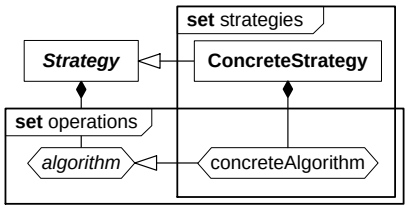


Abbildung 3.27: Ausschnitt der Spezifikation des Strategy-Musters

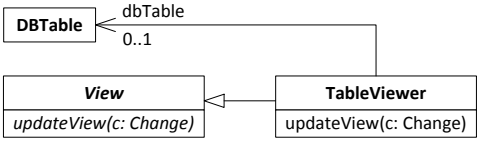


Abbildung 3.28: Zur Spezifikation aus Abb. 3.26 passende Entwurfsstruktur (a)

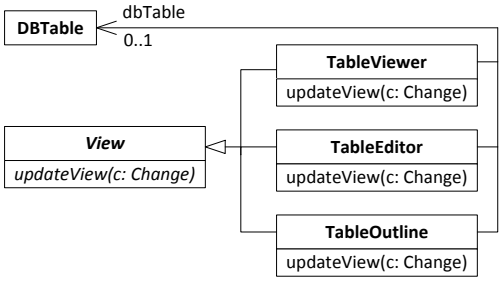


Abbildung 3.29: Zur Spezifikation aus Abb. 3.26 passende Entwurfsstruktur (b)

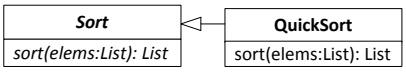


Abbildung 3.30: Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (a)

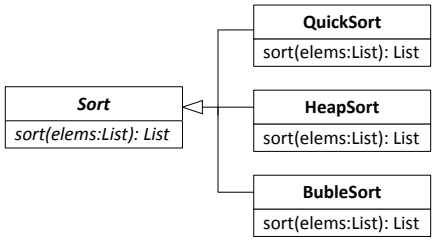


Abbildung 3.31: Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (b)

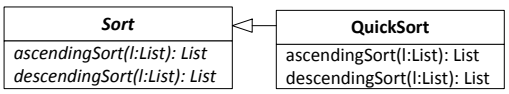


Abbildung 3.32: Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (c)

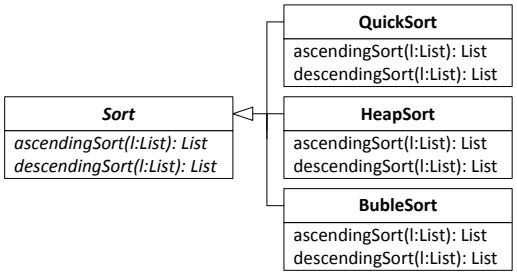


Abbildung 3.33: Zur Spezifikation aus Abb. 3.27 passende Entwurfsstruktur (d)

und endet in der Klasse Subject außerhalb des Set Fragments), müssen auch diese mitvervielfältigt werden.

Weil die Entwurfsteile innerhalb eines Set Fragments bei Musteranwendung mindestens ein Mal vorhanden sein müssen, stellt die Musterimplementierung in Abb. 3.28 die minimale Implementierung der Spezifikation aus Abb. 3.26 dar. Jedoch entspricht auch die Implementierung aus Abb. 3.29 der Spezifikation, weil hier je eine ConcreteObserver-Klasse nur zusammen mit der zugehörigen concreteUpdate-Operation und der subject-Referenz vorliegt.

Beispiele

Die wiederholbaren Entwurfsstrukturen in Entwurfsmustern sind nicht immer unabhängig von anderen sich wiederholenden Strukturen. Darum erlaube ich auch die Überschneidung von Set Fragments wie sie in der Abb. 3.27 dargestellt ist. Hier ist ein Teil des Strategy-Musters spezifiziert, bei dem es beliebig viele ConcreteStrategy-Klassen und beliebig viele algorithm-Operationen geben darf. Die im Schnitt der beiden Set Fragments operations und strategies liegende Operation concreteAlgorithm muss dabei für jede ConcreteStrategy-Klasse und jede algorithm-Operation erzeugt werden. Beispiele für die sich aus der Spezifikation ergebenden Implementierungsvarianten sind in den Abb. 3.30 bis 3.33 dargestellt.

Überschneidung

Um die Komplexität der Musterspezifikationssprache zu beschränken und es Entwicklern einfacher zu machen, zu einer Musterspezifikation mit überschneidenden Set Fragments passende Implementierungsvarianten zu erkennen, erlaube ich keine Überschneidungen eines Set Fragments mit mehr als einem anderen Set Fragment, ohne, dass eines davon komplett in einem anderen enthalten ist. Set Fragments dürfen jedoch beliebig tief ineinander geschachtelt werden.

Schachtelung

Verlaufen Kanten (z.B. Referenzen oder Vererbungsbeziehungen) zwischen zwei disjunkten Set Fragments (d.h. sich weder überschneidenden, noch ineinander enthaltenden Set Fragements), dann ist ihre Bedeutung nicht eindeutig definiert. Z.B. ist in Abb. 3.7 (S. 52) für die subject-Referenz zwischen den Set Fragments subjects und observers nicht klar, welche der beliebig vielen ConcreteObserver-Klassen mit welcher der beliebig vielen ConcreteSubject-Klassen per subject-Referenz verbunden werden soll. Das Ziel der Referenz ist nicht eindeutig, weil es zu beiden Set Fragments unabhängig voneinander beliebig viele Vorkommen geben kann. So sind Konstellationen mit nur einer ConcreteObserver-Klasse, aber zwei ConcreteSubject-Klassen und damit zwei potentiellen Zielen einer subject-Referenz möglich. Der Ersteller einer Musterspezifikation trägt die Verantwortung dafür, derartige Kanten (bei meinen Spezifikationen betraf das nur Referenzen und Aktionen) als nicht generierbar zu markieren. Gleichzeitig muss der Anwender eines Musters solche Uneindeutigkeiten auflösen. Bei dem beschriebenen Beispiel könnte z.B. eine zweite subject-Referenz in der einzigen ConcreteObserver-Klasse ergänzt werden, damit je eine Referenz je eine der beiden ConcreteSubject-Klassen referenziert.

Kanten zwischen disjunkten Set Fragments

Semantik von Set Fragments

Im Gegensatz zu den in Abschnitt 3.4 eingeführten DAL-Sprachkonstrukten haben Set Fragments keine direkte Entsprechung im Entwurfsmodell (in Klassen- und Story-Diagrammen). Darum definiere ich ihre Semantik nicht durch eine Abbildung auf Klassen- und Story-Diagramme, sondern durch die Definition aller erlaubten Abbildungen einer Musterspezifikation (mit Set Fragments) auf eine

DAL-Repräsentation (ohne Set Fragments) einer von vielen möglichen Implementierungsvarianten. Das entspricht einem Sprung von Ebene 1 (oben) zu Ebene 2 (Mitte) in Abb. 3.5 (S. 46) und beschreibt eine sogenannte *Auffaltung* einer Musterspezifikation.

Auffaltung

Ich definiere eine Auffaltung als eine Folge von drei speziellen Operationen, welche ich formal durch mathematische Abbildungen zwischen Graphen definiere. Jede dieser Abbildungen entspricht einer Graphtransformation. Darum stütze ich mich bei meiner Formalisierung auf existierende Graphtransformationstheorien und orientiere mich dabei an den von Ehrig et al. [EEPT06, Kap. 2, 8, 13] formulierten Definitionen von typisierten, attributierten Graphen mit und ohne Vererbung. Indem ich die Abbildungen auf beliebigen typisierten, attributierten Graphen definiere, sind die Auffaltung und die Semantik von Set Fragments unabhängig von meiner Musterspezifikationssprache definiert und auf andere Sprachen übertragbar.

Die drei Auffaltungsoperationen bzw. Abbildungen sind *instance*, *add* und π. Sie werden in einer bestimmten Reihenfolge angewandt (siehe Abb. 3.34 und Definition A.15, S. 289) und führen zu einer DAL-Repräsentation einer zur Musterspezifikation konformen Implementierungsvariante.

Instanziierung

Die erste Abbildung *instance* erzeugt eine 1-zu-1-Kopie der Musterspezifikation, in welcher die Set Fragments durch sogenannte Set-Fragment-Instanzen ersetzt werden (siehe Definition A.10, S. 284). Jede Set-Fragment-Instanz repräsentiert eines von ggf. mehreren Vorkommen des in einem Set Fragment spezifizierten Teilgraphen, inklusive ein- und ausgehender Kanten. Die Abbildung *instance* wird in Abb. 3.35 für eine Musterspezifikation mit Set Fragments (einen Graphen G_S) und eine DAL-Repräsentation einer Musterimplementierung mit Set-Fragment-Instanzen (einen Graphen G_I) exemplarisch skizziert. Bei diesem Schritt bzw. dieser Abbildung spreche ich von Instanziierung.

Instanzen von Set Fragments ergänzen

Als Nächstes können zusätzliche Vorkommen der in einem Set Fragment spezifizierten Teilgraphen ergänzt werden. Jede Anwendung einer *add*-Operation (siehe Definition A.12, S. 285) erzeugt eine zusätzliche Set-Fragment-Instanz. Dabei wird eine bereits vorliegende Set-Fragment-Instanz repliziert wie es in der Abb. 3.36 für die Set-Fragment-Instanz i dargestellt ist. Ein Entwickler entscheidet, ob und wie häufig eine *add*-Operation mit welcher Set-Fragment-Instanz als Vorbild angewandt wird, und legt so die gewünschte Implementierungsvariante fest. Einige Anwendungsbeispiele sind im Anhang in Abb. A.32 (S. 288) skizziert.

Projektion

Abschließend wird durch eine Projektion des erzeugten Graphen auf die DAL bzw. durch Weglassen der Set-Fragment-Instanzen (nur die Markieren, nicht die enthaltenen Teilgraphen) durch die Abbildung π (siehe Definition A.11, S. 284) die endgültige DAL-Repräsentation der Implementierungsvariante erzeugt. Dieser Schritt ist in der Abb. 3.37 dargestellt.

Die Semantik von Set Fragments definiere ich als die Menge aller möglichen Auffaltungen (siehe Definition A.17, S. 290). Die vollständige Formalisierung ist dem Anhang A.3.3 (S. 280 ff.) zu entnehmen.

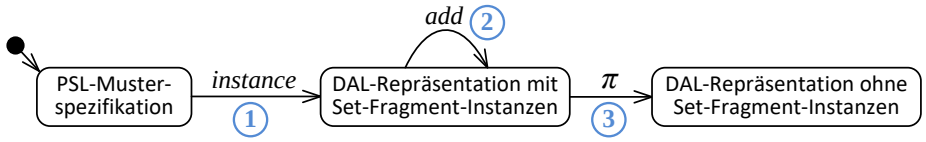


Abbildung 3.34: Abbildungen (bzw. Operationen) bei einer Auffaltung

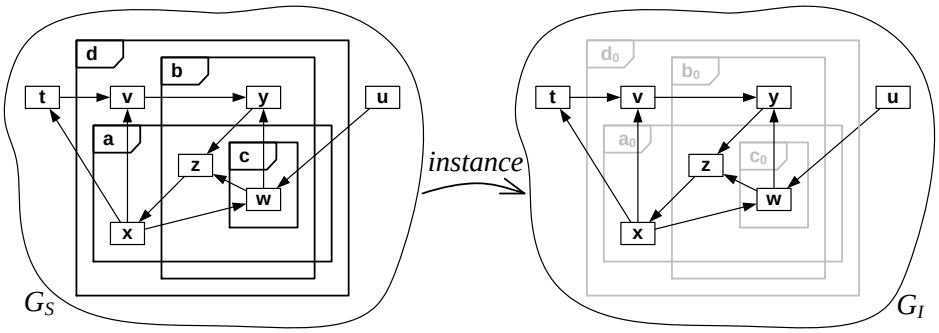


Abbildung 3.35: Instanziierung – Abbildung *instance*

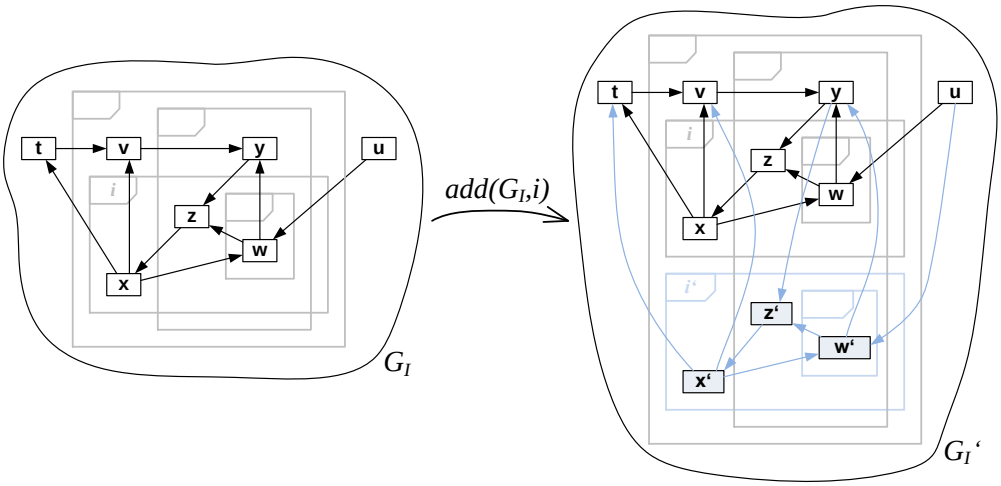


Abbildung 3.36: Hinzufügen einer Set-Fragment-Instanz – Abbildung *add*

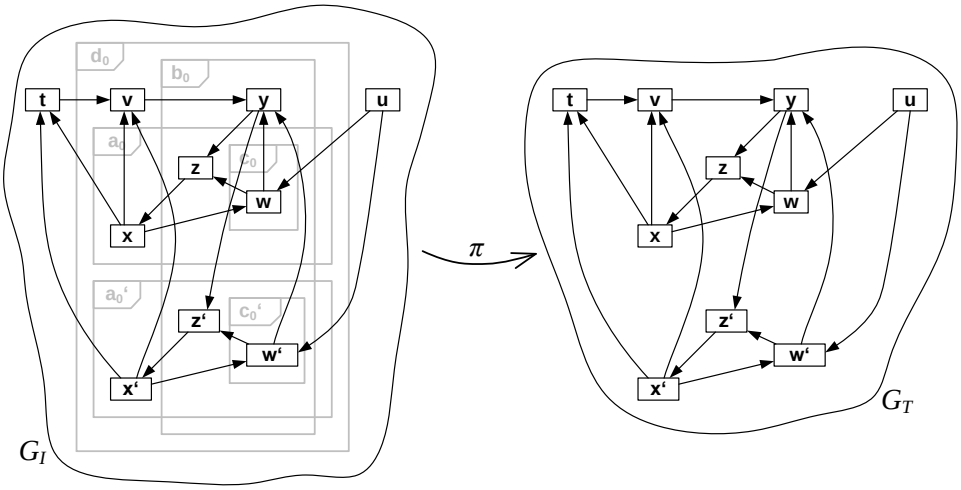


Abbildung 3.37: Projektion – Abbildung π

3.5.3 Bedingungen zu Abhängigkeiten und Kopplung

Die PSL bietet Sprachkonstrukte zur Beschreibung gewollter und ungewollter Abhängigkeiten zwischen Teilen eines Softwaresystems. Mit Hilfe solcher Sprachkonstrukte kann z.B. bei dem Facade-Muster (siehe Abb. 8.7, S. 177) [GHJV95] spezifiziert werden, dass das hinter der Fassade verborgene Teilsystem von dem übrigen System entkoppelt sein soll. Bei dem MVC-Paradigma kann spezifiziert werden, dass die Teilsysteme Model, View und Controller voneinander weitestgehend zu entkoppeln sind. Die Begriffe Abhängigkeiten und Kopplung verwende ich in diesem Zusammenhang synonym.

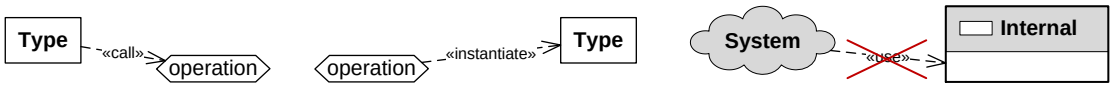


Abbildung 3.38: Drei Beispiele für Kopplungsregeln

Kopplungskanten

In welcher Form Teile eines Softwaresystems voneinander abhängig sein sollen und wie sie es nicht sein dürfen, beschreibe ich durch Kopplungskanten zwischen Subsystemen, Typen, Operationen, Referenzen und Attributen (siehe Abb. 3.38). Eine Kopplungskante geht immer von einem Subsystem, einem Typen oder einer Operation aus. Dargestellt wird eine Kopplungsbeziehung als gestrichelte Linie mit offener Pfeilspitze und einem die Art der Kopplung angebenden Label. Zu vermeidende Abhängigkeiten werden als durchgestrichene Kanten dargestellt (weitere Beispiele in Tab. A.6, S. 264).

Umgebung

Zur Beschreibung von Mustern wie dem Facade-Muster, wo das hinter einer Fassade verborgene Teilsystem von dem übrigen System entkoppelt sein soll, habe ich das Konzept der *Umgebung* (Environment) eines in einer Musterspezifikation beschriebenen Teilsystems eingeführt. Dargestellt wird die Umgebung durch ein Wolkensymbol mit einem wählbaren Label (rechts in Abb. 3.38). Unter einer Umgebung verstehe ich ein spezielles Subsystem, welches alle Typen repräsentiert, die in der Musterspezifikation nicht explizit als Typ oder durch ein anderes Subsystem modelliert sind. Die Umgebung beschreibt also das gesamte Softwaresystem bis auf die in der Musterspezifikation explizit modellierten Teile des Softwaresystems. Bei dem Facade-Muster ist die Umgebung also das übrige System, welches von dem beschriebenen Teilsystem durch eine Fassade entkoppelt werden soll (vgl. Abb. 8.7 und 8.8, S. 177).

Arten der Kopplung

Ich unterscheide in meiner Musterspezifikationssprache zwischen verschiedenen Arten der Kopplung. So unterscheide ich z.B. Lese- und Schreibzugriffe auf

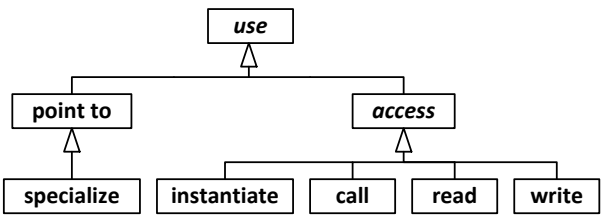


Abbildung 3.39: Hierarchie der Kopplungsarten

Variablen wie Attribute und Referenzen, Aufrufe von Operationen, Vererbungsabhängigkeiten von Typen (erben von) und Instanziieren von Typen.

use Drückt jede Art der Kopplung oder Abhängigkeit aus, insbesondere Folgende.

point to Jeder Verweis auf bzw. jede Verwendung von einem Typen. Ein Beispiel ist die Angabe eines Typen als Parametertyp oder Rückgabetyp einer Operation.

specialize Das Erben von einem anderen Typen bzw. das Verwenden eines Typen als Obertyp. Ist eine spezielle Form der *point-to*-Kopplung.

access Fasst alle Zugriffsarten auf Typen, Operationen, Attribute und Referenzen zusammen. Dazu zähle ich die Folgenden.

instantiate Zugriff auf einen Typ durch Instanziierung.

call Zugriff auf eine Operation durch Aufruf.

read Lesender Zugriff auf ein Attribut oder eine Referenz, z.B. durch die Übergabe eines Wertes als Argument eines Operationsaufrufs.

write Schreibender Zugriff auf ein Attribut oder eine Referenz, z.B. durch Zuweisung eines neuen Wertes bei Attributen oder durch Ändern einer Verknüpfung zwischen zwei Objekten bei Referenzen.

Die in der Musterspezifikationssprache verwendeten Arten der Kopplung sind hierarchisch strukturiert. Ihre Hierarchie ist in der Abb. 3.39 als Klassendiagramm dargestellt. Die Kopplungsarten *use* und *access* sind hier (kursiv) als abstrakte Klassen dargestellt, weil sie nur zum Zusammenfassen der spezielleren Kopplungsarten dienen.

indirekte Abhängigkeiten

Damit z.B. Lesezugriffe u.a. zwischen Subsystemen oder Typen beschrieben werden können, ohne im Detail die zugreifenden Operationen und die Referenzen und Attribute, auf die zugegriffen wird, zu spezifizieren, können die Kopplungsbeziehungen auch auf höherer Abstraktionsebene zwischen Typen und Subsystemen beschrieben werden (Details dazu und alle gültigen Kombinationen von Quelle und Ziel einer Kopplungskante sind dem Anhang A.2.5 zu entnehmen, insb. Tab. A.8).

Spezifikationsbeispiele

Durch Erweitern der Musterspezifikationssprache um Subsysteme und Kopplungsbeziehungen lassen sich Muster wie Facade [GHJV95] und Model-View-Controller (MVC) [BMR+96, Fow02] spezifizieren. Die drei Systemteile Model, View und Controller des MVC-Paradigmas lassen sich als Subsysteme ausdrücken (siehe Abb. 3.40). Durch diverse Kopplungskanten lässt sich nun beschreiben, dass ein Controller zwar Zugriff auf das Modell und die View erhält, diese aber unabhängig vom Controller sein sollen. Die View erhält Lese-, aber keinen Schreibzugriff auf das Modell. Das Modell wiederum ist von der View völlig unabhängig und darf mit Ausnahme des Controllers von keinem anderen Teil des Softwaresystems modifiziert werden (kein Schreibzugriff von der Umgebung aus).

Das Facade-Muster lässt sich spezifizieren, indem wie in der Abb. 3.41 dargestellt ein Typ Facade mit einer beliebigen Anzahl von request-Operationen spezifiziert wird, die auf ein Subsystem zugreifen, um die eigentliche Funktionalität zu

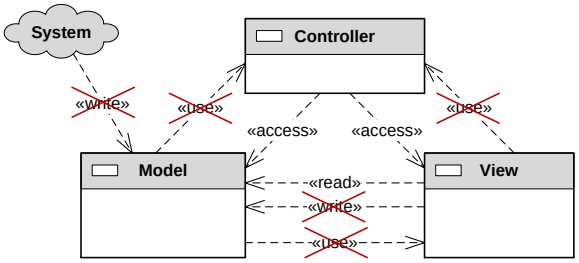


Abbildung 3.40: Spezifikation des MVC-Entwurfsparadigmas

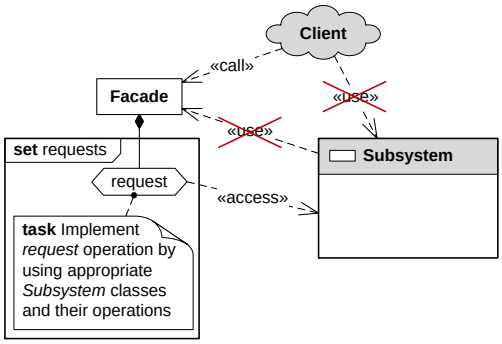


Abbildung 3.41: Spezifikation des Entwurfsmusters Facade

realisieren. Das sonstige Softwaresystem, beschrieben durch die Umgebung Client, ruft zwar Operationen der Fassade auf, hat jedoch keinerlei Abhängigkeiten zu dem dahinter verborgenen Subsystem.

3.5.4 Entwurfsaufgaben für nicht formalisierbare Teile von Entwurfslösungen

Jede Domänen-spezifische Sprache ist zwecks Fokussierung auf das Wesentliche und zwecks vereinfachter Repräsentation des Modellierten in ihrer Ausdrucksmächtigkeit beschränkt. Dasselbe gilt auch für meine Musterspezifikationssprache. Als Konsequenz daraus lassen sich nicht alle Aspekte eines Entwurfsmusters in der Musterspezifikationssprache ausdrücken.

Meine Musterspezifikationssprache hat nicht zum Ziel, ein Entwurfsmuster komplett und mit allen Details zu erfassen. Ich gehe davon aus, dass Entwickler zusätzlich zu den formalen Spezifikationen der Entwurfsmuster auch die zugehörige informelle Beschreibung der Muster zur Verfügung haben. Dennoch soll eine Musterspezifikation die wesentliche Lösungsidee eines Musters möglichst vollständig erfassen. Einige Konzepte hinter einem Entwurfsmuster scheinen sich jedoch nicht formal ausdrücken zu lassen.

nicht alles formalisierbar

Zum Beispiel wird bei dem Observer-Muster (siehe Abb. 1.1 auf S. 4) beschrieben, dass das beobachtete Subjekt einen von den Beobachtern beobachtbaren Zustand haben soll, der von den Beobachtern in ihrer update-Operation abgefragt wird. Was der Zustand ist, hängt erheblich vom Anwendungsfall ab und könnte durch den Wert eines Attributs, einer Referenz oder einer evtl. komplexen Kombination aus mehreren solchen Eigenschaften zusammengesetzt sein. Aus diesem Grund wird der Zustand bei der Musterbeschreibung absichtlich nicht näher beschrieben. Ebenso wird aufgrund der Abhängigkeit vom Anwendungsfall nicht näher beschrieben, wie die Beobachter bei Aufruf ihrer update-Operation auf beobachtete Zustandsänderungen reagieren oder wann und bei welchen Subjekten sich die Beobachter registrieren und deregistrieren.

Für das Verstehen einer Musterspezifikation und zur vollständigen, korrekten Anwendung eines Musters sind solche Informationen wichtig. Sie beschreiben wichtige Teile der Entwurfslösung hinter einem Muster. Damit diese Informationen

Abbildung 3.42: Notation von Entwurfsaufgaben

Hinweise für Entwickler

nicht verloren gehen oder übersehen werden, biete ich in meiner Musterspezifikationssprache die Möglichkeit, zusätzliche, informell in Textform erfasste Informationen an den Rollen einer Musterspezifikation zu hinterlegen. Diese Informationen werden in Form von Entwurfsaufgaben formuliert (TaskDescription). Solche Aufgaben werden in einem Notizsymbol mit dem Label task visualisiert (siehe Abb. 3.42) und mit einer gestrichelten Linie mit der zugehörigen Rolle einer Musterspezifikation verknüpft.

Der formale Teil einer Musterspezifikation dient dazu, den Schritt der Musteranwendung in Teilen zu automatisieren und den Entwurf während seiner Evolution zu prüfen. Im Gegensatz dazu dienen Aufgabenbeschreibungen in den Musterspezifikationen dazu, nicht automatisierbare Entwurfsaufgaben für die Musteranwendung mit zu erfassen und Entwickler auf diese aufmerksam zu machen.

Erledigungsstatus

Als ergänzende Unterstützung kann ein Entwickler für jede Anwendungsstelle mit Hilfe von Werkzeugen festhalten, ob eine Aufgabe bereits erledigt ist (siehe Abschnitt 5.6.2).

3.6 Erweiterbarkeit der Musterspezifikationssprache

Die Musterspezifikationssprache ist entwickelt worden, um eine Entwurfslösung zu einem Entwurfsmuster sehr abstrakt, aber formal und dennoch menschenlesbar zu erfassen. Dabei ist die Sprache in zwei Teile zerlegt (siehe Abb. 3.4, S. 45). Ein allgemeiner Teil dient dazu, Entwurfsmuster-spezifische Informationen wie den Namen eines Musters, seine Varianten, Restriktionen bzgl. der Kopplung seiner Bestandteile, etc. zu erfassen.

PSL fest

Ein anderer Domänen-spezifischer Teil, die DAL, dient dazu, den eigentlichen Entwurf und seine Bestandteile zu beschreiben. Dieser Teil der Sprache soll zukünftig bei Bedarf durch zusätzliche Sprachkonstrukte auf weitere Domänen ausgedehnt werden.

DAL erweiterbar

Bisher besteht mein Domänen-spezifischer Teil (DAL) aus Sprachkonstrukten, die speziell Klassenstrukturen in objektorientierter Software und bis zu einem gewissen Grad das zugehörige Interaktionsverhalten beschreiben können. Um Muster auch auf höherem Abstraktionslevel beschreiben zu können, z.B. das Facade-Muster oder das MVC-Paradigma, habe ich diesen Teil der Sprache um Subsysteme erweitert.

potentielle Erweiterungen

Zur Beschreibung von Entwurfsmustern aus anderen Domänen wie der verteilten, nebenläufigen, komponentenbasierten oder eingebetteten Softwaresysteme oder zur Beschreibung von Entwurfsmustern für Frameworks oder Plattformen wie Java EE[15], AUTOSAR[16] oder Android[17] kann der Domänen-spezifische Anteil der Musterspezifikationssprache um Konstrukte zur Repräsentation von Konzepten

[15] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[16] http://www.autosar.org/
[17] http://www.android.com/

aus diesen Domänen erweitert werden. Die allgemeinen Teile der Musterspezifikationssprache wie Set Fragments oder Kopplungsrestriktionen können dann auch für diese Erweiterungen verwendet werden.

Eine mögliche Erweiterung zur Unterstützung von Mustern in eingebetteten Systemen wären hierarchische Komponenten und zeitbehaftete hierarchische Automaten oder Realtime Statecharts. Mit Hilfe solcher Erweiterungen ließen sich vermutlich auch Echtzeitkoordinationssmuster [DBHT12] in dem Mechatronic-UML-Ansatz [BBB+12] spezifizieren. Bei diesem Ansatz werden Kommunikationsrollen und das Protokollverhalten zwischen den beteiligten Rollen durch Realtime-Statecharts spezifiziert. Angewendet werden solche Muster auf hierarchischen Komponenten durch Zuordnen der Kommunikationsrollen zu den Ports der Systemkomponenten des Systems und partielle Generierung und manuelle Anpassung des Komponentenverhaltens auf Basis des Kommunikationsprotokolls.

3.7 Einschränkungen

Wie alle Sprachen bringt auch die vorgestellte Musterspezifikationssprache einige Einschränkungen in ihrer Ausdrucksmächtigkeit mit sich. Die DAL limitiert die beschreibbaren Softwareentwürfe auf objektorientierte Klassenstrukturen. Konzepte aus funktionalen oder logischen Programmiersprachen stehen nicht zur Verfügung. Es gibt keine Sprachkonstrukte für Komponenten und ihre Schnittstellen (z.B. können keine Ports modelliert werden). Ebenso fehlt eine Deployment-Sicht, sodass die Verteilung von Softwarekomponenten auf Hardwareknoten nicht beschrieben werden kann. Verhalten lässt sich nur sehr eingeschränkt durch vordefinierte Aktionen (z.B. Delegation) beschreiben. Beliebiger Kontrollfluss mit Schleifen, Fallunterscheidungen, Methodenaufrufen wie in UML-Aktivitäten- oder UML-Sequenzdiagrammen lässt sich nicht spezifizieren. Objektstrukturen wie sie zur Laufzeit entstehen (z.B. eine Chain of Responsibility [GHJV95]) können nicht modelliert werden (bei Story Patterns geht das, siehe Abschn. 2.4.1).

begrenzter Sprachumfang, spezielle Domäne

Es lassen sich nicht alle Arten von Implementierungsvarianten beschreiben. Zum Beispiel können aufgrund fehlender Sprachkonstrukte Implementierungen mit Reflection, Dependency Injection oder Aspekten (AOP[18]) nicht repräsentiert werden. Mit Set Fragments können sich wiederholende Teilstrukturen eines Entwurfs beschrieben werden. Alternativen wie sie z.B. durch Feature-Diagramme [CHE05] beschrieben werden können, lassen sich jedoch bisher nicht ausdrücken. Stattdessen muss jede Alternative in einer separaten Spezifikation modelliert werden.

Variabilität begrenzt ausdrückbar

Keine Musterkomposition

Kompositionen mehrerer Entwurfsmuster in einer Musterspezifikation wie z.B. bei Smith [Smi11] (siehe Abb. 9.14, S. 226) sind mit meiner Musterspezifikationssprache (bisher) nicht möglich.

Grenzen bei Vagheit

Aufgrund der absichtlich vagen und teilweise unvollständigen Beschreibung von Entwurfsmustern bzw. der zugehörigen Implementierungsvarianten lassen sich unterspezifizierte Teile eines Musters nicht formalisieren. In meiner Musterspezifikationssprache lassen sie sich nur durch separate Spezifikation aller bekannten Implementierungsvarianten (Aufzählen) oder in natürlicher Sprache in Entwurfsaufgaben (Tasks) erfassen (siehe Abschn. 3.5.4).

[18] AOP = Aspekt-orientierte Programmierung

gewöhnungsbedürftige Notation

Insbesondere durch die Verwendung von Set Fragments und ihrer Notation weicht die konkrete Syntax der Musterspezifikationssprache deutlich von der UML ab. Das kann sich negativ auf die Lesbarkeit der Sprache für Entwickler und Architekten auswirken.

3.8 Zusammenfassung und Ausblick

Musterspezifikationssprache

In diesem Kapitel stelle ich eine neue Sprache zur Spezifikation objektorientierter Entwurfsmuster vor. Musterspezifikationen sollen verständlich sein und sowohl zur Generierung als auch zur Prüfung von Musterimplementierungen dienen.

Klassenstruktur & Interaktionen

Es kann sowohl die Struktur als auch das Verhalten der zu einem Entwurfsmuster gehörenden Entwurfslösung beschrieben werden. Struktur wird vergleichbar zu UML-Klassendiagrammen in Form von Typen, Operationen, Assoziationen und Vererbungsbeziehungen beschrieben. Verhalten wird durch eine Reihe vordefinierter Verhaltensmuster, sogenannter Aktionen, repräsentiert. Diese werden zusammen mit der Struktur spezifiziert und beschreiben z.B. Delegationen oder Klasseninstanziierungen. Ergänzend dazu können Einschränkungen bzgl. der Abhängigkeiten in Musterimplementierungen formuliert werden, z.B. die Intention, bestimmte Klassen voneinander zu entkoppeln. Lässt sich ein Teil einer Entwurfslösung nicht präzise modellieren, kann er in natürlicher Sprache formuliert und als Entwurfsaufgabe in eine Spezifikation aufgenommen werden.

Kopplungsregeln

Aufgaben

Implementierungsvarianten

Die Sprache zeichnet sich insbesondere durch ihre Fähigkeit aus, zahlreiche Implementierungsvarianten eines Musters kompakt in einer Spezifikation zu repräsentieren. Das wird zum einen durch Ausnutzen transitiver Beziehungen (z.B. Vererbung) und zum anderen durch die hier eingeführten Set Fragments ermöglicht. Beides erlaubt bei einer Musterimplementierung Abweichungen von der Musterspezifikation in einem vordefinierten Rahmen.

Semantikdefinition

Die Semantik der Sprache ist vor allem durch Abbildung der DAL-Sprachkonstrukte auf Ecore-Klassen- und Story-Diagramm-Modelle festgelegt. Die Semantik von Set Fragments ist formal durch Beschreiben einer Operation für das Vervielfältigen der in einem Set Fragment beschriebenen Struktur definiert.

Musterspezifikationen

Die Eignung der Sprache für die Spezifikation von Entwurfsmustern wurde durch die Spezifikation der 23 GoF-Muster sowie vierer Architekturmuster und zweier Idiome eingeschätzt (siehe Anhang B).

Erweiterbarkeit

Zur Beschreibung anderer als der bisher betrachteten Softwaremuster könnte die DAL z.B. um Sprachkonstrukte für Softwarekomponenten erweitert werden.

Ausblick

Zur weitergehenden Evaluation der Musterspezifikationssprache könnten weitere Entwurfsmuster spezifiziert sowie der Nutzen und die Grenzen der Spezifikationen z.B. in Bezug auf Lesbarkeit, Vollständigkeit und Einsatz im Rahmen des in dieser Arbeit beschriebenen Verfahrens an praktischen Beispielen, ggf. in Feldstudien untersucht werden.

4 Modellierung und Visualisierung von Musteranwendungsstellen

Das Erstellen einer detaillierten Dokumentation angewandter Entwurfsmuster ist mit hohem Aufwand verbunden. Daher wird dieser Schritt oft vernachlässigt. Die dadurch fehlende oder unvollständige Dokumentation von Musterimplementierungen und damit von getroffenen Entwurfsentscheidungen erschwert die Einarbeitung in unbekannte Softwaresysteme [PULPT02, BTGH06]. Übersehene oder falsch verstandene Musterimplementierungen können zu Entwurfsfehlern und damit langfristig zu Design-Erosion führen [vGB02].

Zweck der Modellierung von Anwendungsstellen

Um das Risiko für solche Entwurfsfehler zu reduzieren, verfolge ich mit meiner Arbeit das Ziel, Musterimplementierungen zwecks Nachvollziehbarkeit als solche erkennbar und mit dem zugehörigen Muster (bzw. einer Musterspezifikation) vergleichbar zu machen. Dazu sollen Musterimplementierungen in einem Modell erfasst werden, welches zum einen zur Darstellung und zum anderen zur automatischen Prüfung von Musterimplementierungen dienen soll (Ziele 2 und 3 in Abschn. 1.2). Ergänzend dazu soll das Modell zur Repräsentation von Anwendungsstellen[1] vor einer semiautomatischen Musteranwendung dienen.

Herausforderungen

Damit eine Musterimplementierung mit allen ihren Details dargestellt und auf Korrektheit geprüft werden kann, ist eine detaillierte Zuordnung der Entwurfsteile zu den Elementen einer Musterspezifikation nötig. Diese Aufgabe ist bei meinem Ansatz besonders herausfordernd, weil die Zuordnung neben den spezifizierten Musterrollen auch Aktionen und Set Fragments umfassen muss. Aktionen spezifizieren das zu einem Muster gehörende Verhalten und müssen einem Verhaltensmodell zugeordnet werden. Set Fragments beschreiben in einer Musterimplementierung als Ganzes mehrfach vorkommende Entwurfsstrukturen. In beiden Fällen ist keine 1-zu-1-Zuordnung von Entwurfs- zu Musterspezifikationselementen möglich. Aus diesem Grund ist nicht nur das Modellieren einer Anwendungsstelle bzw. einer Musterimplementierung schwierig, auch ihre Visualisierung mit allen relevanten Details stellt eine Herausforderung dar.

Grenzen verwandter Arbeiten

Bis auf wenige Ausnahmen beschränken sich bisherige Arbeiten auf die Zuordnung von Musterrollen (siehe Abschn. 9.2) und vernachlässigen dabei das spezifizierte Verhalten oder mit Set Fragments vergleichbare Entwurfsstrukturen. Bei der Zuordnung von Musterrollen werden häufig nur Klassen, aber z.B. keine Methoden und Assoziationen berücksichtigt. Bei der Visualisierung werden Klassen als Teil einer Musterimplementierung markiert, es ist jedoch nicht erkennbar, welche Methoden, Assoziationen, Vererbungsbeziehungen und welches Verhalten Teil welcher Musterimplementierung sind (z.B. Abb. 9.8 – 9.10, S. 222). In anderen Arbeiten

[1] Zwecks Unterscheidung der Begriffe „Musterimplementierung“ und „Anwendungsstelle“ siehe Abb. 2.1, S. 26

wird eine Musterimplementierung zwar detailliert dokumentiert, ihre Darstellung ist jedoch aufgrund zu vieler Details unübersichtlich (z.B. Abb. 9.11, S. 222). Eine übersichtliche Darstellung mehrerer, klar unterscheidbarer Anwendungsstellen ist nicht immer möglich (z.B. nicht bei Eden et al., S. 224).

Lösungsansatz

Bei meinem Ansatz werden alle Anwendungsstellen und Musterimplementierungen in einem separaten, ein Entwurfsmodell ergänzenden Modell festgehalten. Es werden alle Rollen einer Musterspezifikation mit den sie einnehmenden Teilen eines Entwurfsmodells verknüpft. Zu den Rollen zählen sämtliche in der DAL (Abschn. 3.4) beschreibbaren Elemente einer Musterspezifikation, wozu u.a. Klassen, Operationen, Assoziationen, Attribute, Parameter und Aktionen gehören. Die komplexe Zuordnung einer Aktion zu einem Verhaltensmodell (einem Story-Diagramm) erfolgt mit Hilfe baumartiger Token-Strukturen. Mit Hilfe eines anderen baumartigen Konstrukts werden die zu einem Set Fragment gehörenden Entwurfsstrukturen zugeordnet. Damit das Erstellen eines Modells aller Anwendungsstellen möglichst wenig Aufwand eines Entwicklers erfordert, wird es während einer Musteranwendung automatisch erzeugt (Kap. 5), es kann jedoch auch nachträglich manuell erstellt werden. Neben der Visualisierung von Musterimplementierungen wird das Modell insb. zur Konsistenzprüfung genutzt (Kap. 6).

Zur Visualisierung einer Anwendungsstelle vor einer Musteranwendung und zur Darstellung von Musterimplementierungen nach erfolgter Musteranwendung stelle ich mehrere, auf unterschiedliche Bedürfnisse zugeschnittene und in ihrem Detailgrad anpassbare Sichten auf das Modell der Anwendungsstellen bereit. Anwendungsstellen können einzeln oder zusammen mit anderen Anwendungsstellen direkt im Entwurfsmodell dargestellt werden.

Wissenschaftliche Beiträge

Mit diesem Kapitel beantworte ich die dritte der vier Forschungsfragen aus Kapitel 1 (siehe S. 10). Ich stelle einen Weg zur detaillierten Erfassung und Visualisierung von Anwendungsstellen und Musterimplementierungen vor. Zu den wesentlichen wissenschaftlichen Beiträgen dieses Kapitels gehören:

- ein generisches[2] Konzept zur Verknüpfung von Musterspezifikationen mit ihren Implementierungen in Entwurfsmodellen
- ein Konzept zur Modellierung der Zuordnung von Set Fragments zu den sie implementierenden, ggf. mehrfach vorkommenden Entwurfsstrukturen
- eine Möglichkeit, eine komplexe Zuordnung wie die einer Aktion zu einem wesentlich komplexeren Verhaltensmodell mit Hilfe von Tokens baumartig in mehrere Teilzuordnungen zu zerlegen
- mehrere für unterschiedliche Zwecke konzipierte Notationen für Anwendungsstellen und Musterimplementierungen

Kapitelstruktur

Im Folgenden erläutere ich meinen Ansatz zunächst an einem Beispiel und gehe auf meine Annahmen und Anforderungen ein. Anschließend erkläre ich die abstrakte Syntax für Anwendungsstellenmodelle, bevor ich die verschiedenen Notationen und Sichten vorstelle. Abschließend gehe ich auf die Grenzen des Ansatzes ein und gebe eine Zusammenfassung.

[2] Der Ansatz ist nicht auf die exemplarisch verwendeten Ecore- und Story-Diagramm-Modelle beschränkt und könnte auf andere Modellierungssprachen übertragen werden.

4.1 Überblick

Als Teil dieser Arbeit habe ich mir zum Ziel gesetzt, ein möglichst allgemein verwendbares Verfahren zur Erfassung von Musterimplementierungen zu entwickeln, genauer: zur Erfassung des Zusammenhangs zwischen einer Musterspezifikation und zugehöriger Musterimplementierungen in einem Entwurfsmodell auf Basis von Rollenzuordnungen (siehe Abb. 4.1).

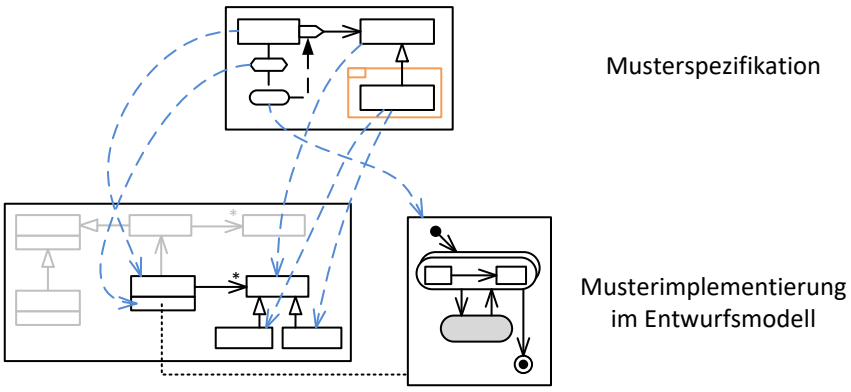


Abbildung 4.1: Zusammenhang einer Musterspezifikation und ihrer Implementierung

Annahmen

Dabei gehe ich davon aus, dass Muster in einem Modell spezifiziert wurden und in einem Softwareentwurfsmodell angewendet werden. Die Musterspezifikationen und Softwareentwurfsmodelle entsprechen dabei getypten attributierten Graphen mit einer Containment-Hierarchie[3], wobei eine Musterspezifikation die Struktur einer oder mehrerer Musterimplementierungen abstrakt repräsentiert. Ein Knoten in einer Musterspezifikation repräsentiert ein oder mehrere zusammenhängende Knoten in einer Musterimplementierung in einem Entwurfsmodell. Als Beispiel für eine Musterspezifikationssprache verwende ich die in Kapitel 3 eingeführte PSL.

Anforderungen

Jede Musterimplementierung in einem Entwurfsmodell soll als Ganzes unabhängig von anderen Musterimplementierungen erfasst werden können. Jede Rolle (jeder Knoten) der zugehörigen Musterspezifikation muss einem oder mehreren die Rolle einnehmenden Knoten in der Musterimplementierung zugeordnet werden können (gestrichelte Pfeile in Abb. 4.1). Gleichzeitig soll ein Knoten in einer Musterimplementierung (z.B. eine Klasse) auch mehrere Rollen einer oder mehrerer Musterspezifikationen einnehmen können. Alle Knoten einer Musterimplementierung, die zu einem Teilgraphen gehören, der laut Musterspezifikation mehrfach in einer Musterimplementierung vorkommen darf, sollen als Einheit erfasst werden. Das orange dargestellte Rechteck in Abb. 4.1 markiert einen solchen Teilgraphen und entspricht einem Set Fragment (siehe Abschn. 3.5.2).

Beispiel

In der Abb. 4.2 soll der Zusammenhang einer Musterimplementierung und der zugehörigen Musterspezifikation verdeutlicht werden. Hier sind ein Entwurfsmodell in Form eines Klassen- und eines Story-Diagramms (unten im Bild) und eine Musterspezifikation in der PSL (oben) dargestellt. Die Rollen des spezifizierten

[3] Jeder Knoten hat genau einen Elternknoten oder ist ein Wurzelknoten. Diese Eigenschaft ist z.B. für EMF-basierte [SBPM08] Modelle erfüllt.

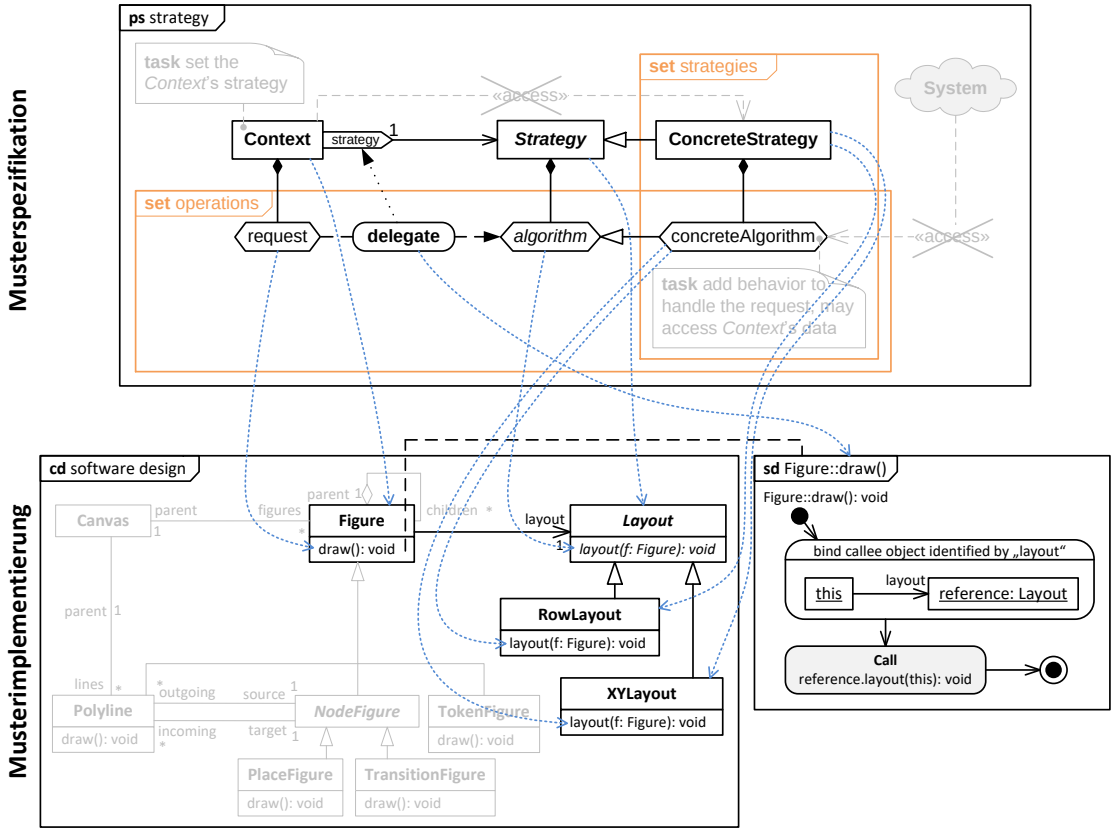


Abbildung 4.2: Zusammenhang einer Musterspezifikation und ihrer Implementierung am Beispiel des Strategy-Musters

Musters Strategy sind schwarz hervorgehoben, während Set Fragments orange und sonstige Informationen hellgrau dargestellt sind. Alle zur Musterimplementierung gehörenden Elemente sind ebenfalls schwarz hervorgehoben, der übrige Teil des Softwareentwurfs ist hellgrau dargestellt. Die blauen, gestrichelten Pfeile stellen die Zuordnung der Musterrollen zu den sie einnehmenden Entwurfselementen dar.

PAL und Korrespondenzmodell

Um Musterimplementierungen und Musterspezifikationen möglichst unabhängig von den verwendeten Modellierungssprachen durch Rollenzuordnungen miteinander verknüpfen zu können, habe ich eine generische Sprache (bzw. ein Meta-Modell), die *Pattern Application Language* (PAL), zur Beschreibung von Korrespondenzen zwischen Musterrollen und Entwurfselementen entwickelt. Ein Korrespondenzknoten verknüpft genau eine Musterrolle mit einem oder mehreren die Rolle einnehmenden Entwurfselementen. Spezielle Korrespondenzknoten ordnen einer Musterspezifikation alle zugehörigen Musterimplementierungen oder einem Set Fragment alle zugehörigen Set-Fragment-Instanzen innerhalb einer Musterimplementierung zu. Mit Hilfe von Tokens kann eine Korrespondenz in mehrere Teilkorrespondenzen zerlegt werden. Korrespondenzen werden in einem sogenannten Dekorationsmodell persistiert (grau hinterlegt in Abb. 4.3), welches das Entwurfsmodell ergänzt und mit Informationen über Rollenzuordnungen anreichert.

Weil die PAL nicht nur zur Dokumentation von Musterimplementierungen, sondern u.a. zur automatisierten Musteranwendung eingesetzt wird, enthält sie einige zusätzliche Konstrukte. Zum Beispiel wird für jede Aufgabenbeschreibung aus

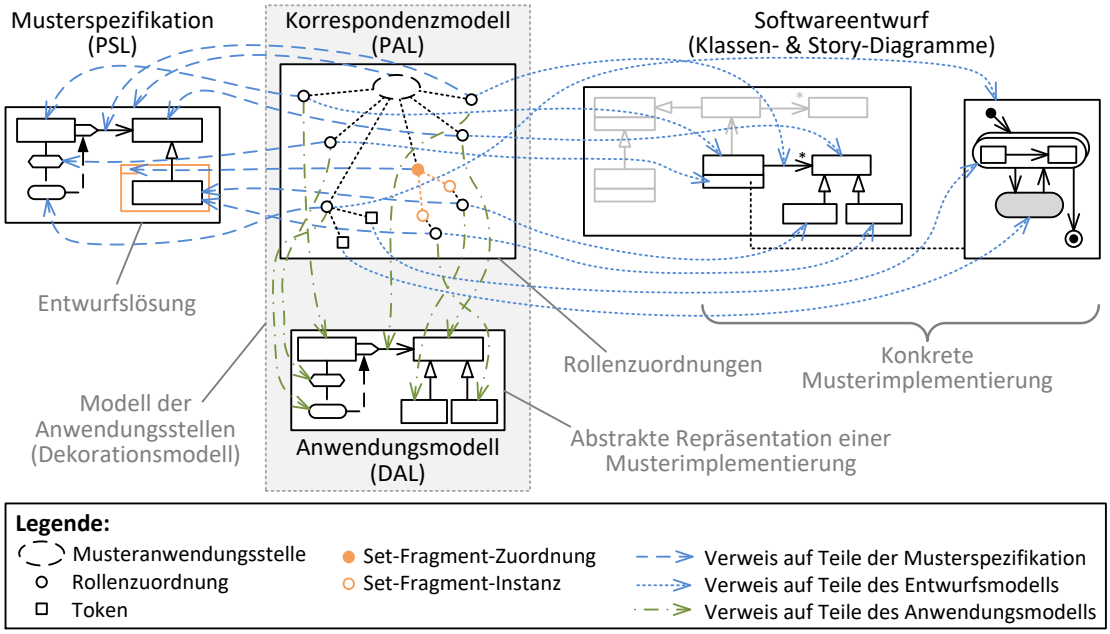


Abbildung 4.3: Verknüpfung einer Musterimplementierung mit der Musterspezifikation (vgl. Abb. 3.5, S. 46)

einer Musterspezifikation eine entsprechende zu erledigende Aufgabe für eine bestimmte Musterimplementierung erstellt.

Anwendungsmodell

Das Dekorationsmodell enthält zusätzlich zu den Korrespondenzknoten ein sogenanntes Anwendungsmodell, welches für die Synthese einer Musterimplementierung benötigt und automatisch erstellt wird (Details in Abschnitt 5.4.1). Im Prinzip enthält ein Anwendungsmodell eine Eins-zu-eins-Repräsentation einer Musterimplementierung in der DAL, wobei alle Set Fragments einer Musterspezifikation durch ein Ein- oder Mehrfachvorkommen der darin spezifizierten Struktur ersetzt werden (siehe Abb. 4.3).

Regeln für Korrespondenzen

Die PAL erlaubt es, eine Rolle in einer Musterspezifikation beliebigen Elementen in einem Entwurfsmodell zuzuordnen. Dadurch können für die Spezifikationen und den Softwareentwurf zwar beliebige Modellierungssprachen verwendet werden, allerdings ist nicht jede Kombination von einer Rolle und zugeordneten Elementen sinnvoll. Für die PSL sowie Klassen- und Story-Diagramm-Modelle habe ich eine Vorschrift für die Zuordnung definiert. Diese beschreibe ich zusammen mit der Übersetzungstransformation im Abschnitt 5.5.4 (und im Anhang C.3.2 & C.3.3).

Visualisierung

Zur besseren Nachvollziehbarkeit (Traceability) sollen Rollenzuordnungen und andere Korrespondenzen nicht nur erfasst (modelliert), sondern auch geeignet visualisiert werden. Dadurch soll sowohl die Anwendung eines Musters und somit die Übertragung einer spezifizierten Entwurfslösung in einen existierenden Softwareentwurf verdeutlicht als auch das Wiedererkennen bereits implementierter Muster im Softwareentwurf vereinfacht werden. Zu diesem Zweck habe ich mehrere Notationen entwickelt, welche den Zusammenhang zwischen einer Musterimplementierung und einer Musterspezifikation auf der einen Seite und zwischen einem Entwurfsmodell und den darin vorkommenden Musterimplementierungen auf der anderen Seite visualisieren. Diese Notationen stelle ich im Folgenden vor.

Sie werden vor allem durch das detaillierte Korrespondenzmodell ermöglicht.

Implementierung in der Spezifikation

In der *Musteranwendungssicht* wird die Spezifikation eines anzuwendenden oder bereits angewendeten Musters mit Informationen über die existierenden Korrespondenzen, insb. Rollenzuordnungen und Set-Fragment-Instanzen, angereichert. Diese Darstellung ist unabhängig von der für den Softwareentwurf eingesetzten Modellierungssprache und orientiert sich allein an der Musterspezifikationssprache aus Kapitel 3.

Implementierungen im Entwurf

Damit Musterimplementierungen auch direkt im Entwurf visualisiert werden, habe ich die für den Softwareentwurf eingesetzten Klassen- und Story-Diagramme mit Hilfe von Markierungen mit Informationen über Korrespondenzen bzw. Rollenzuordnungen angereichert. Bei der so angereicherten Darstellung können zwei verschiedene Detailgrade verwendet werden. In der einfachen *Entwurfssicht* werden alle an Musterimplementierungen beteiligten Entwurfselemente bloß hervorgehoben / markiert. In der detaillierteren *Musterrollen-im-Entwurf-Sicht* werden zusätzlich Informationen zu angewandten Mustern, eingenommenen Rollen und zugehörigen Set-Fragment-Instanzen eingeblendet.

Softwareentwurfsdiagramme können komplex und unübersichtlich werden. Die an einer Musterimplementierung Beteiligten können in solchen Diagrammen weit verstreut sein oder sich über mehrere Diagramme verteilen. Sind einige Klassen an mehreren Musterimplementierungen beteiligt, erschwert das die Situation zusätzlich. Damit man auch in diesen Fällen den Überblick zu einer bestimmten Musterimplementierung behält,

Implementierung in Entwurfsausschnitt

stelle ich zusätzlich eine *Musterrollen-im-Entwurf-Ausschnittssicht* bereit. Darin werden alle Beteiligten einer Musterimplementierung (insb. Klassen, Methoden, Assoziationen) alleine in einem separaten Diagramm dargestellt. Die Notation wird von den Softwareentwurfsdiagrammen (hier: Klassen- & Story-Diagramme) übernommen.

Die Rollenzuordnung in der Musteranwendungssicht stelle ich in Abschnitt 4.3 vor, während die entwurfsbasierten Sichten in Abschnitt 4.4 vorgestellt werden.

4.2 Pattern Application Language – Erfassung von Korrespondenzen

Jede Zuordnung von Entwurfselementen zu einer Rolle wird durch einen speziellen Korrespondenzknoten, nämlich durch ein RoleBinding-Objekt repräsentiert. Zum

Beispiel

Beispiel wird die Zuordnung der Rolle Strategy zur Klasse Layout aus Abb. 4.2 in der Pattern Application Language (kurz: PAL) repräsentiert wie durch die Objektstruktur in der Abb. 4.4 exemplarisch dargestellt (oben im Bild).

In derselben Abbildung ist auch die Zuordnung des Set Fragments strategies (vgl. Abb. 4.2) zu den beiden Set-Fragment-Instanzen und den zugehörigen Entwurfsteilen bestehend aus je einer Klasse und einer Operation dargestellt. Alle Implementierungen der in einem Set Fragment spezifizierten Struktur (hier in Form der Klassen RowLayout und XYLayout mit zugehörigen Operationen) werden durch je ein SetFragmentInstance-Objekt repräsentiert, in einem SetFragmentBinding-Objekt zusammengefasst und dem Set Fragment zugeordnet. Jedes SetFragmentInstance-Objekt ist mit allen dazugehörigen Rollenzuordnungen (RoleBinding) verbunden.

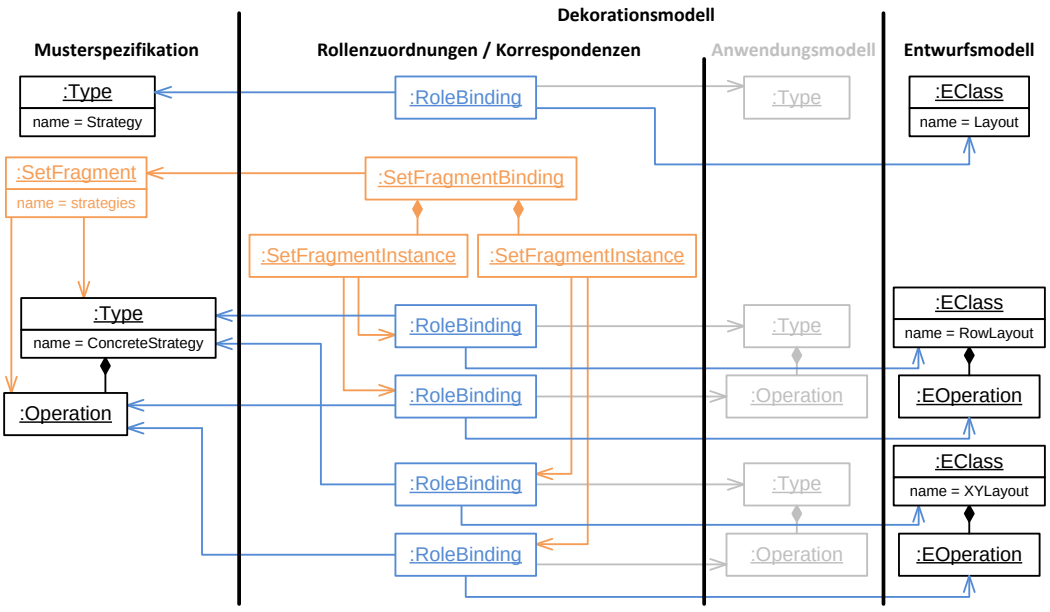


Abbildung 4.4: Ausschnitt eines Dekorationsmodells in abstrakter Syntax

In der Abb. 4.4 ist außerdem ein Teil eines Anwendungsmodells dargestellt. Dieses verdeutlicht den Zusammenhang mit Set Fragments und die Eins-zu-eins-Beziehung zu den Elementen im Entwurf. Eine detaillierte Beschreibung des Anwendungsmodells folgt in Abschnitt 5.4.1.

Anwendungsmodell

Das Meta-Modell der PAL ist in der Abb. 4.5 als Klassendiagramm dargestellt. Alle Musterimplementierungen werden in einem PatternApplications-Objekt zusammengefasst. Jede Musterimplementierung (bzw. jede Anwendungsstelle) wird durch ein AppliedPattern-Objekt repräsentiert. Dieses ist immer mit der zugehörigen Musterspezifikation (PatternSpecification) verknüpft und fasst alle zur Implementierung gehörenden Korrespondenzen (Rollen- und Set-Fragment-Zuordnungen) zusammen.

Anwendungsstelle

Eine Rollenzuordnung (RoleBinding) verknüpft eine Rolle der Musterspezifikation (DesignElement) über die Assoziation role mit allen sie einnehmenden Elementen im Entwurf (EObject) über die Assoziation modelElements. Zu jeder Rollenzuordnung gibt es außerdem ein Duplikat der Rolle (DesignElement) im Anwendungsmodell (ApplicationModel). Dieses Duplikat wird über die Assoziation applicationModelElement verknüpft (vgl. Abb. 4.4).

Rollenzuordnung

Soll einer Musterrolle eine komplexe Struktur des Entwurfsmodells – z.B. einer Aktion ein Story-Diagramm – zugeordnet werden, so kann dies mit Hilfe von Tokens (Token) in mehreren Schritten erfolgen. Dazu kann die Rolle in mehrere Teilaspekte zerlegt und diese einzeln den Entwurfsteilen zugeordnet werden. Einem RoleBinding können ein oder mehrere baumartig strukturierte Tokens zugeordnet werden. Ein Token repräsentiert jeweils einen Teilaspekt der Rolle und verknüpft diesen über die qualifizierte Assoziation mapsTo mit Elementen im Entwurf, die diesem Teilaspekt entsprechen. Ein Schlüssel der mapsTo-Assoziation repräsentiert bestimmte Zuordnungen eines Teilaspekts. Zum Beispiel kann ein abstraktes Konzept wie eine call-Aktion in Teilaspekte zerlegt werden wie die Beschreibung der aufgerufenen Operation und die Beschreibung der Aufrufargumente. Jeder dieser

Tokens

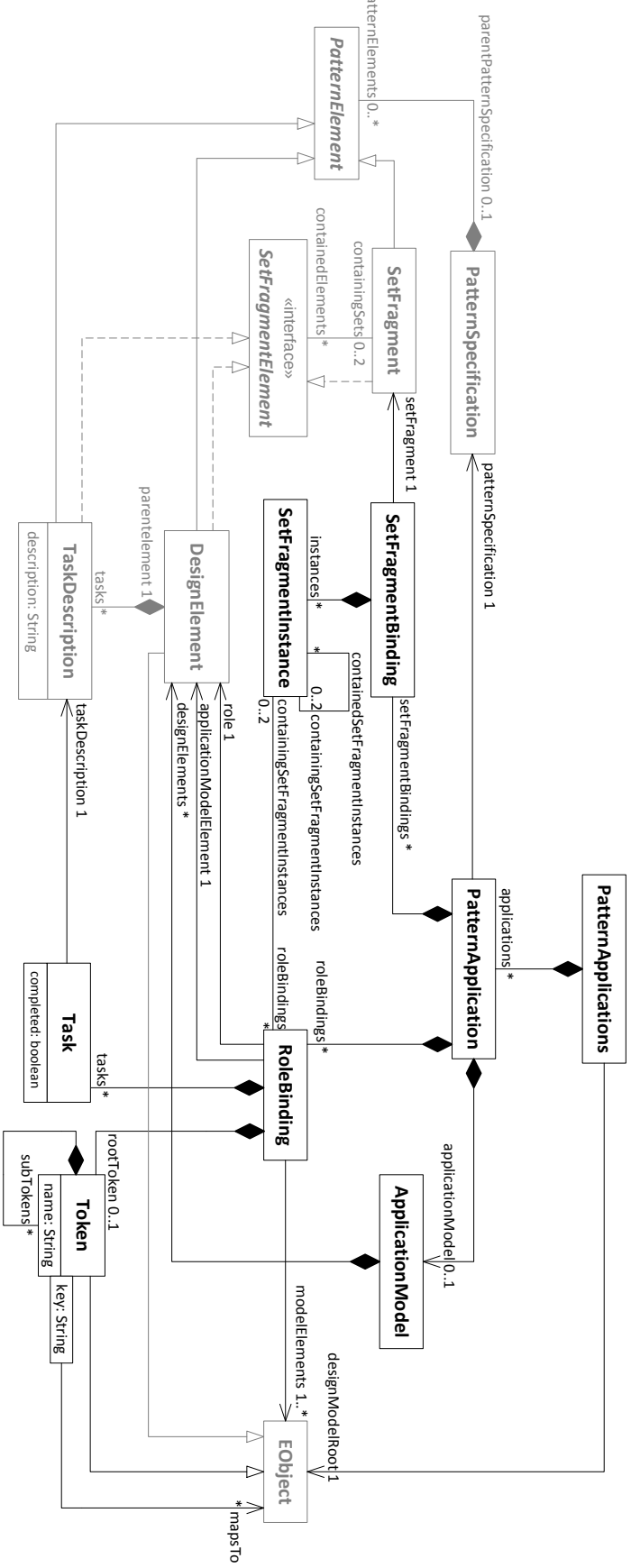


Abbildung 4.5: Abstrakte Syntax der Pattern Application Language (PAL)

Aspekte kann durch je ein Token repräsentiert werden. Einem Token zugeordnete Entwurfselemente wie ein Aktivitätenknoten oder ein Methodenaufrufausdruck können mit Hilfe eindeutiger Schlüssel einzeln über die mapsTo-Assoziation mit einem Token verknüpft werden.

Aufgaben

Zu jeder Aufgabenbeschreibung aus der Musterspezifikation (TaskDescription, siehe Abschnitt 3.5.4) wird eine Aufgabe (Task) angelegt und an eine Rollenzuordnung geheftet. Jede Aufgabe wird bei ihrer Erstellung als unerledigt markiert (completed = false) und kann von einem Entwickler nach seinem Ermessen als erledigt markiert werden.

Set-Fragment-Instanzen

Liegt eine Rolle in einem Set Fragment, wird die zugehörige Rollenzuordnung (RoleBinding) einer Set-Fragment-Instanz (SetFragmentInstance) und über ein SetFragmentBinding-Objekt dem Set Fragment zugeordnet (vgl. Abb. 4.4). Dabei kann eine Set-Fragment-Instanz nicht nur beliebig viele Rollenzuordnungen, sondern auch beliebig viele andere Set-Fragment-Instanzen enthalten und somit analog zu Set Fragments beliebig tief geschachtelt vorkommen. Analog zu Set Fragments können sich auch Set-Fragment-Instanzen überschneiden, weswegen eine Set-Fragment-Instanz aber auch eine Rollenzuordnung bis zu zwei sie enthaltenden Set-Fragment-Instanzen zugeordnet werden kann.

Das vorliegende Meta-Modell ist auf die in Kapitel 3 eingeführte Musterspezifikationssprache, jedoch nicht auf eine bestimmte Modellierungssprache des Entwurfsmodells festgelegt[4].

4.3 Rollenzuordnungen in der Musteranwendungssicht

Die Musteranwendungssicht ist vor allem für die Anwendung eines spezifizierten Musters gedacht (Details in Abschn. 5.3), eignet sich jedoch auch für die Betrachtung von Musterimplementierungen. In dieser Ansicht wird die Spezifikation eines anzuwendenden oder bereits angewendeten Musters dargestellt und um die Zuordnungen von Musterrollen zu Elementen im Entwurfsmodell erweitert.

Darstellung

Ein Beispiel für die Darstellung von Rollenzuordnungen in der Musteranwendungssicht ist der Abb. 4.6 zu entnehmen. Hier wird die Spezifikation des Musters Strategy dargestellt. Die Musterrollen, die Elementen des Entwurfs zugeordnet wurden, sind gelb hervorgehoben. Zusätzlich zum Rollennamen wird in diesem Fall auch der Name des zugeordneten Entwurfselements dargestellt. Zum Beispiel wird die Rolle Context der Klasse Figure zugeordnet und die Rolle request der Operation draw.

Rollenzuordnung

Wurde eine Rolle noch keinem Entwurfselement zugeordnet, behält sie die ursprüngliche Darstellung aus der Musterspezifikation. In diesem Fall wird bei der Musteranwendung ein neues Entwurfselement erzeugt. Wenn nichts anderes angegeben ist, wird das neue Entwurfselement nach dem Rollennamen benannt. Ein Entwickler hat jedoch die Möglichkeit, noch vor der Musteranwendung einen anderen Namen für das zu erzeugende Entwurfselement zu vergeben. In diesem Fall wird die Rolle hellblau hervorgehoben und der zu verwendende Entwurfselement-

[4]Die einzige Einschränkung für das Entwurfsmodell ist, dass der Typ jedes Objekts im Entwurfsmodell aus technischen Gründen ein Untertyp der Klasse EObject sein muss, was bei allen EMF-basierten Modellen der Fall ist [SBPM08].

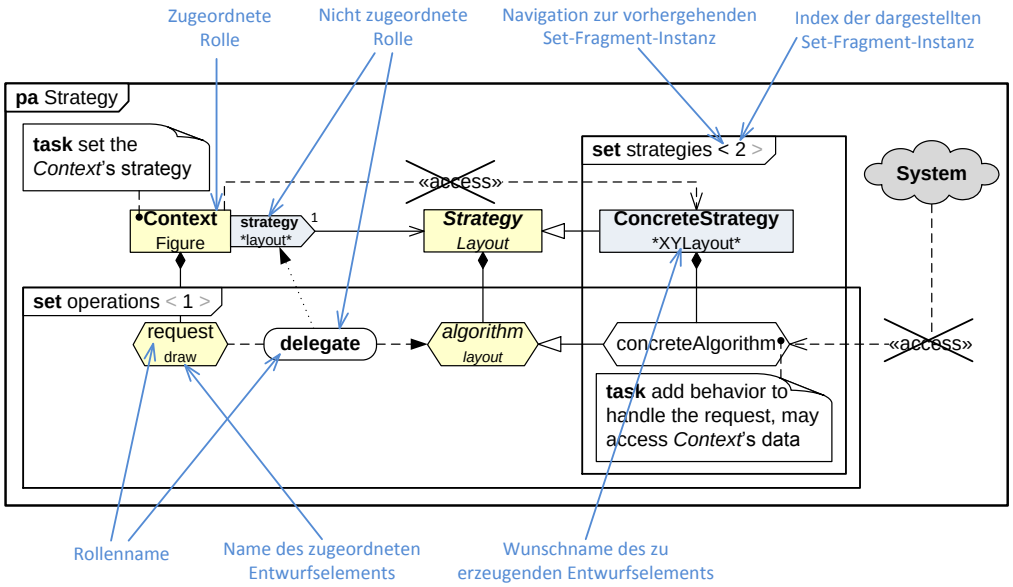


Abbildung 4.6: Musteranwendungssicht: Rollenzuordnungen und Musterspezifikation

name eingeklammert zwischen *-Symbolen unter dem Rollennamen mitdargestellt. So erhält z.B. die zu erzeugende Klasse zur Rolle ConcreteStrategy aus Abb. 4.6 den Namen XYLayout (eines speziellen Koordinaten-basierten Layout-Algorithmus).

Set-Fragment-Instanzen

Set Fragments in Musterspezifikationen kennzeichnen Entwurfsstrukturen, die in einer Musterimplementierung im Entwurf mehrfach auftreten können. Jedes dieser Auftreten nenne ich Set-Fragment-Instanz. In der Musteranwendungssicht stellt jedes Rechteck, welches in der Musterspezifikation einem Set Fragment entspricht, eine Set-Fragment-Instanz dar. Um das Layout der Musteranwendungssicht durch gleichzeitige Darstellung aller Set-Fragment-Instanzen nicht zu verkomplizieren und damit den Zusammenhang zur Musterspezifikation nicht durch ein anderes Layout zu verschleiern[5], wird in der Musteranwendungssicht immer nur eine Set-Fragment-Instanz pro Set Fragment dargestellt. Zu Unterscheidung der dargestellten Set-Fragment-Instanzen wird ihr Index abgebildet. Zum Beispiel wird in der Abb. 4.6 für das Set Fragment operations die Instanz mit Index 1 und für das Set Fragment strategies die Instanz mit Index 2 dargestellt. Mit Hilfe von Navigationspfeilen (< und >) in der Musteranwendungssicht kann zur vorhergehenden oder nachfolgenden Set-Fragment-Instanz gewechselt werden.

Hervorheben durch gleichzeitige Selektion

Man beachte, dass eine Rolle nicht nur einem, sondern mehreren Entwurfselementen zugeordnet werden kann. In diesem Fall wird eines der Entwurfselemente als Hauptelement betrachtet und der Name dieses Elements, sofern vorhanden, unter dem Rollennamen dargestellt. Da der Name eines zugeordneten Entwurfselements nicht unbedingt eindeutig ist und nicht immer existiert, wird die Zuordnung einer Musterrolle zu den entsprechenden Elementen im Softwareentwurf zusätzlich durch simultane Selektion verdeutlicht. Dazu werden bei Selektion einer

[5] Insbesondere im Fall von sich überschneidenden Set Fragments würde sich die gleichzeitige Darstellung aller Set-Fragment-Instanzen stark von der Musterspezifikation unterscheiden. (Siehe Abb. A.32, S. 288. Hier entspricht der Graph G_I einer Musterimplementierung mit je einer Set-Fragment-Instanz pro Set Fragment. Die daraus abgeleiteten Graphen G'_I, G''_I und G'''_I ergeben sich durch Ergänzen weiterer Set-Fragment-Instanzen.)

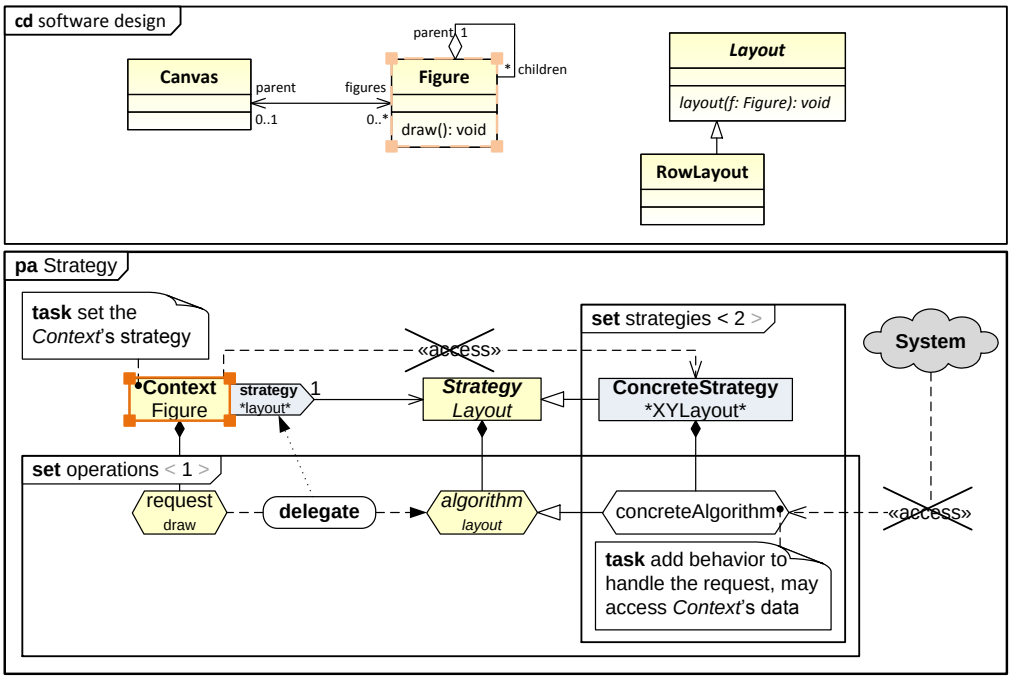


Abbildung 4.7: Selektion einer Musterrolle und zugehörigem Entwurfselement

Musterrolle in der Musteranwendungssicht auch die diese Rolle einnehmenden Entwurfselemente in einem Klassen- oder Story-Diagramm selektiert. Zum Beispiel wird bei Selektion der Musterrolle Context in der Musteranwendungsansicht auch die zugeordnete Klasse Figure im Klassendiagramm selektiert (siehe Abb. 4.7).

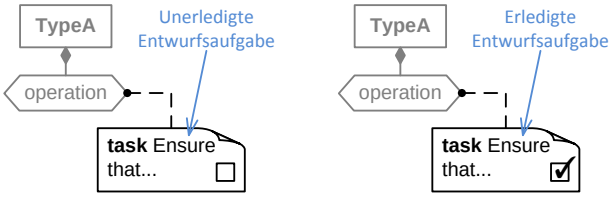


Abbildung 4.8: Darstellung von offenen und erledigten Entwurfsaufgaben

Erledigungsstatus

Der Erledigungsstatus von Entwurfsaufgaben wird in der Musteranwendungssicht wie in der Abb. 4.8 visualisiert. Erledigte Aufgaben werden durch einen Haken gekennzeichnet. Der Status kann vom Entwickler jederzeit geändert werden und dient nur als Erinnerung.

4.4 Musterimplementierungen in Klassen- und Story-Diagrammen

Damit beim Modellieren oder Modifizieren des Softwareentwurfs direkt erkennbar ist, wo bereits Entwurfsmuster eingesetzt wurden, werden die an einer Musterimplementierung beteiligten Entwurfselemente direkt in Klassen- und Story-

Diagrammen markiert und bei Bedarf mit detaillierteren Information zu den Musterimplementierungen angereichert.

4.4.1 Entwurfssicht

alle Musterimplementierungen

In der Entwurfssicht wird jedes in einem Diagramm dargestellte und mit einem Label versehene Entwurfselement, welches zu einer Musterimplementierung gehört, mit einem Puzzle-Symbol markiert (siehe Abb. 4.9 und 4.12). Ansonsten bleibt die Darstellung des Entwurfs in Klassen- oder Story-Diagrammen unverändert. Auf diese Weise können bereits eingesetzte Entwurfsmuster bei Änderungen des Softwareentwurfs nicht mehr ohne Weiteres übersehen werden. Diese Art der Markierung ist sehr generisch und lässt sich auf andere Diagrammarten übertragen. Im Vergleich zu anderen Ansätzen zur Darstellung von Musterimplementierungen (siehe Kap. 9) ist die Notation sehr kompakt und behält die ursprüngliche Anordnung der Elemente in einem Diagramm bei.

Nur die Information, dass ein Element an einer Musterimplementierung beteiligt ist, reicht zum Verstehen des Entwurfs nicht aus. Um die Diagramme kompakt zu halten und dennoch alle nötigen Informationen verfügbar zu machen, werden bei dieser Darstellung auf Wunsch weitere Informationen zu einer Musterimplementierung eingeblendet. Es lassen sich Informationen zu dem eingesetzten Muster sowie den Korrespondenzen zwischen den Musterrollen, Set Fragments und den Entwurfsteilen einblenden. Diese Informationen lassen sich für einzelne Entwurfsteile oder für eine komplette Musterimplementierung einblenden.

Korrespondenzen eines Elements

Wenn ein an einer Musterimplementierung beteiligtes Entwurfselement wie in der Abb. 4.10 selektiert wird, wird ein Label eingeblendet, in dem die implementierten Muster und alle zugehörigen Korrespondenzen aufgeführt werden. In der Abbildung ist die Klasse XYLayout selektiert. Das eingeblendete Label stellt dar, dass die Klasse an einer Implementierung des Musters Strategy beteiligt ist. P steht dabei für *pattern instance* und Strategy[1] kennzeichnet nicht nur das implementierte Muster, sondern ordnet die Klasse einer Musterimplementierung zu, welche durch die Zahl 1 eindeutig identifiziert wird (Durchnummerierung aller Implementierungen des Musters im vorliegenden Entwurfsmodell). S steht für *set fragment instance*. Der Eintrag strategies[2] gibt an, dass die Klasse XYLayout zu einer Set-Fragment-Instanz mit der ID 2 gehört. R steht für *role*. In diesem Beispiel nimmt die Klasse XYLayout die Rolle ConcreteStrategy des Musters Strategy ein. Bei mehr als einem implementierten Muster wird jede Musterimplementierung durch eigene Einträge in dem eingeblendeten Label beschrieben.

komplette Musterimplementierung

Zur Visualisierung einer kompletten Musterimplementierung im Entwurf in ihrem Kontext lassen sich durch Selektion einer Musterimplementierung (in der Baum-artigen Eclipse Outline View) alle an einer Musterimplementierung beteiligten Entwurfsteile farblich hervorheben und alle zugehörigen Korrespondenzen in zusätzlich eingeblendeten Labels darstellen. Zum Beispiel ist in der Abb. 4.11 die Implementierung des Musters Strategy hervorgehoben. Da die darzustellende Musterimplementierung zuvor ausgewählt wurde, muss diese (im Gegensatz zur Darstellung aus Abb. 4.10) nicht in den Labels dargestellt werden, wodurch die Darstellung relativ kompakt bleibt. Man beachte, dass bei dieser Notation nicht nur beschriftete Entwurfselemente in einem Diagramm farblich hervorge-

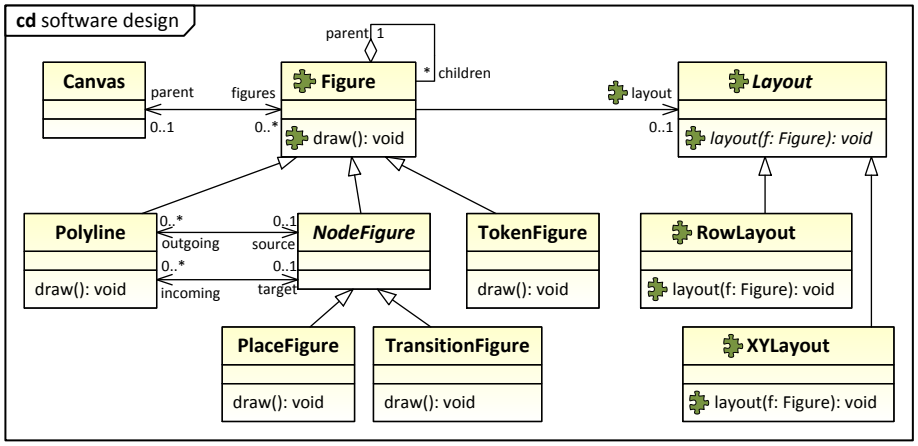


Abbildung 4.9: Klassendiagramm-Entwurfssicht: an Musterimplementierungen beteiligte Entwurfsteile sind markiert

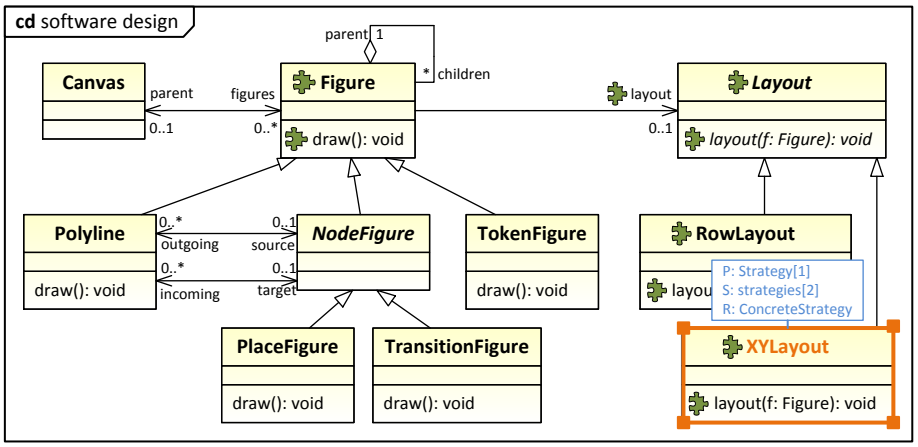


Abbildung 4.10: Klassendiagramm-Entwurfssicht mit Musterimplementierungsdetails eines selektierten Elements

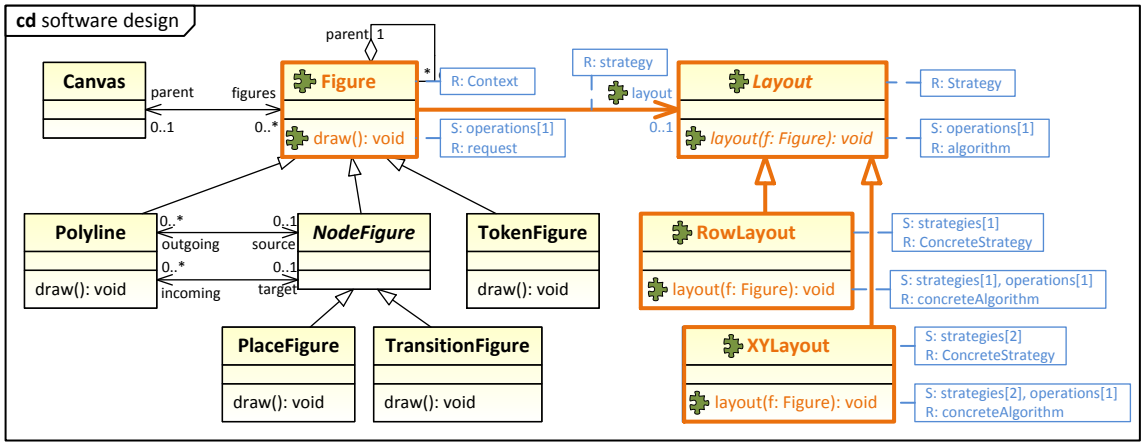


Abbildung 4.11: Klassendiagramm-Entwurfssicht mit Musterimplementierungsdetails einer selektierten Musterimplementierung

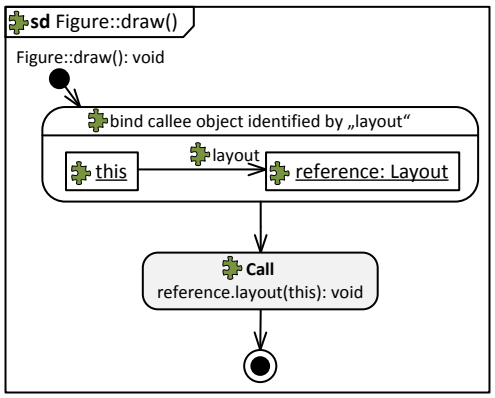


Abbildung 4.12: Story-Diagramm-Entwurfssicht: an Musterimplementierungen beteiligte Entwurfsteile sind markiert

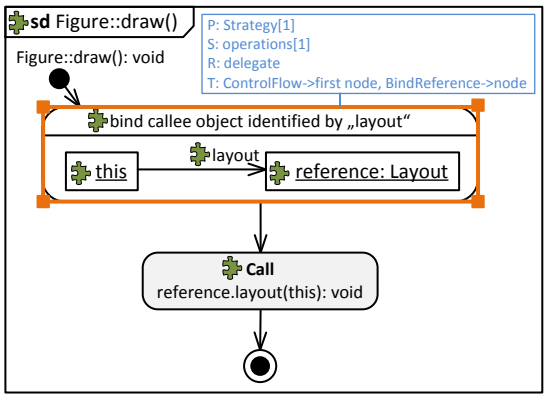


Abbildung 4.13: Story-Diagramm-Entwurfssicht mit Musterimplementierungsdetails eines selektierten Elements

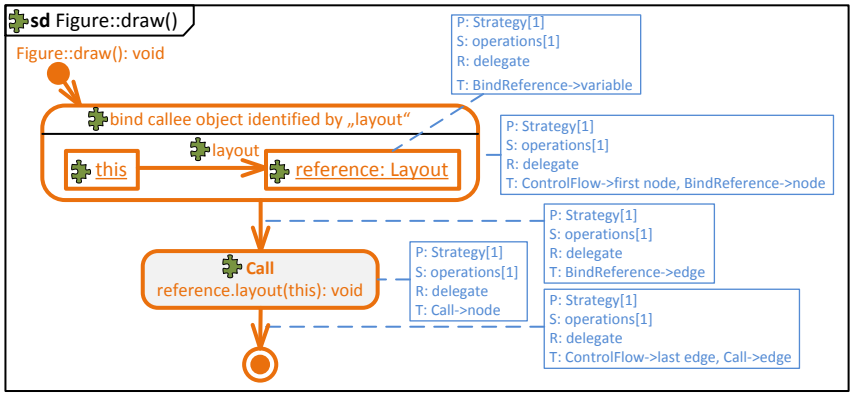


Abbildung 4.14: Story-Diagramm-Entwurfssicht mit Musterimplementierungsdetails einer selektierten Musterimplementierung

hoben werden, sondern auch Elemente ohne Label wie z.B. Vererbungsbeziehungen. Somit werden nicht nur mit einem Puzzle-Symbol versehbare (vgl. Abb. 4.9), sondern alle visualisierten Elemente einer Musterimplementierung hervorgehoben. Neben der Musterimplementierung selbst bleibt auch ihr Kontext im Diagramm sichtbar und hilft dabei, den Zusammenhang der Musterimplementierung mit dem übrigen Softwaresystem zu erkennen.

analog in Story-Diagrammen

Analog zur Visualisierung von Musterimplementierungen in Klassendiagrammen erfolgt auch die Visualisierung von Musterimplementierungen in Story-Diagrammen (Abb. 4.12). Das Markieren einzelner Entwurfselemente blendet ihre zugehörigen Korrespondenzen zu einem Muster, den Musterrollen und Set-Fragment-Instanzen ein (Abb. 4.13). Ebenso kann auch die gesamte Musterimplementierung, soweit sie in einem Diagramm darstellbar ist, hervorgehoben werden (Abb. 4.14).

Im Unterschied zu den bisherigen Klassendiagramm-Beispielen kommen in den Story-Diagrammen aus Abb. 4.13 und 4.14 zusätzlich zum Muster, den Set-Fragment-Instanzen und den Musterrollen auch die mit einem Entwurfselement ver-

knüpfte Tokens dazu. Es werden der Tokenname und der Name der Verknüpfung zum Entwurfselement dargestellt. T steht hierbei für *token*. Zum Beispiel wird in Abb. 4.14 mit dem Ausdruck T: BindReference -> variable für die Objektvariable reference (die Objektvariable ist vom Typ Layout und befindet sich im ersten Aktivitätenknoten des Story-Diagramms) angegeben, dass diese mit einem BindReference-Token über eine variable-Verknüpfung verbunden ist (vgl. Abb. C.32, S. 338). Diese Zuordnung auf Token-Ebene soll auch komplexe Rollenzuordnungen nachvollziehbar machen (siehe Abschnitt C.3.3, S. 328 ff.).

Token-Korrespondenzen

4.4.2 Musterrollen-im-Entwurf-Sicht

Zusätzlich zur simplen, aber übersichtlichen Entwurfssicht lässt sich der Entwurf auch in einer detaillierteren, dafür weniger übersichtlichen Ansicht darstellen, in der Musterrollen-im-Entwurf-Sicht. Der wesentliche Unterschied zur Entwurfssicht ist die zusätzliche Verwendung von UML-Kollaborationen [OMG11b] in Klassen- und Story-Diagrammen. Mit Hilfe der Kollaborationen wird angegeben, welche Muster eingesetzt und welche Musterrollen eingenommen wurden.

UML-Kollaborationen

In der Abb. 4.15 wird z.B. dieselbe Musterimplementierung wie in Abb. 4.11 (S. 91) dargestellt. Die Kollaboration (als Ellipse dargestellt) gibt den Namen des eingesetzten Musters an. Die Namen der eingenommen Rollen (hier Context, Strategy und ConcreteStrategy) werden an den Kollaborationskanten (gestrichelte Linien) angegeben. Anders als in der UML werden zusätzlich Set-Fragment-Instanzen samt ihrer IDs angegeben, z.B. strategies[1]. Im Gegensatz zur Entwurfssicht sind in dieser Darstellung die Kollaborationen und damit die Muster und Rollen jederzeit sichtbar.

permanent sichtbar

Wird eine Musterimplementierung selektiert, werden analog zur Entwurfssicht alle an der ausgewählten Musterimplementierung beteiligten Entwurfsteile und Beziehungen farblich hervorgehoben (siehe Abb. 4.16, vgl. Abb. 4.11, S. 91). Werden einzelne Entwurfselemente selektiert, so werden wie in der Entwurfssicht Labels mit Musterrollen und Set-Fragment-Instanzen eingeblendet (siehe Abb. 4.10, S. 91). Die Darstellung mit UML-Kollaborationen wird analog zu Klassendiagrammen auch in Story-Diagrammen verwendet (siehe Abb. 4.17).

Einblendungen & Hervorhebungen

Beide entwurfsbasierten Ansichten – einfach und detailliert, Entwurfssicht und Musterrollen-im-Entwurf-Sicht – werden in den Abb. 4.18 und 4.19 (S. 95) mit Hilfe eines komplexeren Beispiels in Klassendiagrammen gegenübergestellt. Das Beispiel wird u.a. zur Evaluation meines Ansatzes (siehe Abschnitt 8.2.3) und für die Gegenüberstellung von verwandten Arbeiten verwendet (siehe Abschnitt 9.2.2). Außerdem werden die verschiedenen Hervorhebungen (bei Selektionen) an diesem Beispiel vorgeführt (siehe Abb. 4.20 und 4.21, S. 95).

Vergleich der Darstellungen

4.4.3 Musterrollen-im-Entwurf-Ausschnittssicht

Da Entwurfsdiagramme umfangreich und unübersichtlich werden und sich z.B. an einem Muster beteiligte Klassen über mehrere Diagramme verteilen können, habe ich noch eine weitere Sicht auf den Entwurf vorgesehen, die Musterrollen-im-Entwurf-Ausschnittssicht. Dabei handelt es sich um eine partielle Sicht auf

Entwurfsausschnitt

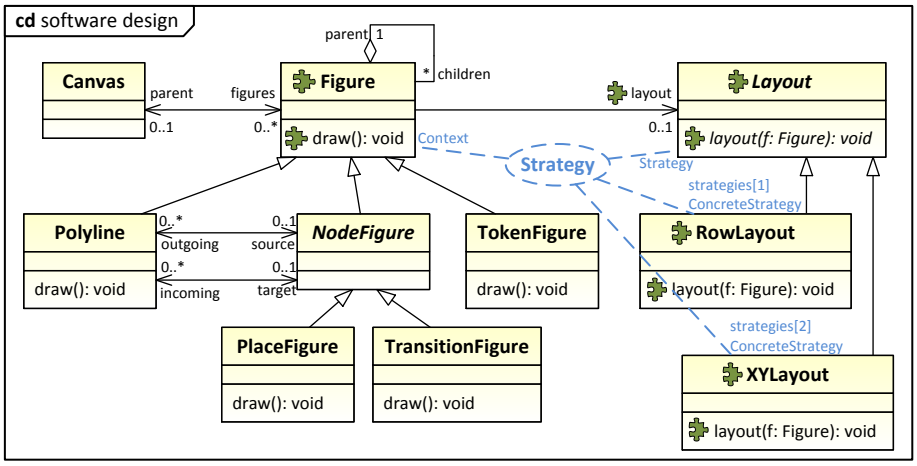


Abbildung 4.15: Musterrollen-im-Entwurf-Sicht (Klassendiagramm): Entwurf mit Markierungen und UML-Kollaborationen

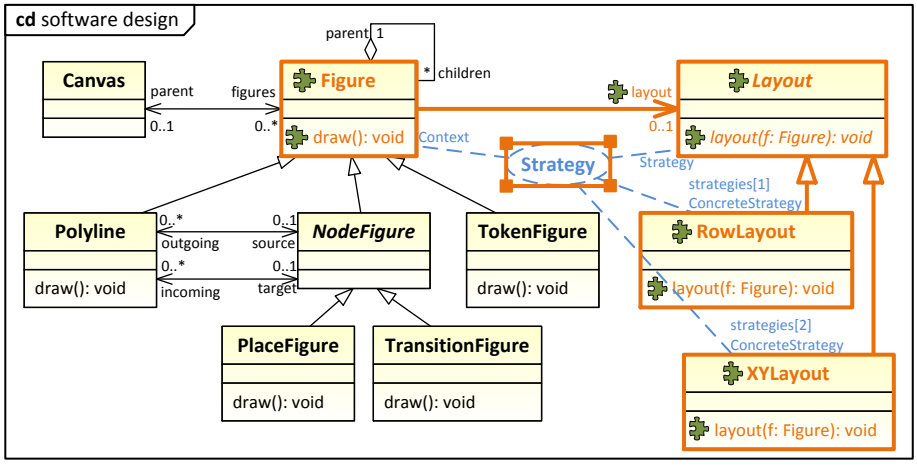


Abbildung 4.16: Musterrollen-im-Entwurf-Sicht (Klassendiagramm): Entwurf mit Markierungen, UML-Kollaborationen und selektierter Musterimplementierung

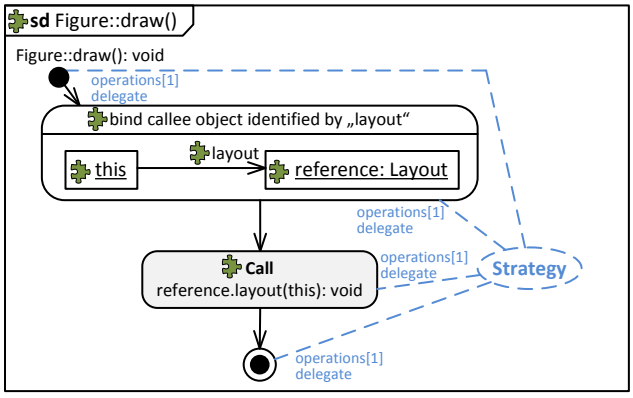


Abbildung 4.17: Musterrollen-im-Entwurf-Sicht (Story-Diagramm): Entwurf mit Markierungen und UML-Kollaborationen

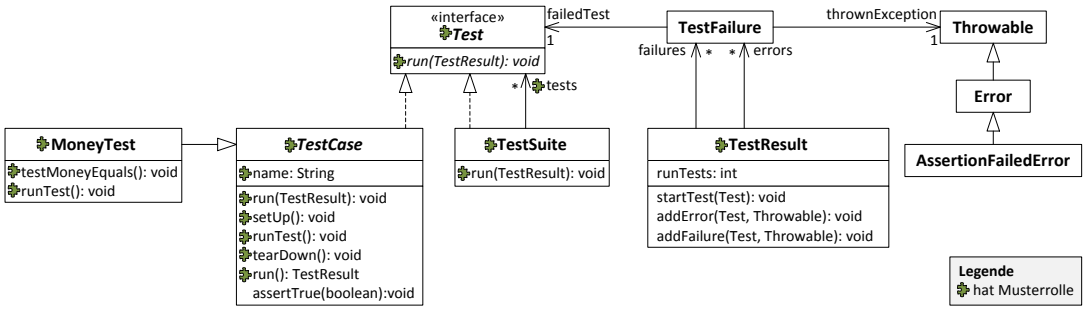


Abbildung 4.18: Entwurfssicht: Visualisierung von Anwendungsstellen, niedriger Detailgrad

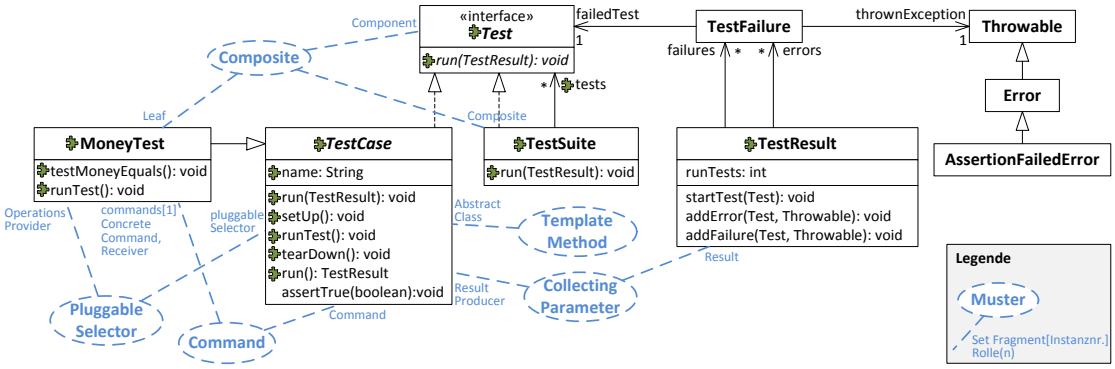


Abbildung 4.19: Musterrollen-im-Entwurf-Sicht: Visualisierung von Anwendungsstellen, hoher Detailgrad

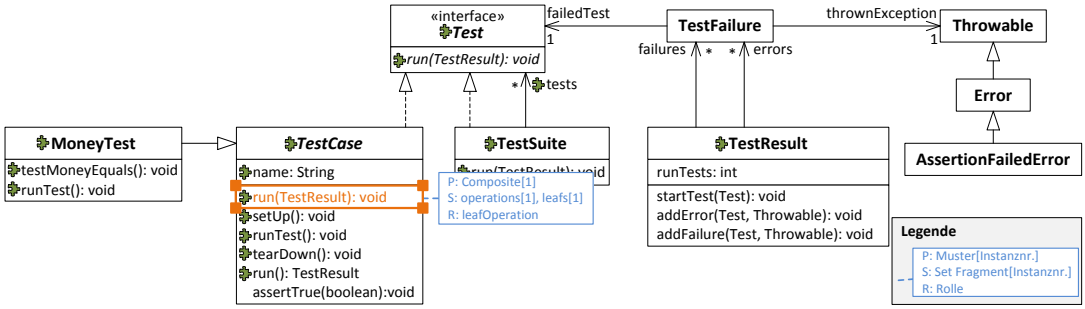


Abbildung 4.20: Entwurfssicht mit selektiertem Entwurfselement

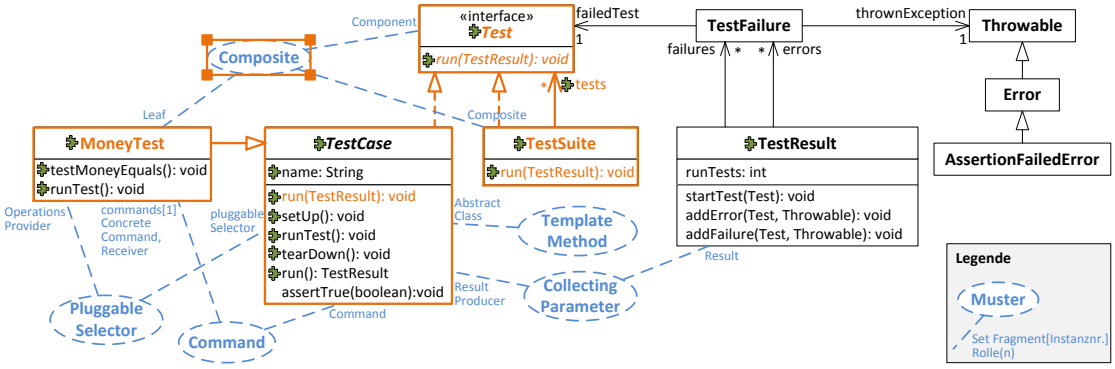


Abbildung 4.21: Musterrollen-im-Entwurf-Sicht mit selektierter Musterimplementierung

den Entwurf, bei welcher ausschließlich die an einer bestimmten (vorher ausgewählten) Musterimplementierung beteiligten Entwurfsteile dargestellt werden, z.B. ausschließlich die Klassen, Methoden, Attribute, Assoziationen und Vererbungsbeziehungen, welche an einer bestimmten Strategy-Musterimplementierung beteiligt sind.

Anordnung analog zur Spezifikation

Ein Beispiel für drei Ausprägungen einer solchen Ansicht ist in den Abb. 4.22 bis 4.24 (S. 97) aufgeführt. Die Ansicht verwendet die Notation der verwendeten Entwurfssprache, hier die der Klassendiagramme, und ordnet die beteiligten Entwurfselemente (sofern möglich[6]) analog zu ihren Entsprechungen in der zugehörigen Musterspezifikation an (vgl. Strategy-Musterspezifikation oben in Abb. 4.2, S. 82).

Set-Fragment-Instanz

Set Fragments in Musterspezifikationen beschreiben mehrere mögliche Ausprägungen bestimmter Entwurfsteile. In der Musterrollen-im-Entwurf-Ausschnittssicht wird nur eine davon dargestellt. Die dargestellte Ausprägung / Instanz eines Set Fragments wird durch eine ID angegeben. In der Abb. 4.22 z.B. wird die Set-Fragment-Instanz Nr. 2 zu dem Set Fragment strategies und Set-Fragment-Instanz Nr. 1 zu dem Set Fragment operations dargestellt.

Alle an einer Set-Fragment-Instanz beteiligten Entwurfsteile lassen sich hervorheben, um die ggf. mehrfach vorkommenden Teile des Entwurfs und variablen Anteile der Ansicht kenntlich zu machen. In der Abb. 4.23 sind alle Beteiligten der Set-Fragment-Instanz 2 zu dem Set Fragment strategies farblich hervorgehoben, während in der Abb. 4.24 alle Beteiligten der Set-Fragment-Instanz 1 zu dem Set Fragment operations farblich hervorgehoben sind.

4.5 Einschränkungen

keine Korrespondenzen bei Kanten

Eine Einschränkung meiner Modellierung von Anwendungsstellen ist die fehlende Möglichkeit, die Beziehungen einer Musterspezifikation (z.B. Vererbungsrelationen) den zugehörigen Beziehungen einer Entwurfsmusterimplementierung im Entwurfsmodell zuordnen zu können. Bisher bietet das Meta-Modell für Anwendungsstellen (siehe Abb. 4.5, S. 86) nur die Möglichkeit, einer Musterrolle ein oder mehrere Entwurfselemente zuzuordnen. (Ein RoleBinding-Objekt ordnet einem DesignElement-Objekt ein oder mehrere EObject-Objekte zu. Da Vererbung und andere Beziehungen in Ecore nicht als EObjects repräsentiert werden, wurde auf diese Möglichkeit der Einfachheit halber verzichtet.)

Keine Musterkomposition

Zusammenhänge zwischen unterschiedlichen Musterimplementierungen wie z.B. eine Komposition mehrerer Entwurfsmuster in überlappenden Musterimplementierungen lassen sich nicht explizit beschreiben. Die Rollenzuordnungen zu jeder einzelnen Musterimplementierung werden separat voneinander modelliert.

Vorteile der Sichten nicht belegt

Für die vorgestellten Sichten auf Musterimplementierungen und Rollenzuordnungen konnte bisher nicht empirisch nachgewiesen werden, dass sie denen anderer Ansätze überlegen sind. Zusätzlich zur Diskussion der Vor- und Nachteile im Vergleich zu verwandten Arbeiten (siehe Abschnitt 9.2.2) sollte ihr Nutzen zukünftig anhand größerer, praxisnaher Anwendungsbeispiele evaluiert und ggf. anhand ei-

[6] In Klassendiagrammen können nur Klassen und einige Beziehungen so angeordnet werden wie in der Musterspezifkation. Methoden und Attribute werden entsprechend der UML-Notation immer innerhalb der Klassen positioniert.

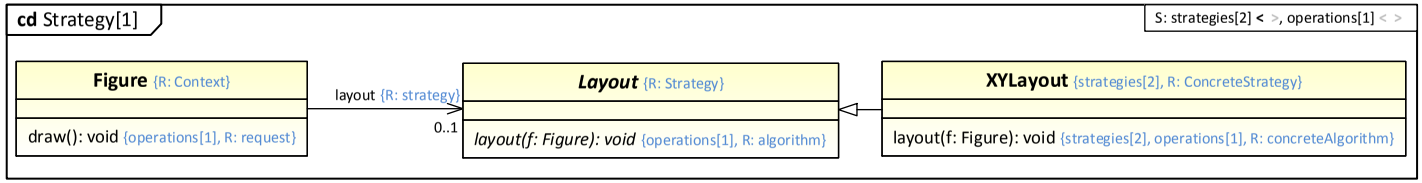


Abbildung 4.22: Isolierte Darstellung einer Musterimplementierung in einem Klassendiagramm

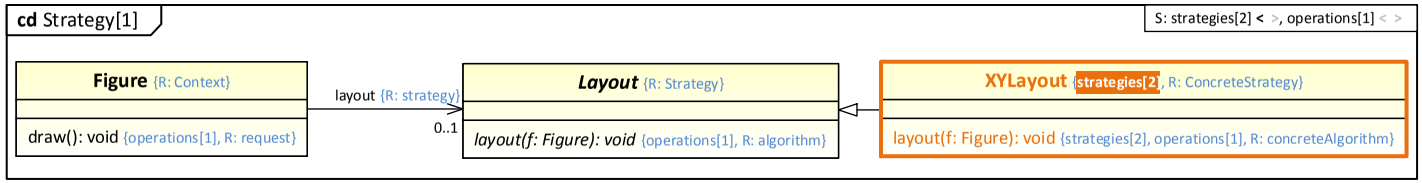


Abbildung 4.23: Musterimplementierung mit Hervorhebung der Elemente in der Set-Fragment-Instanz strategies[2]

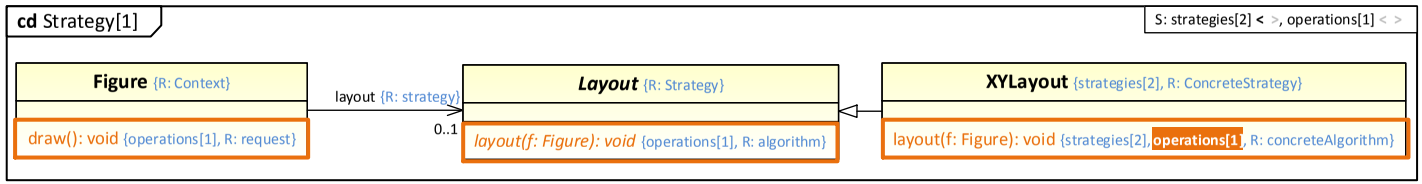


Abbildung 4.24: Musterimplementierung mit Hervorhebung der Elemente in der Set-Fragment-Instanz operations[1]

ner empirischen Studie (z.B. einer vergleichenden Fallstudie, einer Feldstudie oder eines kontrollierten Experiments) [Pre01] belegt werden.

4.6 Zusammenfassung und Ausblick

Anwendungsstellen erfassen

In diesem Kapitel wird ein Verfahren zur Modellierung und Visualisierung von Musteranwendungsstellen bzw. von Musterimplementierungen vorgestellt. Eine Anwendungsstelle oder Musterimplementierung wird im Wesentlichen durch Zuordnung der Musterrollen einer Musterspezifikation zu den sie einnehmenden Elementen eines Entwurfsmodells modelliert. Auf ähnliche Weise werden Zuordnungen von Set Fragments zu entsprechenden Strukturen im Entwurfsmodell erfasst.

unterteilte Korrespondenzen

Mit Hilfe baumartiger Token-Strukturen kann eine komplexe Zuordnung in mehrere partielle Zuordnungen unterteilt werden. Das ist z.B. bei der Zuordnung einer Aktion zu den sie implementierenden Teilen eines Story-Diagramms sinnvoll.

diverse Visualisierungen

Zur Repräsentation vorliegender Musterimplementierungen im Entwurf werden mehrere Darstellungen mit verschiedenen Detailgraden vorgeschlagen. Es können an Musterimplementierungen beteiligte Teile des Entwurfs in Klassen- oder Story-Diagrammen markiert oder detaillierte Informationen zu implementierten Mustern, eingenommenen Rollen und Set Fragments präsentiert werden. Ergänzend dazu können Anwendungsstellen und Musterimplementierungen in einer zu den Musterspezifikationen sehr ähnlichen Notation dargestellt werden. In allen detaillierten Darstellungen werden Set Fragments und Set-Fragment-Instanzen geeignet visualisiert und werden in der Musteranwendungssicht, insb. zur Musteranwendung, besonders kompakt dargestellt.

Ausblick

Zukünftig wäre eine genauere Untersuchung der vorgestellten Sichten auf den Entwurf und die Musterimplementierungen hilfreich, um den Nutzen der Darstellungen im Vergleich zu bisherigen Ansätzen an größeren, praxisnahen Anwendungsbeispielen, idealerweise anhand von empirischen Studien besser einschätzen zu können.

5 Synthese von Musterimplementierungen

Mit meiner Arbeit verfolge ich insbesondere das Ziel, Entwurfsfehler und Design-Erosion zu reduzieren, indem Musterimplementierungen visualisiert (Ziel 2 in Abschn. 1.2) und nach jeder Entwurfsänderung automatisch geprüft werden können (Ziel 3 in Abschn. 1.2). Um den Aufwand für das Erstellen der dazu notwendigen, detaillierten Modelle aller Musterimplementierungen zu minimieren, sollen solche Modelle bei meinem Ansatz weitestgehend automatisch und schon während der initialen Musteranwendung erzeugt werden. Das Generieren von Musterimplementierungen samt zugehöriger Anwendungsstellenmodelle soll den Aufwand bei der Musteranwendung und der Dokumentation der Anwendungsstellen senken. Eine interaktive Benutzerführung und geeignete Visualisierungen von Anwendungsstellen sollen ergänzend dazu die Wahrscheinlichkeit für Fehler bei der Musteranwendung senken (Ziel 1 in Abschn. 1.2).

Zweck der Generierung von Musterimplementierungen

Es ist eine besondere Herausforderung, möglichst viele Schritte einer Musteranwendung zu automatisieren, ohne die bei Entwurfsmustern besonders wichtige Flexibilität bei der Wahl einer Musterimplementierung zu verlieren. Besonders schwierig ist die Übersetzung einer Musterspezifikation in eine Musterimplementierung. Das Ergebnis der Übersetzung hängt in vielen Fällen nicht nur von der Musterspezifikation, sondern zusätzlich vom Kontext im Entwurfsmodell ab. Zum Beispiel enthält die Spezifikation einer Operation in den meisten Fällen keine Signatur. Bei der Übersetzung einer solchen Operation in eine Methode im Entwurfsmodell kann die Methodensignatur von einer bereits existierenden, zu überschreibenden Methode im Entwurfsmodell abhängen. Das Ergebnis einer solchen Übersetzung hängt dann von der Anwendungsstelle ab. Eine weitere Herausforderung stellt die Generierung eines Anwendungsstellenmodells dar. Die Schwierigkeit besteht darin, alle für eine Visualisierung und eine Prüfung von Musterimplementierungen nötigen Details zu erfassen und dabei insbesondere die Zuordnung zu Set Fragments und spezifiziertem Verhalten zu erfassen.

Herausforderungen

In diversen Arbeiten wurde gezeigt, dass einige Schritte einer Entwurfsmusteranwendung automatisiert werden können. Die bisherigen Ansätze haben jedoch signifikante Schwächen (Details in Abschn. 9.3). Durch Erzeugen einer Art Beispielimplementierung oder Verwenden einer Implementierungsschablone (Quellcode oder ein UML-Modell-Ausschnitt) ist die Flexibilität bei der Musteranwendung eingeschränkt. Die Möglichkeit, Teile eines vorliegenden Entwurfs wiederzuverwenden und eine Musterimplementierung durch Vervollständigen des Entwurfs zu erzeugen, bieten nur wenige Arbeiten. Den meisten Ansätzen fehlt ein mit Set Fragments vergleichbares Konzept zur Vervielfältigung sich als Ganzes wiederholender Strukturen einer Entwurfslösung wie z.B. der Fabriken und Produkte des Musters Abstract Factory (siehe Abb. 3.3, S. 43). Bis auf wenige Ausnahmen wird bei der Musteranwendung nur Struktur, aber kein Verhalten generiert. Außerdem wird ein detailliertes Modell der Anwendungsstellen, welches zur Visualisierung

Grenzen verwandter Arbeiten

und Prüfung von Musterimplementierungen genutzt werden kann, in den wenigsten Fällen bei einer Musteranwendung miterzeugt.

Lösungsansatz

Bei meinem Ansatz werden Entwickler bei der Anwendung von Entwurfsmustern geleitet. Dazu wird der aktuelle Stand einer Musteranwendung im Entwurfsmodell und in der Spezifikation des anzuwendenden Musters visualisiert. Die Musteranwendung erfolgt semiautomatisch. Nach der manuellen Zuordnung der Musterrollen zu existierenden Entwurfsteilen erfolgt die Übertragung der spezifizierten Entwurfslösung in den Entwurf automatisiert. Dabei wird auch ein Modell der Anwendungsstelle generiert, welches die anschließende Visualisierung und Überprüfung der Musterimplementierung ermöglicht.

Die Anwendung eines Entwurfsmusters erfolgt in zwei Schritten. Im ersten Schritt werden die Anwendungsstelle und die zu erzeugende Implementierungsvariante durch Benutzerinteraktionen bestimmt und automatisch in einem Modell persistiert. Im zweiten Schritt wird die Implementierungsvariante in das vorliegende Entwurfsmodell übersetzt. Dabei werden Details der Implementierung wie z.B Methodensignaturen abhängig von dem vorliegenden Entwurfsmodell an die konkrete Situation angepasst. Neben der Klassenstruktur werden auch Verhaltensmodelle, Story-Diagramme (siehe Abschn. 2.4), erzeugt.

Damit Muster in nahezu jeder Situation angewendet werden können, ordnet ein Entwickler Teile des vorliegenden Entwurfs den Musterrollen zu und die bisher nicht eingenommenen Rollen werden in Entwurfsteile übersetzt. Zusätzliche Flexibilität wird durch die Möglichkeit erreicht, durch Set Fragments markierte Teile einer Entwurfslösung zu vervielfältigen und eine generierte Musterimplementierung nachträglich anzupassen (z.B. Operationen in eine Oberklasse verschieben).

Wissenschaftliche Beiträge

Mit diesem Kapitel beantworte ich die zweite der vier Forschungsfragen aus Kapitel 1 (siehe S. 10). Ich zeige auf wie Entwurfsmuster halbautomatisch angewandt und Entwickler durch die Anwendung geführt werden können. Zu den wesentlichen wissenschaftlichen Beiträgen dieses Kapitels gehören:

- ein flexibles, halbautomatisches Verfahren zur Anwendung spezifizierter Entwurfsmuster
- eine Methode zur Bestimmung gültiger Implementierungsvarianten durch sogenanntes *Auffalten* von Set Fragments (Abschn. 5.4), was durch vordefinierte Operationen zur Vervielfältigung von in Set Fragments spezifizierten Teilen einer Entwurfslösung geschieht
- eine modulare, Kontext-abhängige Modelltransformation zur Übersetzung einer Musterspezifikation in eine Musterimplementierung
- ein Verfahren zur automatischen Verknüpfung einer Musterimplementierung (Struktur und Verhalten) mit der zugehörigen Musterspezifikation

Kapitelstruktur

Nach einem Überblick zu meinem Musteranwendungsverfahren erläutere ich das Vorgehen an einem Beispiel. In Abschnitt 5.3 erkläre ich wie eine Anwendungsstelle und die gewünschte Implementierungsvariante interaktiv bestimmt werden. Die Generierung des Anwendungsstellenmodells und seine Übersetzung in den Entwurf beschreibe ich in den Abschnitten 5.4 und 5.5. Zum Abschluss einer Musteranwendung ggf. notwendige Schritte und mögliche, nachträgliche Anpassungen einer Musterimplementierung erläutere ich in den Abschnitten 5.6 und 5.7. Abschließend diskutiere ich die Grenzen des Ansatzes und fasse zusammen.

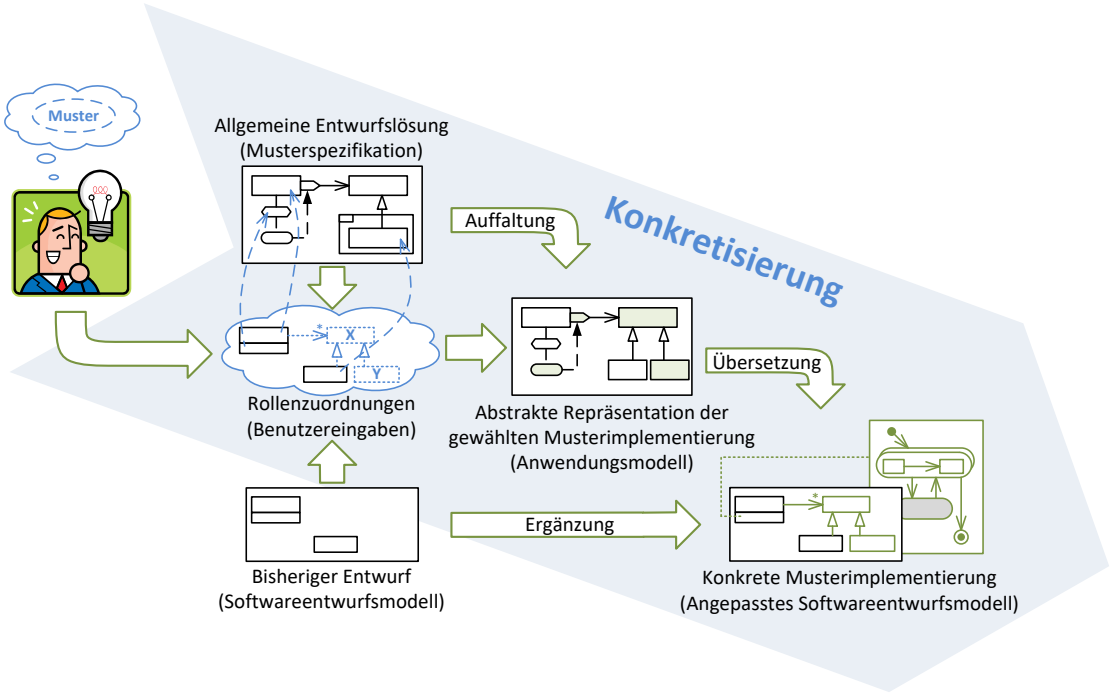


Abbildung 5.1: Synthese einer Musterimplementierung (vgl. Abb. 3.5, S. 46 und Abb. 4.3, S. 83)

5.1 Überblick

Eine Musterimplementierung im Entwurfsmodell wird bei meinem Ansatz in zwei Schritten aus einer Musterspezifikation und sie ergänzenden Benutzereingaben generiert. Diese Schritte sind in der Abb. 5.1 skizziert. Zunächst wird durch Auffaltung der Set Fragments aus der Musterspezifikation (siehe Abschn. 5.4) unter Verwendung von Benutzereingaben und Auswertung des schon vorhandenen Softwareentwurfsmodells ein *Anwendungsmodell* generiert. Anschließend wird das Anwendungsmodell unter Berücksichtigung des schon existierenden Softwareentwurfsmodells in eine Musterimplementierung durch Anpassen des Softwareentwurfsmodells übersetzt. Dabei wird die allgemeine, abstrakt beschriebene Entwurfslösung aus der Musterspezifikation schrittweise zu der tatsächlich verwendeten Entwurfslösung konkretisiert, was ich im Folgenden genauer erläutere.

Rollenzuordnung

Während der initialen Entwicklung einer Software oder bei späteren Anpassungen einer existierenden Software identifiziert ein Entwickler eine Stelle im Entwurf, an der er ein Entwurfsmuster anwenden möchte. Er wählt die zugehörige Musterspezifikation und ordnet den Elementen in seinem Entwurf die Musterrollen zu, die von den Entwurfselementen eingenommen werden sollen. Die Entwicklungsumgebung leitet den Entwickler bei dieser Aufgabe, indem sie auf fehlende Entwurfsteile hinweist und nur gültige Rollenzuordnungen erlaubt. Wurde einer Musterrolle noch kein Entwurfselement zugeordnet, so wird angenommen, dass es bei der Synthese der Musterimplementierung automatisch ergänzt werden soll. Für jedes Set Fragment der Musterspezifikation gibt der Entwickler an, wie häufig die darin definierte Entwurfsstruktur in seinem Softwareentwurf vorkommen soll.

Das Ergebnis dieser manuellen Schritte ist ein automatisch generiertes Modell der Rollenzuordnungen, das Korrespondenzmodell (Abb. 4.3, S. 83).

Ausgehend von der Musterspezifikation, den Benutzereingaben und dem bisherigen Softwareentwurf lässt sich die gewünschte Musterimplementierung im Entwurfsmodell ableiten. Die zu diesem Zweck entwickelte Modelltransformation habe ich aus Gründen der Komplexität und zwecks Wiederverwendbarkeit unterteilt.

Auffaltung

Zunächst wird eine abstrakte Repräsentation der gewählten Musterimplementierung, das Anwendungsmodell, automatisch hergeleitet (Abb. 5.1). Diesen Schritt nenne ich Auffaltung der Musterspezifikation. Eine Musterspezifikation lässt je Set Fragment offen wie häufig der im Set Fragment spezifizierte Teil der Entwurfslösung in einer Musterimplementierung vorkommen darf. Im Gegensatz dazu legt ein Anwendungsmodell für jedes Set Fragment eine bestimmte Anzahl der Vorkommen fest und repräsentiert die daraus resultierende, nämlich die vom Entwickler für seine konkrete Situation gewählte Implementierungsvariante (siehe Abb. 3.5, S. 46 und Abb. 4.3, S. 83). Insbesondere enthält das Anwendungsmodell keine Set Fragments, sondern den darin enthaltenen, ggf. vervielfältigten Teil der Entwurfslösung. Zum Beispiel wird bei Anwendung des Musters Observer (Musterspezifikation in Abb. 3.7, S. 52) jede Klasse, welche die in einem Set Fragment spezifizierte Rolle ConcreteObserver einnehmen soll, durch ein eigenes Typ-Objekt im Anwendungsmodell repräsentiert und erhält alle laut Musterspezifikation geforderten Eigenschaften wie eine concreteUpdate-Operation. Die Set Fragments aus der Musterspezifikation werden sozusagen aufgefaltet, indem die vom Entwickler gewünschte Anzahl der im Set Fragment spezifizierten Entwurfsstruktur im Anwendungsmodell erzeugt wird.

Übersetzung

Eine Implementierung des Musters im Entwurfsmodell entsteht durch die selektive Übersetzung des Anwendungsmodells in die Modellierungssprache, die im Entwurfsmodell eingesetzt wird. Dabei werden nur die Musterrollen übersetzt, zu denen entsprechende, das Muster implementierende Entwurfsteile fehlen. Bei meiner Umsetzung der vorgestellten Konzepte habe ich als Modellierungssprache für den Entwurf eine Kombination aus Ecore (bzw. EMOF[1]) und Story-Diagrammen [NZJ13, vDHH+12, FNTZ00] (siehe Abschn. 2.4.2) verwendet. Das Entwurfsmodell ist in meinem Fall ein Ecore-Klassenmodell (basierend auf dem EMF-Framework[2] [SBPM08]), welches zur Beschreibung des Verhaltens von Operationen mit Story-Diagrammen verknüpft ist. Das Anwendungsmodell wird also in ein Ecore-Klassenmodell mit Story-Diagrammen übersetzt. Dabei wird das Entwurfsmodell ausschließlich ergänzt und es werden nur die Elemente im Anwendungsmodell übersetzt, denen vom Entwickler kein entsprechendes Element aus dem Entwurfsmodell zugeordnet wurde. Es werden fehlende Klassen, Operationen, Attribute, Assoziationen und Vererbungsbeziehungen, aber auch ganze Story-Diagramme ergänzt. Bereits zum Teil modelliertes Verhalten kann (bisher) nicht automatisch vervollständigt werden.

Anwendungsstellen erfassen

Während einer Musteranwendung werden das entstehende Anwendungsmodell und die Zuordnung der das Muster implementierenden Entwurfsteile zu ihren Rollen (das Korrespondenzmodell) automatisch in einem das Entwurfsmodell ergän-

[1] siehe Fußnote auf S. 13
[2] EMF: http://www.eclipse.org/modeling/emf/

zenden Dekorationsmodell[3], dem Modell der Anwendungsstellen, persistiert (siehe Abb. 4.3, S. 83). Auf diese Weise werden ohne Mehraufwand für den Entwickler auch die Anwendungsstellen markiert und die Musterimplementierung explizit modelliert.

Davon profitieren Entwickler in zweierlei Hinsicht. Zum einen geht auf diese Weise die Information über angewendete Muster nicht verloren und erleichtert das Verstehen des Entwurfs bei späteren Anpassungen. Zum anderen kann diese Art der Dokumentation von Musteranwendungen dazu genutzt werden, den Entwurf automatisch auf Konsistenz zu den verwendeten Musterspezifikationen zu prüfen. Beides hilft schließlich dabei, Design-Erosion einzudämmen, weil Entwurfsänderungen im Bewusstsein über angewendete Entwurfsmuster erfolgen und vesehentliche Abweichungen von den ursprünglich eingesetzten Mustern vermieden werden.

Anwendungsszenarien

Das entwickelte Musteranwendungsverfahren ist durch die nahezu beliebige Zuordnung von Musterrollen zu Entwurfselementen und die automatische Ergänzung von fehlenden Entwurfsteilen flexibel einsetzbar. Insbesondere werden durch dieses Vorgehen vier Musteranwendungsszenarien unterstützt:

alles generiert

1. Hat der Entwickler keiner Musterrolle ein Element aus dem Entwurfsmodell zugeordnet, wird die gesamte in der Musterspezifikation definierte Entwurfsstruktur inklusive des zugehörigen Verhaltens neu angelegt.

Teile generiert

2. Bei einer partiellen Zuordnung der Musterrollen wird nur Fehlendes im Entwurfsmodell ergänzt. Durch die Auswahl der Rollen, die Entwurfselementen zugeordnet werden, kann ein Muster in verschiedensten Situationen angewendet werden.

nur Korrespondenzen

3. Durch die Zuordnung aller Musterrollen zu schon existierenden Entwurfselementen kann ein zuvor angewandtes Muster nachträglich als solches markiert, visualisiert und überprüft werden. So wird ein Mustervorkommen im Entwurf dokumentiert, indem es durch Verknüpfung mit einer Musterspezifikation und in Form von Rollenzuordnungen explizit modelliert wird.

nachträgliche Anpassung

4. Zeitlich zurückliegende Musteranwendungen mit einer vorliegenden Rollenzuordnung können nachträglich angepasst werden. Zum Beispiel kann in einer Implementierung des Strategy-Musters nachträglich eine zusätzliche Implementierung einer Strategie (ConcreteStrategy, siehe Abb. 4.6, S. 88) ergänzt werden. Die schon vorliegenden Rollenzuordnungen werden in diesem Fall weitergenutzt und ergänzt.

Wird das Entwurfsmodell nach Musteranwendung geändert, wird man auf ggf. entstandene Inkonsistenzen zur Musterspezifikation und auf verlorengegangene Korrespondenzen hingewiesen, damit man die Abweichungen im Entwurf beheben und die Korrespondenzen nachpflegen kann.

Granularitätslevel

Die zwei Transformationsschritte bei der Synthese einer Musterimplementierung im Entwurfsmodell, also die Auffaltung und die Übersetzung, erstrecken sich wie

[3]Mit Dekorationsmodell meine ich ein Modell M_2, welches ein anderes Modell M_1 ohne Änderungen an M_1 erweitert (M_1 „dekoriert“). So wird sichergestellt, dass das ursprüngliche Modell M_1 mit den bisherigen Werkzeugen bearbeitbar bleibt.

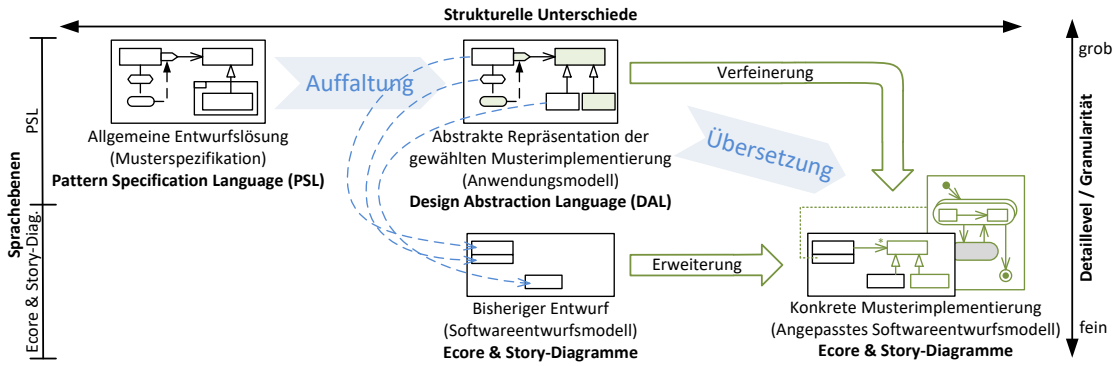


Abbildung 5.2: Sprachebenen und Granularität bei der Auffaltung und Übersetzung

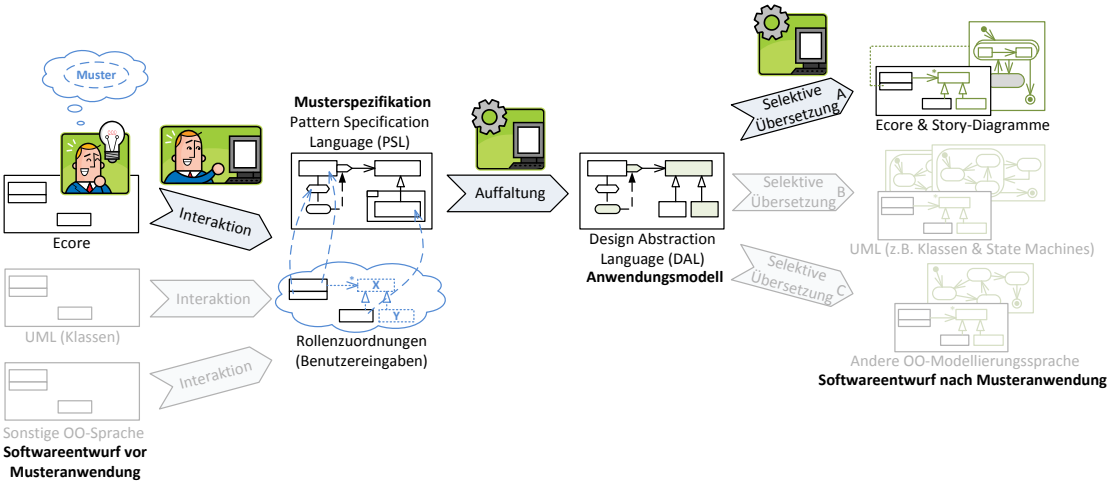


Abbildung 5.3: Erweiterbarkeit des Ansatzes auf andere Modellierungssprachen

in der Abb. 5.2 dargestellt über verschiedene Sprachebenen und Modellgranularitätslevel. Bei der Auffaltung ändert sich nur die Struktur (horizontale Achse). Set Fragments werden durch mehrfache Vorkommen ihres Inhalts ersetzt. Die Modellierungssprache bleibt im Wesentlichen dieselbe, weil das Anwendungsmodell in der DAL beschrieben wird, welche eine Teilmenge der PSL, also der Musterspezifikationssprache ist. Damit bleibt auch die Granularität der Modelle während der Auffaltung erhalten. Bei der Übersetzung dagegen ändert sich nicht nur die Struktur, sondern auch die Granularität (vertikale Achse). Das in der DAL beschriebene, abstrakte und damit kompakte Anwendungsmodell wird in einen Ausschnitt eines wesentlich detaillierteren Entwurfsmodells (Ecore- & Story-Diagramm-Modelle) übersetzt und somit verfeinert bzw. konkretisiert. Gleichzeitig wird ein schon existierendes Entwurfsmodell durch Ergänzen von ein Muster implementierenden Elementen strukturell modifiziert. Die Modellierungssprache und damit auch der Abstraktionsgrad des Entwurfsmodells bleiben bei dieser Änderung erhalten.

Übertragbarkeit auf andere Sprachen

Weil das Anwendungsmodell in der abstrakten DAL beschrieben wird (siehe Abschn. 3.2), ist es – wie die Musterspezifikation auch – unabhängig von der für den Softwareentwurf verwendeten Modellierungssprache. Dadurch ließe sich die Auffaltungstransformation prinzipiell wiederverwenden, wenn man die für den Softwareentwurf verwendete Modellierungssprache wechseln würde (sie-

he Abb. 5.3). So könnte der Ansatz z.B. dahingehend erweitert werden, dass nicht Ecore-Modelle mit Story-Diagrammen als Entwurfsmodellierungssprache verwendet werden, sondern ein bestimmter UML-Dialekt oder eine Teilmenge der UML wie Klassen- und State-Machine-Modelle. Auch andere, auf bestimmte Domänen zugeschnittene, objektorientierte Modellierungssprachen wie UWE (UML-based Web Engineering) [KKZB08] sind hier denkbar.

5.2 Beispiel einer Musteranwendung

Wie ein Muster bei meinem Ansatz mit Hilfe der Entwurfsumgebung zur Anwendung kommt, wird im Folgenden an einem Beispiel erläutert.

Angenommen, ein Entwickler eines grafischen Editors für Petrinetze möchte das Observer-Muster in seinem Softwareentwurf einsetzen. Ein Teil des Softwareentwurfsmodells ist schon vorhanden (Schritt 1 in Abb. 5.4).

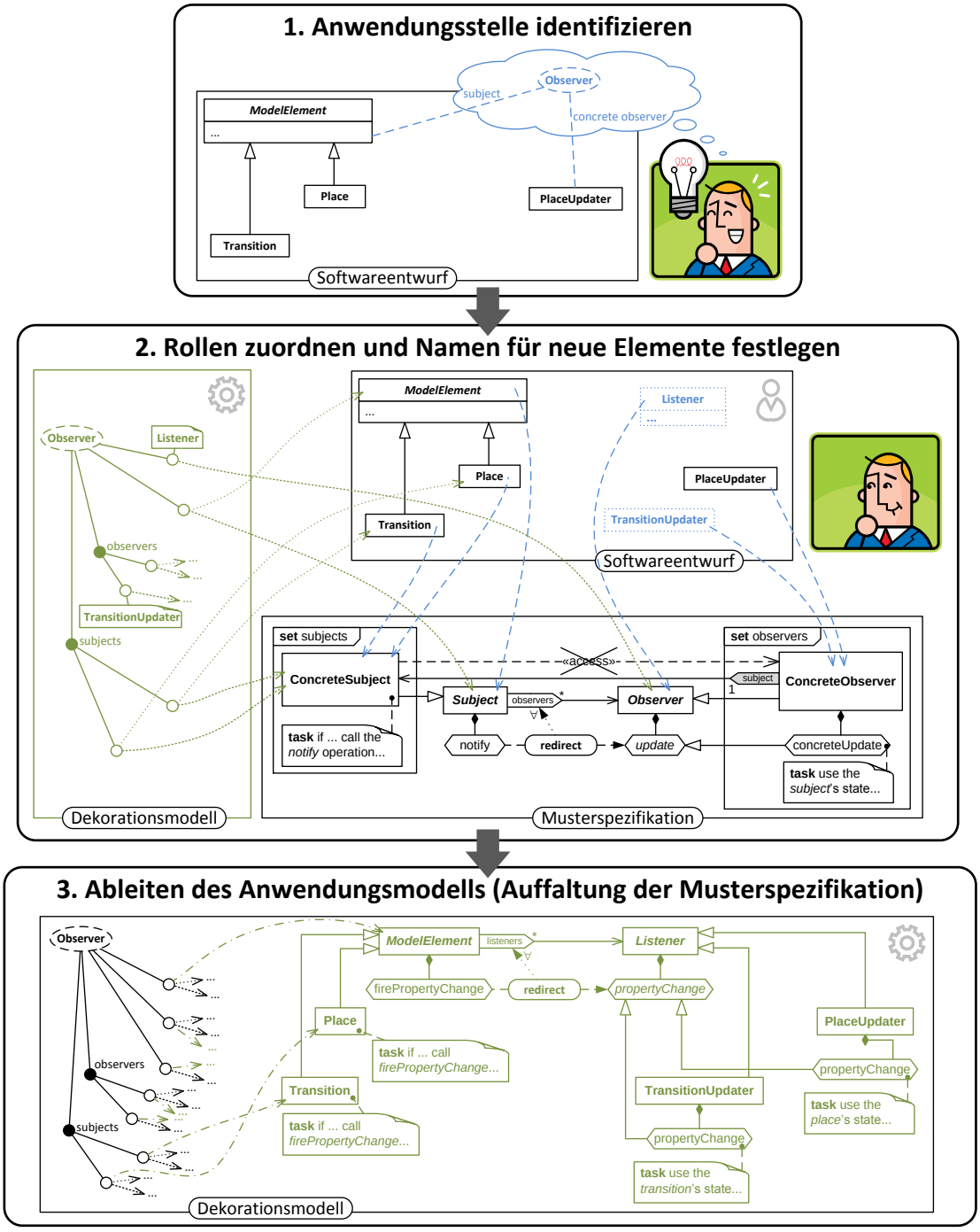
Rollenzuordnung

Um das Observer-Muster mit Hilfe der Entwurfsumgebung anzuwenden und die Anwendungsstelle automatisch als solche zu markieren, öffnet er die Spezifikation des Musters – diese wurde zuvor von einem Experten erstellt und in einem Musterkatalog abgelegt – und beginnt damit, die schon vorliegenden Klassen den Musterrollen zuzuordnen, welche die Klassen einnehmen sollen (gestrichelte, blaue Pfeile in Schritt 2 in Abb. 5.4). Zum Beispiel nehmen die Klassen Transition und Place die Rolle des Typs ConcreteSubject ein. Fehlen Klassen für bestimmte Rollen, kann der Entwickler Namen für zu erzeugenden Klassen festlegen. Zusätzlich zur schon existierenden Klasse PlaceUpdater soll eine neue Klasse TransitionUpdater die Rolle des Typs ConcreteObserver einnehmen, für die Rolle Observer soll eine Klasse Listener erstellt werden. Während der Entwickler diese Eingaben tätigt wird von der Entwurfsumgebung im Hintergrund automatisch ein Modell der Anwendungsstelle (Dekorationsmodell, Abb. 4.3, S. 83) angelegt und die eingegebenen Rollenzuordnungen darin festgehalten (neue Elemente sind grün dargestellt). Details zur Benutzerinteraktion werden in Abschnitt 5.3 (S. 108 ff.) beschrieben, während das Erstellen der Korrespondenzen in Abschnitt 5.4 (S. 112 ff.) beleuchtet wird.

Auffaltung

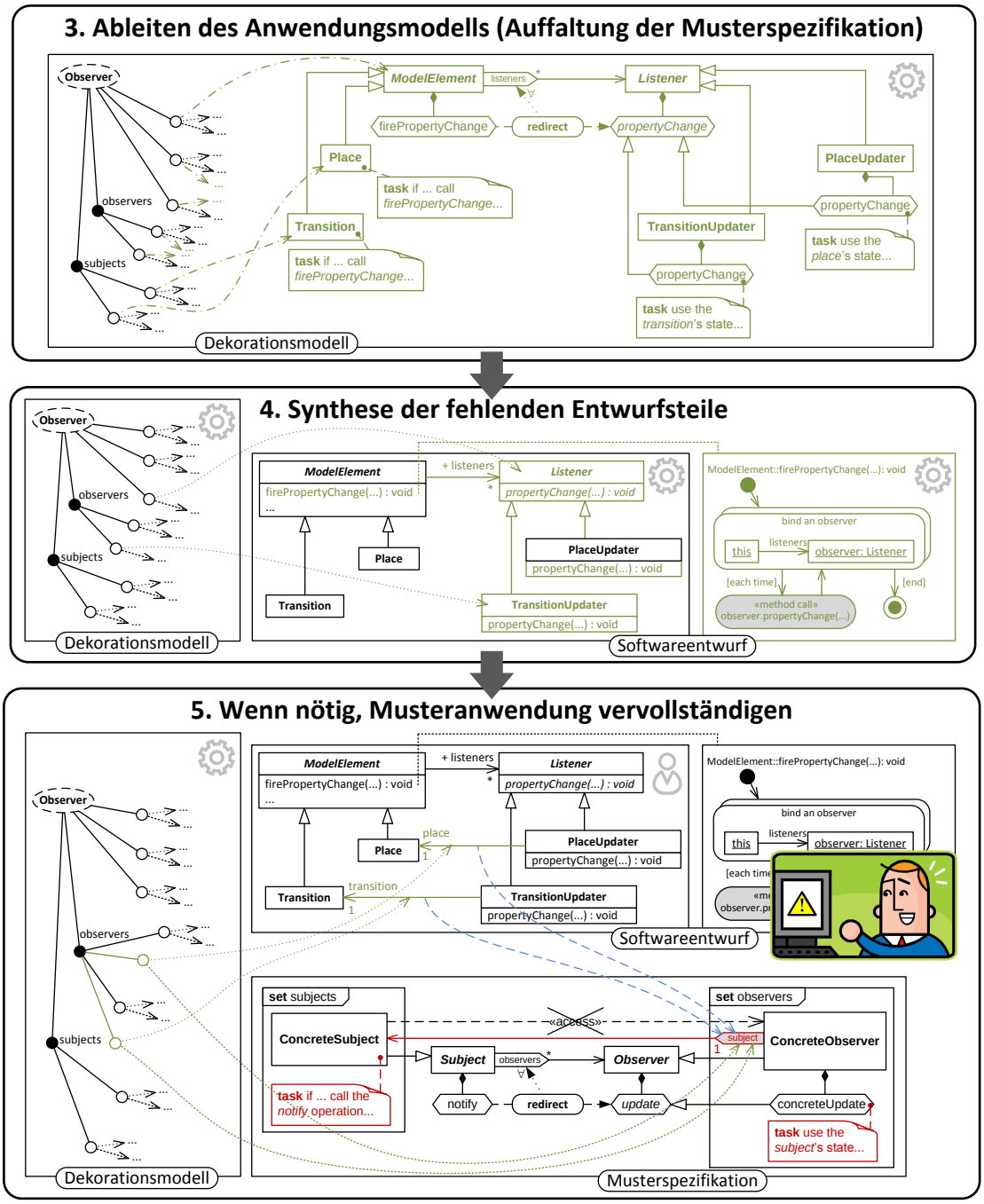
Hat der Entwickler alle existierenden Elemente des Entwurfs, welche eine Musterrolle einnehmen sollen, der entsprechenden Rolle zugeordnet, startet er die Generierung der von ihm durch seine Eingaben gewählten Musterimplementierung. Die Entwurfsumgebung leitet aus den Rollenzuordnungen eine DAL-Repräsentation der Musterimplementierung ab[4], das Anwendungsmodell (Schritt 3 in Abb. 5.4, vgl. Abb. 4.3, S. 83). Dieser Schritt entspricht der Auffaltung einer Musterspezifikation (siehe Abb. 5.1, S. 101). Dabei wird eine Kopie der Musterspezifikation angelegt, in der Set Fragments entfernt und die darin spezifizierten Strukturen in ggf. mehrfacher Ausprägung angelegt werden (siehe Abb. 3.34, S. 71). Zum Beispiel wird für das Set Fragment observers die Struktur bestehend aus einem ConcreteObserver-Typ, der zugehörigen concreteUpdate-Operation und einer Aufgabe (task) zwei Mal im Anwendungsmodell angelegt. Die Beziehungen dieser Elemente zu anderen Elementen im Anwendugsmodell werden konform zur Musterspezifikation angelegt. Details zu diesem Schritt werden in Abschnitt 5.4 (S. 112 ff.) beschrieben.

[4]Genau genommen wird Schritt 3 verzahnt mit Schritt 2 ausgeführt.



Legende: = manuelle Modelländerung, = automatisierte Modelländerung

Abbildung 5.4: Schritte bei einer Musteranwendung am Beispiel Observer – Teil 1



Legende: = manuelle Modelländerung, = automatisierte Modelländerung

Abbildung 5.5: Schritte bei einer Musteranwendung am Beispiel Observer – Teil 2

Übersetzung

Als Nächstes wird das Entwurfsmodell von der Entwurfsumgebung schrittweise erweitert (Schritt 4 in Abb. 5.5). Das Anwendungsmodell dient hierbei als Vorlage für die zu erstellende Musterimplementierung. Elemente im Anwendungsmodell, zu denen es im Entwurfsmodell keine Entsprechung gibt, werden übersetzt und die Ergebnisse mit den zugehörigen Rollen der Musterspezifikation verknüpft. Bei dem Beispiel aus Abb. 5.5 werden die Klassen Listener und TransitionUpdater, mehrere Methoden, eine Assoziation und Vererbungsbeziehungen angelegt. Außerdem wird ein neues Story-Diagramm erstellt, welches das Verhalten der Methode firePropertyChange definiert und der redirect-Aktion im Anwendungsmodell und in der Musterspezifikation entspricht. Details zu diesem Schritt werden in Abschnitt 5.5 (S. 121 ff.) beschrieben.

manuelle Schritte

Wenn in einer Musterspezifikation manuell zu erledigende Aufgaben (tasks) oder aufgrund fehlender Informationen nicht generierbare Elemente wie die subject-Referenz vorkommen (in Abb. 5.5 in Schritt 5 rot dargestellt), erfordern diese von der vorliegenden Situation abhängige Tätigkeiten zum Abschließen der Musteranwendung. Im vorliegenden Anwendungsfall (Abb. 5.5) ist z.B. aus der Musterspezifikation (bzw. aus dem Entwurfsmuster) nicht ersichtlich, was der interne Zustand einer ConcreteSubject-Klasse ist. Darum kann der Aufruf der notify-Operation als Reaktion auf eine Zustandsänderung nicht generiert werden. Da solche Tätigkeiten nicht automatisierbar sind, müssen sie manuell von Entwicklern durchgeführt werden. Details dazu werden in Abschnitt 5.6 (S. 136 ff.) beschrieben.

Bei dem Beispiel aus Abb. 5.5 muss der Entwickler bei jeder die Rolle ConcreteObserver einnehmenden Klasse im Entwurfsmodell eine Referenz ergänzen, die der subject-Referenz in der Musterspezifikation entspricht. Dazu legt der Entwickler die Referenzen place und transition im Entwurfsmodell an und ordnet sie der subject-Referenz in der Musterspezifikation zu. Die Entwurfsumgebung prüft die Konformität der zugeordneten Referenzen zur Musterspezifikation (passende Kardinalität, Rollen, etc.) und persistiert die Rollenzuordnung im Dekorationsmodell.

Für jede Aufgabenbeschreibung (Task) in der Musterspezifikation wird im Anwendungsmodell mindestens eine entsprechende, unerledigte Aufgabe erstellt (in Schritt 3). Alle diese Aufgaben muss der Entwickler erledigen, um die Musteranwendung abzuschließen und das spezifizierte Muster komplett zu implementieren. Bei dem Beispiel muss er dafür sorgen, dass die firePropertyChange-Methode bei Zustandsänderungen von Place- und Transition-Objekten aufgerufen wird. Außerdem muss er das Verhalten der propertyChange-Methoden modellieren, um auf die Zustandsänderung des beobachteten Objekts angemessen zu reagieren.

5.3 Rollenzuordnung: Musteranwendung aus Benutzersicht

Bevor eine Musterimplementierung komplett oder teilweise generiert werden kann, gibt der Anwender anhand von einigen Benutzerinteraktionen mit der Entwurfsumgebung an, welche Implementierungsvariante an welcher Stelle im Entwurf gewünscht ist und welche Rollen des anzuwendenden Musters vorliegende Teile des Entwurfs (z.B. Klassen oder Operationen) ggf. einnehmen sollen. Erst danach werden die zur Vervollständigung der gewählten Implementierungsvariante fehlenden

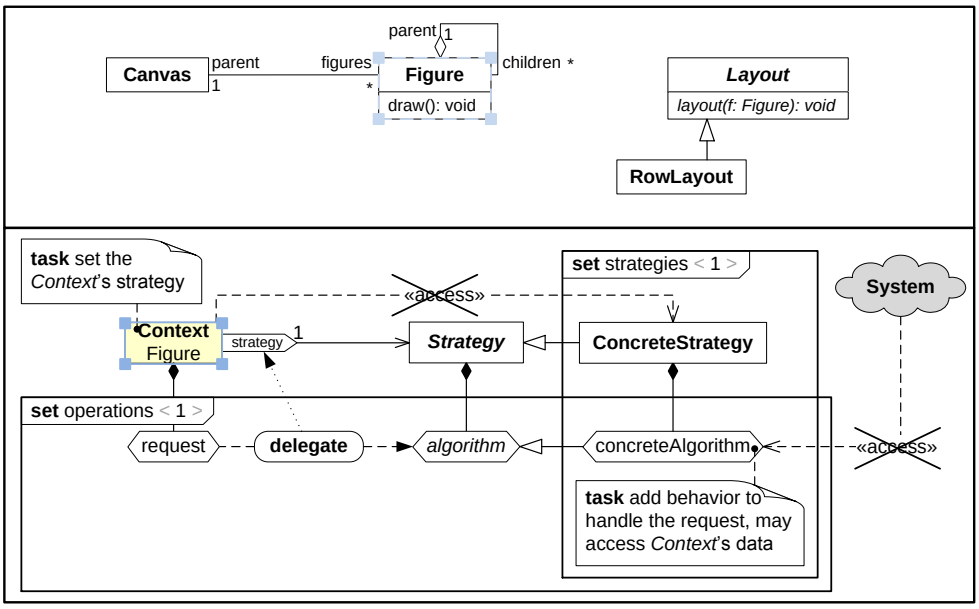


Abbildung 5.6: Zuordnung einer Klasse zu ihrer Musterrolle aus Benutzersicht

Elemente und Eigenschaften (soweit möglich) automatisch im Entwurf ergänzt. Wie ein Anwender dabei mit der Entwurfsumgebung interagiert, die Rollenbelegungen festlegt und die Set Fragments instanziiert, beschreibe ich im Folgenden.

5.3.1 Ablauf bei initialer Musteranwendung

Zur Anwendung eines Musters öffnet der Entwickler sein Entwurfsmodell und einen Katalog mit Musterspezifikationen in einer für diesen Zweck vorgesehenen Entwurfsumgebung. Aus dem Katalog mit zuvor spezifizierten Mustern und zugehörigen Entwurfslösungen wählt der Entwickler die für seine Situation passende Musterspezifikation und beginnt mit der Vorbereitung der Musteranwendung und bestimmt vor allem die Rollenzuordnung.

Dazu bietet die Entwurfsumgebung eine spezielle zweigeteilte Ansicht, welche in der Abb. 5.6) skizziert ist. Zum einen wird darin die Klassenstruktur des bisherigen Softwareentwurfs dargestellt (oben im Bild), zum anderen die gewählte Musterspezifikation und Rollenbelegung in der Musteranwendungsansicht (unten im Bild, Details in Abschnitt 4.3).

Musteranwendungsansicht

Sollen Teile des vorliegenden Softwareentwurfs bei der Implementierung des Musters wiederverwendet werden, ordnet der Anwender diese Teile den zugehörigen Rollen der Musterspezifikation zu. Durch Markieren eines Elements wie einer Klasse, einer Operation oder eines Assoziationsendes in der Klassenansicht (oder mehrerer zusammen dieselbe Rolle einnehmenden Elemente) und des zugehörigen Elements in der Musterspezifikation sowie durch anschließendes Ausführen einer Rollenzuordnungsanweisung der IDE legt der Entwickler die von dem markierten Entwurfselement zu spielende Rolle fest. In der Abb. 5.6 wird festgelegt, dass die im Entwurf bereits existierende Klasse Figure die Rolle Context der Musterspezifikation zum Strategy-Muster einnehmen soll.

Rollenzuordnung

Bei dieser Ansicht wird jede zugeordnete Musterrolle in der Musterspezifikation

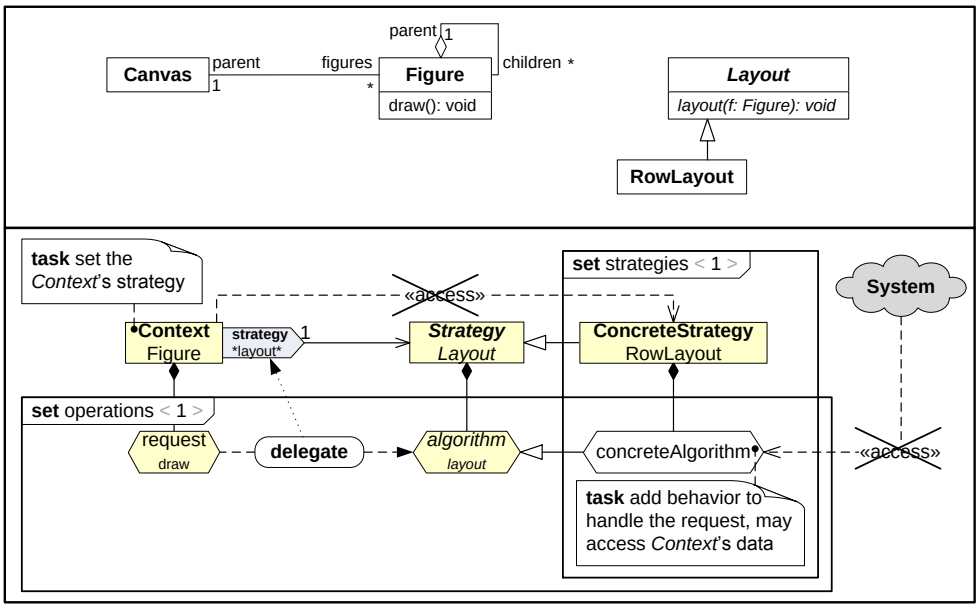


Abbildung 5.7: Eine mögliche Rollenbelegung aus Benutzersicht

farblich hervorgehoben und der Name des sie einnehmenden Entwurfselements ergänzt (hier: der Name der Klasse Figure). Kommt in einer Musterspezifikation ein Subsystem vor, können diesem mehrere Klassen zugeordnet werden. In diesem Fall werden die Namen aller zugeordneten Klassen dargestellt (siehe Abb. 8.26, S. 199).

Bei der Rollenzuordnung ist der Anwender flexibel. Die Reihenfolge bei der Zuordnung ist beliebig. Bestimmt er für eine Musterrolle kein sie einnehmendes Element aus dem Entwurf, wird das Element bei der Musteranwendung generiert.

Namen für neue Elemente

In diesem Fall kann ein Wunschname für das zu erzeugende Element, z.B. für eine Klasse, festgelegt werden. Wird keiner Rolle ein Entwurfselement zugeordnet, wird die komplette in der Musterspezifikation beschriebene Entwurfslösung in dem Entwurfsmodell neu erzeugt. Wird jeder Rolle ein Entwurfselement zugeordnet, so werden nur die ggf. fehlenden Eigenschaften und Beziehungen ergänzt.

Für das Beispiel aus Abb. 5.6 kann das Ergebnis nach weiteren Rollenzuordnungen wie in Abb. 5.7 aussehen. Hier sind fast alle Musterrollen den zugehörigen Entwurfsteilen zugeordnet worden. Ausnahmen bilden a) die Rolle und Operation concreteAlgorithm, weil bisher keine entsprechende Operation in der Klasse RowLayout vorhanden ist, b) die Referenz strategy, weil im Entwurf keine entsprechende Assoziation vorhanden ist, und schließlich c) die Aktion delegate, weil noch kein Verhaltensmodell zu der Operation draw der Klasse Figure angelegt wurde.

Für die zu erzeugende Referenz zur Rolle strategy wurde der Wunschname layout angegeben. Solche Namensgebungen werden in der Musterspezifikationsansicht mit dem Symbol * und farblich (blau) hervorgehoben.

Soll die in einem Set Fragment spezifizierte Struktur mehr als ein Mal in der Musterimplementierung vorkommen, kann der Entwickler weitere Vorkommen dieser Struktur bestimmen. Dazu wählt er ein Set Fragment, zu dem er eine zusätzliche Set-Fragment-Instanz erstellen lassen möchte. In der Ansicht aus Abb. 5.7 kann der Entwickler z.B. das Set Fragment strategies wählen. Daraufhin wird durch die

Instanzen von Set Fragments

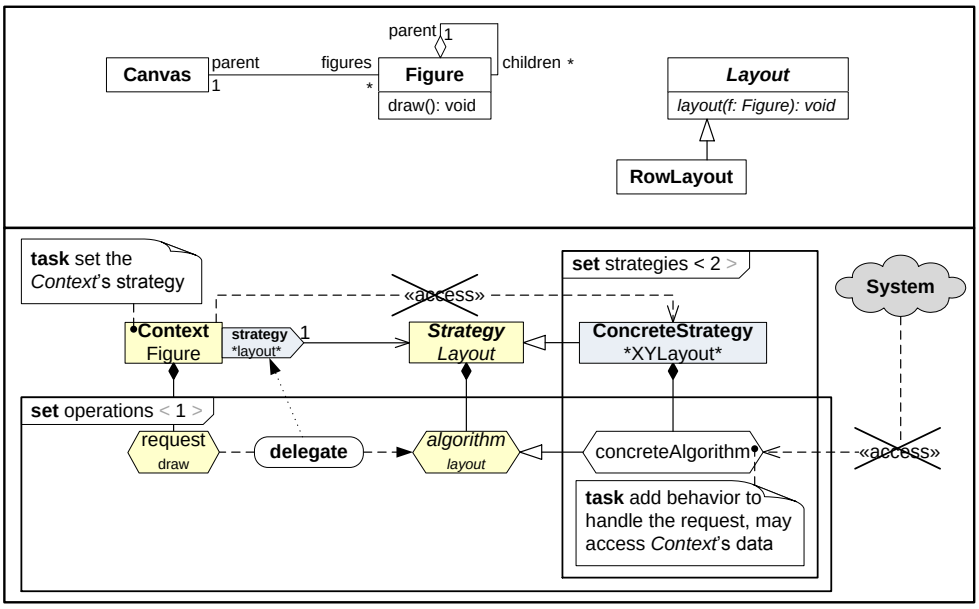


Abbildung 5.8: Zusätzliche Set-Fragment-Instanz aus Benutzersicht

Entwurfsumgebung automatisch eine neue Set-Fragment-Instanz erstellt, also ein weiteres Vorkommen der in dem Set Fragment beschriebenen Struktur im Anwendungsmodell und zugehörige Korrespondenzknoten (Role Bindings). Ergänzend dazu erhält der Anwender eine angepasste Ansicht der Musterspezifikation und der Rollenbelegung, wo er die Rollen aus dem neu instanziierten Set Fragment neu belegen kann. In der Abb. 5.8 wird die neue Set-Fragment-Instanz durch den Index 2 hinter dem Namen des Set Fragments strategies gekennzeichnet. Nun können die Rollen ConcreteStrategy und concreteAlgorithm für ein zusätzliches Vorkommen der Struktur aus dem Set Fragment strategies neu vergeben werden. Im dargestellten Beispiel wurde der Name für eine neue, die Rolle ConcreteStrategy einnehmende Klasse festgelegt (hier: XYLayout).

In der Musteranwendungsansicht (unten in Abb. 5.8) kann mit Hilfe von Vor- und Zurück-Buttons (hier mit < und > dargestellt) zwischen den Instanzen zu einem Set Fragment hin- und hergewechselt werden. Der Index gibt die gerade sichtbare Instanz an (sie sind alle nach dem Zeitpunkt ihrer Erzeugung durchnummeriert).

Nachdem alle Entwurfsteile, die eine Musterrolle einnehmen sollen, vom Entwickler der zugehörigen Rolle zugeordnet wurden, und genauso viele Vorkommen der Strukturen in Set Fragments erzeugt wurden wie vom Entwickler gewünscht, sind das Anwendungsmodell und die Beschreibung der Musteranwendung vollständig. Die automatische Musteranwendung kann durch selektive Übersetzung des abstrakten Anwendungsmodells in eine Musterimplementierung in dem Entwurfsmodell erfolgen (siehe Abschnitt 5.5).

5.3.2 Nachträgliches Ändern einer Musteranwendung

Wurde an einer bestimmten Stelle im Entwurf ein Muster bereits angewandt, so kann die Musteranwendung nachträglich verändert (erweitert) werden. Details

Instanzen von Set Fragments ergänzen

dazu werden in Abschnitt 5.7 beschrieben. Zum Beispiel lassen sich bei einem angewandten Strategy-Muster nachträglich zusätzliche konkrete Strategien oder Operationen ergänzen (Rollen ConcreteStrategy oder request, siehe Abb.5.8 auf S.111).

Das Vorgehen ist dabei analog zur initialen Musteranwendung. Der Unterschied besteht darin, dass eine komplette Rollenzuordnung (Korrespondenzmodell) und ein Anwendungsmodell bereits vorliegen. Für zu ergänzende Teile der Entwurfslösung wie eine weitere ConcreteStrategy wird durch den Entwickler eine weitere Set-Fragment-Instanz ergänzt und die zugehörigen Rollen bereits existierenden Entwurfsteilen zugeordnet oder Namen für zu generierende Entwurfsteile festgelegt. Anschließend werden fehlende Teile analog zur initialen Musteranwendung im Entwurfsmodell ergänzt.

Kein Entfernen von Set-Fragment-Instanzen

Destruktive Operationen wie das Entfernen einer Implementierung zu einem Set Fragment sind bisher nicht vorgesehen, weil insbesondere nach manueller Anpassung einer Musterimplementierung nicht zweifelsfrei entschieden werden kann, ob die einst generierten und inzwischen manuell geänderten oder mit neuen Beziehungen versehenen Entwurfsteile wieder vollständig entfernt werden können. Manuelle Ergänzungen (z.B. Beziehungen, Operationen oder Attribute) könnten noch benötigt werden, ihr Entfernen könnte zu Inkonsistenzen im Entwurfsmodell führen (z.B. Beziehungen bzw. Kanten ohne Ursprung oder Ziel), die Semantik des Modells könnte unabsichtlich von der gewünschten abweichen (z.B. wenn eine Assoziation oder Vererbungsbeziehung unabsichtlich verschwindet).

Im Gegensatz dazu lassen sich Rollenzuordnungen jederzeit nach einer Musteranwendung wieder entfernen. Das Entwurfsmodell bleibt dabei jedoch unverändert und muss bei Bedarf manuell angepasst werden. Nach dem Lösen von Rollenzuordnungen lassen sich auch generierte Entwurfsteile (manuell) wieder entfernen.

5.4 Auffaltung: Generieren des Anwendungsmodells

Nachdem ich im vorhergehenden Abschnitt beschrieben habe wie ein Entwickler bei der Anwendung eines Entwurfsmusters vorgeht, beleuchte ich im Folgenden, was dabei im Hintergrund durch die Entwurfsumgebung automatisch erledigt wird. Dabei beschreibe ich wie die Anwendungsstelle angelegt und wie die Auffaltung der Set Fragments bzw. der Musterspezifikation funktioniert (Schritte 2 und 3 in Abb. 5.4, S. 106). Vorher erläutere ich den Sinn hinter einem Anwendungsmodell.

5.4.1 Anwendungsmodell

1-zu-1 die gewünschte Implementierungsvariante

Ein Anwendungsmodell repräsentiert die vom Entwickler für seinen Anwendungsfall gewählte Implementierungsvariante der in einer Musterspezifikation beschriebenen Entwurfslösung. Die Implementierungsvariante ist im Prinzip die Entwurfsmodellstruktur, die vom Entwickler zur Implementierung des Musters in dem konkreten Anwendungsfall gewählt wurde. Diese Entwurfsmodellstruktur wird in der DAL beschrieben und repräsentiert 1-zu-1 alle Typen, Beziehungen, Operationen und deren Interaktionen, die zusammen das anzuwendende Muster implementie-

ren sollen. Die DAL (siehe Abschnitt 3.4) ist Teil der Musterspezifikationssprache (PSL) (siehe Abb. 3.4, S. 45), enthält jedoch insbesondere keine Set Fragments. Stattdessen kommt jede in einem Set Fragment definierte Entwurfsmodellstruktur ein Mal oder mehrfach im Anwendungsmodell vor.

Beispiel

In der Abb. 5.9 (S. 114) ist ein Beispiel für ein Anwendungsmodell dargestellt (unten im Bild). Es ist das Ergebnis der in Abschnitt 5.3.1 beschriebenen Schritte und wurde durch schrittweises Zuordnen von Musterrollen zu Entwurfsteilen und schrittweises Ergänzen von Set-Fragment-Instanzen – also den Vorkommen der in einem Set Fragment definierten Entwurfsmodellstrukturen – erstellt.

Bei dem abgebildeten Beispiel hat ein Entwickler entschieden, dass er von dem Set Fragment operations nur eine und von dem Set Fragment strategies zwei Ausprägungen wünscht. Für jede dieser Ausprägungen enthält das Anwendungsmodell separate Rollenzuordnungen, z.B. wurde die Rolle ConcreteStrategy einmal mit der schon existierenden Klasse RowLayout und einmal mit der zu generierenden Klasse XYLayout belegt. Für jede Aufgabenbeschreibung enthält das Anwendungsmodell eine zu erledigende Aufgabe. Alle in der Musterspezifikation vorkommenden Beziehungen spiegeln sich ebenso im Anwendungsmodell wieder.

Wie hilft es?

Das Anwendungsmodell dient zum einen dazu, die komplexe Transformation einer Musterspezifikation in eine Implementierung des Musters im Entwurfsmodell in kleinere Schritte zu zerlegen und dadurch besser beherrschbar zu machen. Nach dem Auffalten der Musterspezifikation und dem Erzeugen des Anwendungsmodells muss die im Anwendungsmodell abstrakt beschriebene Implementierungsvariante des Musters „nur noch“ in ein detaillierteres Modell, nämlich die konkrete Implementierung des Musters im Entwurfsmodell, übersetzt werden. Zum anderen dient das Anwendungsmodell dazu, einen Teil der Transformation (die Auffaltung) wiederverwendbar zu machen, wenn der Ansatz auf andere Modellierungssprachen übertragen wird. Der Algorithmus für die Auffaltung kann erhalten bleiben, wenn die spezifizierten Muster nicht mehr in Ecore- und Story-Diagramm-Modellen, sondern in Modellen basierend auf anderen Modellierungssprachen implementiert werden sollen (vgl. Abb. 5.3, S. 104). Das gleiche gilt auch bei Erweiterung der Musterspezifikationssprache durch Ergänzen zusätzlicher Sprachkonstrukte in der DAL, z.B. bei Erweiterung um Komponenten (siehe Abb. 3.4, S. 45).

Herausforderungen

Durch über Set-Fragment-Grenzen hinaus gehende Beziehungen und sich überschneidende Set Fragments, wie sie auch in der Musterspezifikation in der Abb. 5.9 oben zu sehen sind, entstehen relativ komplexe Abhängigkeiten zwischen den Musterrollen. Zum Beispiel darf die in einem Set Fragment beschriebene Struktur immer nur als Ganzes in einer Musterimplementierung im Entwurf auftauchen. Bei mehrfachen Vorkommen solcher Strukturen in einer Musterimplementierung müssen eine Musterrolle einnehmende Entwurfselemente korrekt miteinander verbunden werden. Zum Beispiel müssen Vererbungsbeziehungen und Assoziationen zwischen den richtigen Klassen erstellt werden. Insbesondere bei sich überschneidenden Set Fragments ist diese Aufgabe nicht trivial. Wie die in Set Fragments eingeschlossenen Strukturen bei einer Musteranwendung vervielfältigt und die im Softwareentwurf erzeugten Elemente miteinander verbunden werden dürfen, wird durch die Semantik von Set Fragments in Abschnitt 3.5.2 (bzw. im Anhang A.3.3) formal definiert. Wie ein Anwendungsmodell automatisch erzeugt wird, beschreibe ich im Folgenden.

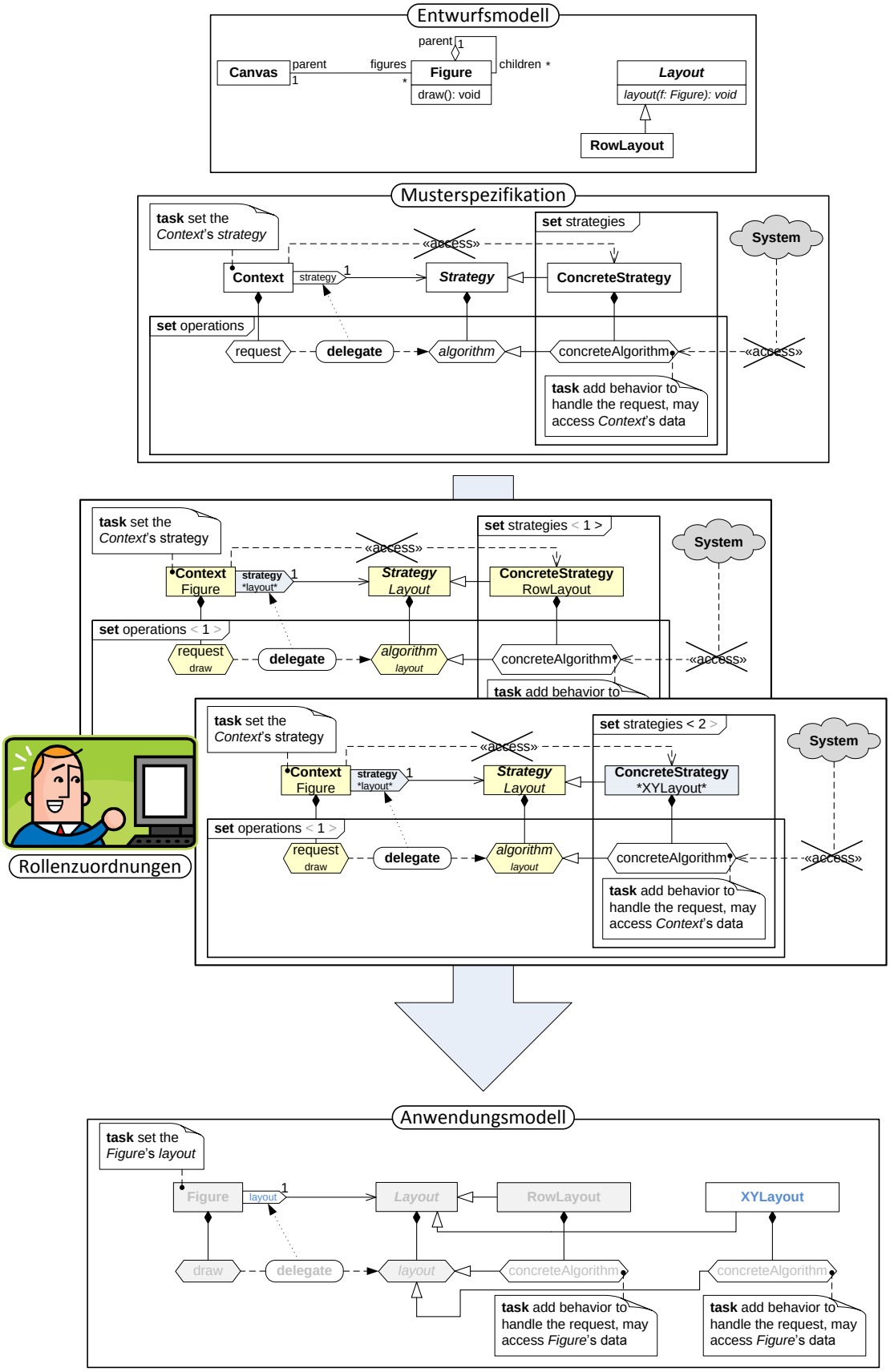


Abbildung 5.9: Ein aus Benutzereingaben hergeleitetes Anwendungsmodell für das Muster Strategy

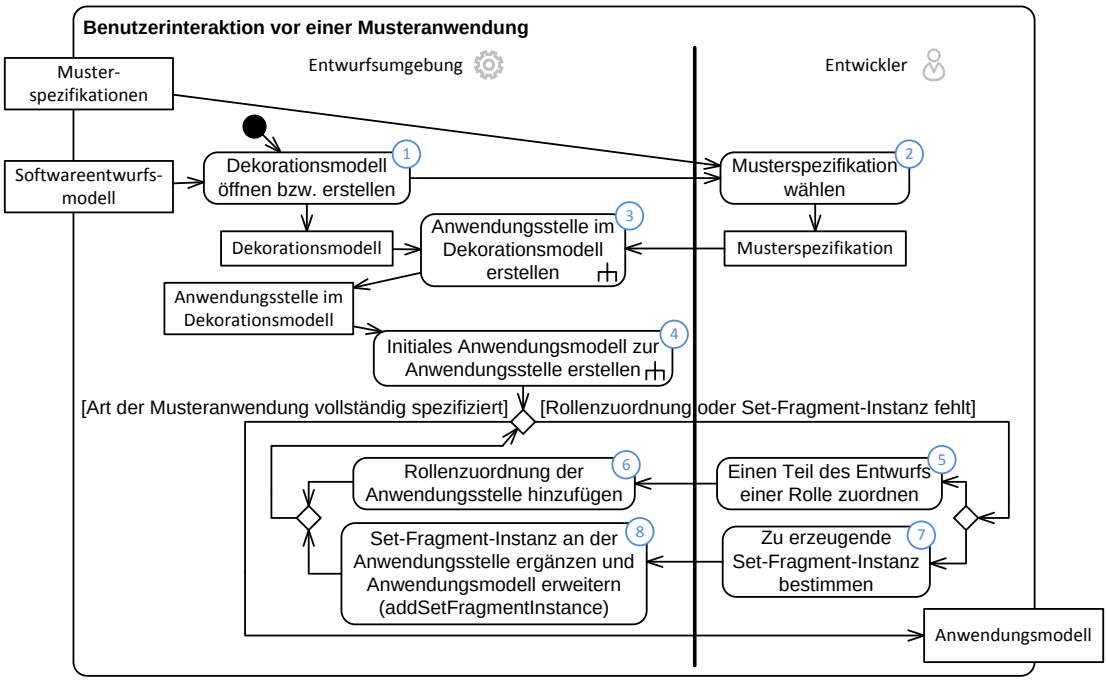


Abbildung 5.10: Vorbereiten einer Musteranwendung und Auffaltung.
Anmerkung: Rollenzuordnungen und Set-Fragment-Instanzen können auf die gleiche Weise wieder entfernt werden, wie sie hinzugefügt werden. Die zugehörigen Schritte wurden zur Vereinfachung ausgeblendet.

5.4.2 Ablauf der Rollenzuordnung und Auffaltung

Die Rollenzuordnung und die damit verbundene Auffaltung der Set Fragments erfolgt interaktiv. Nach Abschluss dieses Vorgangs liegt ein Anwendungsmodell vor, welches die gewünschte Musterimplementierungsvariante repräsentiert und durch Übersetzung der darin enthaltenen Elemente in ein Entwurfsmodell übertragen werden kann. Das Ergebnis der in Abschnitt 5.3.1 beschriebenen Vorbereitung einer Musteranwendung wäre z.B. das in Abb. 5.9 unten dargestellte Anwendungsmodell.

Schritte

Die wesentlichen Schritte bei der Vorbereitung einer Musteranwendung und beim Festlegen der von Entwurfsteilen einzunehmenden Musterrollen sind in der Abb. 5.10 in einem Aktivitätendiagramm dargestellt. Das Diagramm umfasst das Erstellen der Rollenzuordnungen und die gleichzeitige Auffaltung einer Musterspezifikation entsprechend der Übersichtsgrafik 5.1 (S. 101). Die im Aktivitätendiagramm nummerierten Schritte werden im Folgenden erläutert.

Ein Entwickler bearbeitet ein Entwurfsmodell. Wenn in dem Entwurfsmodell noch keine Muster angewandt oder zumindest nicht in einem Dekorationsmodell erfasst wurden, wird ein Dekorationsmodell zur Erfassung von Musteranwendungsstellen und Musterimplementierungen automatisch erzeugt (Schritt 1). Existiert ein Dekorationsmodell, dann wird es geladen.

Aus einem Katalog mit zuvor von Experten spezifizierten Mustern und zugehörigen Entwurfslösungen wählt der Entwickler die für seine Situation passende Musterspezifikation zwecks Musteranwendung aus (Schritt 2).

Korrespondenzmodell

Zur Erfassung der Anwendungsstelle und später auch der Musterimplementierung wird im Dekorationsmodell automatisch ein Modell der Anwendungsstelle, das Korrespondenzmodell mit den Rollenzuordnungen (siehe Abb. 4.3, S. 83), angelegt (Schritt 3). Das Korrespondenzmodell soll die vom Entwickler einzugebenden Zuordnungen von Entwurfsteilen zu den davon einzunehmenden Musterrollen persistieren und dient als eine Art von automatisch erstellter Dokumentation der Musteranwendung (siehe Kapitel 4). Bei diesem Schritt wird die verwendete Musterspezifikation festgehalten. Zu jeder Musterrolle wird ein Korrespondenzknoten (RoleBinding) angelegt, der die Zuordnung der Rolle zu einem oder mehreren Entwurfsteilen speichern soll. Zu jedem Set Fragment der Musterspezifikation wird eine Set-Fragment-Instanz[5] angelegt, um jede Ausprägung der Struktur in einem Set Fragment bei der Musterimplementierung zu erfassen. Dieser Schritt wird im Abschnitt 5.4.3 (S. 117 ff.) erläutert. Details dazu sind in einem weiteren Aktivitätendiagramm im Anhang C.1 dargestellt (Abb. C.2, S. 305).

Anwendungsmodell

Als Nächstes wird die Anwendungsstelle (das Korrespondenzmodell) um ein initiales Anwendungsmodell ergänzt (Schritt 4). Ein Anwendungsmodell repräsentiert die vom Entwickler gewählte Implementierungsvariante und dient als Zwischenprodukt bei der Musteranwendung. Das schrittweise Erstellen des Anwendungsmodells bezeichne ich als Auffaltung der Musterspezifikation (siehe Abb. 3.34, S. 71), weil man durch Vervielfältigen der Strukturen in den Set Fragments der Musterspezifikation zu der gewählten Implementierungsvariante gelangt und dieser Vorgang dem Auffalten eines gefalteten Papierblatts ähnelt. Bei dem Auffalten wird für jedes Set Fragment festgelegt wie häufig die darin enthaltene Struktur in der gewählten Implementierungsvariante vorkommen soll und diese Struktur entsprechend häufig im Anwendungsmodell erstellt. Das Anwendungsmodell wird in der DAL beschrieben und enthält somit keine Set Fragments mehr (siehe Abb. 3.4, S. 45). Beim initialen Erstellen des Anwendungsmodells wird für jedes Set Fragment genau ein Vorkommen der darin enthaltenen Struktur im Anwendungsmodell erstellt (das entspricht dem Schritt 1 in Abb. 3.34, S. 71). Jedes Vorkommen einer solchen Struktur im Anwendungsmodell bzw. in der Musterimplementierung stellt eine Set-Fragment-Instanz dar und wird zusammen mit der Anwendungsstelle im Dekorationsmodell erfasst. Das initiale Erstellen des Anwendungsmodells ist automatisiert. Dieser Schritt wird im Abschnitt 5.4.4 (S. 118 ff.) erläutert. Details dazu sind in einem weiteren Aktivitätendiagramm im Anhang C.2.1 dargestellt (Abb. C.5, S. 309).

Ist das Anwendungsmodell erstellt, kann die eigentliche Musteranwendung beginnen. Erfolgt die Musteranwendung direkt nach dem Erstellen des initialen Anwendungsmodells, dann wird genau die in der Musterspezifikation beschriebene Struktur im Entwurfsmodell neu erstellt.

Rollenzuordnung

Sollen Teile des schon existierenden Entwurfs bei der Implementierung des Musters wiederverwendet werden, ordnet der Entwickler diese Teile den zugehörigen Musterrollen zu (Schritt 5, Abb. 5.10). Solche Zuordnungen werden automatisch in den in Schritt 3 erstellten Korrespondenzknoten (Role Bindings) persistiert (Schritt 6, Abb. 5.10) und bei der Musteranwendung berücksichtigt.

[5] Repräsentiert ein Vorkommen des in einem Set Fragment spezifizierten Teils einer Entwurfslösung

Weitere Set-Fragment-Instanzen

Soll die in einem Set Fragment spezifizierte Struktur in der Musterimplementierung mehrfach vorkommen, bestimmt der Entwickler weitere Vorkommen dieser Struktur (Schritt 7, Abb. 5.10, das entspricht dem Schritt 2[6] in Abb. 3.34, S. 71). Dazu wählt er ein Set Fragment, zu welchem er eine zusätzliche Set-Fragment-Instanz erstellen lassen möchte. Daraufhin erstellt die Entwurfsumgebung automatisch ein weiteres Vorkommen der in dem Set Fragment beschriebenen Struktur im Anwendungsmodell sowie zugehörige Korrespondenzknoten im Korrespondenzmodell (Schritt 8, Abb. 5.10). Der Algorithmus zu diesem Schritt wird in Abschnitt5.4.5 erläutert und im Anhang C.2.2 ausführlich, u.a. in Form von Pseudocode (Codefragment C.1, S. 312) beschrieben.

Die anschließende Musteranwendung durch Übersetzung des Anwendungsmodells in eine Musterimplementierung in einem Entwurfsmodell wird in Abschnitt 5.5 beschrieben.

5.4.3 Erstellen einer Anwendungsstelle (eines Korrespondenzmodells)

Rollenzuordnungen

Während ein Entwickler ein Entwurfsmuster für die Anwendung auswählt und anschließend Teile seines Softwareentwurfs den Musterrollen zuordnet wird automatisch ein Modell der Anwendungsstelle im Dekorationsmodell erstellt. Das Modell einer Anwendungsstelle stellt Teile des Softwareentwurfs, die eine Rolle des anzuwendenden Musters einnehmen sollen, mit den Rollen und Set Fragments einer Musterspezifikation in Beziehung. Die Anwendungsstelle wird somit als eine Menge von Korrespondenzen zwischen Teilen des Softwareentwurfs und Teilen der Musterspezifikation modelliert, weswegen ich das Modell der Anwendungsstelle auch Korrespondenzmodell nenne. Das Korrespondenzmodell enthält Informationen darüber, welche Musterspezifikation verwendet werden soll und welche Rollen schon existierende Entwurfsteile einnehmen sollen. Ergänzend dazu werden Korrespondenzen zwischen Set Fragments und den zugehören Strukturen (Set-Fragment-Instanzen) im Entwurf erfasst (Details in Kapitel 4).

Nach Musteranwendung wird das Korrespondenzmodell automatisch vervollständigt, indem alle automatisch erzeugten Elemente im Entwurf ebenfalls mit ihren Rollen verknüpft werden und so das im Entwurf implementierte Muster mit allen seinen Beteiligten vollständig erfasst wird.

Dadurch dient das Korrespondenzmodell als Dokumentation der Musteranwendung, zur Visualisierung der Anwendungsstelle (siehe Kapitel 4) und kann zur automatischen Prüfung des Entwurfs nach weiteren, ggf. manuellen Entwurfsänderungen herangezogen werden (siehe Kapitel 6).

Korrespondenzknoten

Beim Anlegen der Korrespondenzen für eine Musteranwendungsstelle wird zur jeder Rolle und zu jedem Set Fragment einer Musterspezifikation ein Korrespondenzknoten angelegt. Auf Details zu diesem Schritt gehe ich im Anhang C.1 ein.

[6] Der Schritt 3 aus Abb. 3.34 entfällt, weil die Markierungen von Set-Fragment-Instanzen separat im Korrespondenzmodell festgehalten werden und sich von vornherein nicht im DAL-Graphen befinden.

5.4.4 Instanziieren eines Anwendungsmodells

Eine erste Version eines Anwendungsmodells wird vollautomatisch durch die Entwurfsumgebung erzeugt, sobald ein Entwickler eine Musterspezifikation zur Anwendung auswählt (Schritt 4 in Abb. 5.10, S. 115 und Schritt 1 in Abb. 3.34, S. 71). Bei seiner initialen Erzeugung erhält ein Anwendungsmodell ein exaktes Abbild der in der Musterspezifikation beschriebenen Entwurfsmodellstruktur, jedoch ohne Set Fragments und Kopplungsregeln. Zu jeder Rolle der Musterspezifikation befindet sich im Anwendungsmodell ein der Rolle entsprechendes Entwurfselement (z.B. Typ, Operation, Referenz, Aktion). Alle solchen Abbilder von Musterrollen sind genauso untereinander verbunden wie die als Vorbild dienenden Rollen der Musterspezifikation.

Abbild der Spezifikation ohne Set Fragments

In der Abb. 5.11 stellt das Anwendungsmodell$_0$ das Ergebnis dieses Schrittes exemplarisch für die Spezifikation des Musters Strategy (oben im Bild) dar. Alle nicht zum Anwendungsmodell gehörenden Teile der Musterspezifikation wie die Rollennamen oder Kopplungsrestriktionen sind hier ausgegraut (hell) dargestellt. Auch die Instanzen von Set Fragments gehören nicht zum Anwendungsmodell, werden hier jedoch hell orange angedeutet, um nachvollziehbar zu machen, welche Teile des Anwendungsmodells zu welchem Set Fragment bzw. zu welcher Set-Fragment-Instanz gehören[7].

Weitere Details zur initialen Erzeugung eines Anwendungsmodells werden im Anhang C.2.1, insb. in den Abb. C.5 (S. 309) und C.4 (S. 308) beschrieben.

5.4.5 Ergänzen einer Set-Fragment-Instanz

Nachdem ein Anwendungsmodell erstellt wurde, können darin zusätzliche Vorkommen der in einem Set Fragment spezifizierten Entwurfsmodellstrukturen ergänzt werden. Jedes dieser Vorkommen wird durch einen speziellen Korrespondenzknoten, eine Set-Fragment-Instanz, repräsentiert und dem zugehörigen Set Fragment zugeordnet (siehe Abb. 4.3, S. 83).

spezieller Korrespondenzknoten

Das Hinzufügen einer Set-Fragment-Instanz und der zugehörigen Struktur im Anwendungsmodell stellt eine spezielle Operation dar (ich nenne sie *add*-Operation, siehe Abb. 3.36, S. 71). Im Prinzip wird dabei eine Set-Fragment-Instanz samt der zugehörigen Elemente im Anwendungsmodell repliziert. Dabei dient eine Set-Fragment-Instanz als Vorbild für die anzulegende Kopie. Das einmalige Ausführen dieser Operation entspricht dem Schritt 8 in der Abb. 5.10 (S. 115) und dem Schritt 2 in Abb. 3.34 (S. 71).

Kopieren einer Set-Fragment-Instanz

Zur Veranschaulichung sind zwei Anwendungen der Operation in der Abb. 5.11 (S. 119) dargestellt. Das Anwendungsmodell$_1$ ist aus dem Anwendungsmodell$_0$ entstanden, indem die Set-Fragment-Instanz strategies 1 repliziert und eine neue Instanz strategies 2 erzeugt wurde. Anschließend ist das Anwendungsmodell$_2$ aus dem Anwendungsmodell$_1$ durch das Replizieren der Set-Fragment-Instanz operations 1 entstanden.

Neben den 1-zu-1 kopierten Elementen (Knoten in der Musterspezifikation) wurden auch ihre Beziehungen bzw. Relationen (Kanten in der Musterspezifikation)

[7] Set-Fragment-Instanzen sind Teil des Korrespondenzmodells, gehören aber nicht zum Anwendungsmodell (siehe Abb. 4.3, S. 83).

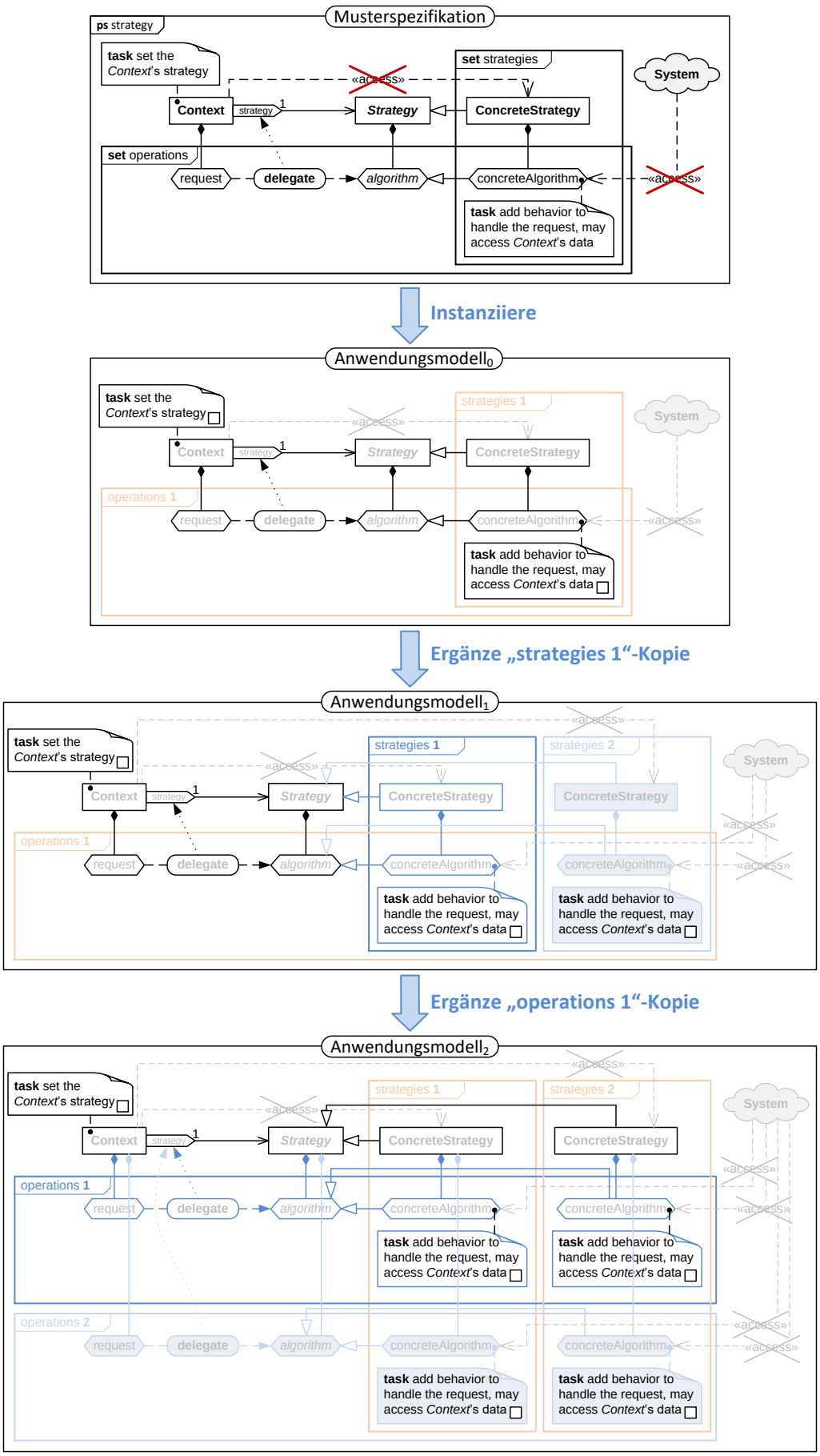


Abbildung 5.11: Instanziieren und Auffalten eines Anwendungsmodells

Bedingungen beim Kopieren

so repliziert, dass jedes Element im entstandenen Anwendungsmodell alle in der Musterspezifikation definierten Bedingungen (Constraints) erfüllt. Insbesondere bei sich überschneidenden oder geschachtelten Set Fragments ist diese Operation nicht trivial. Damit das Anwendungsmodell nach jeder *add*-Operation konsistent zur Musterspezifikation bleibt, müssen mehrere Bedingungen erfüllt werden, insb. folgende:

1. Zu jedem DAL-Knoten und jeder DAL-Kante in der Musterspezifikation muss es (mindestens) eine Entsprechung im Anwendungsmodell geben (das ist laut Anmerkung A.16 (ii) erfüllt).
2. Jeder Knoten und jede Kante im Anwendungsmodell muss genau einem Knoten bzw. genau einer Kante in der Musterspezifikation zugeordnet sein (das ist laut Anmerkung A.16 (iii) erfüllt).
3. Jede Kante muss eine Quelle und ein Ziel haben (das ist laut Anmerkung A.16 (i) erfüllt).
4. Konsistenz zur Spezifikation (Teilgraphenisomorphie)
   (a) Jede zu einem Set Fragment gehörende Gruppe von Knoten und Kanten im Anwendungsmodell (eine Set-Fragment-Instanz) muss isomorph[8] zu den Knoten und Kanten im zugehörigen Set Fragment sein.
   (b) Für jede Auswahl von Set-Fragment-Instanzen zu einem Anwendungsmodell, bei welcher zu jedem Set Fragment genau eine Set-Fragment-Instanz ausgewählt wurde, sind die zugehörigen Knoten und Kanten des Anwendungsmodells isomorph zur Musterspezifikation (das ist laut Anmerkung A.16 (iv) erfüllt, außerdem erfüllt 4 (b) auch 4 (a)).

Durch das Kopieren einer Set-Fragment-Instanz, für welche alle genannten Eigenschaften bereits erfüllt sind (das gilt insb. nach der Instanziierung des Anwendungsmodells), und das korrekte[9] Ziehen von Kanten zwischen den Knoten in- und außerhalb der neuen Set-Fragment-Instanz erhalte ich die geforderten Eigenschaften. Die Details zu dieser Operation beschreibe ich ausführlich, u.a. in Form von Pseudocode im Anhang C.2.2 und orientiere mich dabei an meiner formalen Definition von Set Fragments (Details dazu im Anhang A.3.3 und C.2.3).

Einschränkungen für Kanten in Musterspezifikationen

Die *add*-Operation kopiert Kanten ungeachtet der Multiplizität der zugehörigen Assoziationen im Meta-Modell der DAL, was die Menge der sinnvollen Musterspezifikationen einschränkt. Die Einschränkung tritt bei 0..1-zu-n- oder 1-zu-n-Assoziationen wie der Vererbungsbeziehung in der DAL auf. Die DAL erlaubt nur Einfachvererbung und damit höchstens eine Oberklasse. Dadurch wäre eine Musterspezifikation, bei welcher die Vererbungsbeziehung zwischen den Klassen Strategy und ConcreteStrategy aus Abb. 5.11 in umgekehrter Richtung verläuft, nicht sinnvoll, weil man dann laut spezifiziertem Set Fragment mehrere ConcreteStrategy-Oberklassen erzeugen könnte, diese aber aufgrund der Einfachvererbung nicht mit der einen Strategy-Unterklasse in Beziehung setzen könnte (die Vererbungskante würde nur einmal statt mehrfach existieren). Die Verantwortung für das Erstellen ausschließlich zulässiger Musterspezifikationen in Bezug auf die beschriebene Einschränkung liegt zurzeit beim Erzeuger der Spezifikationen.

[8] deckungsgleich (d.h. bijektiv und homomorph)
[9] korrekt = der Definition A.12 (S. 285) entsprechend

5.5 Übersetzung des Anwendungsmodells in den Entwurf

Nachdem ein Entwickler durch Rollenzuordnung festgelegt hat wie ein Muster in einer bestimmten Situation angewendet werden soll und aus diesen Angaben ein Anwendungsmodell generiert wurde, wird das Anwendungsmodell in das Entwurfsmodell übersetzt und das bisherige Entwurfsmodell vervollständigt (rechts in Abb. 5.2, S. 104).

Art der Modelltransformation

Die Übersetzung des Anwendungsmodells bzw. das Vervollständigen des Entwurfsmodells ist eine *exogene* Modelltransformation[10] [CH06]. Die abstrakte Repräsentation einer Musterimplementierung, beschrieben in der DAL, wird abhängig von dem schon existierenden Softwareentwurf und der Rollenzuordnungen in eine konkrete Implementierung des spezifizierten Entwurfsmusters im deutlich detaillierteren Softwareentwurfsmodell, beschrieben in Ecore und Story-Diagrammen, übersetzt. Dabei wird das Entwurfsmodell nur erweitert, was einer nicht destruktiven Update-Transformation entspricht. Diese Modell-zu-Modell-Transformation ist jedoch in vielerlei Hinsicht besonders, worauf ich als Nächstes eingehen möchte, bevor ich die Transformation genauer beschreibe.

5.5.1 Herausforderungen

Aufbau

Zur Veranschaulichung sind in der Abb. 5.12 alle an der Transformation beteiligten Modelle skizziert. Als Eingabe für die Transformation dienen das Quellmodell (das Anwendungsmodell), das zum Teil schon vorhandene Zielmodell (das Entwurfsmodell) und die Relationen zwischen den einander zugeordneten Elementen im Quell- und Zielmodell (das Korrespondenzmodell). Abhängig davon, welche Musterrollen (Elemente im Anwendungsmodell) Elementen im Entwurf zugeordnet sind, werden fehlende Übersetzungen der Musterrollen im Entwurf ergänzt und das Korrespondenzmodell vervollständigt. Das Anwendungsmodell bleibt dabei unverändert.

mehrfacher Nutzen der Korrespondenzen

Das Korrespondenzmodell dient nicht nur dazu, während der Transformation nachzuvollziehen, welche Elemente bereits übersetzt wurden, sondern wird auch zur Dokumentation einer Musteranwendung und zur Überprüfung der zugehörigen Musterimplementierung verwendet. Daraus und aus dem gewählten Lösungsansatz ergibt sich ein von klassischen exogenen Modelltransformationen abweichendes Szenario und neue Herausforderungen, auf die ich im Folgenden näher eingehe.

Von außen vorgegebene Struktur für Korrespondenzen

Das Korrespondenzmodell wird nicht nur zum Zweck der Traceability während der Transformation eingesetzt, sondern dient auch zur Persistierung von Musteranwendungen und macht Musterimplementierungen in einem Softwareentwurfsmodell explizit. Darum ist das Korrespondenzmodell mit zusätzlichen Informationen angereichert (z.B. Verknüpfung mit Elementen der Musterspezifikation), welche für die Transformation allein nicht nötig wären. Die Struktur von Korrespondenzmodellen wird in Kapitel 4 durch ein Meta-Modell festgelegt und näher erläutert.

[10] Eine Übersetzungstransformation, die aus einem Modell getypt über einem Typmodell ein anderes Modell getypt über einem anderen Typmodell ableitet.

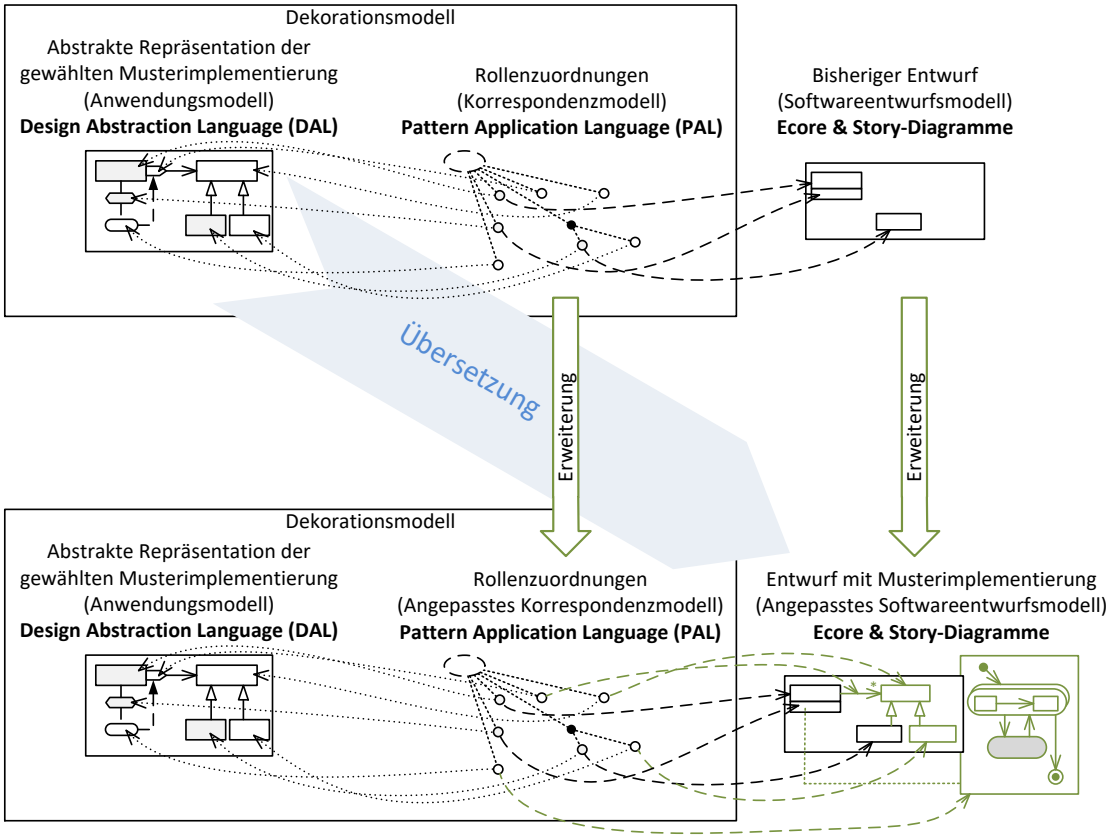


Abbildung 5.12: Ein- und Ausgaben bei der Übersetzungstransformation

Somit kann nicht das Traceability-Modell einer beliebigen Transformations-Engine verwendet werden.

Manuell festgelegte Korrespondenzen als Startpunkt

Vor Anwendung eines Musters legt ein Entwickler fest, welche Teile des vorliegenden Softwareentwurfs bestimmte Rollen des Musters einnehmen sollen. Durch diese Festlegung wird schon vor der Übersetzung des Anwendungsmodells ein Teil des Korrespondenzmodells erstellt. Diese Korrespondenzen legen den Startpunkt für die Übersetzungstransformation fest (Korrespondenzen r_1, r_2, r_3 in Abb. 5.13 b)). Im Gegensatz dazu werden bei klassischen, exogenen Modelltransformationen Baum-artig strukturierte Quell- und Zielmodelle an der Wurzel beginnend durchlaufen und schrittweise übersetzt bzw. synchronisiert (Abb. 5.13 a)).

Ergebnis der Übersetzung bildet nur einen Teil des Zielmodells

Während bei klassischen exogenen Modelltransformationen ein Quellmodell $M1$ wie in der Abb. 5.13 a) skizziert komplett in ein Zielmodell $M2 = M1'$ übersetzt wird, bildet die Übersetzung $M1'$ eines kompletten Anwendungsmodells $M1$ (eines Quellmodells) wie in der Abb. 5.13 b) skizziert nur einen Teil des Softwareentwurfsmodells $M2$ (des Zielmodells), nämlich nur den das Muster implementierenden Teil des Entwurfsmodells.

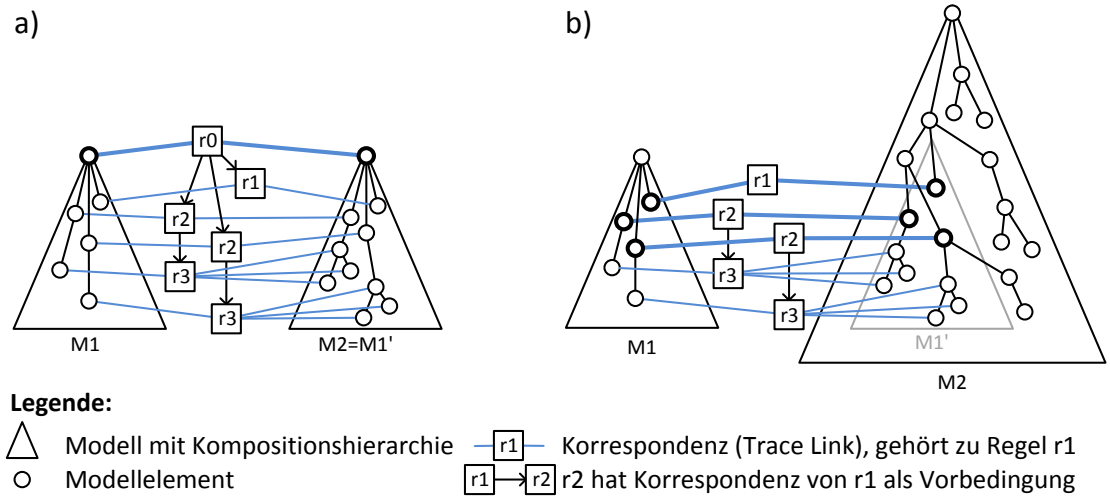


Abbildung 5.13: Korrespondenzen und Startpunkt bei Modelltransformationen:

a) Klassische exogene Modelltransformation: Ein Modell $M1$ wird beginnend an der Wurzel vollständig in ein Modell $M2$ übersetzt. $M2$ wird neu erzeugt oder mit $M1$ synchronisiert.
b) Meine Übersetzungstransformation: Ein Anwendungsmodell $M1$ wird in einen Teil eines Entwurfsmodells $M2$ übersetzt, beginnend an Stellen in $M2$, die durch manuelle Rollenzuordnungen vorbestimmt sind (hervorgehoben).

Übersetzung variiert abhängig vom Kontext im Zielmodell

Wie ein Element im Quellmodell übersetzt wird, hängt nicht nur vom Informationsgehalt im Quellmodell ab, sondern in einigen Situationen auch von der Umgebung des übersetzten Elements im Zielmodell.

Betrachten wir zur Verdeutlichung die Übersetzung von Operationen, die anhand von drei in der Abb. 5.14 dargestellten Fällen erläutert wird. Hier sind jeweils die Ausgangssituation in Form von Anwendungsmodell (Quellmodell), Entwurfsmodell (Zielmodell) und den vom Entwickler festgelegten Korrespondenzen zwischen den Elementen sowie das Übersetzungsergebnis im Entwurfsmodell illustriert.

Im einfachsten Fall a) wird eine Operation concreteAlgorithm direkt übersetzt, indem eine Methode gleichen Namens ohne Ein- und Ausgabeparameter erzeugt wird. Ist eine Operation wie in den Fällen b) und c) eine Spezialisierung einer anderen Operation, so wird damit gefordert, dass die Signaturen der Übersetzungen der Operationen zueinander konform sind[11]. Dadurch variiert die Übersetzung der Operation concreteAlgorithm abhängig davon, welche Methode die Rolle der algorithm-Operation einnimmt. Die Übersetzung eines Elements hängt also nicht nur von seinen Eigenschaften im Anwendungsmodell bzw. der Musterspezifikation ab, sondern auch von Elementen im Entwurfsmodell, die andere Musterrollen einnehmen.

[11] Bei den meisten OO-Sprachen wie z.B. Java sind die Signaturen konform zueinander, wenn sie identisch sind (gleicher Name, gleiche Anzahl, Reihenfolge und Typ von Parametern sowie gleicher Rückgabetyp) oder der Rückgabetyp der spezielleren (überschreibenden) Methode eine Spezialisierung des Rückgabetyps der allgemeineren (überschriebenen) Methode ist.

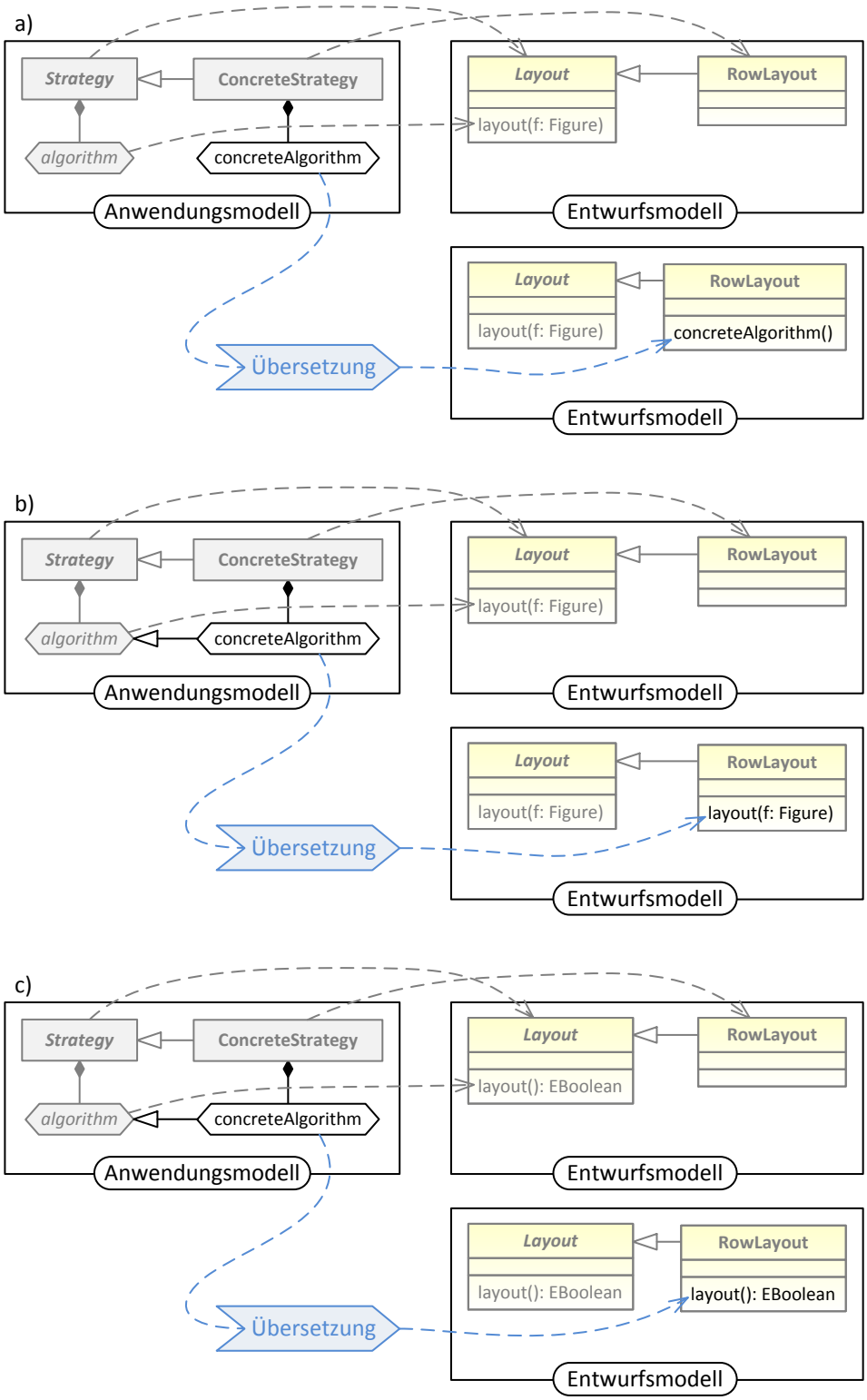


Abbildung 5.14: Kontextabhängige Übersetzung von Operationen

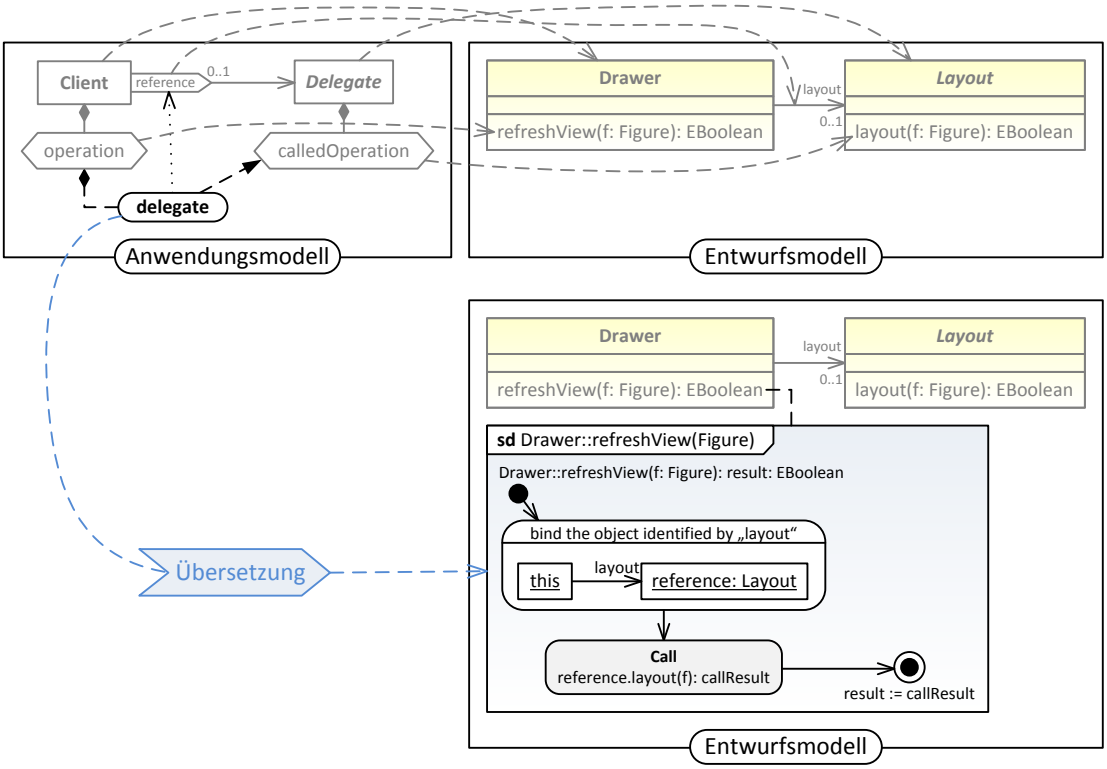


Abbildung 5.15: Unterschiedliche Abstraktions- und Detaillevel bei der Übersetzung

Stark abweichende Abstraktions- und Detaillevel bei Quell- und Zielmodell

Die DAL wurde so konzipiert, dass die wesentlichen Teile eines objektorientierten Softwareentwurfs sehr kompakt und allgemein verwendbar beschrieben werden. Vor allem Aktionen in der DAL sind im Vergleich zu den daraus abzuleitenden Story-Diagramm-Anteilen im Entwurfsmodell sehr abstrakt und bestehen im Wesentlichen aus nur einem Knoten mit Label und bis zu drei Kanten. Die aus einer Aktion abgeleiteten Story-Diagramm-Anteile sind dagegen deutlich detaillierter und können aus mehreren Aktivitätenknoten, Story Patterns und sogar Schleifen bestehen. Zum Beispiel wird die delegate-Aktion aus Abb. 5.15 in ein Story-Diagramm mit drei Aktivitätenknoten und einem Story Pattern übersetzt. Der Methodenaufruf wird als zusammengesetzter Ausdruck modelliert und ist in abstrakter Syntax wesentlich komplexer als hier dargestellt (vgl. Abb. A.22, S. 273). Ein weiterer zusammengesetzter Ausdruck wird für die Rückgabe des Ergebnisses im Endknoten verwendet (vgl. Abb. A.23, S. 274). Für die Übersetzung bedeuten stark abweichende Abstraktions- und Detaillevel, dass ggf. viele Fallunterscheidungen und evtl. Zwischenschritte notwendig sind. Außerdem stellt dieser Umstand eine besondere Herausforderung an die Nachvollziehbarkeit der Transformation und der zugehörigen Ergebnisse dar.

5.5.2 Wahl der Transformationssprache

Es gibt verschiedene Ansätze zur Realisierung von Modelltransformationen. Insbesondere gibt es dedizierte Sprachen zur Beschreibung von Modelltransforma-

dedizierte Transformationssprachen

tionen und Engines zur Ausführung damit beschriebener Transformationen. So wären z.B. die Modelltransformationssprachen Tripelgraphgrammatiken (TGG) [Sch94, KW07, SK08, GR10, HLG+13] und Query/View/Transformation (QVT) [OMG11d] sowie zugehörige Engines wie der TGG-Interpreter[12], die QVT-Engine[13] des Eclipse Modeling Projects[14] oder die Medini QVT-Engine [15] potentielle Kandidaten zur Beschreibung und Ausführung meiner Modelltransformation.

bidirektional

TGG und QVT Relational (kurz: QVT-R) sind zur deklarativen, relationalen Beschreibung von exogenen Modelltransformationen entwickelt worden, womit sich nicht nur uni-, sondern auch bidirektionale Transformationen beschreiben lassen. Bei solchen Transformationen wird ein Modell inkrementell in ein anderes übersetzt. Somit sind u.a. Update- und Synchronisationsszenarien sogar zwischen Modellen auf verschiedenen Abstraktionsleveln [Anj14, Rie14, LAS17] realisierbar.

Grenzen der Sprachen

Dennoch sprechen einige Punkte gegen die Verwendung dieser Ansätze für meine Transformation. Das Korrespondenzmodell ist bei meinem Ansatz vorgegeben, weil es neben der Transformation auch der Modellierung und Validierung von Musterimplementierungen dient. Bei QVT wird das Traceability-Modell implizit im Hintergrund aufgebaut, sodass mein Korrespondezmodell gesondert erzeugt werden müsste. Bei TGGs werden Korrespondenzknoten in einer Transformationsregel explizit mitmodelliert, müssen aber vorgegebene Schnittstellen erfüllen. Bei beiden Ansätzen werden komplette Modelle übersetzt, wodurch Transformationsregeln überall im Modell angewendet werden können. Im Gegensatz dazu wird bei meinem Ansatz ein Modell in einen Teil eines anderen bereits exisitierenden Modells übersetzt, die Transformation soll also auf diesen Teil beschränkt ausgeführt werden. Während bei TGG und QVT-R Axiome und Top-Relationen – Zuordnungen der Wurzeln von Quell- und Zielmodell zueinander – den Startpunkt einer Transformation festlegen, wird bei meinem Ansatz der Startpunkt manuell durch einen Entwickler und seine Rollenzuordnungen mitten im Modell festgelegt (vgl. Abb. 5.13, S. 123).

Die aufgezählten Herausforderungen bei der Realisierung meiner Transformation mit TGG oder QVT-R lassen sich vermutlich mit hohem Aufwand umgehen. Allerdings hängt die Übersetzung einzelner Elemente bei meinem Ansatz nicht nur vom Quellmodell, sondern auch vom Zielmodell ab. Das erfordert einen komplexen Kontrollfluss in der Transformation, der z.B. die Signatur einer Methode abhängig von einer anderen Methode bestimmt (siehe Bsp. in Abb. 5.14, S. 124) oder abhängig von der Kardinalität einer Referenz im Entwurfsmodell entscheidet, ob eine Aktion ein Mal oder mit Hilfe einer Schleife mit jedem referenzierten Objekt ausgeführt werden soll. Derartige Transformationen lassen sich nur schwierig oder gar nicht mit deklarativen Transformationssprachen wie TGG und QVT-R beschreiben, was ein K.O.-Kriterium für diese Ansätze und der Hauptgrund für meine Entscheidung gegen TGG und QVT-R ist.

Neben QVT-R, der deklarativen Transformationssprache im QVT-Standard, gibt es auch eine operationale Transformationssprache: QVT Operational (kurz:

[12] http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html
[13] http://projects.eclipse.org/projects/modeling.mmt.qvt-oml
[14] http://www.eclipse.org/modeling/
[15] http://projects.ikv.de/qvt

QVT-O). Auch das war eine in Frage kommende Transformationssprache für meinen Ansatz, denn damit lassen sich auch komplexe Fallunterscheidungen und Hilfsoperationen realisieren und das Korrespondenzmodell lässt sich als eigene Domäne (zusätzlich zum Quell- und Zielmodell) behandeln. Doch auch bei QVT-O und den zugehörigen Engines geht man davon aus, dass ein komplettes Modell in ein komplettes anderes Modell übersetzt wird. Das Beginnen der Transformation an vorgegebener Stelle ist nicht vorgesehen. Zusammen mit einigen technischen Problemen[16] war das der Grund für meine Entscheidung gegen QVT-O und für eine eigene Implementierung der Transformation in der Programmiersprache Java.

Ausweg: Java

5.5.3 Aufbau der Transformation

Die Übersetzung des Anwendungsmodells in eine Implementierung des zugehörigen Musters im Entwurfsmodell wird bei meinem Ansatz nicht durch eine dedizierte Transformationssprache, sondern operational durch eine Programmiersprache, nämlich Java, beschrieben. Dadurch erhält man einen besonders hohen Freiheitsgrad bei der Realisierung der Transformation, kann die Einschränkungen der dedizierten Transformationsprachen und Transformations-Engines umgehen und die Transformation an die speziellen Anforderungen meines Ansatzes anpassen. Man erhält die volle Kontrolle über die Ausführungsstelle, Ausführungsreihenfolge sowie das Traceability-Modell der Transformation. Allerdings erhält man diese Vorteile auf Kosten der Komplexität der Transformation und damit auch ihrer Wartbarkeit und Fehleranfälligkeit. Der Entwickler der Transformation ist selbst für die Strukturierung der Transformation zuständig[17] und für die korrekte Ausführung und somit für die richtige Anwendungsstelle und Reihenfolge verantwortlich.

Vor- & Nachteile

Auch wenn ich keine Transformationssprache verwende, orientiere ich mich beim Aufbau und der Strukturierung meiner Transformation an existierenden Transformationssprachen.

Da bei meiner Übersetzung im Zielmodell, also dem Entwurfsmodell, fehlende Teile nur ergänzt werden, handelt es sich bei der Transformation um eine nicht destruktive Update-Transformation. Solche Transformationen sind typischerweise aufgebaut wie in der Abb. 5.16 (angelehnt an die Darstellung von Czarnecki & Helsen [CH06], vgl. Abb. 2.5, S. 30) skizziert. Die Transformation besteht aus mehreren Transformationsregeln und einer Transformations-Engine. Während der Transformation werden Traceability Links erzeugt, welche die Relationen zwischen den einander zugeordneten Elementen im Ziel- und Quellmodell repräsentieren. Die Transformations-Engine liest das Quellmodell und das in Teilen schon vorhandene Zielmodell sowie die zugehörigen Traceability Links ein und erweitert das Ziel- sowie das Trace-Modell durch Ergänzen von neuen Elementen bzw. Traceability Links. Alle beteiligten Modelle sind Instanzen der zugehörigen Meta-Modelle,

Aufbau der Transformation

[16] Die Spezifikation von QVT ist bisher in keiner Engine komplett umgesetzt worden und die existierenden Realsierungen sind vor allem in unüblichen Szenarien wie in meinem fehlerhaft.

[17] Bei Transformationssprachen wie TGG und QVT ist die Struktur zum Teil durch die Sprache selbst vorgegeben, z.B. durch die Zerlegung der Transformation in mehrere Transformationsregeln, die explizite Definition der beteilgiten Domänen und der Relationen zwischen den Elementen sowie die Wiederverwendbarkeit und Komponierbarkeit von Transformationsregeln.

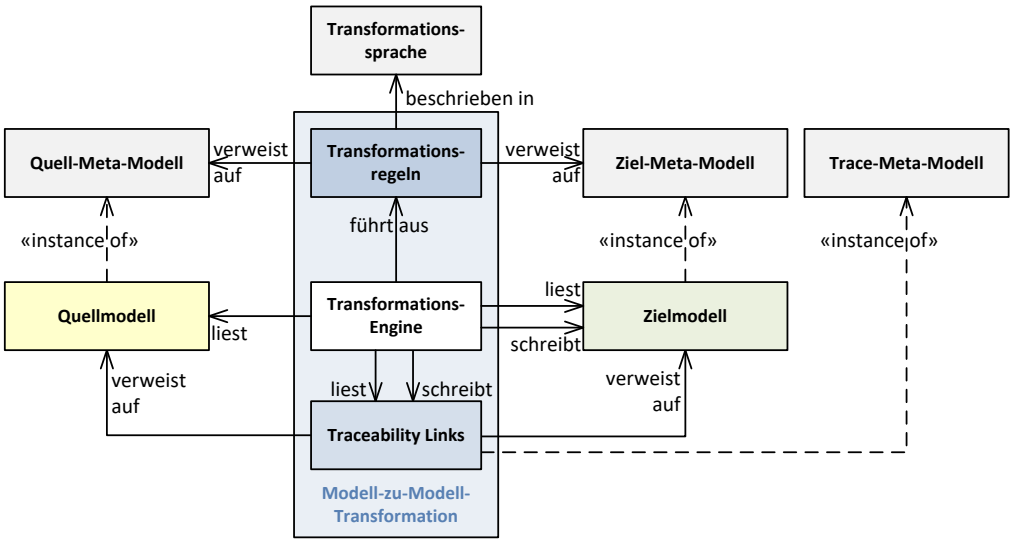


Abbildung 5.16: Typischer Aufbau einer exogenen Update-Transformation (Erweiterung der Darstellung von Czarnecki & Helsen [CH06])

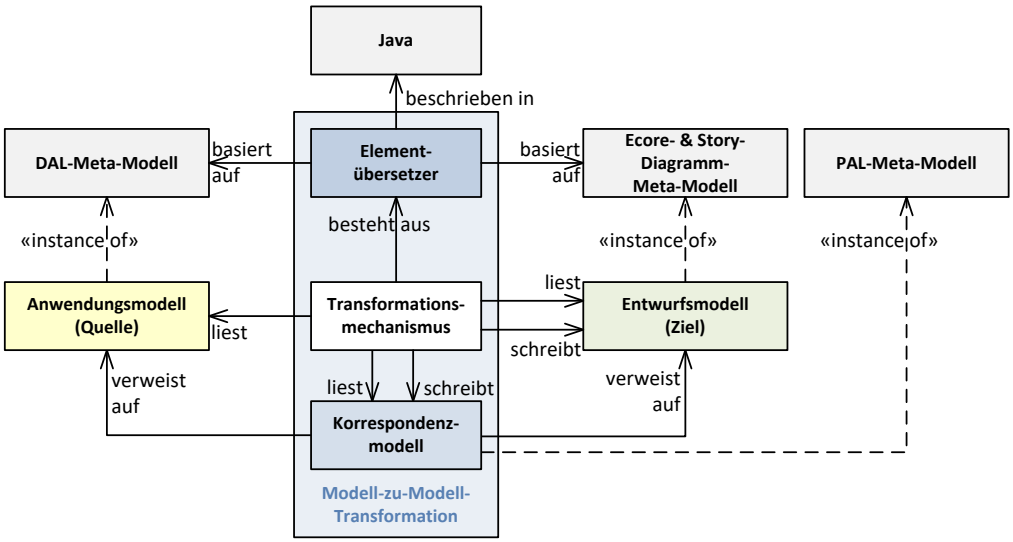


Abbildung 5.17: Aufbau meiner Übersetzungstransformation

die Transformationsregeln werden in der entsprechenden Sprache beschrieben (vgl. Begriffsdefinitionen *Meta-Modell* & *Sprache* auf S. 30, Abschn. 2.3.2).

Angelehnt an diesen Aufbau habe ich meine Übersetzungstransformation strukturiert und in der Abb. 5.17 skizziert. Statt der in einer Transformationssprache formulierten Transformationsregeln verwende ich in Java implementierte Übersetzer, welche jeweils für bestimmte Elemente im Quellmodell verantwortlich sind. An die Stelle einer Transformations-Engine tritt bei mir ein in Java implementierter Transformationsmechanismus, welcher analog zu einer Transformations-Engine das Scheduling der Elementübersetzer (Transformationsregeln) übernimmt.

Regel-Scheduling

Der Transformationsmechanismus besitzt eine Warteschlange, in der Paare von einem zu übersetzenden Element aus dem Anwendungsmodell und dem dafür

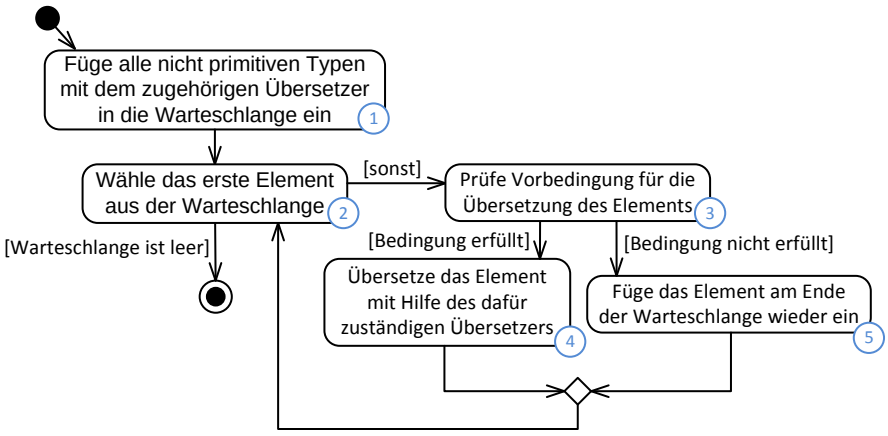


Abbildung 5.18: Verhalten des Transformationsmechanismus bei meiner Übersetzung

zuständigen Elementübersetzer auf ihre Übersetzung warten. Die Elemente im Anwendungsmodell werden beginnend mit den Typen mit Hilfe der zuständigen Elementübersetzer und der Warteschlange nach und nach in Elemente im Entwurfsmodell übersetzt (siehe Abb. 5.18). Die Übersetzung endet, wenn die Warteschlange leer ist.

Alle Elementübersetzer haben eine einheitliche Schnittstelle, welche die Prüfung von einzuhaltenden Bedingungen vor der Übersetzung ermöglicht (Schritt 3 in Abb. 5.18). Das entspricht im Wesentlichen einer Vorbedingung einer Transformationsregel. Ist die Vorbedingung erfüllt, wird das Element übersetzt (Schritt 4 in Abb. 5.18), wenn nicht, wird es für eine spätere Ausführung wieder in die Warteschlange eingefügt (Schritt 5 in Abb. 5.18). Falsche (insb. nicht erfüllbare) Vorbedingungen können zum Nichtterminieren der Transformation führen.

Regelvorbedingung

Die Übersetzung beginnt mit den Typen, weil die zugehörigen Übersetzer die einzigen sind, die keine Vorbedingungen haben (vergleichbar mit Axiomen).

Axiome

Ein Elementübersetzer übersetzt ein Element entweder komplett selbst oder delegiert Teile der Übersetzung an andere Übersetzer. Die Elementübersetzer sind nach dem Entwurfsmuster Chain of Responsibility [GHJV95] organisiert und sind jeweils für bestimmte Elemente des Anwendungsmodells verantwortlich. Ein Elementübersetzer kann andere Übersetzer sofort oder über die Warteschlange zeitversetzt einsetzen und direkt auswählen oder indirekt über den Transformationsmechanismus auf Basis der Verantwortlichkeiten wählen lassen. Auf diese Weise lässt sich die Transformation nach dem divide-and-conquer-Prinzip zerlegen und die Elementübersetzer lassen sich weitestgehend unabhängig voneinander implementieren. Zum Beispiel kann ein Typ von einem Elementübersetzer direkt in eine Klasse übersetzt werden, die Vererbungsbeziehung zeitversetzt durch einen anderen Übersetzer ergänzt werden, nachdem der Obertyp ebenfalls übersetzt wurde (das wird durch eine Vorbedingung sichergestellt), und die Kindelemente des Typs, nämlich seine Attribute, Operationen und Referenzen, können zeitversetzt durch von dem Transformationsmechanismus automatisch gewählte, dafür zuständige Elementübersetzer übersetzt werden.

Regelabhängigkeiten

Bei stark abweichenden Abstraktions- und Detailleveln bei den Elementen im Quell- und Zielmodell werden Tokens als Platzhalter für später zu übersetzende

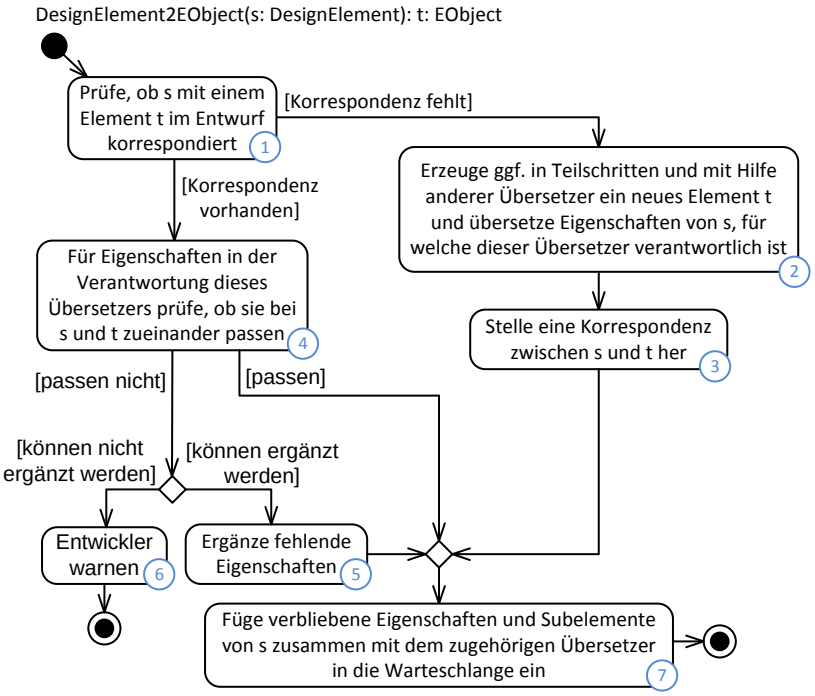


Abbildung 5.19: Verhalten eines Elementübersetzers

Token ≈ zu übersetzender Teilaspekt

Teile verwendet. Für solche Tokens gibt es zuständige Elementübersetzer. Durch den Einsatz von Tokens kann ein komplexer Transformationsschritt und der entsprechende Elementübersetzer in mehrere Schritte und Übersetzer zerlegt werden.

Aufbau einer Regel

Alle Elementübersetzer gehen bei der Übersetzung auf ähnliche Weise vor. Dieses Vorgehen ist im Aktivitätendiagramm in der Abb. 5.19 dargestellt. Ein Übersetzer erhält immer ein zu übersetzendes Element s aus dem Anwendungsmodell (oder ein Token) und übersetzt dieses in ein oder mehrere Elemente t des Entwurfsmodells, also des Ecore- oder Story-Diagramm-Modells. Da ausschließlich fehlende Teile im Entwurf ergänzt werden sollen, wird zuerst geprüft, ob das Element s bereits übersetzt oder manuell einem oder mehreren Elementen im Entwurf zugeordnet wurde (Schritt 1). Wenn nicht, wird das Element s übersetzt und in Relation mit dem Ergebnis der Übersetzung t gesetzt, also im Korrespondenzmodell entsprechend miteinander verknüpft (Schritte 2 und 3). Wenn das zu übersetzende Element s bereits einem Element t im Entwurf zugeordnet ist, werden diejenigen Eigenschaften des zugeordneten Elements t überprüft, deren Übersetzung in der Verantwortung des Übersetzers liegen (Schritt 4). Wenn sie nicht zu den Eigenschaften des zugehörigen Elements s im Anwendungsmodell passen, werden sie nach Möglichkeit ergänzt (Schritt 5). Ist das Ergänzen nicht möglich, wird der Entwickler auf die Abweichung hingewiesen (Schritt 6). Abschließend werden die Kindelemente und Eigenschaften des zu übersetzenden Elements s behandelt, welche außerhalb der Verantwortung des Elementübersetzers liegen. Die Übersetzung dieser wird an andere Übersetzer delegiert, indem die zu übersetzenden Elemente mit den zugehörigen (direkt oder indirekt gewählten) Übersetzern in die Übersetzungswarteschlange eingefügt werden (Schritt 7).

Der Transformationsmechanismus ist vergleichsweise einfach gehalten. Weil kein

Einschränkungen

Backtracking implementiert wurde, wird nicht garantiert, dass eine Übersetzung immer zum Erfolg führt, wenn es eine zum Erfolg führende Ausführungsreihenfolge der Regeln gibt. Die Reihenfolge der Regelanwendungen und die Terminierung der Transformation hängen insb. von den Vorbedingungen der Elementübersetzer ab sowie von ihrer Übersetzungsimplementierung (z.B. vom Anstoßen weiterer Übersetzer). Die Verantwortung für die Korrektheit der Transformation liegt derzeit bei ihrem Entwickler[18].

5.5.4 Transformationsregeln

Die Transformation ist aus 26 Transformationsregeln bzw. Elementübersetzern aufgebaut. Die Übersetzer sind in Java implementiert. Darum kann ich keine Spezifikation der Übersetzer als Transformationsregeln in einer der existierenden Transformationssprachen präsentieren. Um dennoch einen Eindruck von der Realisierung der einzelnen Übersetzer zu geben, stelle ich im Folgenden einige Übersetzer und die wesentlichen Konzepte dahinter exemplarisch in einer an relationale Transformationssprachen angelehnten, aus Platzgründen jedoch hauptsächlich auf konkreter Syntax basierenden Notation dar. Ergänzend dazu verwende ich Aktivitätendiagramme zur Beschreibung des Kontrollflusses einzelner Übersetzer.

Eine Übersicht aller Transformationsregeln und ihrer Zusammenhänge sowie detaillierte Beschreibungen weiterer Regeln befinden sich im Anhang C.3 (Regeln und ihre Abhängigkeiten in Abb. C.8, S. 322).

Übersetzen von Typen

Wie zuvor beschrieben beginnt meine Übersetzung mit den Typen im Anwendungsmodell (siehe Abb. 5.18, S. 129 und C.8, S. 322). Für die Übersetzung von nicht primitiven, als generierbar markierten Typen ist der Übersetzer Type2EClass verantwortlich, dessen wesentliches Verhalten ich in der Abb. 5.20 skizziert habe. Er erzeugt eine zu dem Typ TypeA passende Klasse gleichen Namens im Ecore-Modell (ein EClass-Objekt) und verknüpft diese mit dem zugehörigen Typ im Anwendungsmodell. Diese Korrespondenz – blau dargestellt – wird als Kreis mit zwei ausgehenden Kanten abgebildet. Die erzeugten Elemente werden grün hervorgehoben und mit einem ++ markiert.

Notation einer Regel

Das Verhalten des Übersetzers wird in der Abb. 5.22 durch ein Aktivitätendiagramm beschrieben (vgl. Abb. 5.19 auf S. 130). Wie bei allen Übersetzern wird zunächst geprüft, ob der zu übersetzende Typ s bereits übersetzt oder einer existierenden Klasse zugeordnet wurde. Wenn nicht, wird eine neue Klasse (EClass) mit den zum abstract-Attribut des Typen passenden Eigenschaften erzeugt und mit dem Typen in Korrespondenz gesetzt. Existiert eine mit dem Typ korrespondierende Klasse t schon, so wird nur die Konformität des abstract-Attributwerts des Typen zu den Eigenschaften der Klasse geprüft und der Entwickler im Fall einer Diskrepanz gewarnt. Ist der Typ s ein Untertyp eines anderen, wird anschließend die Übersetzung der Vererbungsbeziehung vorbereitet, indem der dafür zuständige Übersetzer TypeInheritance2EClassInheritance zusammen mit dem Typen s in die Übersetzungswarteschlange eingefügt wird. Abschließend werden alle

[18] Bei der Implementierung werden Korrektheit und Terminierung durch Tests sichergestellt.

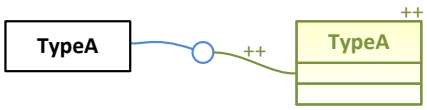


Abbildung 5.20: Type2EClass

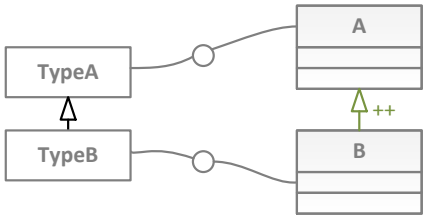


Abbildung 5.21: TypeInheritance2EClassInheritance

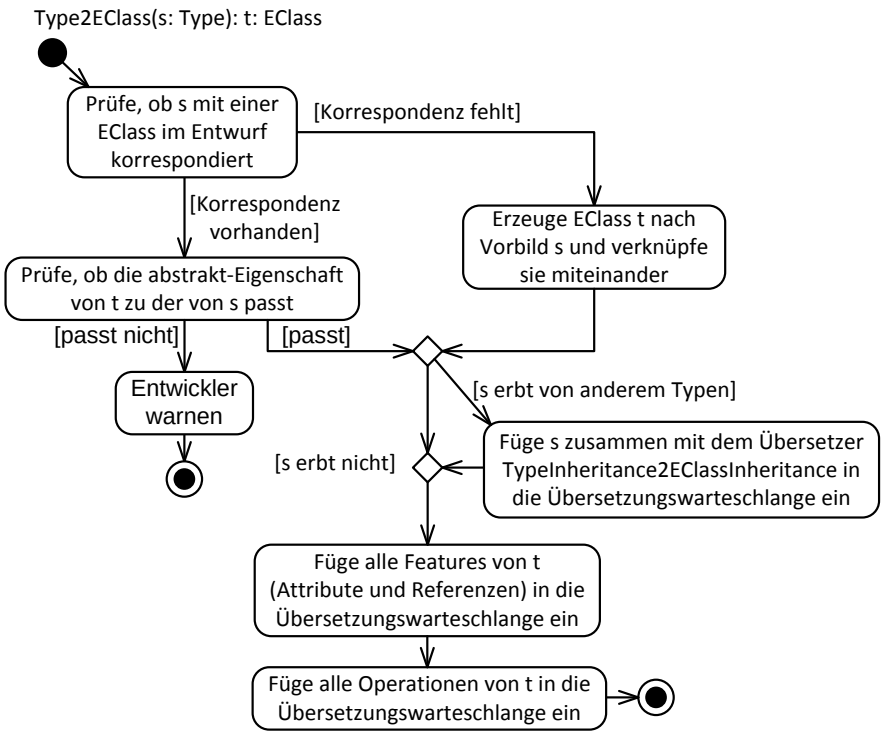


Abbildung 5.22: Kontrollfluss des Übersetzers Type2EClass

Kindelemente des Typen, nämlich Attribute, Referenzen und Operationen, jeweils mit einem durch den Transformationsmechanismus bestimmten Übersetzer in die Warteschlange eingefügt und somit für eine spätere Übersetzung vorbereitet.

Die Vererbungsbeziehung ist nur eine Eigenschaft eines Typ-Objekts, wird also im Anwendungsmodell (bzw. der Musterspezifikation) nicht durch ein eigenes Objekt repräsentiert. Darum wird stellvertretend das Typ-Objekt, zu dem die Eigenschaft gehört, in die Übersetzungswarteschlange eingefügt sowie der Übersetzer direkt bestimmt und nicht vom Transformationsmechanismus gewählt.

Übersetzen von Vererbungsbeziehungen zwischen Typen

Die Übersetzung einer Vererbungsbeziehung eines Typen zu einem Obertyp wird vom Übersetzer TypeInheritance2EClassInheritance übernommen. Sein Verhalten ist in der Abb. 5.21 skizziert. Dieser Übersetzer hat eine Vorbedingung, die erfüllt sein muss, bevor die Übersetzung erfolgen kann. Zu den beiden Typen, zwischen denen die Vererbungsbeziehung modelliert ist, muss jeweils eine damit korrespon-

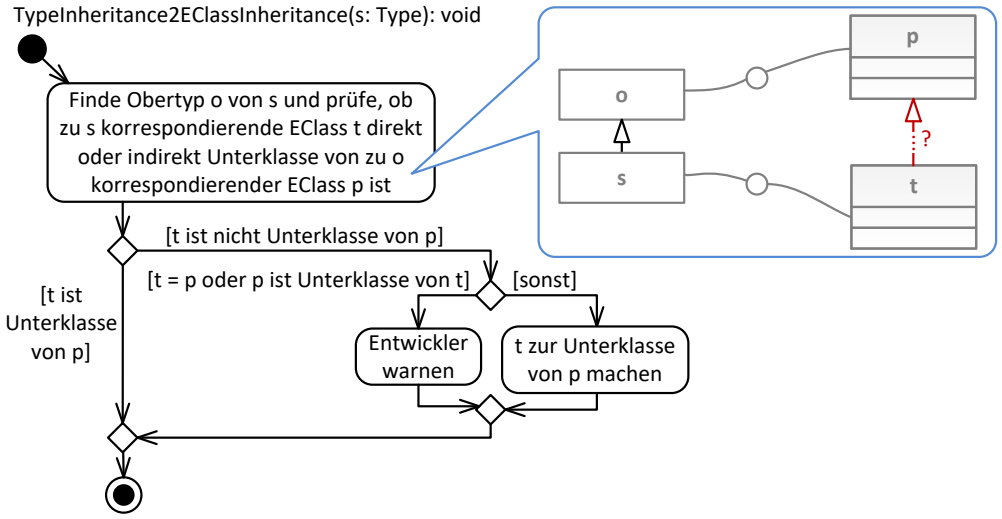


Abbildung 5.23: Kontrollfluss des Übersetzers TypeInheritance2EClassInheritance

dierende Klasse im Entwurfsmodell existieren. Ist die Bedingung erfüllt, so wird zwischen den entsprechenden Klassen im Entwurfsmodell eine Vererbungsbeziehung erstellt. Andernfalls wird die Übersetzung durch den Transformationsmechanismus verzögert (siehe Schritte 3–5 in Abb. 5.18 auf S. 129).

Notation der Vorbedingung

Die Vorbedingung besteht bei diesem und anderen Übersetzern aus Elementen im Anwendungs- und Entwurfsmodell, die vor der Übersetzung vorhanden sein und auf bestimmte Weise in Korrespondenz zueinander stehen müssen (analog zu TGG und QVT). Solche Vorbedingungen stelle ich im Folgenden in Grau dar, die zu übersetzenden Elemente werden schwarz oder farbig dargestellt, die bei der Übersetzung erzeugten Elemente werden durch grüne Kanten und grüne Schrift sowie das Label ++ hervorgehoben.

Sonderfall: Beziehung übersetzen

In der Abb. 5.23 wird das Verhalten des Übersetzers TypeInheritance2EClassInheritance als Aktivitätendiagramm dargestellt. Weil das zu übersetzende Element, die Vererbungsbeziehung, in diesem Fall nicht durch ein Objekt im Anwendungsmodell repräsentiert wird, weicht das Verhalten etwas von dem eines Elementübersetzers ab. Es wird keine Korrespondenz eines Objekts im Anwendungsmodell zu einem im Entwurfsmodell geprüft, sondern die zu übersetzende Eigenschaft, die Vererbungsbeziehung der mit dem übergebenen Typen s korrespondierenden Klasse t.

Sonderfall: Transitivität

Eine Vererbungsbeziehung zwischen zwei Typen im Anwendungsmodell repräsentiert eine direkte oder indirekte Vererbungsbeziehung in einem Entwurfsmodell. Darum wird geprüft, ob zwischen den Klassen t und p im Entwurfsmodell, welche mit den in Vererbungsbeziehung stehenden Typen s und o im Anwendungsmodell korrespondieren, eine direkte oder indirekte Vererbungsbeziehung vorliegt (siehe Skizze oben rechts in Abb. 5.23). Liegt sie nicht vor, wird eine direkte Vererbungsbeziehung ergänzt, falls nicht schon eine Vererbungsbeziehung in umgekehrter Richtung vorliegt.

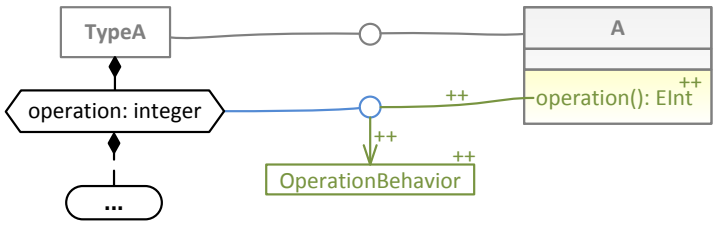


Abbildung 5.24: Operation2EOperation ohne Spezialisierung

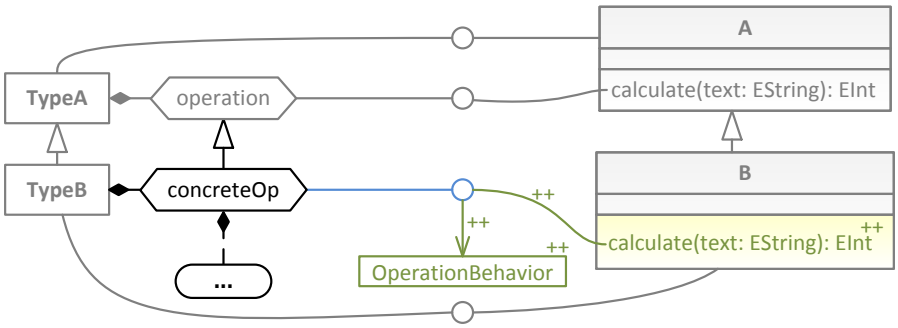


Abbildung 5.25: Operation2EOperation mit Spezialisierung

Übersetzen von Operationen

Sonderfall: Kontext-abhängige & schrittweise Übersetzung

Die meisten Übersetzer sind sehr ähnlich nach dem in der Abb. 5.19 (S. 130) beschriebenen Prinzip aufgebaut. Eine Besonderheit tritt bei der Übersetzung von Operationen auf. Zum einen ist hier die Übersetzung nicht nur von der zu übersetzenden Operation im Anwendungsmodell abhängig, sondern auch von schon existierenden Methoden im Entwurfsmodell. Zum anderen werden hier Tokens eingesetzt, um die Übersetzung des Operationsverhaltens in mehreren Schritten und dennoch nachvollziehbar durchzuführen. Das Verhalten des Übersetzers ist in den Abb. 5.24 und 5.25 skizziert. Hier werden zwei Fälle unterschieden.

Im einfachsten Fall soll eine Operation übersetzt werden, die nicht in einer Spezialisierungsbeziehung zu einer anderen Operation steht (Abb. 5.24). In diesem Fall müssen ausschließlich die Eigenschaften der zu übersetzenden Operation im Anwendungsmodell betrachtet werden. Gibt es keine mit der Operation korrespondierende Methode im Entwurfsmodell, so wird eine neue Methode gleichen Namens erzeugt. Ist ein Rückgabetyp spezifiziert, wird dieser entsprechend übersetzt, andernfalls wird void angenommen. Sind Parameter spezifiziert, werden auch diese mit Hilfe des zuständigen Übersetzers übersetzt.

Token

Ist für die Operation Verhalten in Form von Aktionen spezifiziert, so wird ein OperationBehavior-Token (dargestellt als Kasten mit dem Tokentyp als Label) erstellt und an den zur Operation gehörenden Korrespondenzknoten gehängt. Das Token repräsentiert das gesamte Verhalten einer Operation[19] und damit eine logische, nicht explizit im Anwendungsmodell vertretene Einheit, welche einzeln durch einen dafür zuständigen Übersetzer übersetzt werden soll. In diesem Fall repräsentiert das Token ein noch zu erstellendes Story-Diagramm-Modell.

[19] Das Verhalten einer Operation kann durch mehrere Aktionen beschrieben werden.

In dem Fall, dass die zu übersetzende Operation als Spezialisierung einer anderen Operation spezifiziert wurde (Abb. 5.25), müssen bei der Übersetzung nicht nur die Eigenschaften der zu übersetzenden Operation im Anwendungsmodell betrachtet werden, sondern auch der Kontext der bei der Übersetzung zu erzeugenden Methode im Entwurfsmodell. Wird die Operation concreteOp aus Abb. 5.25 übersetzt, ergibt sich die Signatur der erzeugten Methode aus der Signatur derjenigen Methode, die mit der spezialisierten – d.h. implementierten oder überschriebenen – Operation operation korrespondiert. Obwohl für die Operation concreteOp im Anwendungsmodell keine Parameter und kein Rückgabetyp spezifiziert wurden, erhält die Übersetzung der Operation beides.

Signatur „erben“

Die Unterscheidung der beiden Fälle und die Behandlung des letzteren führen zu einem relativ komplexen Kontrollfluss des Übersetzers für Operationen. Dieser ist im Anhang in der Abb. C.18 (S. 328) dargestellt.

Übersetzen von Aktionen

Das Verhalten einer Operation wird in Musterspezifikationen und somit auch in Anwendungsmodellen durch Aktionen beschrieben. Bei der Übersetzung des auf diese Wiese beschriebenen Verhaltens wird je ein Story-Diagramm generiert und mit der zur Operation gehörenden Methode im Ecore-Modell verknüpft (siehe Abb. 5.26). Die zur Operation gehörenden Aktionen werden jeweils in Kontrollflussausschnitte im Story-Diagramm übersetzt. Diese können aus mehreren Aktivitätenknoten bestehen und werden analog zu den Aktionen zu einer Sequenz zusammengesetzt.

Sequenz von Kontrollflussstücken

Anders als bei der Übersetzung der Struktur in ein Ecore-Modell unterscheiden sich die Abstraktions- und Detaillevel der zu übersetzenden Aktionen und der daraus abgeleiteten Story-Diagramm-Strukturen deutlich. Aus diesem Grund wird die Übersetzung einer Aktion mit Hilfe von Tokens in mehrere kleinere Übersetzungsschritte und der Korrespondenzknoten in mehrere Teilkorrespondenzen zerlegt.

Sonderfall: stark abweichender Detailgrad

Tokens ermöglichen zum einen eine logische Zerlegung der Elemente im Anwendungsmodell, insb. Aktionen, in einzeln übersetzbare Einheiten, wobei ein Token je eine übersetzbare Einheit repräsentiert. Zum anderen dokumentieren Tokens im Detail woraus die Story-Diagramm-Elemente schrittweise entstanden sind.

Warum Tokens?

Die Übersetzung einer Operation wird mit Hilfe eines OperationBehavior-Tokens (siehe Abb. 5.24, 5.25) in zwei Schritte zerlegt. Das Token repräsentiert das gesamte Verhalten einer Operation und wird einzeln übersetzt. Dadurch wird die Übersetzung des Operationsverhaltens in ein Story-Diagramm von der Übersetzung einer Operation in eine Methode entkoppelt.

Für die Übersetzung des OperationBehavior-Tokens bzw. des Operationsverhaltens ist der Übersetzer OperationBehavior2StoryDiagram zuständig. Sein Verhalten ist in der Abb. 5.26 skizziert. Er übersetzt alle Aktionen einer Operation in ein entsprechendes Verhaltensmodell – ein Story-Diagramm-Modell – und verknüpft dieses mit Hilfe des Tokens mit der Operation. Die Übersetzung der einzelnen Aktionen wird an die jeweils zuständigen Übersetzer delegiert und ihre Ergebnisse zu einem vollständigen Story-Diagramm kombiniert. Im Detail ist diese Übersetzung in einem Aktivitätendiagramm im Anhang C.3.3 dargestellt (Abb. C.20, S. 330).

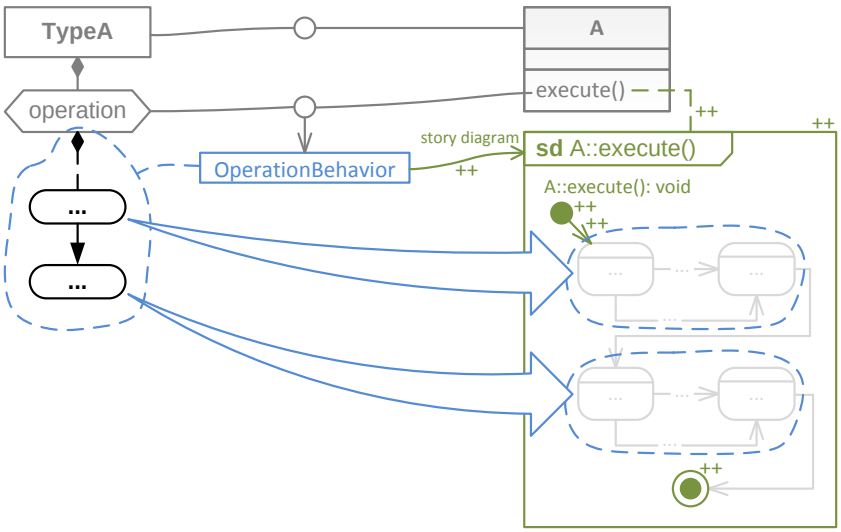


Abbildung 5.26: OperationBehavior2StoryDiagram

weitere Übersetzer im Anhang

Die vielen Teile eines Story-Diagramm-Ausschnitts, welcher das Verhalten einer Aktion repräsentiert, werden mit Hilfe weiterer Tokens schrittweise mit der zugehörigen Aktion verknüpft. Die Tokens bilden dabei eine Baum-artige Struktur. Wie genau die verschiedenen Aktionen in Teile eines Story-Diagramms übersetzt und zu einem vollständigen Story-Diagramm kombiniert werden, beschreibe ich ebenfalls im Anhang C.3.3 (S. 328 ff.).

5.6 Verbleibende, nicht automatisierbare Schritte

Anwendung abschließen

Nach der automatischen Synthese einer Musterimplementierung im Entwurfsmodell wie sie in den Abschnitten 5.3 bis 5.5 beschrieben wird können zum Abschluss der Musteranwendung manuell durchzuführende Schritte nötig sein. Zum einen müssen die in der Musterspezifikation beschriebenen Entwurfsaufgaben erledigt werden. Zum anderen müssen nicht generierbare Entwurfselemente einer Musterspezifikation von einem Entwickler in seinen Entwurf übertragen und den zugehörigen Musterrollen zugeordnet werden. Erst wenn allen Musterrollen entsprechende Entwurfsteile zugeordnet und alle Aufgaben erledigt sind, ist eine Musteranwendung abgeschlossen.

ggf. Anpassungen

Darüber hinaus kann ein Entwickler bei Bedarf die generierte Musterimplementierung – die kanonische (triviale) Implementierungsvariante eines Musters[20] – innerhalb eines bestimmten Rahmens an seine konkrete Situation anpassen und z.B. zusätzliche Klassen in die Vererbungshierarchie einfügen, ohne dabei von der Musterspezifikation abzuweichen.

[20] Neben der kanonischen gibt es meist unzählige weitere geringfügig davon abweichende, aber zur Musterspezifikation konforme Musterimplementierungen (siehe Abb. 3.12, S. 57, Bsp.: Abb. 3.10 & 3.11, S. 55).

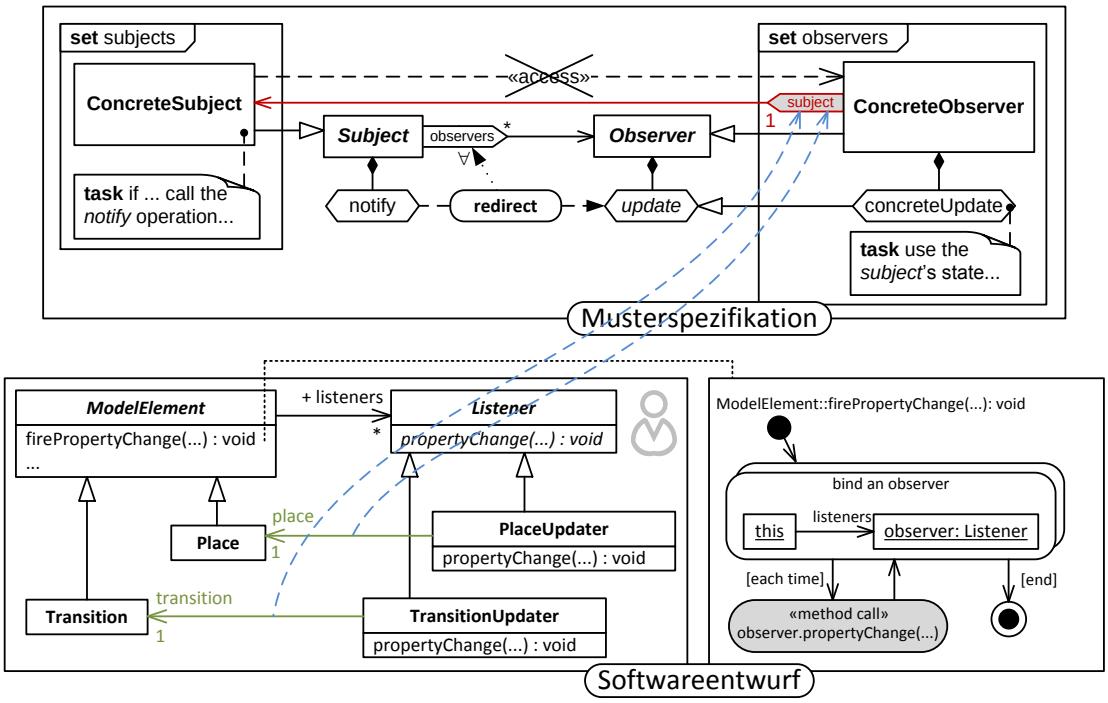


Abbildung 5.27: Manuelles Ergänzen einer nicht eindeutig spezifizierbaren Referenz

5.6.1 Nicht generierbare Elemente ergänzen

unvollständige Spezifikation

Sind in einer Musterspezifikation Teile der Spezifikation als nicht generierbar markiert, so liegt das daran, dass die Spezifikation unvollständig oder nicht eindeutig ist. In solchen Fällen muss ein Entwickler die fehlenden Teile des Musters in der Musterimplementierung selbst ergänzen. Zum Beispiel kann bei einer call-Aktion das Zielobjekt oder die aufgerufene Operation fehlen, wodurch ein Aufruf nur unvollständig beschrieben wäre. read- und write-Aktionen gelten grundsätzlich als nicht generierbar. Weil nicht definiert ist, was mit einem ausgelesenen Wert passieren soll bzw. welcher Wert einer Variable zugewiesen werden soll, sind diese immer unvollständig spezifiziert und dienen nur der kompakten Musterspezifikation und einer rudimentären Überprüfung von Musterimplementierungen auf Konformität mit der Spezifikation. Zum Beispiel kann das völlige Fehlen von Methodenaufrufen in einem Story-Diagramm auf eine fehlende Implementierung einer spezifizierten call-Aktion hindeuten (mehr dazu in Abschnitt 6.2.4).

nicht eindeutige Spezifikation

Im Zusammenhang mit Set Fragments können Situationen auftreten, in denen Teile einer Musterspezifikation nicht eindeutig sind. Das ist der Fall bei der subject-Referenz des Observer-Musters (siehe Bsp. in Abb. 3.7, S. 52 und Abschn. 5.2, S. 105 ff.). Die zugehörige Musterspezifikation ist der Abb. 5.27 zu entnehmen. Die nicht eindeutige und somit nicht generierbare Referenz ist rot dargestellt. Eine Referenz in einer Musterspezifikation repräsentiert eine Referenz zwischen zwei Klassen im Entwurf. Da die durch die Referenz verknüpften Typen in zwei verschiedenen Set Fragments liegen, kann es in einer Musterimplementierung beliebig viele ConcreteSubject-Klassen und unabhängig davon beliebig viele ConcreteObserver-Klassen mit je einer subject-Referenz geben. Welche ConcreteObserver-Klasse welche ConcreteSubject-Klasse referenzieren soll, ist somit nicht klar und muss

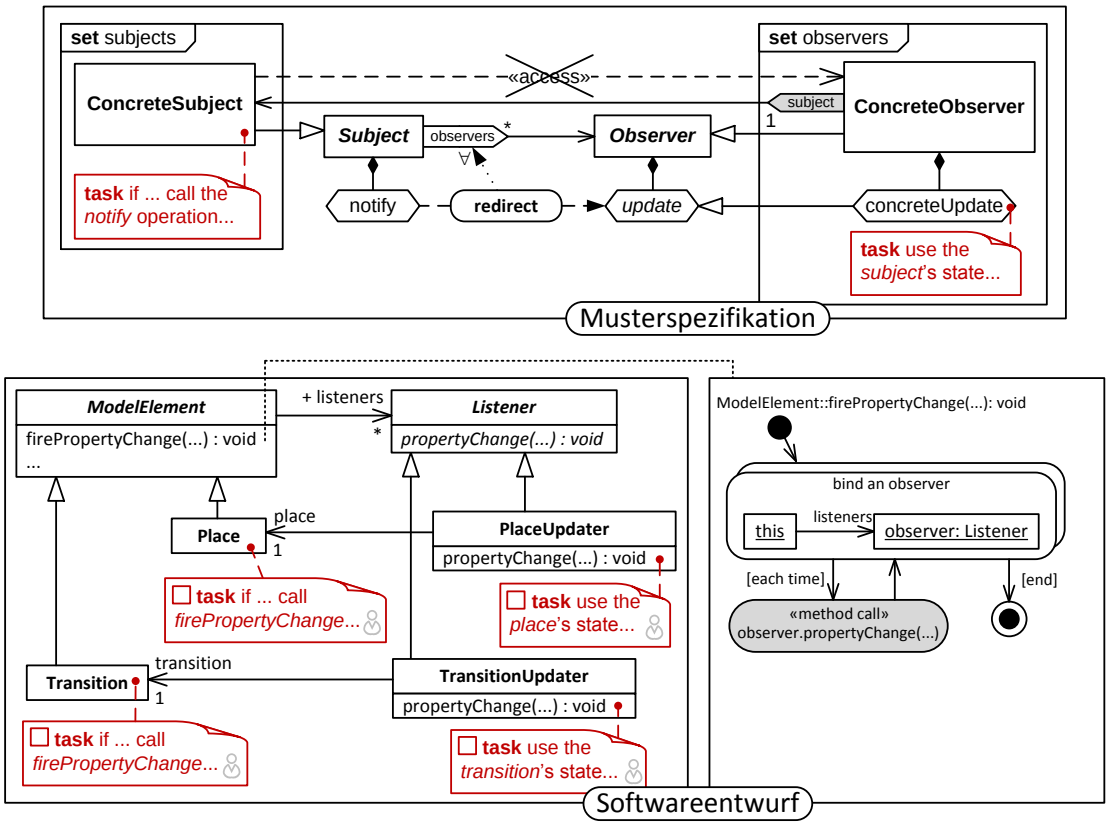


Abbildung 5.28: Manuell zu erledigende Aufgaben nach dem Generieren einer Observer-Musterimplementierung

vom Entwickler entschieden werden. Dazu legt er manuell der subject-Referenz entsprechende Referenzen im Entwurf an (grün dargestellt). Zur Dokumentation seiner Entscheidung und zu Validierungszwecken ordnet er die erstellten Referenzen der Rolle subject zu (blau gestrichelt dargestellt). Erst jetzt gibt es zu jeder Musterrolle eine Entsprechung im Entwurfsmodell.

5.6.2 Aufgaben abarbeiten

Als Entwurfsaufgaben (Task bzw. TaskDescription) werden zu erledigende Schritte spezifiziert, die stark von der vorliegenden Situation abhängen, in Musterbeschreibungen nur grob umrissen sind und daher nicht automatisiert werden können. Bei dem Strategy-Muster wird z.B. das Modellieren / Implementieren des Strategiealgorithmus als Aufgabe formuliert, beim State-Muster u.a. das Setzen des initialen Zustands.

Auch beim Observer-Muster aus dem Beispiel in Abschnitt 5.2 (S. 105 ff.) sind Aufgaben spezifiziert (siehe Abb. 5.28). Eine Aufgabe stellt das Implementieren eines Aufrufs der notify-Operation bei Zustandsänderungen dar. Eine weitere Aufgabe ist das Reagieren auf die Zustandsänderung. Für beide Aufgabenbeschreibungen aus der Musterspezifikation sind bei der Generierung der Musterimplementierung entsprechende, unerledigte Aufgaben im Anwendungsmodell angelegt und mit den Entwurfselementen verknüpft worden, zu denen sie gehören (Aufga-

benbeschreibungen und Aufgaben sind in Abb. 5.28 rot dargestellt). Erledigt der Entwickler eine Aufgabe, kann er sie durch Abhaken als erledigt markieren. Somit dienen Aufgaben hauptsächlich als Gedächtnisstütze bei der Fertigstellung einer Musterimplementierung. Ist eine Aufgabe noch nicht als erledigt markiert, erhält der Entwickler einen entsprechenden Hinweis.

Erledigungsstatus als Erinnerung

5.7 Mögliche Anpassungen einer generierten Musterimplementierung

Grundsätzlich stehen einem Entwickler alle Möglichkeiten offen, eine generierte Musterimplementierung an seine Bedürfnisse anzupassen. Um die Intention des Musters zu erhalten, muss die geänderte Musterimplementierung jedoch immer noch konform zur Musterspezifikation sein. So können nicht nur zusätzliche Vorkommen der in Set Fragments beschriebenen Entwurfsstrukturen nachträglich ergänzt werden, sondern auch Vererbungshierarchien erweitert oder direkte durch indirekte Beziehungen ausgetauscht werden sowie Methoden, Attribute und Referenzen in der Hierarchie verschoben werden.

Während das Ergänzen zusätzlicher Vorkommen der in Set Fragments beschriebenen Entwurfsstrukturen zum Teil automatisiert werden kann, müssen andere Anpassungen komplett manuell durchgeführt werden.

5.7.1 Nachträglich zusätzliche Set-Fragment-Instanzen ergänzen

Nachdem ein Muster mit dem vorgestellten Verfahren angewendet wurde, existieren eine Rollenzuordnung und ein Anwendungsmodell, womit eine Musterimplementierung detailliert modelliert ist. Soll für ein Set Fragment der Musterspezifikation eine zusätzliche Set-Fragment-Instanz im Entwurfsmodell erstellt werden, nutzt man denselben Mechanismus wie bei der initialen Musteranwendung. In diesem Fall werden das Dekorations- und Anwendungsmodell sowie die Musterspezifikation geladen und man fährt direkt mit den Schritten 5 und 7 aus Abb. 5.10 (S. 115) fort. Man bestimmt die zu ergänzende Set-Fragment-Instanz, woraufhin das schon existierende Anwendungsmodell erweitert und anschließend das Entwurfsmodell ergänzt wird. Im Prinzip „faltet“ man die Musterspezifikation weiter auf und setzt die ursprüngliche Musteranwendung fort. Schon existierende Rollenzuordnungen bleiben erhalten. Bei der Musterimplementierung aus Abb. 5.28 z.B. kann eine weitere ConcreteSubject-Klasse Token ergänzt werden, welche automatisch erzeugt wird und wie Place und Transition von der Klasse ModelElement erbt.

Auffaltung fortsetzen

5.7.2 Direkte durch indirekte Beziehungen ersetzen

In meiner Musterspezifikationssprache betrachte ich Spezialisierungsbeziehungen (also Vererbungsbeziehungen zwischen Typen und Überschreib- bzw. Implementierungsbeziehungen zwischen Operationen) als transitiv. Die den Musterrollen entsprechenden Klassen bzw. Operationen können also direkt oder indirekt auf

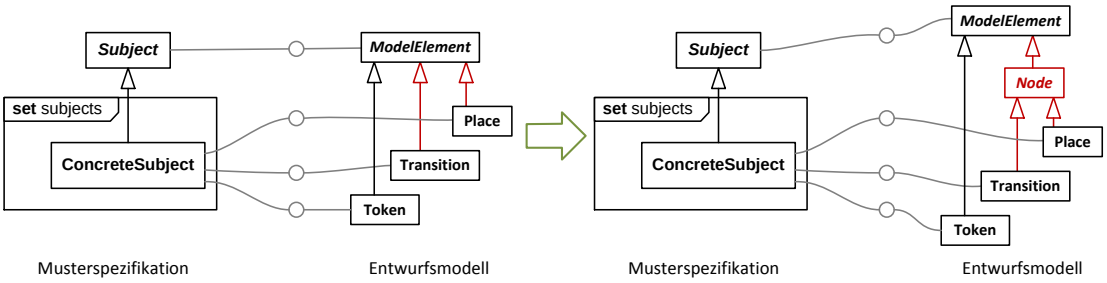


Abbildung 5.29: Ersetzen einer direkten durch eine indirekte Vererbungsbeziehung

die spezifizierte Weise in Beziehung stehen. In beiden Fällen sind sie konform zur Musterspezifikation.

Bei der Synthese einer Musterimplementierung wird nur die einfachste Implementierungsvariante generiert und demnach nur direkte Beziehungen verwendet. So erben z.B. die Klassen Place und Transition aus Abb. 5.28 direkt von der Klasse ModelElement, analog zu den zugehörigen Rollen ConcreteSubject und Subject der Observer-Musterspezifikation.

Klassenhierarchie erweitern

Möchte der Entwickler eine gemeinsame Oberklasse Node für die Klassen Place und Transition einführen, so muss er die Klasse Node selbst erstellen und die Vererbungsbeziehungen manuell anpassen. Die Rollenzuordnungen bleiben dabei erhalten, sodass die Musterimplementierung weiterhin automatisch validiert werden kann. Durch die Änderung entsteht wie in der Abb. 5.29 dargestellt eine indirekte Vererbungsbeziehung zwischen den Klassen Place und ModelElement sowie Transition und ModelElement. Somit ist die Musterimplementierung weiterhin konform zur Musterspezifikation.

indirektes Überschreiben

Eine Spezialisierungsbeziehung zwischen Operationen wird analog zu einer Vererbungsbeziehung zwischen Typen behandelt. Auch hier kann eine weitere Methode im Entwurf ergänzt werden, sodass eine Methode nicht mehr direkt, sondern indirekt eine andere Methode implementiert bzw. überschreibt.

5.7.3 Methoden und Referenzen in Vererbungshierarchien verschieben

Auch eine direkte Kompositionsbeziehung (die Zugehörigkeit einer Operation, eines Attributs oder einer Referenz zu einem Typ) kann manuell durch eine indirekte Komposition ersetzt werden. Hierbei wird berücksichtigt, dass eine Klasse die Operationen, Attribute und Referenzen der Oberklasse erbt. Wird z.B. in einer Musterspezifikation wie in Abb. 5.28 (S. 138) ein Typ ConcreteObserver mit einer Operation conreteUpdate definiert, so muss die zugehörige Klasse PlaceUpdater im Entwurf eine der Rolle conreteUpdate entsprechende Methode – hier propertyChange – nicht zwingend selbst enthalten, sondern kann diese von einer Oberklasse erben.

Damit ist es bei diesem Beispiel laut Musterspezifikation zulässig, eine neue Klasse ViewUpdater mit einer propertyChange-Methode zu erstellen und die propertyChange-Methoden der Klassen PlaceUpdater und TransitionUpdater wie in der Abb. 5.30 dargestellt zu ersetzen.

Durch das manuelle Entfernen der propertyChange-Methoden aus den Klassen

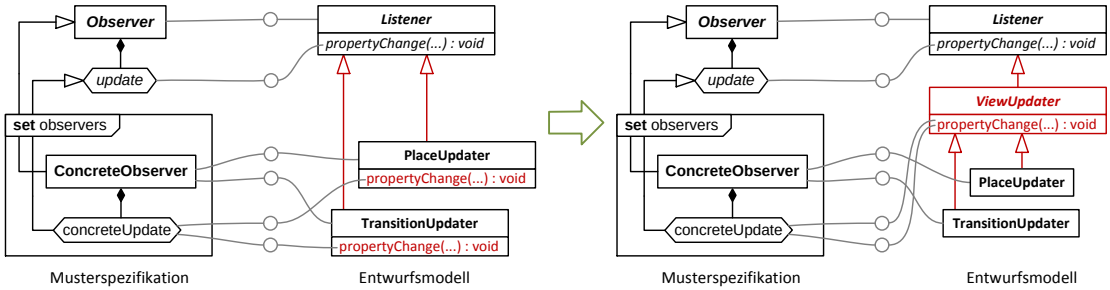


Abbildung 5.30: Ersetzen einer direkten durch eine indirekte Kompositionsbeziehung

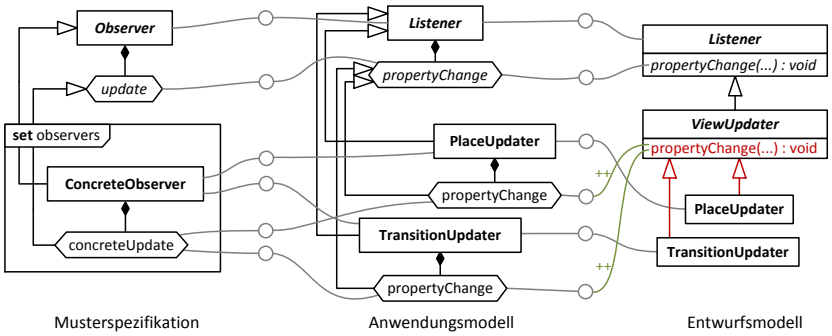


Abbildung 5.31: Rollenzuordnung bei einer indirekten Kompositionsbeziehung

verlorene Korrespondenzen wiederherstellen

PlaceUpdater und TransitionUpdater werden allerdings die Zuordnungen der concreteUpdate-Rolle und der zugehörigen Operationen im Anwendungsmodell invalide. Sie verweisen auf keine Methode mehr, was zu einer Warnung des Entwicklers beim Validieren der Musterimplementierung führt. Um für folgende Entwickler und das Validierungssystem erkennbar zu machen, dass die propertyChange-Methode der Klasse ViewUpdater nun die Rolle concreteUpdate übernimmt, muss der Entwickler in diesem Fall die unvollständig gewordene Rollenzuordnung wieder vervollständigen. Dazu ordnet er den beiden propertyChange-Operationen im Anwendungsmodell die neue propertyChange-Methode im Entwurfsmodell zu (siehe Abb. 5.31). Für diese Zuordnung kann der Entwickler genauso durch die Set-Fragment-Instanzen navigieren wie bei der initialen Musteranwendung (siehe Abschnitt 5.3.1, S. 109 ff.).

Analog zu Operationen können auch Attribute in der Vererbungshierarchie nach oben verschoben werden. Das gleiche gilt für Referenzen, obwohl deren Kompositionsbeziehung nicht explizit als Kante dargestellt wird, sondern durch die Anordnung des Referenz-Labels am Typ (siehe Abb. 3.16, S. 60 oder Tab. A.2, S. 254).

5.8 Einschränkungen

spezielle Entwurfsmodelle

Die vorgestellte Übersetzung der DAL (siehe Abschn. 5.5) schränkt die Anwendbarkeit des Ansatzes auf Klassen- und Story-Diagramm-Modelle ein. Durch die Entwicklung von Übersetzungen in andere Modellierungssprachen (z.B. UML-Klassenmodell statt Ecore-Modell oder State Machines statt Story-Diagramme) lässt sich die Anwendbarkeit des Ansatzes ausweiten (siehe Abb. 5.3, S. 104).

MDSD / MDA

Änderungen nur im Modell, nicht im Code

Die Anwendung eines Musters erfolgt bei meinem Ansatz auf Modell-Ebene, nicht im Quellcode. Der Grund dafür ist meine Annahme eines vollständig modellgetriebenen Entwicklungsprozesses, wo ausführbarer Quellcode komplett aus Modellen generiert wird. Da aus Ecore-Klassen- und Story-Diagrammen vollständig ausführbarer Code generiert werden kann und modellgetriebene Softwareentwicklung verbreitet und Industrie-erprobt ist, ist das ein durchaus realistisches Szenario. Dieser Entwicklungsprozess nimmt Entwicklern jedoch die Wahl zwischen Entwurfsänderungen im Modell und im Quellcode, da sämtliche Änderungen im Modell erfolgen müssen.

kein Vervollständigen von Verhalten

Nach der Zuordnung von im Entwurf bereits existierenden Klassen, Methoden, Attributen und Assoziationen zu den Rollen eines anzuwendenden Entwurfsmusters lässt sich der fehlende Teil einer Entwurfsmusterimplementierung ergänzen. Bereits existierende Teile eines Story-Diagramms lassen sich nicht auf diese Weise wiederverwenden. Zu einer spezifizierten Aktion eines Entwurfsmusters wird ein Story-Diagramm nur dann generiert, wenn es bis dahin komplett fehlt. Zum einen liegt das daran, dass ich für das Vervollständigen eines partiell modellierten Verhaltens noch keine Lösung habe und das besonders schwierig zu sein scheint. Zum anderen ist das Zuordnen bestimmter Teile eines Story-Diagramms zu Teilen einer Entwurfsmusterspezifikation zu detailliert (Zuordnung über mehrere Tokens, siehe Abschn. C.3.3) und damit zu aufwändig, um sie manuell zu erstellen.

kein Nachdokumentieren von Verhalten

Als Konsequenz daraus kann eine Musterimplementierung in Story-Diagrammen nicht durch nachträgliches Modellieren der Anwendungsstelle erfasst werden. In Klassendiagrammen ist das möglich.

Ebenso lässt sich ein durch automatische Anwendung eines Musters erstelltes Story-Diagramm nicht nachträglich anpassen und validieren, da die Rollenzuordnung auf Ebene von Story-Diagrammen nicht ohne weiteres angepasst werden kann. Eine automatisch erzeugte Musterimplementierung lässt sich also bisher nur im Klassenmodell anpassen, nicht aber in Story-Diagramm-Modellen. Das schränkt die Freiheiten eines Entwicklers bei der Implementierung eines Musters deutlich ein. Entweder verzichtet man auf abweichende Implementierungsvarianten oder man verzichtet auf die automatische Validierung des modellierten Verhaltens.

kein Anpassen von Verhalten

manuelle Schritte

Aufgrund der absichtlich unpräzise und unvollständig beschriebenen Entwurfsmuster und zugehöriger Implementierungsvarianten lassen sich einige Teile einer Entwurfslösung nur ebenso unpräzise und unvollständig in einer Musterspezifikation erfassen wie ihre Beschreibung in der Literatur. Eine Folge daraus sind eingeschränkte Möglichkeiten, eine Musterimplementierung automatisch zu erzeugen. So lassen sich vage beschriebene Teile der Entwurfslösung nur als Aufgabe (Task) beschreiben und müssen nach der Generierung einer Musterimplementierung manuell ergänzt werden. Ebenso verhält es sich mit unterspezifizierten Entwurfsteilen (diese sind als nicht generierbar markiert und werden in Spezifikationen grau dargestellt) wie einer read-Aktion, wo zwar spezifiziert wird, dass z.B. ein Attributwert gelesen wird, aber offen gelassen ist, wie darauf zugegriffen wird bzw. wie der Wert verwendet wird (z.B. in einer Zuweisung oder als Argument in einem Methodenaufruf). Bei generierten Operationen müssen ggf. Parameter manuell ergänzt oder Parametertypen angepasst werden, wenn die Parameter oder ihre Typen in der Spezifikation fehlen.

Das Konzept der Set Fragments und die zugehörige Auffaltungsoperation sind

formal definiert. Bisher ist jedoch die Korrektheit der Auffaltungsoperation nicht bewiesen (siehe Anmerkung A.16, S. 289), z.B. wurde nicht nachgewiesen, dass ein durch Auffaltung erzeugter Graph weiterhin zusammenhängend ist und alle Knoten und Kanten mit denen des ursprünglichen Graphen (der Spezifikation) korrespondieren (ein Homomorphismus existiert).

Aussagen über Auffaltung nicht bewiesen

Die Transformationsregeln sind in Java implementiert. Dadurch lassen sich Beweisverfahren wie z.B. Erreichbarkeitsanalysen nicht so einfach auf die vorgestellte Transformation anwenden wie bei Transformationssprachen mit bewiesenen Eigenschaften (z.B. bei TGG [Sch94, SK08, LAST16]).

Beweise schwierig

5.9 Zusammenfassung und Ausblick

In diesem Kapitel stelle ich ein Verfahren zur semiautomatischen Anwendung spezifizierter Entwurfsmuster vor. Bei dem Verfahren werden sowohl eine Implementierung des Musters im Entwurfsmodell als auch ein Modell der Anwendungsstelle generiert. Letzteres ist insb. zur Visualisierung und Prüfung von Musterimplementierungen notwendig und erspart aufwändiges Dokumentieren von Anwendungsstellen[21]. Die Musteranwendung erfolgt in einem Softwareentwurfsmodell bestehend aus einem Ecore-Klassenmodell für die Struktur und einem Story-Diagramm-Modell für das Verhalten der modellierten Softwareanwendung.

interaktive Musteranwendung & Korrespondenzenerfassung

Zur Anwendung eines Musters ordnet ein Entwickler die Rollen einer Musterspezifikation den Elementen eines Entwurfsmodells zu und nutzt dazu eine spezielle, an die Musterspezifikation angelehnte Sicht auf die Anwendungsstelle, die Musteranwendungssicht. Alle im Entwurfsmodell fehlenden Teile einer Musterimplementierung werden automatisch erzeugt und zur Nachvollziehbarkeit mit den zugehörigen Musterrollen verknüpft.

automatische Vervollständigung

Die Generierung einer Musterimplementierung ist in zwei Schritte unterteilt: (a) die Auffaltung einer Musterspezifikation – also das Vervielfältigen der in Set Fragments spezifizierten Entwurfsteile und das Erzeugen einer DAL-Repräsentation einer Musterimplementierung – und (b) die Übersetzung der DAL-Repräsentation in eine konkrete Implementierung im vorliegenden Entwurfsmodell. Für die Auffaltung sind Operationen definiert, welche eine zur Musterspezifikation konforme Vervielfältigung der in Set Fragment spezifizierten Entwurfsteile sicherstellen. Die Übersetzung ist als modulare Modelltransformation aufgebaut, bei welcher das Ergebnis der Übersetzung nicht nur von der Spezifikation, sondern auch von dem Kontext einer zu erzeugenden Musterimplementierung abhängt (z.B. wird in einigen Fällen die Signatur einer generierten Methode von einer im Entwurf bereits existierenden Methode übernommen) und welche an durch den Anwender vorgegebenen Stellen im Zielmodell (vorgegebene Korrespondenzen) beginnt.

Auffaltung & DAL-Übersetzung

spezielle Modelltransformation

Nach der Generierung einer Musterimplementierung, zu welcher neben der Klassenstruktur auch ein Verhaltensmodell gehört, kann die Implementierung in einem bestimmten Rahmen angepasst werden (z.B. durch Austausch einer direkten durch eine indirekte Vererbungsbeziehung). Wird der Rahmen verlassen, kann die automatische Konformitätsprüfung der Musterimplementierung auf die Abweichung von der Spezifikation hinweisen.

Anpassbarkeit der Implementierung

[21] Entwurfsentscheidungen müssen jedoch weiterhin geeignet dokumentiert werden.

Ausblick

In Zukunft sollte die Praxistauglichkeit des vorgestellten Verfahrens zu Musteranwendung genauer evaluiert werden, z.B. anhand von Feldstudien und Umfragen. Es ist interessant, herauszufinden, in welchem Umfang das Vorgehen beim Einsatz von Entwurfsmustern hilft, wie die Akzeptanz dieses Vorgehens bei Entwicklern und Architekten ist und mit welchen Hindernissen oder Einbußen es in der Praxis verbunden ist.

Der Beweis der beschriebenen Konsistenzeigenschaften der formal definierten Auffaltungsoperation wäre eine sinnvolle Ergänzung der vorliegenden Arbeit. Es könnte insbesondere gezeigt werden, dass Auffaltungen ausschließlich zusammenhängende und zur Spezifikation bzgl. Set Fragments konsistente Graphen erzeugen.

Außerdem wäre es spannend, zu untersuchen, ob die hier eingeführten Operationen Auffaltung und Übersetzung sich mit existierenden, bidirektionalen Transformationssprachen (ggf. mit Hilfe einiger Erweiterungen) vollständig beschreiben lassen und die Transformationen wartbarer (z.B. lesbarer, kompakter, leichter anpassbar) oder performanter machen. Möglicherweise eignet sich das in dieser Arbeit beschriebene Szenario der Musteranwendung als Fallbeispiel für Vergleiche von Graphtransformationssprachen.

Das Verfahren zur Musteranwendung könnte auf die Quellcode-Ebene ausgeweitet werden, sodass Musterimplementierungen und Entwurfsänderungen wahlweise im Entwurfsmodell oder im Quellcode erfolgen können und die Artefakte konsistent zueinander gehalten werden.

6 Validierung der Konsistenz von Musterimplementierungen zu ihren Musterspezifikationen

Aufgrund fehlender oder unzureichender Dokumentation und wechselnder Entwickler können in einem Softwaresystem angewandte Entwurfsmuster leicht übersehen oder missverstanden werden [PULPT02]. Wenn die mit dem Einsatz eines Entwurfsmusters verbundene Intention und der Entwurf unklar sind, können spätere Systemanpassungen zu ungünstigen Entwurfsentscheidungen und langfristig zu Design-Erosion führen [BTGH06, vGB02]. Sind Entwurfsmusterimplementierungen dokumentiert, können bei Entwurfsänderungen zumindest Flüchtigkeitsfehler auftreten.

Zweck der Validierung

Um die Wahrscheinlichkeit für solche Entwurfsfehler zu reduzieren, sollen Entwurfsmusterimplementierungen nicht nur expliziert erfasst und visualisiert werden (Kap. 4), sie sollen insb. nach manuellen Entwurfsanpassungen auf Korrektheit geprüft werden. Dabei soll die Konformität aller Musterimplementierungen zu den zugehörigen Musterspezifikationen sichergestellt werden. Entwickler sollen sich bewusst für oder gegen den Einsatz eines bestimmten Entwurfsmusters entscheiden. Versehentliche Entwurfsabweichungen sollen vermieden werden.

Herausforderungen

Zu den Herausforderungen bei der Validierung von Musterimplementierungen gehören die Formalisierung der Entwurfslösung eines Entwurfsmusters, die Erfassung aller Musterimplementierungen mit allen für eine Validierung nötigen Details und der Abgleich der Entwurfslösung mit einer zugehörigen Implementierung. Während die ersten zwei Punkte bereits in den Kapiteln 3 und 4 behandelt werden, gehe ich in diesem Kapitel auf den letzten der drei Punkte ein. Hierbei stellt sich die Frage, welche Eigenschaften einer Musterimplementierung sich automatisch prüfen lassen und wie.

Grenzen verwandter Arbeiten

Bisherige Verfahren prüfen im Wesentlichen, ob eine Musterimplementierung strukturell 1-zu-1 einer Musterspezifikation entspricht (siehe Abschn. 9.4). Es wird also sichergestellt, dass zu jeder Musterrolle eine Entsprechung in der Musterimplementierung vorhanden ist und alle spezifizierten Beziehungen – Assoziationen und Vererbung – in gleicher Form vorhanden sind. Strukturelle Abweichungen von der Spezifikation wie das Einfügen zusätzlicher Klassen in eine Vererbungshierarchie werden (mit Ausnahme von Eden et al., S. 235) nicht toleriert, was Entwickler in ihrer Flexibilität einschränkt. In einigen Fällen müssen Musterimplementierungen vor der Analyse durch Reverse-Engineering-Verfahren erkannt (Kim & Shen, S. 236) oder eine Zuordnung zur Musterspezifikation (Musterrollen) manuell vorgenommen werden (Eden et al., S. 235). Das ist fehleranfällig und aufwändig. Verhalten und Abhängigkeiten werden in den Prüfungen bisheriger Ansätze kaum oder gar nicht berücksichtigt.

Lösungsansatz

Durch das automatische Erstellen eines detaillierten Modells einer Musterimplementierung schon bei der Musteranwendung (Kap. 5) sind Reverse Engineering und manuelle Rollenzuordnungen bei meinem Ansatz (in der Regel[1]) nicht nötig. Bei der Prüfung der implementierten Entwurfsstruktur nutze ich die Transitivität von z.B. Vererbungsbeziehungen aus, erlaube mehr Implementierungsvarianten und räume dadurch Entwicklern mehr Freiheiten bei der Implementierung eines Musters ein. Neben der Struktur wird zum Teil auch das implementierte Verhalten berücksichtigt, z.B. indem das Vorhandensein und die Reihenfolge bestimmter Operationen sichergestellt werden. Spezifizierte Kopplungsrestriktionen werden ebenfalls in den Prüfungen berücksichtigt. Die Prüfungen erfolgen auf Basis von modular aufgebauten, in Java implementierten Prüfoperationen.

Wissenschaftlicher Beitrag

Der Hauptbeitrag dieses Kapitels aus wissenschaftlicher Sicht ist die Identifizierung von automatisch aufdeckbaren Abweichungen einer Musterimplementierung von ihrer Musterspezifikation. Ein Proof of Concept in Form einer prototypischen Implementierung aller Prüfoperationen ist bisher nicht erfolgt.

Kapitelstruktur

Im Folgenden gebe ich zunächst einen Überblick zu meinem Validierungsverfahren und stelle anschließend die automatisch überprüfbaren Merkmale einzeln vor. Abschließend zeige ich die Grenzen des Ansatzes auf und fasse zusammen.

6.1 Überblick

überlappende Änderungen

Entwurfsmusteranwendungen erfolgen häufig zusammen mit weiteren Entwurfsänderungen, insb. bei Anpassung eines Entwurfs an neue Anforderungen. Es werden schrittweise Klassen, Beziehungen und Operationen ergänzt oder geändert. Wenn dabei Entwurfsmuster zum Einsatz kommen, können mehrere Entwurfsmuster überlappend angewandt werden. Bei solchen Entwurfsänderungen können u.a. vorherige Musterimplementierungen bewusst oder unbewusst verändert werden. Versehentliche Änderungen und Flüchtigkeitsfehler können dabei Inkonsistenzen zu ursprünglich eingesetzten Entwurfsmustern erzeugen (siehe Abb. 6.1) und zum Verlust gewünschter Systemeigenschaften führen. Eine mit Hilfe eines Entwurfsmusters besonders flexibel ausgelegte Stelle im Entwurf (z.B. eine zur Laufzeit austauschbare Strategie) könnte ihre Flexibilität verlieren und somit zukünftige Softwareanpassungen erschweren (z.B. wenn eine bestimmte Strategie nicht über ihre Schnittstelle, sondern direkt verwendet wird).

potentieller Intentionsverlust

Zur Vermeidung ungewollter Abweichungen von ursprünglich eingesetzten Entwurfsmustern setze ich ein automatisches Konsistenzprüfungsverfahren ein, welches vorliegende Musterimplementierungen auf Vollständigkeit und Konsistenz zu zugehörigen Musterspezifikationen prüft. Dabei werden insb. folgende Eigenschaften geprüft:

Arten der Prüfung

- Implementieren aller Musterrollen und des spezifizierten Verhaltens
- Einhalten spezifizierter Eigenschaften (z.B. abstrakt / konkret, Kardinalität)
- Einhalten spezifizierter Beziehungen (z.B. Spezialisierung, Komposition)
- Einhalten von Kopplungsregeln

[1] Ausnahmen sind in Abschnitten 5.6 und 5.7 beschrieben

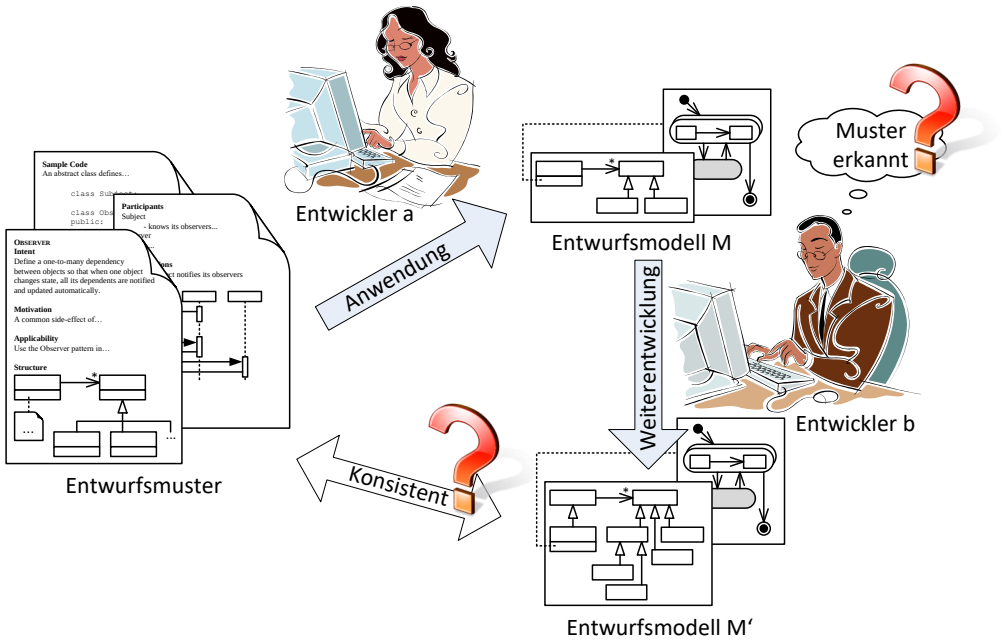


Abbildung 6.1: Notwendigkeit der Überprüfung von Musterimplementierungen

Triggern der Prüfungen

Für jede Art der Prüfung wird eine separate Prüfungsregel implementiert. Jede Prüfung wird aus den in Abschnitt 5.5 beschriebenen Übersetzern heraus aufgerufen (Schritte 1, 4, 6 in Abb. 5.19, S. 130). Erfolgt die Prüfung während einer Musteranwendung, wird die fehlende Eigenschaft nach Möglichkeit automatisch ergänzt. Erfolgt die Prüfung nach einer Musteranwendung (Prüfung einer vorhandenen, ggf. modifizierten Musterimplementierung)[2], wird der Entwickler auf die Abweichung von der Musterspezifikation hingewiesen. Dann kann er entscheiden, ob er die Musterimplementierung korrigiert oder bewusst eine zum angewandten Muster alternative Lösung sucht und die ursprüngliche Musteranwendung rückgängig macht.

Aufzeigen von Abweichungen

6.2 Abweichungen von Musterspezifikationen aufdecken

Die eingeführte Musterspezifikationssprache (Kap. 3) bietet verschiedene Möglichkeiten zur Beschreibung einer Entwurfslösung zu einem Entwurfsmuster. Neben der Klassenstruktur wird Verhalten, Abhängigkeitsregeln und zu erledigende Aufgaben spezifiziert. Bei der Anwendung eines so spezifizierten Musters wird neben der Musterimplementierung auch ein Modell der Anwendungsstelle generiert (Kap. 5). Eine Anwendungsstelle wird insb. durch die Zuordnung aller spezifizierten Musterrollen zu den zugehörigen Teilen im Softwareentwurfsmodell charakterisiert (Kap. 4). Das detaillierte Modell der Anwendungsstelle wird dazu genutzt, die Konformität einer Musterimplementierung zu ihrer Musterspezifikation automatisch zu prüfen.

Das Modell der Anwendungsstelle besteht im Wesentlichen aus zwei Teilen: dem Korrespondenzmodell (den Rollenzuordnungen) und dem Anwendungsmodell (sie-

[2]Dieser Fall ist in Abb. 5.19 nicht explizit dargestellt.

he Abb. 4.3, S. 83). Während die Rollenzuordnungen die Anwendungsstelle eines Musters im Softwareentwurfsmodell markieren, repräsentiert das Anwendungsmodell die für diese Stelle vorgesehene Implementierungsvariante der Entwurfslösung (insb. eine bestimmte Anzahl von Set-Fragment-Instanzen).

Vollständigkeit der Korrespondenzen

Mit Hilfe der Rollenzuordnungen kann geprüft werden, ob ein Muster vollständig angewandt wurde (ob allen spezifizierten Musterrollen Elemente des Entwurfsmodells zugeordnet wurden). Das Anwendungsmodell enthält insb. eine 1-zu-1-Repräsentation der für eine Anwendungsstelle vorgesehenen Klassenstruktur samt aller zugehörigen Beziehungen. Mit Hilfe des Anwendungsmodells kann geprüft werden, ob alle spezifizierten Beziehungen und Eigenschaften bei einer Musterimplementierung vorhanden und alle spezifizierten Bedingungen eingehalten sind.

Erledigungsstatus

Außerdem enthält das Anwendungsmodell den Abarbeitungsstatus von Entwurfsaufgaben, sodass bisher unerledigte Aufgaben identifiziert werden können.

implizite & explizite Prüfungen

Die Validierung einer Anwendungsstelle bzw. einer Musterimplementierung erfolgt bei meinem Ansatz entweder implizit während einer Musteranwendung (während der automatischen Übersetzung eines Anwendungsmodells in den Entwurf) oder explizit durch einen Entwickler angestoßen (nach einer vorherigen Musteranwendung oder Rollenzuordnung). In beiden Fällen werden dieselben Prüfungen vorgenommen.

Aufbau einer Prüfung

Die Prüfungen sind in den Übersetzern verankert, welche ein Anwendungsmodell in das Softwareentwurfsmodell übersetzen. In der Abb. 6.2 ist der Kontrollfluss aller Übersetzer schematisch dargestellt (das ist eine detailliertere Darstellung des Verhaltens aus Abb. 5.19 S. 130). Die Validierung einer Anwendungsstelle erfolgt hier in den Schritten 1 und 4. Soll ein Muster nicht angewandt, sondern nur validiert werden, so wird der in dieser Abbildung dargestellte Algorithmus per *checkOnly*-Parameter auf die Schritte 1, 4, 6 und 7 beschränkt. Eine konkrete Ausprägung eines Übersetzers ist z.B. in der Abb. C.18 (S. 328) dargestellt. Darin wird bei der Validierung einer Anwendungsstelle insbesondere die Konsistenz einer spezifizierten mit der implementierten Methodensignatur geprüft.

Scheduling

Die Reihenfolge der Prüfungen wird analog zur Reihenfolge der Übersetzeraufrufe bestimmt, indem das Baum-artige Modell einer Musterspezifikation mit Hilfe einer Warteschlange top-down traversiert (siehe Abschn. 5.5.3, S. 127 ff.) und ihre Entsprechung im Entwurfsmodell geprüft wird.

Präsentation der Prüfergebnisse

Anwendungsstellen können einzeln oder alle auf einmal validiert werden. Alle während der Validierung entdeckten Abweichungen von einer Musterspezifikation werden zwecks Korrekturmöglichkeit gruppiert nach Muster, Anwendungsstelle und Art der Verletzung aufgelistet.

Welche in einer Musterspezifikation definierten Eigenschaften einer Musterimplementierung auf welche Weise geprüft werden, erläutere ich im Folgenden.

6.2.1 Vollständigkeit der Rollenzuordnung

1-zu-n-Korrespondenzen

Das Modell einer Anwendungsstelle enthält zu jeder Rolle der Musterspezifikation einen Korrespondenzknoten – ein RoleBinding-Objekt. Ein Role Binding speichert alle Zuordnungen von Entwurfsmodellelementen zu einer Musterrolle (siehe Bsp. in Abb. 4.4, S. 85). Einer Musterrolle können beliebige viele Entwurfselemente zugeordnet werden. Für eine vollständige Anwendung eines Musters muss jeder

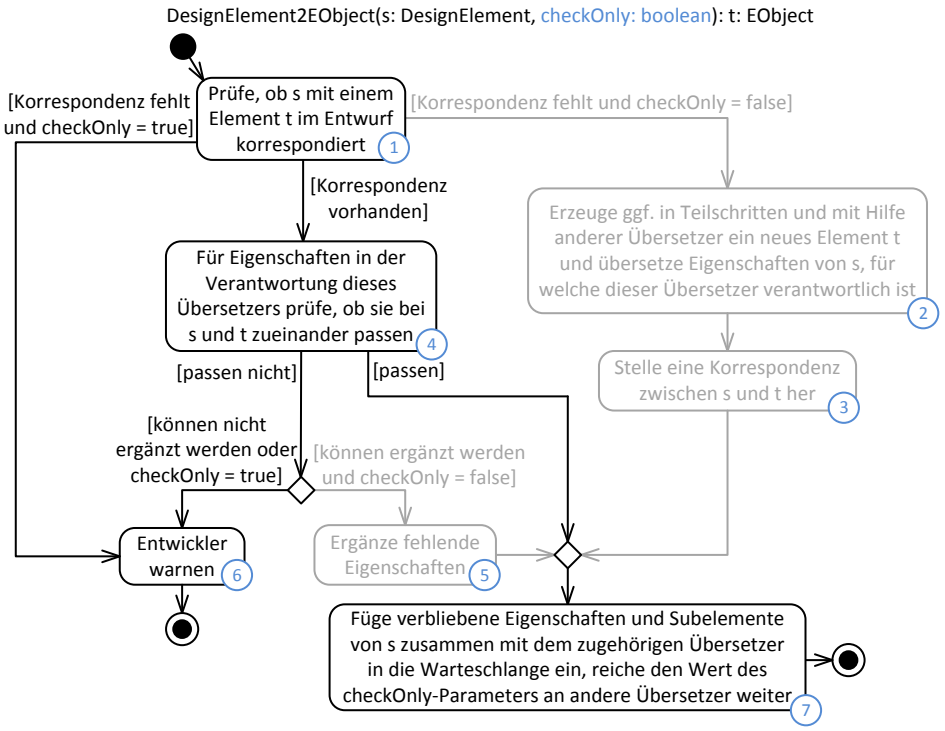


Abbildung 6.2: Validierung im Verhalten eines Elementübersetzers

Rolle der Musterspezifikation mindestens ein Element im Entwurfsmodell zugeordnet und mit der Musterrolle verknüpft sein.

frühe Prüfbarkeit

Role Bindings werden angelegt, sobald ein Entwickler eine Musterspezifikation zur Anwendung in einem Entwurfsmodell auswählt und damit eine Anwendungsstelle erstellt (Schritt 3 in Abb. 5.10, S. 115, Bsp. in Abb. C.3, S. 306). Ab diesem Zeitpunkt lässt sich die Anwendungsstelle bereits auf Vollständigkeit prüfen.

Set-Fragment-Instanzen separat prüfbar

Werden zu einem Set Fragment mehrere Ausprägungen in einer Musterimplementierung vorgesehen, so werden für jede der Ausprägungen separate Role Bindings erstellt (Bsp. in Abb. C.6, S. 311) und einzeln geprüft.

Vollständige Zuordnung & Typkonformität

Bei der Validierung einer Anwendungsstelle werden sämtliche Rollen der zugehörigen Musterspezifikation durchlaufen (d.h. alle DAL-Knoten einer Musterspezifikation, siehe Abschn. 3.4) und die zugehörigen Rollenzuordnungen auf Verknüpfung mit mindestens einem Entwurfselement geprüft. Jedes verknüpfte Entwurfselement wird außerdem auf Typkonformität zur Rolle geprüft, z.B. wird geprüft ob ein Typ der Musterspezifikation (Typ Type) nur Klassen oder Datentypen (Typen EClass und EDataType) und nicht anderen Elementen im Entwurfsmodell zugeordnet ist. Ob der Typ des verknüpften Elements passt, entscheidet der zugehörige DAL-Elementübersetzer.

6.2.2 Einhalten spezifizierter Eigenschaften einer Rolle

Sämtliche Musterrollen werden durch Elemente der DAL beschrieben (Abschn. 3.4). Rollen können in Musterspezifikationen mit einigen Eigenschaften versehen werden, z.B. um auszudrücken, dass eine bestimmte Klasse oder Methode abstrakt oder konkret sein soll. Ist eine Eigenschaft spezifiziert, ist sie in der Musterimple-

¬spezifiziert ⇒ ¬relevant

mentierung gefordert. Fehlt sie, interpretiere ich sie als für eine Musterimplementierung irrelevant. Bei der Validierung einer Anwendungsstelle werden folgende Eigenschaften bei jedem einer Rolle zugeordneten Entwurfselement geprüft, falls die jeweilige Eigenschaft für eine Rolle spezifiziert ist.

abstrakt bzw. konkret Ist ein Typ oder eine Operation als abstrakt bzw. als konkret spezifiziert (es gibt auch den Fall *beliebig*, siehe Abb. A.15, S. 267), so wird sichergestellt, dass die zugeordnete Klasse oder Methode im Entwurfsmodell ebenfalls abstrakt bzw. konkret ist (bei *beliebig* entfällt die Prüfung).

Attributtyp Es wird sichergestellt, dass der Typ eines Attributs in der Musterspezifikation zum Attributtypen im Entwurfsmodell passt. D.h., dass z.B. bei boolean in der Spezifikation EBoolean im Entwurfsmodell erwartet wird.

Arrays Ist ein Attribut in der Musterspezifikation als Array definiert (Multiplizität *), so wird dieselbe Multiplizität bei dem zugeordneten Attribut im Entwurfsmodell erwartet.

Kardinalität von Referenzen Ist bei einer Referenz in der Musterspezifikation die Kardinalität 0..1 oder * angegeben, so wird sichergestellt, dass bei der zugehörigen Assoziation im Entwurfsmodell (bei bidirektionalen Assoziationen am zur Referenz gehörenden Assoziationsende) eine dazu passende Kardinalität modelliert ist.

Erledigungsstatus von Aufgaben Enthält eine Musterspezifikation eine Entwurfsaufgabenbeschreibung zu einer Rolle, so wird sichergestellt, dass die zugehörige Aufgabe an der Anwendungsstelle den Status *erledigt* hat.

6.2.3 Konsistenz zu spezifizierten Rollenbeziehungen

Neben den spezifizierten Eigenschaften einzelner Rollen werden bei der Validierung einer Anwendungsstelle auch Beziehungen zwischen Entwurfselementen entsprechend der spezifizierten Beziehungen zwischen zugehörigen Rollen geprüft. Dabei wird u.a. die Transitivität der Beziehungen berücksichtigt. Folgendes wird geprüft.

Spezialisierung bei Typen Bei einer Beziehung in einer Musterspezifikation der Form *ein Typ B ist eine Spezialisierung von einem Typ A*, wird sichergestellt, dass die zugehörigen Klassen im Entwurfsmodell in der gleichen Relation zueinander stehen. D.h., eine Klasse Y in der Rolle B ist ein Untertyp der Klasse X in der Rolle A. Dabei ist aufgrund der Transitivität der Vererbungsbeziehung irrelevant, ob die Klasse Y direkt oder indirekt von X erbt (vgl. Abb. 3.18, S. 61).

Methodensignatur

Spezialisierung bei Operationen Bei einer Beziehung in einer Musterspezifikation der Form *eine Operation b ist eine Spezialisierung einer Operation a*, wird sichergestellt, dass die zugehörigen Methoden im Entwurfsmodell in der gleichen Relation zueinander stehen. Das bedeutet, dass eine Methode

y in der Rolle b dieselbe oder eine speziellere Methodensignatur[3] hat wie die Methode x in der Rolle a. Zusätzlich gilt: Liegt die Methode y in einer Klasse Y und die Methode x in einer Klasse X, so ist Y eine Unterklasse von X. Auch hier gilt aufgrund der Transitivität der Spezialisierungsbeziehung, dass Y auch eine indirekte Unterklasse von X sein darf (siehe Bsp. unten in Abb. A.19, S. 268).

Komposition Ist in einer Musterspezifikation angegeben, dass eine Operation, ein Attribut oder eine Referenz a zu einem Typen A gehört, so muss auch die zugehörige Klasse X in der Rolle A eine Methode, ein Attribut bzw. eine Referenz x in der Rolle a bereitstellen. Das bedeutet, dass x in der Klasse X verfügbar sein muss, d.h. in X oder in einer ihrer Oberklassen deklariert sein muss (Bsp. in Abb. 3.15, S. 60). Auch hier gilt also, dass eine direkte Kompositionsbeziehung in der Spezifikation auf eine direkte oder indirekte „Bereitstellen-Beziehung“ im Entwurfsmodell abgebildet wird.

Verfügbarkeit eines Elements

Ist in einer Musterspezifikation ein Parameter p zu einer Operation o definiert, so muss die zur Rolle o gehörende Methode m im Entwurfsmodell einen Parameter in der Rolle p besitzen. Bei mehreren Parametern pro Operation o muss die Methode m die Reihenfolge der Parameter einhalten. m darf allerdings zusätzliche Parameter besitzen, zu denen es keine Entsprechung in o gibt (Bsp. in Abb. 3.15 Mitte, S. 60).

Teilmenge aller Parameter

Ein Subsystem wird einer Menge von Paketen und Klassen im Entwurfsmodell zugeordnet (siehe Abb. 3.25, S. 65). Besteht in einer Musterspezifikation eine Kompositionsbeziehung zwischen Subsystemen und Typen (ein Subsystem enthält ein anderes Subsystem oder Typen), so müssen die diesen Subsystemen zugeordneten Pakete und Klassen im Entwurfsmodell analog zur Spezifikation direkt oder indirekt ineinander enthalten sein. Ist ein Subsystem keinem Paket, sondern nur einer Auswahl von Klassen im Entwurfsmodell zugeordnet, so entfällt diese Prüfung.

Paketstruktur, enthaltene Klassen

Rückgabetyp Ist als Rückgabetyp einer Operation o ein Typ (oder primitiver Datentyp) T spezifiziert, so wird sichergestellt, dass die zur Rolle o gehörende Methode m im Entwurfsmodell die zur Rolle T gehörende Klasse (bzw. den dazu gehörigen primitiven Datentyp) als Rückgabetyp besitzt.

Parametertyp Wurde der Typ eines Parameters in einer Musterspezifikation angegeben, so wird der zugehörige Parametertyp im Entwurfsmodell analog zum Rückgabetyp einer Operation geprüft.

Typ einer Referenz Verweist eine Referenz r laut Musterspezifikation auf einen Typen T, so wird sichergestellt, dass die zur Rolle r gehörende Assoziation (bei bidirektionalen Assoziationen das zugehörige Assoziationsende) auf die zur Rolle T gehörende Klasse verweist (Abb. 3.16, S. 60).

[3] Zwei Methoden haben dieselbe Methodensignatur, wenn sie identische Namen, identische Parametertypen in gleicher Reihenfolge sowie identische Rückgabetypen haben. Eine Methodensignatur m_2 ist spezieller als eine andere m_1, wenn der Rückgabetyp in m_2 ein Untertyp des Rückgabetypen in m_1 ist und die Signaturen ansonsten identisch sind.

6.2.4 Konsistenz zum spezifizierten Verhalten

Im Gegensatz zu laut einer Musterspezifikation geforderten strukturellen Eigenschaften einer Musterimplementierung lässt sich gefordertes Verhalten nur schwer im Entwurfsmodell überprüfen. Im Folgenden beschreibe ich, welcher Teil des spezifizierten Verhaltens sich automatisch prüfen lässt.

Verfügbarkeit eines Verhaltensmodells Enthält eine Musterspezifikation mindestens eine Aktion zu einer Operation *o*, so wird bei der Validierung des Entwurfsmodells sichergestellt, dass es zu der zu *o* gehörenden Methode im Entwurfsmodell auch ein Verhaltensmodell gibt. Bei meinem Ansatz wird das Verhalten in Form von Story-Diagrammen modelliert. Darum wird bei dieser Prüfung zusätzlich die Konformität der Methodensignatur zur Signatur des Story-Diagramms geprüft.

Signaturkonformität

Reihenfolge von Aktionen Einer Aktion in einer Musterspezifikation werden bei einer automatisierten Musteranwendung ein oder mehrere das spezifizierte Verhalten modellierende Aktivitätenknoten eines Story-Diagramms zugeordnet. Sind mehrere aufeinander folgende Aktionen spezifiziert und sind diesen Aktionen Aktivitätenknoten zugeordnet, so wird sichergestellt, dass die Ausführungsreihenfolge der Aktivitätenknoten der spezifizierten Reihenfolge der Aktionen entspricht (siehe Abb. 3.21, S. 62).

Fehlendes oder inkorrektes Verhalten In meiner Musterspezifikationssprache habe ich mehrere Arten von Aktionen definiert, mit Hilfe derer man Verhalten in einem Muster beschreiben kann. Dazu gehören z.B. Methodenaufrufe, Klasseninstanziierungen, Lese- und Schreibzugriffe. Das damit beschriebene Verhalten lässt sich im Entwurfsmodell (in Story-Diagrammen) zum Teil automatisch prüfen. Unter der Annahme, dass alle Aktionen zugehörigen Aktivitätenknoten zugeordnet sind (das passiert nur bei automatischer Musteranwendung) und dass ein Story-Diagramm das durch Aktionen beschriebene Verhalten nicht durch Aufruf anderer Story-Diagramme modelliert, lässt sich für einige der Aktionen prüfen, ob das spezifizierte Verhalten in einem Story-Diagramm vorkommt.

Ausschließen fehlender Aufrufe

Bei call-Aktionen z.B. kann geprüft werden, ob die der Aktion zugeordneten Aktivitätenknoten wenigstens einen Methodenaufruf enthalten (dann muss eine MethodCallExpression in einem StatementNode existieren, siehe Abb. A.22, S. 273) und ob dieser Aufruf auf der richtigen Methode erfolgt. Ob der im Story-Diagramm modellierte Kontrollfluss immer zum Methodenaufruf führt, kann jedoch (insb. nach manuellen Anpassungen des generierten Story-Diagramms) nicht ohne Kontrollflussanalyse beantwortet werden.

Aufrufargumente

Neben der aufgerufenen Methode kann überprüft werden, ob beim Methodenaufruf Parameterwerte übergeben werden. Damit kann z.B. geprüft werden, ob sämtliche erhaltenen Parameterwerte weitergegeben werden wie es bei redirect- und delegate-Aktionen gefordert ist (siehe Abschn. 3.4.2).

Klasseninstanziierung

Bei create- und produce-Aktionen kann sichergestellt werden, dass eine Klasse instanziiert wird und dass die instanziierte Klasse zu dem laut Spezifikation zu instanziierenden Typen passt (sie die richtige Rolle hat). Bei

produce-Aktionen kann zusätzlich sichergestellt werden, dass das erzeugte Objekt direkt als Ergebnis eines Methodenaufrufs zurückgegeben wird.

Andere Eigenschaften von Aktionen und Verhalten wie das Lesen oder Schreiben von bestimmten Variablen (z.B. eines Attributs oder einer Referenz) lässt sich aufgrund der unzähligen Möglichkeiten, das Verhalten in Story-Diagrammen zu modellieren, nur schwer automatisch prüfen und wird hier daher nicht näher betrachtet.

6.2.5 Einhalten von Kopplungsrestriktionen

In Musterspezifikationen lassen sich u.a. gewollte und ungewollte Abhängigkeiten zwischen Entwurfsteilen beschreiben (siehe Abschn. 3.5.3, S. 72 ff.). Die Spezifikation dieser Abhängigkeiten dient zwei Zwecken: Zum einen sollen diese Abhängigkeiten dem Musteranwender bewusst sein und dazu dokumentiert werden. Zum anderen sollen diese bei allen Musteranwendungsstellen automatisch geprüft werden können.

Insgesamt bietet die Musterspezifikationssprache acht verschiedene Arten von Abhängigkeiten (siehe Abb. 3.39, S. 72). Sie können zwischen diversen Musterrollen wie Subsystemen, Typen und Operationen beschrieben werden (siehe Tab. A.6, S. 264 und A.8, S. 265). Zur Prüfung, ob die Kopplungsrestriktionen einer Musterspezifikation in der Implementierung des spezifizierten Musters in einem Entwurfsmodell eingehalten wurden, können statische Analyseverfahren eingesetzt werden. Damit können u.a. Abhängigkeiten zwischen Paketen, Klassen, Methoden und Variablen identifiziert und die Art der jeweiligen Abhängigkeit bestimmt werden. Es kann z.B. zwischen Aufrufen einer Methode durch eine andere (*call*), dem Instanziieren einer Klasse durch eine andere (*instantiate*), Vererbungsbeziehungen zwischen Klassen (*specialize*), dem Verwenden einer Klasse als Typ eines Parameters (*point to*) sowie Lese- und Schreibzugriffen auf Variablen (*read* / *write*) unterschieden werden. So lässt sich für alle durch Kopplungsrestriktionen beschreibbaren Abhängigkeiten (siehe Abschn. 3.5.3, S. 72 ff.) im Entwurfsmodell prüfen, ob die jeweilige Abhängigkeit vorliegt oder nicht und damit auch, ob eine Musterimplementierung eine Kopplungsrestriktion einhält oder nicht.

statische Analyse

Statische Analyseverfahren können entweder direkt auf dem Entwurfsmodell (bei meinem Ansatz auf Ecore- und Story-Diagramm-Modellen) angewandt werden wie es z.B. Reclipse[4] [vDMT10a, vDMT10b] tut oder auf dem zugehörigen Quell- oder Bytecode wie es diverse Analysewerkzeuge wie Structure101[5], SonarQube[6], Teamscale[7], Seerene[8], und X-Ray[9] tun. Da bei meinem Ansatz noch kein Quellcode des entwickelten Softwaresystems vorliegt, muss eine statische Analyse entweder auf Modell-Ebene erfolgen oder es muss eine Codegenerierung mit

Modell- vs. Codeanalyse

[4] Reclipse führt seine Analysen auf dem abstrakten Syntaxgraphen (ASG) eines Modells (z.B. eines UML- oder Matlab-Simulink-Modells [ST08]) oder des Quellcodes einer Programmiersprache (z.B. Java) durch.
[5] http://structure101.com/
[6] http://www.sonarqube.org/
[7] https://www.cqse.eu/de/produkte/teamscale/landing/
[8] https://www.seerene.com/de/produkt/
[9] http://xray.inf.usi.ch/xray.php

Traceability-Links zum Entwurfsmodell ergänzt werden, damit die Ergebnisse einer statischen Codeanalyse auf das Entwurfsmodell übertragen werden können.

Machbarkeit

Im Rahmen meiner Arbeit habe ich kein statisches Analyseverfahren zur Prüfung der spezifizierten Kopplungsrestriktionen entwickelt oder angewandt. Aufgrund existierender Analyseverfahren und Codegeneratoren erscheint so eine Umsetzung jedoch machbar.

6.3 Einschränkungen

keine Konsistenzrelationen

Die größte Einschränkung meines Ansatzes zur Validierung von Musterimplementierungen ist die Notwendigkeit, jede Validierung explizit in Java zu implementieren, also die Validierung zu operationalisieren. Stattdessen wäre eine deklarative Beschreibung der zu prüfenden Eigenschaften in Form von Konsistenzrelationen und ihre automatische Prüfung wünschenswert.

keine Garantien

Die hier beschriebenen Prüfungen decken potenzielle Probleme auf, geben jedoch keine Garantien für die Abwesenheit von Inkonsistenzen. Änderungen an Musterspezifikationen nach einer Musteranwendung werden bei den Konsistenzprüfungen bisher nicht berücksichtigt.

sehr eingeschränkte Verhaltensprüfung

Die Prüfung der generierten Verhaltensmodelle ist nur eingeschränkt möglich. Zum Beispiel kann zwar geprüft werden, ob ein Story-Diagramm einen Methodenaufruf enthält, jedoch kann im Allgemeinen nicht entschieden werden, ob es in jedem Fall zu dem Methodenaufruf kommt (ob alle Bedingungen auf dem Kontrollflusspfad zum Methodenaufruf bei jeder Story-Diagramm-Ausführung erfüllt sind). Ob auf ein Attribut oder eine Referenz lesend oder schreibend zugegriffen wird, kann bei meinem Ansatz bisher nicht geprüft werden.

6.4 Zusammenfassung und Ausblick

Validierung von Anwendungsstellen

In diesem Kapitel zeige ich auf wie die Musterspezifikationssprache, die Modellierung von Anwendungsstellen und das semiautomatische Verfahren zur Musteranwendung (Kap. 3 – 5) gemeinsam dazu genutzt werden können, Musteranwendungsstellen insb. nach manuellen Entwurfsänderungen automatisch auf Korrektheit zu prüfen. Ich nenne automatisch prüfbare Merkmale und stelle eine Möglichkeit zur Implementierung solcher Prüfungen vor.

Ausblick

Als Nächstes könnte eine prototypische Implementierung solcher Prüfungen als Proof of Concept dienen. Alternativ dazu könnte untersucht werden, ob deklarativ beschriebene, automatisch prüfbare Konsistenzbedingungen (vgl. OCL-Constraints [OMG14]) oder Konsistenzrelationen zwischen unterschiedlich detaillierten Modellen [Anj14, Rie14, LAS17] die hier beschriebenen Prüfungen ersetzen und die Implementierung von Prüfoperationen unnötig machen könnten.

7 Prototypische Implementierung des Ansatzes

Proof of Concept

Als Machbarkeitsstudie (Proof of Concept) habe ich den in dieser Arbeit vorgestellten Ansatz prototypisch implementiert. Die Implementierung ist modular aufgebaut und erweitert existierende Werkzeuge zur Modellierung von Klassen- und Story-Diagrammen. Alle entstandenen Werkzeuge sind in einer Werkzeugsammlung, der *Pattern-Oriented Software Engineering Tool Suite* (kurz: POSE) zusammengefasst. Ein Teil davon ist in einem gleichnamigen studentischen Projekt entstanden (siehe Vorarbeiten, S. 207 [BBF+11]) und wurde anschließend von mir grundlegend überarbeitet und erweitert.

POSE

Funktionsumfang

Die wichtigsten Anwendungsfälle mit implementierter Werkzeugunterstützung sind in unten stehender Box zusammengefasst (Links zum Quellcode in Anhang E).

Implementierte Funktionalität

- Spezifikation objektorientierter Entwurfsmuster (Entwurfsstruktur und Interaktionen, inklusive zahlreicher Implementierungsvarianten)
- Semiautomatische Anwendung von Entwurfsmustern in Ecore- und Story-Diagramm-Modellen
- Visualisierung von Musteranwendungsstellen in Ecore-Klassendiagrammen und in der Musteranwendungssicht (Rollenzuordnungen)
- Verknüpfung der Musterrollen mit den sie implementierenden Entwurfsteilen, d.h. Modellierung von Musteranwendungsstellen
- Automatische Generierung eines Korrespondenz- und Anwendungsmodells bei Musteranwendung (in Ecore- und Story-Diagramm-Modellen)
- Nachträgliche Modellierung von Musteranwendungsstellen (manuell, beschränkt auf Ecore-Modelle)

Einschränkungen

Zu den zwei Haupteinschränkungen des Prototypen gehören (a) die fehlende Implementierung der in Kapitel 6 vorgestellten Konsistenzprüfung zwischen einer Musteranwendungsstelle und der zugehörigen Musterspezifikation und (b) die unvollständige Implementierung der Visualisierungskonzepte für Musteranwendungsstellen aus Abschnitt 4.4.

Kapitelstruktur

Im Folgenden stelle ich in Abschnitt 7.1 zunächst die Architektur des Prototypen vor. Anschließend erläutere ich in Abschnitt 7.2 seine Verwendung. Dazu zeige ich exemplarisch wie ein Entwurfsmuster unter Verwendung des Prototypen in einem partiel vorliegenden Entwurf angewandt werden kann und stelle die Benutzungsschnittstelle vor. Abschließend gehe ich in Abschnitt 7.3 auf die Herausforderungen, Ergebnisse und Einschränkungen des Prototypen ein.

7.1 Architektur

Eclipse als Basis

Die prototypische Implementierung der Pattern-Oriented Software Engineering Tool Suite besteht aus diversen Plug-ins für die Entwicklungsumgebung Eclipse[1] [GB03, CR06]. Eclipse stellt eine modulare, erweiterbare Plattform für die Entwicklung von Desktop-Anwendungen[2], insb. von Entwicklungsumgebungen bereit.

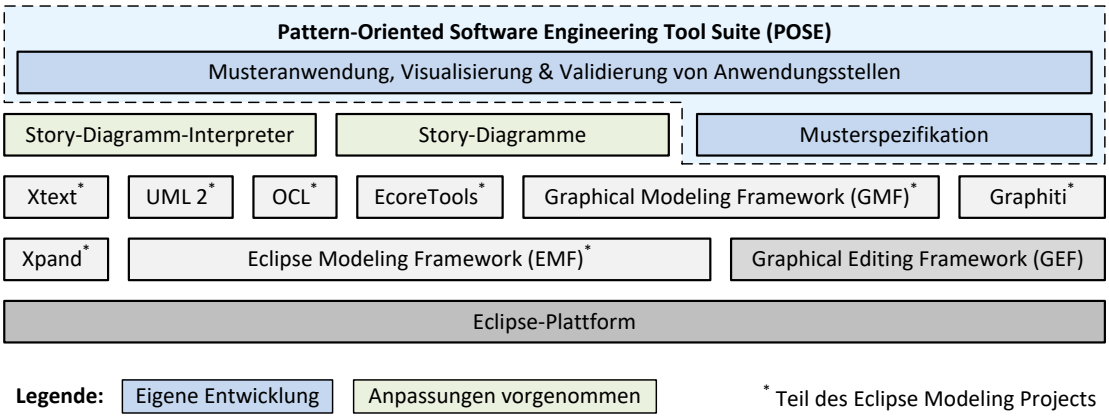


Abbildung 7.1: Architekturüberblick

eingesetzte Frameworks

Die Hauptbausteine des Prototypen sind in der Abb. 7.1 aufgeführt. Die POSE-Werkzeuge (blau) setzen Story-Diagramm-Werkzeuge (grün) und diverse Frameworks (grau) ein, insb. aus dem Eclipse Modeling Project[3] [Gro09]. Dazu gehört vor allem das Eclipse Modeling Framework (EMF)[4] [SBPM08], welches das Ecore-Meta-Modell (eine partielle Implementierung der Meta-Modellierungssprache MOF [OMG11c]) bereitstellt. Die EcoreTools[5] enthalten u.a. einen grafischen Ecore-Klassendiagramm-Editor. Das Graphical Editing Framework (GEF)[6] stellt die Grundfunktionalität zur Entwicklung grafischer Editoren (unabhängig von EMF) basierend auf dem Model-View-Controller-Paradigma [BMR+96, Fow02] bereit. Das Graphical Modeling Framework (GMF)[7] bietet Werkzeuge zur modellgetriebenen Entwicklung von grafischen Editoren basierend auf GEF und EMF. Graphiti[8] ist eine vereinfachende Erweiterung von GEF und eine Alternative zum generativen Ansatz von GMF. Xpand[9] ist ein Template-basiertes Code-Generator-Framework für EMF-Modelle. Xtext[10] baut auf Xpand und ist ein umfangreiches Framework zur Entwicklung von textuellen Domänen-spezifischen Sprachen (DSLs) und zugehöriger Editoren. Die EcoreTools basieren auf EMF und GMF, genauso wie die hier verwendeten Story-Diagramm-Modelle und -Editoren[11].

[1] Eclipse: http://www.eclipse.org
[2] Rich-Client-Platform-Anwendungen: https://wiki.eclipse.org/Rich_Client_Platform
[3] Eclipse Modeling Project: https://eclipse.org/modeling/
[4] EMF: http://www.eclipse.org/modeling/emf/
[5] EcoreTools: http://www.eclipse.org/ecoretools/
[6] GEF: http://www.eclipse.org/gef/
[7] GMF: http://www.eclipse.org/gmf-tooling/
[8] Graphiti: http://www.eclipse.org/graphiti/
[9] Xpand: http://www.eclipse.org/modeling/m2t/?project=xpand
[10] Xtext: http://www.eclipse.org/Xtext/
[11] Story-Diagramm-Tools: http://storydriven.org/, https://github.com/ReEng/sdm-commons

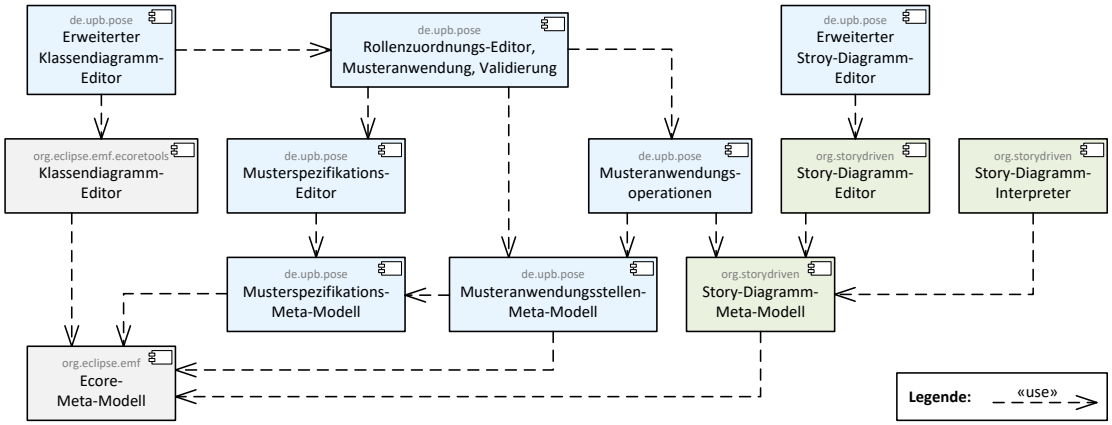


Abbildung 7.2: Komponentenarchitektur des Prototyps

Neuentwicklungen & Erweiterungen

Die POSE Tool Suite erweitert die Funktionalität des Klassendiagramm-Editors der EcoreTools und des Story-Diagramm-Editors um die Visualisierung und nachträgliche Dokumentation von Musteranwendungsstellen sowie um die Funktionalität zur semiautomatischen Anwendung von Entwurfsmustern. Zur Spezifikation von Entwurfsmustern und zur Visualisierung von Musteranwendungsstellen wurden diverse grafische Editoren entwickelt. Diese basieren größtenteils auf dem Graphiti-Framework, während die erweiterten Versionen der existierenden Editoren für Klassen- und Story-Diagramme weiterhin auf GMF basieren.

wichtigste Komponenten

Die Hauptbestandteile der POSE Tool Suite sind im Komponentendiagramm in der Abb. 7.2 dargestellt. Die Pfeile beschreiben die Abhängigkeiten der Komponenten untereinander. Alle Komponenten der POSE Tool Suite sind hellblau hervorgehoben (Paket de.upb.pose). Diese wurden komplett neu entwickelt. Die Werkzeuge zur Modellierung von Story-Diagrammen (hellgrün hervorgehoben) wurden für den Prototypen geringfügig erweitert und angepasst.

Vorgehen und Aufbau

Alle Meta-Modelle (Begriffsdefinition in Abschn. 2.3.2, S. 30) wurden in EMF bzw. Ecore modelliert. Anschließend wurden die Meta-Modell-Klassen und je ein Editor für derartige Modelle mit einer Baum-Darstellung und simplen Editier-Operationen (z.B. Element erstellen oder löschen, Attribute ändern) generiert. Zusätzlich wurde ein grafischer Musterspezifikationseditor basierend auf dem Graphiti-Framework entwickelt. Alle Übersetzungstransformationen (siehe Kap. 5.5), die bei der Musteranwendung zum Einsatz kommen (Musteranwendungsoperationen), wurden in einem eigenständigen Eclipse-Plug-in entwickelt und komplett in Java programmiert. Diese Transformationen enthalten u.a. einige Konsistenzprüfungsoperationen und werden bei der Rollenzuordnung, Musteranwendung und Validierung aufgerufen. Die Musteranwendungssicht (technisch ein Editor, der Rollenzuordnungs-Editor), welche die Rollenzuordnung ermöglicht und visualisiert, nutzt die Notation aus dem Musterspezifikationseditor und erweitert diese um einige Markierungen und Hervorhebungen (vgl. Abb. 7.3, S. 159 und Abb. 7.7, S. 162). Der Klassendiagramm-Editor der EcoreTools (basiert auf GMF) wurde um einige Markierungen und Zusatzoperationen zur Anwendung von Entwurfsmustern erweitert (z.B. das Pattern-Werkzeug, UML-Kollaborationen und Markierungen an Klassen, siehe Abb. 7.7, S. 162). Aus dem erweiterten Klassendiagramm-Editor

wird zur Anwendung eines Musters und der dafür nötigen Rollenzuordnung der Rollenzuordnungseditor aufgerufen. Auch die Validierung von Musteranwendungsstellen wird aus dem Klassendiagramm-Editor angestoßen.

Anpassungen an Story-Diagramm-Werkzeugen

Die Implementierung der Story-Diagramm-Werkzeuge wurde vervollständigt und stellenweise angepasst. Zum Beispiel wurde die Verknüpfung von Ecore-Modellen mit Story-Diagramm-Modellen für den Anwendungsfall, dass Methodenrümpfe durch Story-Diagramme modelliert werden, ergänzt. Die bisher fehlende Unterstützung für Sprachkonstrukte wie z.B. bestimmte textuelle Ausdrücke in Story-Diagrammen, Übergabe von Argumenten und Rückgabewerten beim Aufruf von Story-Diagrammen sowie negative Anwendungsbedingungen (siehe Abschn. 2.4.1) wurde implementiert. Dazu wurden das Story-Diagramm-Meta-Modell und der grafische Editor erweitert sowie der Interpreter entsprechend angepasst.

Größe der Implementierung

Die Abb. 7.2 stellt die Architektur des Prototypen stark vereinfacht dar. Insgesamt besteht die POSE Tool Suite aus 12 Eclipse-Plug-ins und 5 weiteren Eclipse-Projekten (u.a. Update Site, Tests, etc.). Die Implementierung umfasst 95.222 nicht leere[12] Zeilen Java-Code, davon sind 54.911 bzw. 57,66 % aus Meta-Modellen generiert (modellgetrieben entwickelt). Dazu kommen Tests im Umfang von 3.733 Zeilen Java-Code. Die Story-Diagramm-Werkzeuge haben einen noch größeren Umfang von 249.837 Zeilen Java-Code (ohne Tests), wovon 79,6 % generiert ist. Details zum Umfang der Prototyp-Implementierung können dem Anhang entnommen werden (siehe Anhang E und Tab. E.2, S. 364).

7.2 Benutzungsschnittstelle am Beispiel einer Observer-Musteranwendung

Im Folgenden stelle ich am Beispiel des Observer-Musters [GHJV95] und meiner Spezifikation dieses Musters (Abb. B.20, S. 297) die Benutzungsstelle des entwickelten Prototypen vor und erläutere wie ein Muster mit Hilfe der implementierten Werkzeugunterstützung zur Anwendung gebracht wird.

Spezifikation eines Musters

Zunächst wird die Musterspezifikation in einem dafür entwickelten Editor modelliert und in einem Musterspezifikationskatalog gespeichert. Der Editor unterstützt alle in Kapitel 3 eingeführten Sprachkonstrukte. Die damit erstellte Spezifikation des Observer-Musters ist zusammen mit dem Editor in der Abb. 7.3 dargestellt. Die dargestellte Spezifikation entspricht der in Abb. B.20 (S. 297) und wird im Folgenden für eine Musteranwendung verwendet.

Anwendungsszenario

Als Anwendungsbeispiel habe ich eine Stelle im Entwurf eines grafischen Petrinetz-Editors gewählt. Es liegt ein partieller Entwurf vor, in welchem die Domänen-Klassen ModelElement, Transition und Place bereits modelliert wurden (siehe Abb. 7.4). Diese bilden einen Teil des Meta-Modells für Petrinetze. Es wurde auch eine Klasse PlaceUpdater modelliert, welche die grafische Repräsentation von Petrinetzen aktualisieren soll, sobald sich der Petrinetzmodellzustand ändert. Diese Klasse soll also als Beobachter eines Petrinetzmodells – genauer: eines Place-Objekts – agieren und somit Teil einer Observer-Musterimplementierung werden.

[12] Code-Zeilen inklusive Kommentarzeilen. Alle gezählten Zeilen entsprechen folgendem regulären Ausdruck: `\n[\S]*`

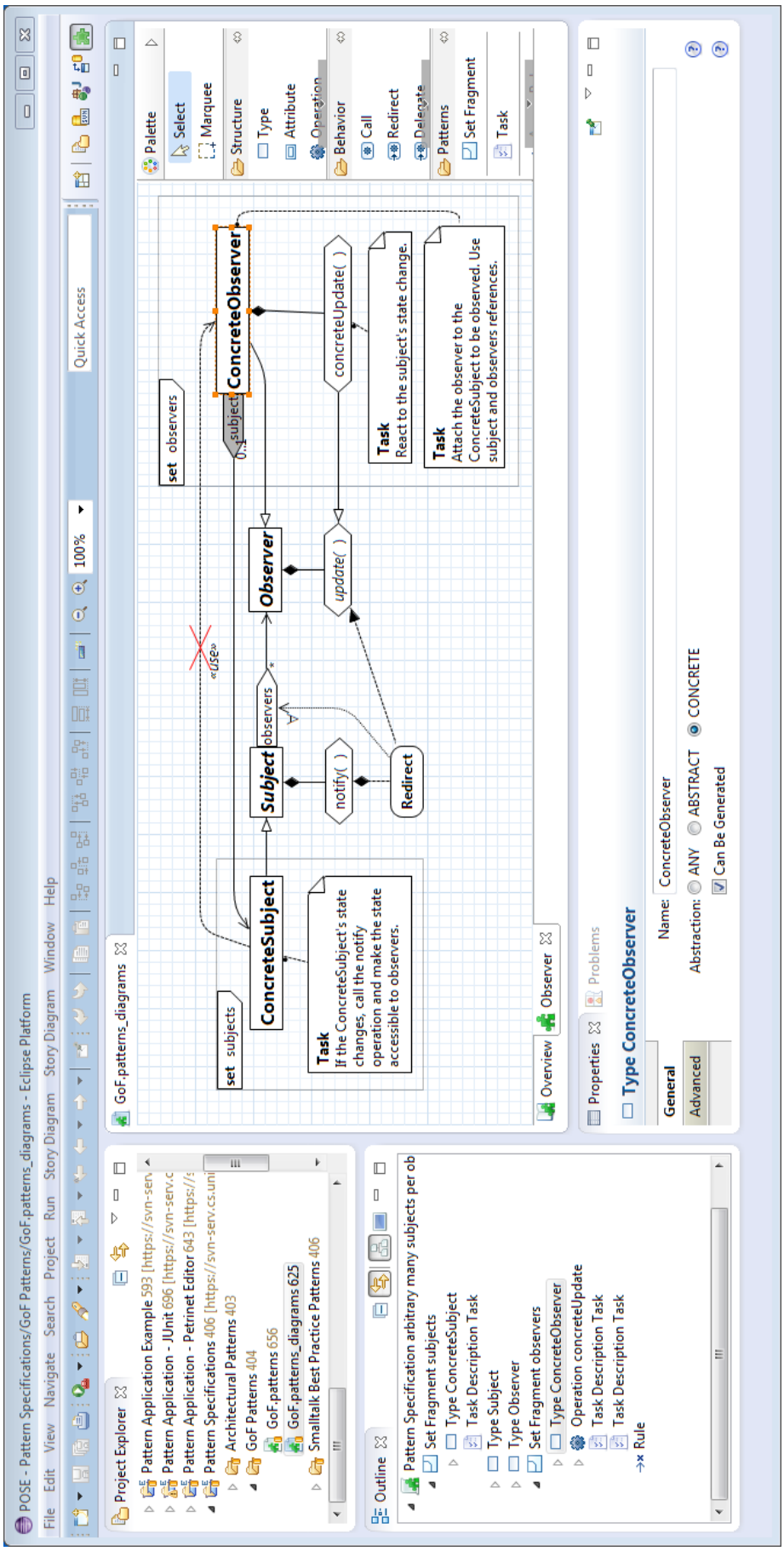


Abbildung 7.3: Spezifikation des Observer-Musters im Pattern Specification Editor

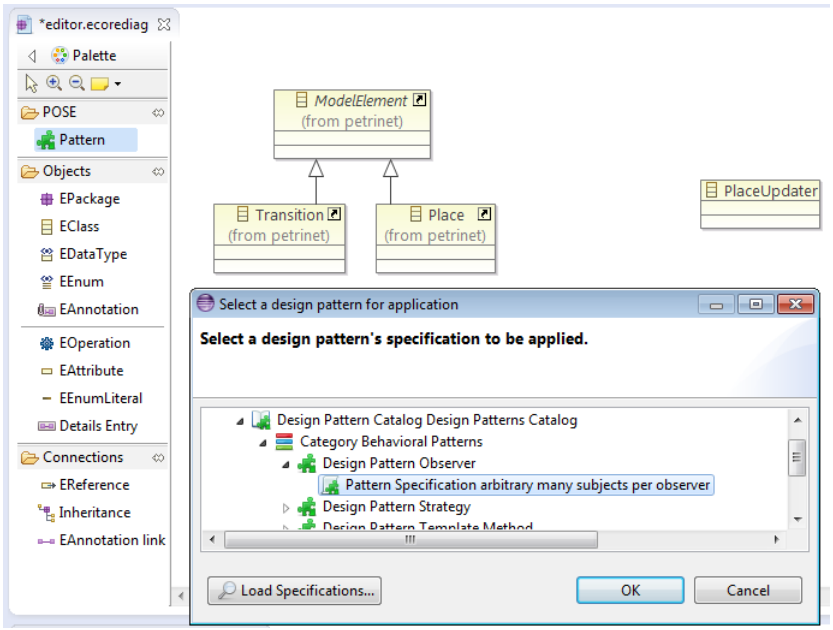


Abbildung 7.4: Auswahl des Observer-Musters aus einem Musterkatalog

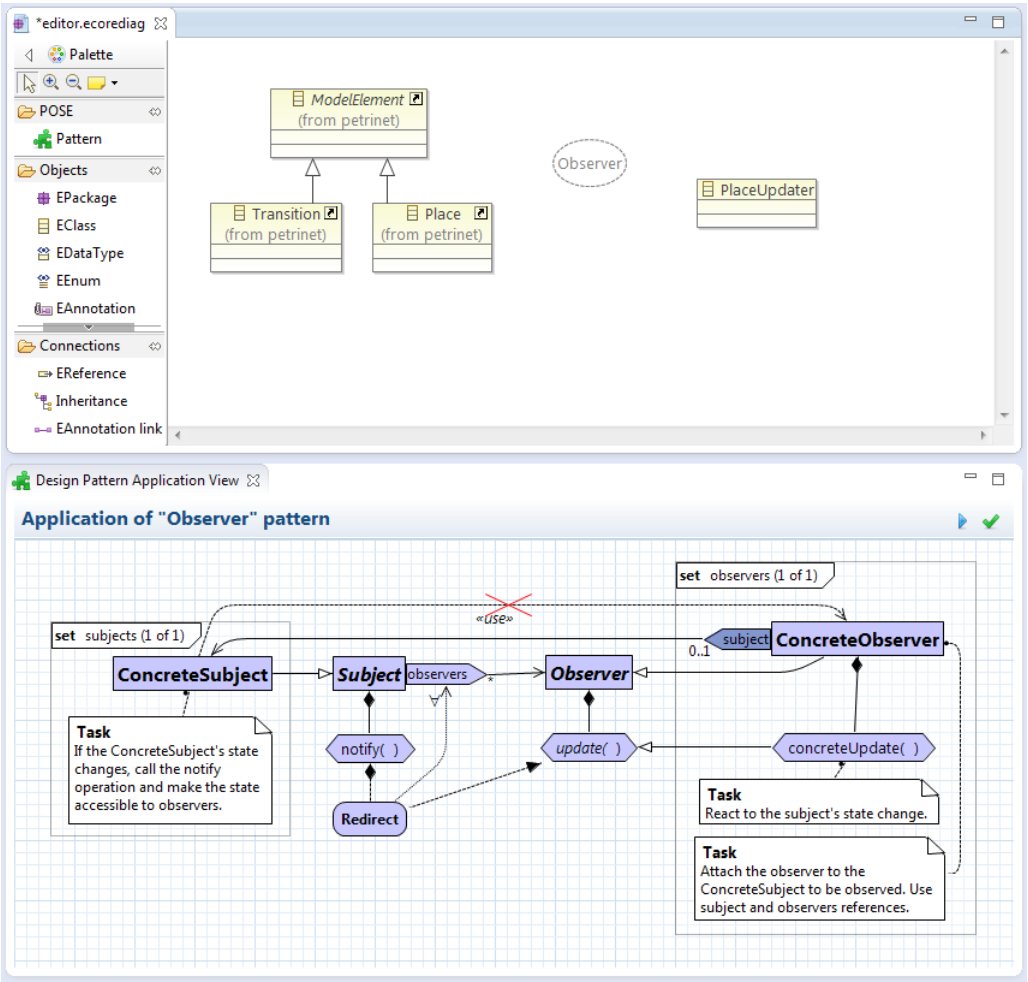


Abbildung 7.5: Initiale Darstellung einer Observer-Muster-Anwendungsstelle

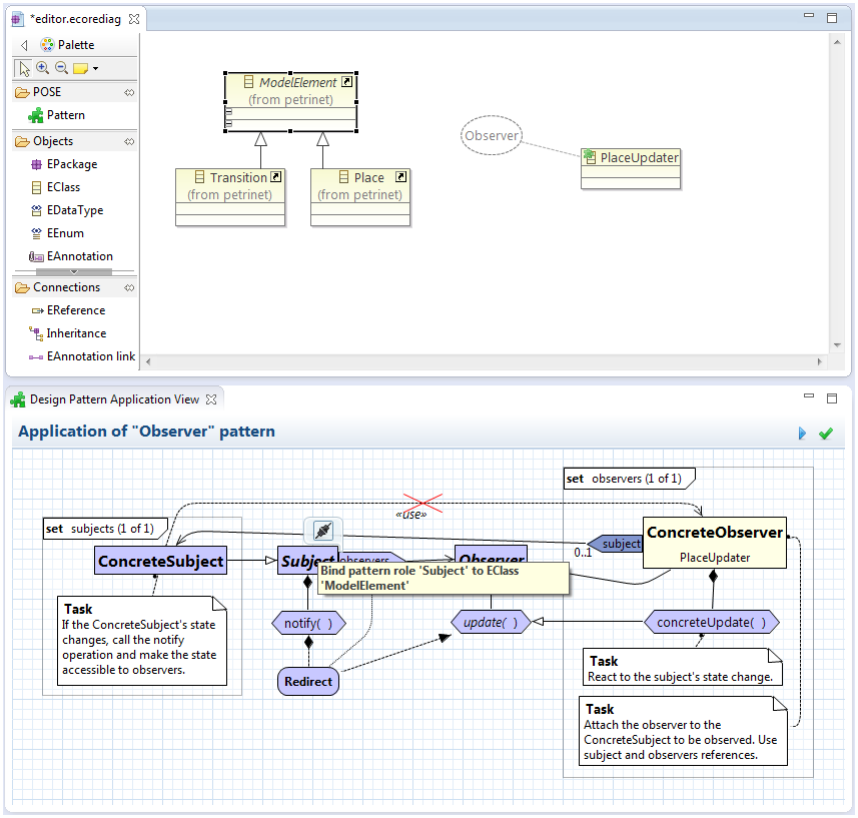


Abbildung 7.6: Zuordnung einer Rolle zu einer Klasse im Entwurf

Beginn einer Musteranwendung

Sobald dem Modellierer bewusst ist, dass hier ein zuvor spezifiziertes Muster angewendet werden soll, wählte er in dem angepassten Ecore-Modell-Editor das Werkzeug für eine Musteranwendung aus (Puzzle-Symbol in der Werkzeugpalette in Abb. 7.4), selektiert einen Musterkatalog und wählt darin wie in Abb. 7.4 dargestellt eine Musterspezifikation zur Anwendung aus. Als Zwischenergebnis erhält er eine Darstellung einer begonnenen Musteranwendung im Klassendiagramm und eine detaillierte Repräsentation der Anwendungsstelle basierend auf der Notation der Musterspezifikation (siehe Abb. 7.5). Letzteres stellt durch blaue farbliche Hinterlegung dar, welche Rollen der Musterspezifikation noch keinem Element im Klassen-Modell zugeordnet wurden.

Rollen zuordnen

Die Zuordnung der Rollen erfolgt durch Markieren eines Elements im Klassendiagramm und Betätigen eines zu einer Rolle gehörenden Buttons in der Musteranwendungsansicht. In der Abb. 7.6 wird die selektierte Klasse ModelElement der Rolle Subject durch Betätigen des beim Überfliegen der Rolle mit dem Mauszeiger auftauchenden Buttons zugeordnet. Die Rolle ConcreteObserver ist in dieser Abbildung bereits der Klasse PlaceUpdater zugeordnet. Das wird im Klassendiagramm durch eine entsprechende Kante und ein Puzzle-Symbol an der betroffenen Klasse dargestellt. Der Name der zugeordneten Klasse wird in der Musteranwendungsansicht unter dem Rollennamen dargestellt und die Rolle gelb hinterlegt dargestellt, um eine bereits erfolgte Zuordnung hervorzuheben.

Namen festlegen

Bei Rollen, zu welchen im Klassendiagramm noch keine Entsprechungen existieren, können Namen für zu erzeugende Elemente angegeben werden. Auch diese

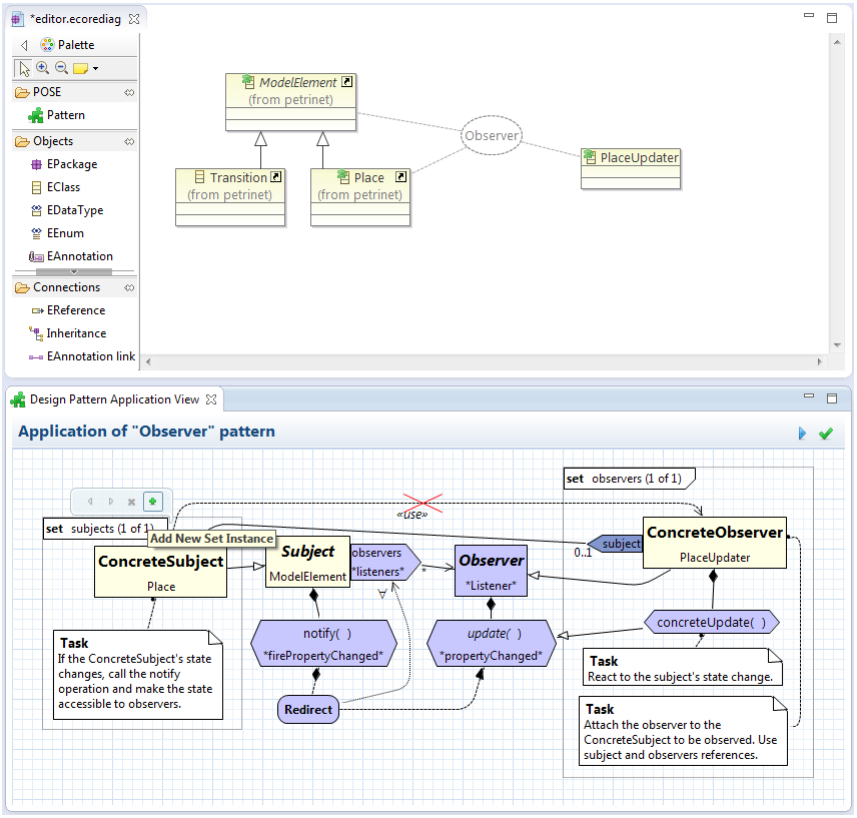


Abbildung 7.7: Ergänzen einer zusätzlichen Set-Fragment-Instanz

werden in der Musteranwendungsansicht unter dem Rollennamen dargestellt. In der Abb. 7.7 wurden z.B. Namen für die zu erzeugende Observer-Klasse und die zugehörige update-Operation mit „Listener“ und „propertyChanged“ vorbelegt. Solche Namen werden zwecks Unterscheidung von Namen zugeordneter Entwurfsteile in * eingeschlossen.

Instanzen von Set Fragments ergänzen

Alle Rollen in der Abb. 7.7 sind, sofern möglich, existierenden Entwurfsteilen oder zu verwendenden Namen zu erzeugender Entwurfselemente zugeordnet. Im Klassendiagramm sind die am Observer-Muster beteiligten Klassen durch Kollaborationskanten und Puzzle-Symbole markiert. Es ist festgelegt worden, dass die Klasse Place die Rolle ConcreteSubject einnimmt. Analog dazu soll auch die Klasse Transition als ConcreteSubject fungieren. Dazu wird eine zusätzliche Ausprägung des Set Fragments subjects, eine Set-Fragment-Instanz, durch einen entsprechenden Button wie in der Abb. 7.7 dargestellt ergänzt (Plus-Button in der Musteranwendungsansicht). Anschließend kann auch für diese Ausprägung der Entwurfsstruktur im Set Fragment subjects eine Zuordnung der Rollen zu Entwurfsteilen erstellt werden, also die Rolle ConcreteSubject der Klasse Transition zugeordnet werden. Analog dazu wird eine zusätzliche Instanz des Set Fragments observers für eine zu erzeugende TransitionUpdater-Klasse erstellt. Das Ergebnis dieser Schritte ist in der Abb. 7.8 dargestellt.

Navigation zwischen Instanzen

Um zwischen den Ansichten der verschiedenen Set-Fragment-Instanzen zu wechseln, stehen weitere Navigations-Buttons je Set Fragment zur Verfügung (dargestellt als Pfeile). Diese sind in der Abb. 7.7 noch ausgegraut, weil zu diesem Zeit-

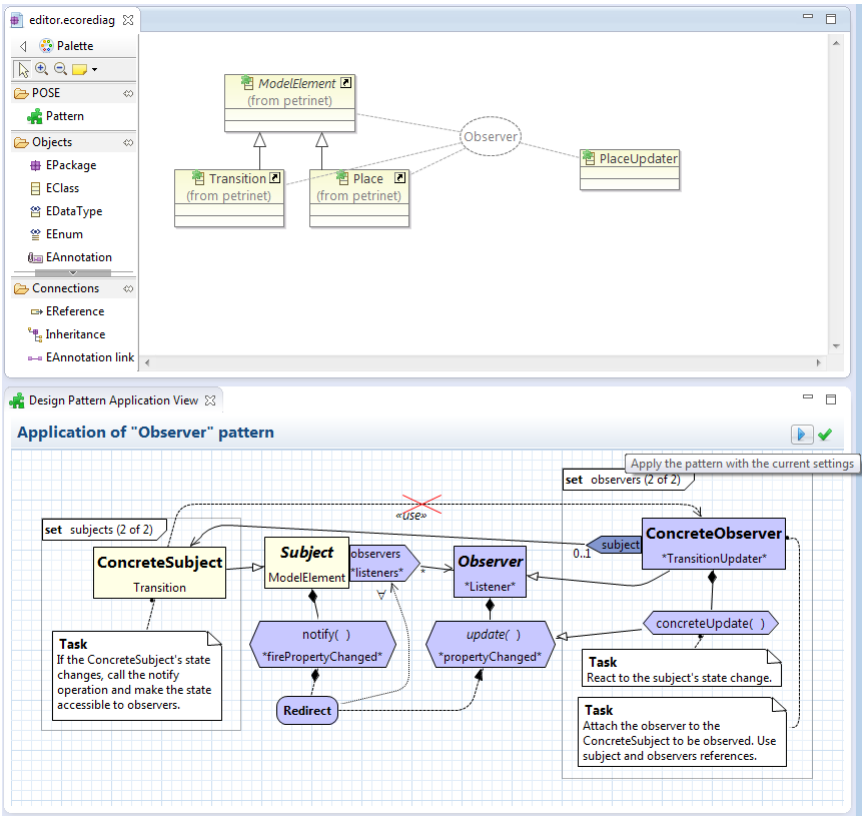


Abbildung 7.8: Abgeschlossene Rollenzuordnung, Auslösen der Musteranwendung

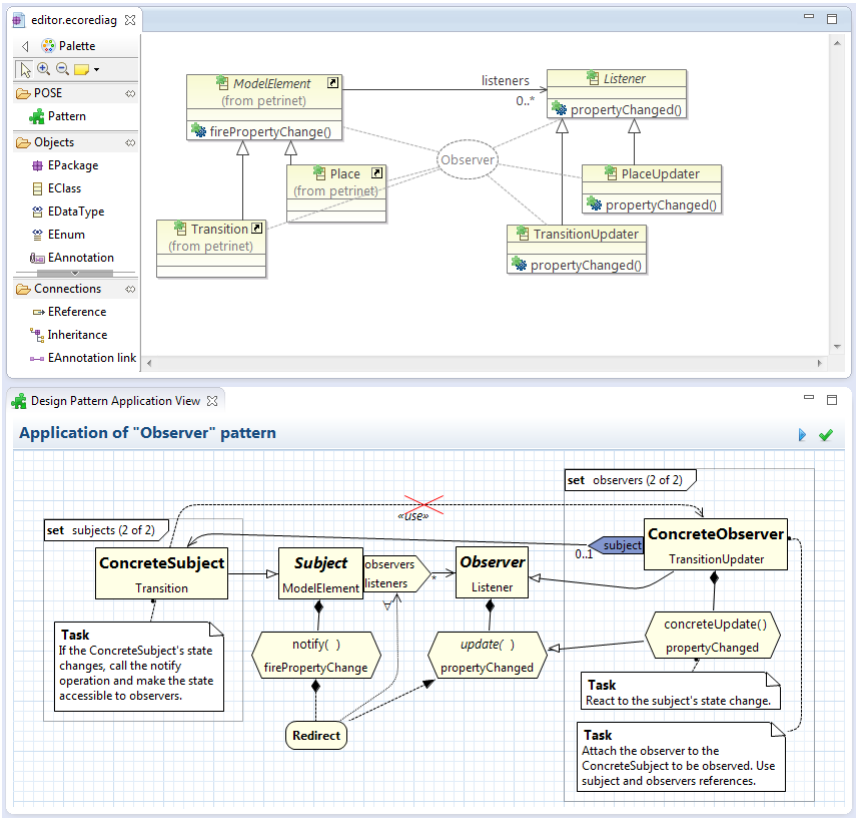


Abbildung 7.9: Zustand nach automatischer Musteranwendung

punkt nur je eine Instanz existiert. Auch das Entfernen einer Instanz ist möglich (Button mit Kreuz-Symbol), jedoch wird bei meinem Ansatz zu jedem Set Fragment mindestens eine Instanz verlangt (darum ist auch dieser Button ausgegraut). Jede Instanz ist mit einer Nummer versehen und wird zusammen mit der Anzahl aller Instanzen am Label des Set Fragments dargestellt. In der Abb. 7.8 steht neben dem Namen des Set Fragments subjects „(2 of 2)“, was die Darstellung der zweiten von zwei Set-Fragment-Instanzen ausdrückt.

Musterimplementierung & Korrespondenzen generieren

Nachdem alle Rollenzuordnungen erfolgt sind, kann die vollautomatische Vervollständigung des existierenden Entwurfs mit dem in der Abb. 7.8 markierten Button ausgelöst werden. Das Ergebnis der Musteranwendung ist in der Abb. 7.9 dargestellt. Alle fehlenden Teile, welche laut Musterspezifikation automatisch erzeugt werden können, wurden im Klassenmodell und im Klassendiagramm ergänzt. Es sind die Klassen Listener und TransitionUpdater mit den spezifizierten Operationen und Vererbungsbeziehungen entstanden. Die unidirektionale Assoziation listeners von der Klasse ModelElement zur Klasse Listener sowie die Operationen firePropertyChange in der Klasse ModelElement und propertyChange in der schon existierenden Klasse PlaceUpdater sind ergänzt worden. Die Rollenzuordnungen sind vervollständigt worden und sind in der Musteranwendungsansicht sichtbar. Die einzige noch nicht zugeordnete Musterrolle ist die subject-Referenz der beiden ConcreteObserver-Klassen, was durch die blaue Hintergrundfarbe in der Musteranwendungsansicht hervorgehoben wird.

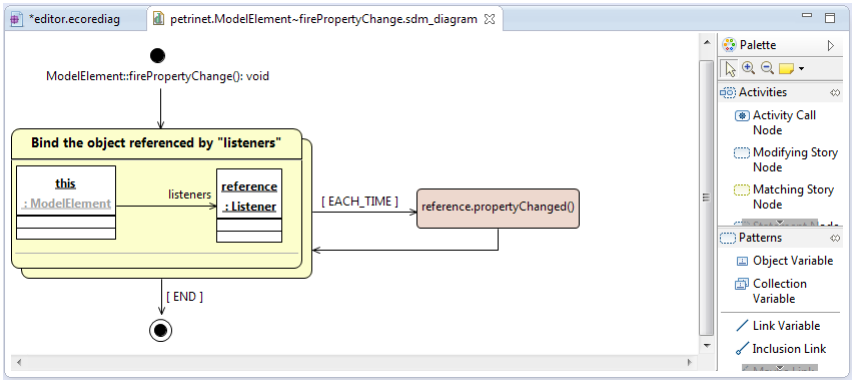


Abbildung 7.10: Erzeugtes Verhaltensmodell (Story-Diagramm) für eine Operation

generiertes Verhaltensmodell

Neben der Ergänzungen im Klassenmodell wurde auch ein Verhaltensmodell für die ergänzte Operation firePropertyChange erstellt. Das Verhalten wird in Form eines Story-Diagramms definiert. Das erzeugte Modell ist in der Abb. 7.10 dargestellt. Dieses Story-Diagramm definiert das durch die redirect-Aktion spezifizierte Verhalten. In diesem Fall werden alle über die Referenz listeners erreichbaren Listener-Objekte in einer Schleife durchlaufen (gelb hinterlegter Activity-Knoten des Story-Diagramms). Auf jedem dieser Objekte, welche in einer Variable namens reference zwischengespeichert werden, wird die Methode propertyChange aufgerufen (rot hinterlegter Activity-Knoten).

manuelle Vervollständigung

Da die Assoziationen zu der spezifizierten Referenz subject nicht automatisch erzeugt werden konnten, müssen sie manuell modelliert werden. In der Abb. 7.11 sind das die nach der automatischen Musteranwendung manuell ergänzten unidirektionalen Assoziationen place und transition. Nachdem sie im Klassenmodell ergänzt

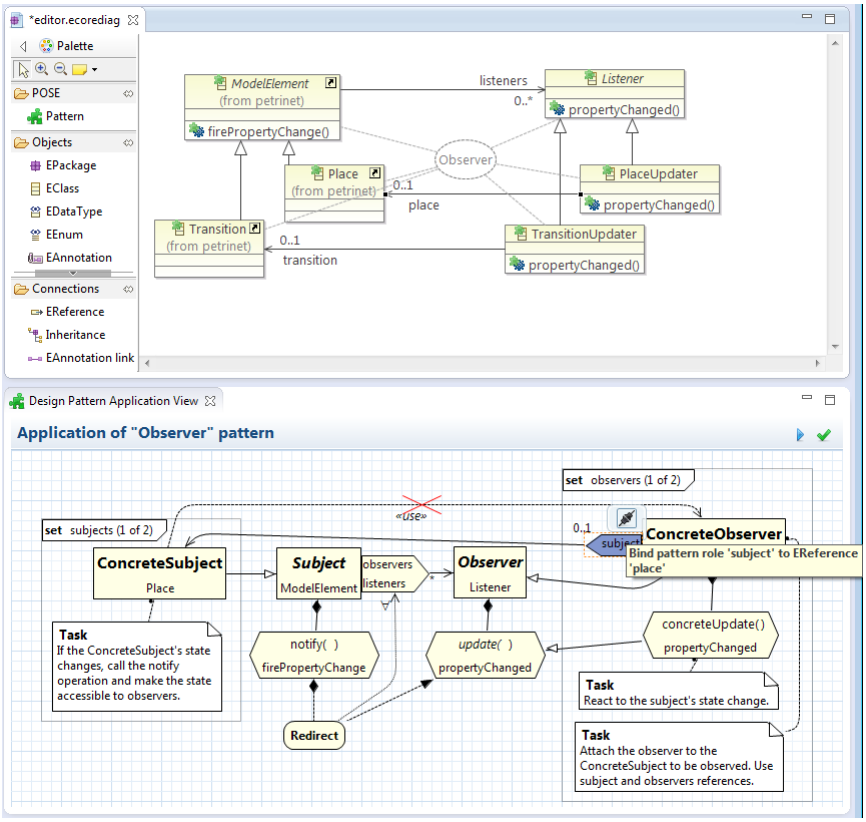


Abbildung 7.11: Rollenzuordnung nach manueller Ergänzung des Entwurfs

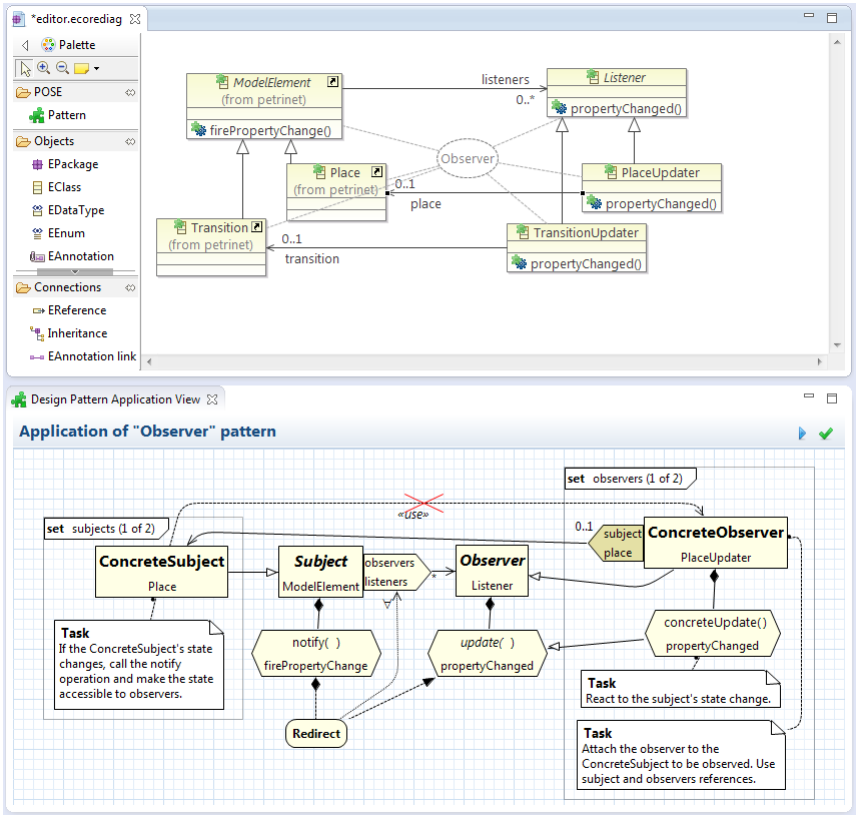


Abbildung 7.12: Klassendiagramm und Musteranwendungsansicht nach vollständiger Musteranwendung

wurden, können sie zur vollständigen Repräsentation der Musterimplementierung und zwecks ihrer späteren Validierbarkeit ihrer Rolle in der Musterspezifikation zugeordnet werden. Diese Zuordnung erfolgt erneut durch Auswahl der Assoziation im Klassendiagramm und durch Betätigung des zugehörigen Buttons in der Musteranwendungsansicht und ist in der Abb. 7.11 dargestellt.

Ergebnis der Anwendung

Nach dieser Zuordnung ist die Anwendung des Observer-Musters bis auf die noch manuell zu erledigenden Entwurfsaufgaben vollständig. Das Ergebnis wird in der Abb. 7.12 dargestellt. Der Erledigungsstatus der Entwurfsaufgaben wird im vorliegenden Prototypen nicht dargestellt, könnte aber nach Erweiterung des Prototypen wie in der Abb. 4.8 (S. 89) dargestellt werden.

Die so entstandene Musterimplementierung, bestehend aus Klassen- und Story-Diagramm-Modell, könnte nach weiteren Anpassungen der Modelle, z.B. nach weiteren Musteranwendungen, validiert werden. Solche Validierungen sind im vorliegenden Prototypen jedoch nicht vollständig implementiert worden.

7.3 Herausforderungen, Ergebnisse und Einschränkungen des Prototypen

Zu den größten Herausforderungen bei der Implementierungen gehörten Folgende. Es mussten zahlreiche, auf unterschiedlichen Konzepten und Frameworks basierende Werkzeuge integriert werden. Zum Beispiel mussten zwei gleichzeitig sichtbare Editoren entwickelt werden (üblicherweise ist in Eclipse genau ein Editor sichtbar). Die verschiedenen Modelle (Ecore-, Story-Diagramm- und Dekorationsmodelle) mussten so miteinander verknüpft werden, dass sie sich mit den Original-Editoren weiterhin bearbeiten lassen. Es mussten generierte Editoren in der Darstellung und Funktionalität erweitert werden (insb. um die Visualisierung von Anwendungsstellen und das Anwenden von Mustern im EcoreTools-Editor), obwohl eine derartige Erweiterung in GMF-generierten Editoren nicht vorgesehen ist.

Integration heterogener Werkzeuge

Werkzeug-Prototypen erweitern

Damit alle in dieser Arbeit eingeführten Aktionen (Abschn. 3.4.2) in Story-Diagramme übersetzt werden können, musste die Funktionalität des Story-Diagramm-Editors und des zugehörigen Interpreters (beides ebenfalls Prototypen) erweitert werden, weil einige der in der Literatur [vDHH+12] beschriebenen, für meine Übersetzung nötigen Konzepte (z.B. Methodenaufrufe, Argumentübergabe, negative Anwendungsbedingungen) bisher komplett oder teilweise fehlten.

automatische Layouts

Hinzu kommen die bei grafischen Editoren bekannten Herausforderungen beim automatischen Layout von Diagrammen. Zum Beispiel ist eine überlappungsfreie, übersichtliche Positionierung generierter Elemente in einem Diagramm schwierig. Ebenso ist die zwecks Wiedererkennung annähernd identische Anordnung der Elemente in der Musteranwendungssicht und der Musterspezifikation aufgrund zusätzlicher Labels und damit anderer Elementgrößen in der Musteranwendungssicht schwierig.

Ergebnisse

Mit der POSE Tool Suite habe ich für die in dieser Arbeit vorgestellten Konzepte eine prototypische Implementierung geliefert und damit ihre Umsetzbarkeit gezeigt (Proof of Concept). Trotz beschränkter Funktionalität des Prototypen konnte das Vorgehen bei der Spezifikation und Anwendung eines Musters, samt Verknüpfung von Entwurf und Spezifikation sowie der Visualisierung von Anwen-

dungsstellen demonstriert werden. Das ermöglicht eine erste Evaluation des Ansatzes (siehe Kap. 8) anhand konkreter Anwendungsbeispiele, z.B. im praxisnahen JUnit-Anwendungsszenario (siehe Anhang D).

Einschränkungen

Die vorliegende Version der POSE Tool Suite implementiert nur einen Teil der in dieser Arbeit vorgestellten Konzepte aus den Kapiteln 3 bis 6. Die wichtigsten dieser Einschränkungen werden im Folgenden erläutert.

Validierung von Anwendungsstellen Die größte Einschränkung des Prototypen ist die fehlende Implementierung der in Kapitel 6 beschriebenen Konsistenzprüfungen zwischen einer Musterspezifikation und einer zugehörigen Musteranwendungsstelle.

Nach Anwendung eines spezifizierten Entwurfsmusters liegt eine detaillierte Zuordnung der Musterrollen zu den diese Rollen einnehmenden Entwurfsteilen in Klassen- und Story-Diagramm-Modellen vor. Der Implementierung der in Abschnitt 6.2 geschilderten Prüfungen steht prinzipiell nichts im Wege. Alle dazu notwendigen Informationen liegen in den Modellen vor und die Prüfoperationen selbst sind nicht sonderlich komplex. Auch wenn solche Prüfungen in der Praxis von großer Hilfe wären, habe ich auf ihre Implementierung verzichtet. Der Grund dafür ist, dass die Implementierung wissenschaftlich wenig herausfordernd, dabei aber sehr aufwändig ist (große Anzahl der Prüfungen, nötige Integration in existierenden Werkzeugen, notwendige Visualisierung der Prüfergebnisse). Leider schränkt das auch die Möglichkeiten zur Evaluation des Konzepts erheblich ein (siehe Abschnitt 8.4).

Detaillierte Visualisierung von Anwendungsstellen Die zweite signifikante Einschränkung der prototypischen Implementierung liegt in den Visualisierungsmöglichkeiten von Musteranwendungsstellen. Der Prototyp stellt nur einige der in Abschnitt 4.4 vorgestellten Ansichten und verwendbaren Detailgrade zur Verfügung.

Während an Musteranwendungsstellen beteiligte Teile des Entwurfs in einem Klassendiagramm mit einem Puzzle-Symbol markiert werden (siehe Abb. 4.9, S. 91), werden sie in Story-Diagrammen bisher nicht auf die gleiche Weise hervorgehoben (siehe Abb. 4.12, S. 92). Die detaillierte Darstellung der von einem Entwurfselement eingenommenen Musterrollen (siehe Abb. 4.10, S. 91 & Abb. 4.13, S 92) sowie die farbliche Hervorhebung aller an einer Musterimplementierung beteiligten Elemente (Knoten und Kanten) bei Selektion einer Musterimplementierung (siehe Abb. 4.11, S. 91 & Abb. 4.14, S 92) in Klassen- und Story-Diagrammen fehlen dem Prototypen ebenfalls.

Bei der Darstellung von UML-Kollaborationen in Klassendiagrammen fehlt die Angabe der Musterrollen an den Kollaborationskanten (vgl. Abb. 4.15, S. 94 und Abb. 7.12 oben). Außerdem fehlt auch hier die farbliche Hervorhebung aller an einer Musterimplementierung beteiligten Entwurfsteile (Abb. 4.16, S. 94) bei ihrer Selektion. In Story-Diagrammen fehlt die Darstellung von Kollaborationen völlig (Abb. 4.17, S. 94).

Die Darstellung der Entwurfsteile in einem Klassendiagramm, welche denen in einer Musteranwendungssicht sichtbaren Entwurfsteilen entsprechen, in

einem separaten Klassendiagramm mit ausschließlich diesen Klassen und Beziehungen sowie zur Musteranwendungssicht analogen Navigationsmöglichkeiten für Set Fragments (vgl. Abb. 4.22, S. 97 & Abb. 4.6, S. 88) wurde ebenfalls nicht implementiert.

Bei allen genannten und in Abschnitt 4.4 vorgestellten Darstellungen von Musteranwendungsstellen besteht kein Zweifel, dass ihre Implementierung möglich ist. Aufgrund des hohen Implementierungsaufwands und des geringen wissenschaftlichen Beitrags durch eine Implementierung habe ich auf diese Funktionalität im Prototypen verzichtet.

Nachträgliche Modellierung der Rollenzuordnung eingeschränkt Wird ein Entwurfsmuster mit Hilfe der in Kapitel 5 beschriebenen Operationen angewandt, so wird ein Modell der Anwendungsstelle automatisch angelegt und kann zur Visualisierung und Validierung der Anwendungsstelle verwendet werden. Dieses Modell lässt sich auch nachträglich durch manuelle Zuordnung der Entwurfsteile zu Musterrollen erstellen, ist jedoch mit dem vorliegenden Prototypen nur auf Ebene der Klassendiagramme möglich.

Für die Zuordnung der Musterrollen, insbesondere der Aktionen (z.B. Delegation, siehe Tab. A.3, S. 259) in einer Musterspezifikation, zu Elementen in einem Story-Diagramm ist eine komplizierte 1-zu-n-Zuordnung in mehreren Schritten mit Hilfe von Tokens nötig (siehe Abschn. C.3.3). Eine derart komplexe, nicht automatisierte Rollenzuordnung ist für Benutzer kaum zumutbar. Darum habe ich auf die Implementierung einer ebenso komplexen Benutzerschnittstelle für eine derartige Rollenzuordnung in Story-Diagrammen verzichtet. Die Konsequenz daraus ist eine eingeschränkte Möglichkeit zur Validierung von Musteranwendungsstellen, falls eine Anwendungsstelle erst nach Musteranwendung durch manuelle Zuordnung der Elemente in einem Klassendiagramm zu Musterrollen modelliert wird.

Erledigungsstatus von Entwurfsaufgaben Im vorliegenden Prototypen wird der Erledigungsstatus von in einer Musterspezifikation definierten Aufgaben in der Musteranwendungssicht abweichend vom Konzept (siehe Abb. 4.8, S. 89) nicht dargestellt. Der Grund dafür ist auch in diesem Fall der hohe Implementierungsaufwand ohne signifikanten wissenschaftlichen Beitrag durch eine Implementierung dieser Funktionalität.

8 Evaluation

Den im Rahmen dieser Arbeit vorgestellten Ansatz zur modellgetriebenen Anwendung von Mustern und der expliziten Modellierung von Anwendungsstellen evaluiere ich im Folgenden anhand mehrerer, relativ kleiner (nicht vergleichender) Fallstudien [Pre01, Definition 3.6]. Alle Fallstudien habe ich selbst durchgeführt. Fallstudien sind im Vergleich zu anderen empirischen Forschungsmethoden wie kontrollierten Experimenten oder Feldstudien weniger aufwändig und eignen sich gut zur Untersuchung von Methoden oder Werkzeugen anhand konkreter Anwendungsbeispiele. Ihre Ergebnisse lassen sich jedoch nur schwer verallgemeinern.

Anwendungsbeispiele und Fallstudien

Im Rahmen der Fallstudien habe ich unterschiedliche Muster mit der in Kapitel 3 vorgestellten Musterspezifikationssprache beschrieben. Zu den spezifizierten Mustern zählen neben den bekannten Gang-of-Four-Entwurfsmustern auch einige weitere Entwurfsmuster sowie Architekturmuster und Idiome. Einige der spezifizierten Muster habe ich in unterschiedlichen Situationen in einem Softwareentwurfsmodell angewandt. Der Entwurf wurde jeweils durch Klassen-[1] und Story-Diagramm-Modelle beschrieben. Für eine möglichst realistische Einschätzung der Praxistauglichkeit in einem umfangreicheren Anwendungsbeispiel habe ich die Entwicklung des JUnit-Frameworks[2] mit meinem Verfahren nachempfunden.

Vorgehen

Eine Evaluierung der automatischen Validierung von Anwendungsstellen (siehe Kap. 6) sowie der detaillierten Visualisierung von Anwendungsstellen in Klassen- und Story-Diagrammen (siehe Abschn. 4.4) war aufgrund fehlender Funktionalität im vorliegenden Prototypen nicht möglich.

Einschränkungen

8.1 Spezifikation von Entwurfsmustern

Zwecks Evaluation der Ausdrucksmächtigkeit der in Kapitel 3 vorgestellten Musterspezifikationssprache wurden diverse Entwurfs- und Architekturmuster spezifiziert. Es wurden alle 23 Muster der Gang of Four [GHJV95] sowie exemplarisch 2 weitere Entwurfsmuster, 2 Architekturmuster und 2 Idiome spezifiziert. Insgesamt wurden 34 Musterspezifikationen erstellt, darunter einige Varianten je Muster. Alle erstellten Spezifikationen sind im Anhang B aufgeführt. Einige dieser Spezifikationen – eine in Bezug auf eingesetzte Sprachkonstrukte und Set-Fragment-Konstellationen möglichst repräsentative Auswahl – stelle ich im Folgenden exemplarisch vor und bespreche sie.

Erstellte Musterspezifikationen

Die Spezifikationssprache wurde zu zwei wesentlichen Zwecken entwickelt (vgl. Abschnitt 3.1): (a) zur Menschen-lesbaren Beschreibung der von einem Entwurfsmuster vorgeschlagenen Entwurfslösung (Klassenstruktur und ggf. zugehöriges Verhalten) und (b) zur präzisen, Maschinen-lesbaren Beschreibung zum Muster

betrachtete Zwecke

[1] Basierend auf Ecore (http://www.eclipse.org/modeling/emf/)
[2] http://junit.org/

konformer Softwareentwürfe (Musterimplementierungen) mit möglichst all seinen Varianten. In diesem Abschnitt gehe ich – soweit möglich – auf die Eignung der Spezifikationssprache für diese beiden Zwecke ein. Die Eignung der Musterspezifikationssprache zur Erstellung von Spezifikationen für eine Werkzeug-gestützte Anwendung eines Musters ergibt sich aus der Evaluation des Verfahrens zur automatischen Musteranwendung (siehe Abschn. 8.2). Auf ihre Eignung für die Prüfung von Musterimplementierungen auf Konsistenz mit einer Musterspezifikation gehe ich u.a. im Abschnitt 8.4 ein.

8.1.1 Spezifikation der GoF-Entwurfsmuster

Alle 23 Muster der Gang of Four (GoF) [GHJV95] konnten in der vorgestellten Musterspezifikationssprache modelliert werden. Wie erwartet und vorgesehen (siehe Abschn. 3.1.1), konnten nur Teile der meist ca. 10-seitigen Musterbeschreibung in einer Spezifikation erfasst und formalisiert werden. Während Erläuterungen und Implementierungsbeispiele weggelassen wurden, wurden die Struktur und Teile des Verhaltens der in einer Musterbeschreibung vorgeschlagenen Softwareentwurfslösung präzisiert und samt zahlreicher Varianten in einer oder einigen wenigen Spezifikationen je Muster zusammengefasst (nur die Entwurfslösung soll erfasst werden, siehe Begriffe in Abb. 2.1, S. 26). Im Folgenden erläutere ich einige der GoF-Spezifikationen exemplarisch.

Reduktion auf das Wesentliche

Observer

Das Observer-Muster wird von der Gang of Four recht ausführlich auf 11 Seiten in ihrem Buch beschrieben (vgl. Abb. 1.1, S. 4) [GHJV95, S. 293 ff.]. Die Struktur des vorgeschlagenen Softwareentwurfs lässt sich wie in Abb. 8.1 skizzieren. Diese Entwurfsstruktur habe ich zusammen mit einem Teil des zugehörigen Verhaltens wie in der Abb. 8.2 dargestellt spezifiziert. Weitere (weniger Implementierungsvarianten umfassende) Spezifikationsvarianten sind den Abbildungen B.21 und B.22 (S. 297) zu entnehmen.

Struktur

Die Spezifikation (Abb. 8.2) enthält alle Klassen, Operationen und Assoziationen aus der Entwurfsskizze (Abb. 8.1). Die Klassen Subject und Observer sowie die zugehörigen Operationen notify und update habe ich 1-zu-1 in die Spezifikation übernommen, genauso wie die unidirektionale Assoziation observers mit der Kardinalität *. Analog zur Entwurfsskizze sind die beiden Klassen abstrakt, die notify-Operation ist konkret, die update-Operation ist abstrakt.

Varianten

Die Entwurfsskizze stellt einen beispielhaften Entwurf dar und deutet beliebig viele Klassen ConcreteSubject1, ConcreteSubject2,... und ConcreteObserver1, ConcreteObserver2,... an. Im Gegensatz dazu verwende ich in meiner Spezifikation nur je eine Klasse ConcreteSubject und ConcreteObserver. Eine beliebige Anzahl dieser Klassen samt all ihrer Eigenschaften wie zugehöriger Operationen und Assoziationen spezifizierte ich mit Hilfe der Set Fragments subjects und observers. Die Anzahlen der ConcreteSubject- und ConcreteObserver-Klassen können damit unabhängig voneinander beliebig variieren. (Ich nehme hier an, dass jeder Beobachter beliebig viele Subjekte beobachten kann. Für den Fall, dass jeder Beobachter genau ein Subjekt beobachtet, habe ich eine alternative Spezifikation in Abb. B.21,

disjunkte Set Fragments

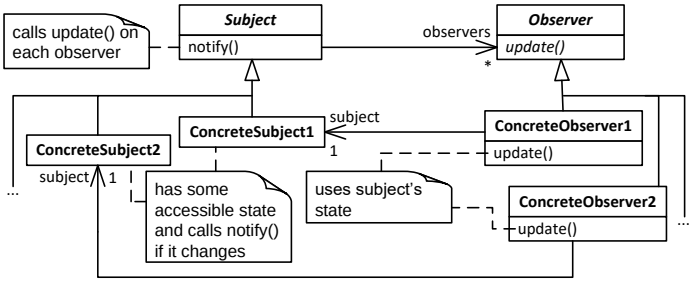


Abbildung 8.1: Grobe Idee der Entwurfslösung hinter dem Observer-Muster

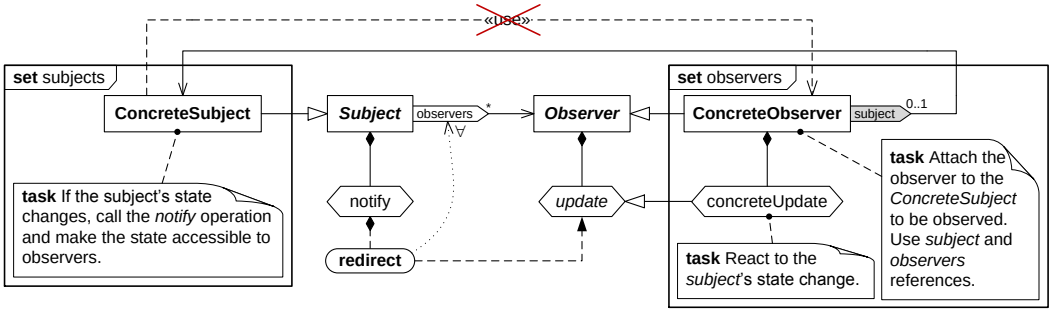


Abbildung 8.2: Spezifikation des GoF-Entwurfsmusters „Observer“ – Variante mit beliebig vielen Subjekten je Beobachter

S. 297 angefertigt.) Durch die Verwendung von Set Fragments wird die Spezifikation bezüglich der Vielfachheit der Klassen und ihrer Eigenschaften kompakter – die Klassen und Operationen werden nur einmal dargestellt – und präziser – es wird klar angegeben, welche Teile des Entwurfs wie vervielfältigt werden können.

Bei der Klasse ConcreteObserver habe ich die Operation concreteUpdate spezifiziert. Diese entspricht der update-Operation der ConcreteObserver-Klassen aus der Entwurfsskizze, hat nun aber einen eindeutigen Rollennamen erhalten. Da die Signatur der Operation nicht näher spezifiziert werden kann, die Signatur jedoch der von der update-Operation entsprechen muss, habe ich eine Implementierungsbeziehung mit einer Kante von concreteUpdate zu update spezifiziert (Notation wie bei der UML-Vererbungsbeziehung). Diese Beziehung ist aus der Entwurfsskizze nicht erkennbar und präzisiert ebenfalls die Spezifikation.

Methodensignatur erben ist explizit

Auch die unidirektionale Assoziation subject habe ich aus der Entwurfsskizze in die Musterspezifikation übernommen. Die Assoziationskante verläuft hier allerdings zwischen zwei voneinander unabhängigen Set Fragments. Für solche Kanten ist keine Auffaltungssemantik definiert (siehe Unklarheiten, S. 52). Aufgrund der Assoziationskardinalität 0..1 und der unabhängig voneinander variierenden Anzahl von ConcreteSubject- und ConcreteObserver-Klassen kann zum Zeitpunkt der Musterspezifikation nicht angegeben werden, welche ConcreteObserver-Klasse ein Beobachter welcher ConcreteSubject-Klasse werden soll (welche Klassen die subject-Kanten nach der Auffaltung verbinden sollen). Aus diesem Grund ist diese Assoziation in der Spezifikation als *nicht generierbar* markiert. Sie dient als Hinweis, dass eine ConcreteObserver-Klasse eine noch nicht bestimmbare ConcreteSubject-Klasse referenzieren soll. Ob eine solche Referenz in einer Musterimplementierung existiert, kann sogar automatisch geprüft werden. Die Entscheidung,

Unklares als *nicht generierbar* markiert, für Validierung genutzt

welche zwei Klassen per subject-Assoziation miteinander verbunden werden, muss bei Anwendung des Musters vom Entwickler getroffen und die Assoziation manuell erstellt werden. (In der alternativen Spezifikation des Observer-Musters, in der jedem ConcreteObserver genau ein ConcreteSubject zugeordnet wird, ist es eindeutig spezifiziert, welche Klassen per Assoziation miteinander verbunden werden (siehe Abb. B.21, S. 297). In diesem Fall kann die subject-Assoziation generiert werden.)

Die Entwurfsskizze in Abb. 8.1 stellt – genauso wie das OMT-Klassendiagramm der Gang of Four [GHJV95, S. 294] – im Wesentlichen nur die Entwurfsstruktur dar. Darum wird die Idee hinter dem Entwurfsmuster in der Musterbeschreibung auf mehreren Seiten zusammen mit dem zugehörigen Verhalten u.a. mit Sequenzdiagrammen und Beispielcode genauer erläutert. Einen Teil dieser zusätzlichen Informationen konnte ich in meine Spezifikation übernehmen. Dazu gehört insbesondere das Informieren aller Beobachter über eine Änderung des internen Zustands eines Subjekts.

Verhalten präzise & kompakt als Aktionen erfasst

Das Weitergeben der Information über eine Zustandsänderung wird durch einen Aufruf der update-Operation auf allen registrierten Observer-Objekten erreicht. Dieses Verhalten ist durch die redirect-Aktion in der notify-Operation spezifiziert. Während die gestrichelte Kante auf die aufzurufende Operation verweist, gibt die gepunktete Kante an, auf welchen Objekten die Operation aufzurufen ist. In diesem Fall wird per Allquantor ∀ angegeben, dass die update-Operation auf jedem Objekt aufgerufen werden soll, welches von einem Subject-Objekt aus über die observers-Referenz erreichbar ist. Außerdem gibt die redirect-Aktion an, dass sämtliche Argumente des notify-Aufrufs auch an die aufzurufende update-Operation weitergereicht werden. So werden ggf. Zustandsänderungsinformationen in den Parametern der notify-Operationen 1-zu-1 an die update-Operation weitergegeben. Während die Entwurfsskizze nur eine knappe Notiz zu diesem Verhalten enthält („calls update() on each observer“), konnte ich diese Information in der Musterspezifikation präzise und kompakt festhalten.

zu vage Hinweise zumindest informell erfasst

Da der interne Zustand eines Subjekts stark von dem Anwendungsfall abhängt, kann er in der Musterbeschreibung und der Musterspezifikation nicht genauer angegeben werden. Für derart vage Informationen habe ich analog zur Entwurfsskizze nur eine informelle Beschreibung in Form von Kommentaren bzw. Entwurfsaufgaben vorgesehen. So verwende ich Entwurfsaufgaben, um anzugeben, dass die notify-Operation von jedem Subjekt aufgerufen werden soll, sobald sich der interne Zustand ändert, dass dieser Zustand den Beobachtern zugänglich gemacht werden soll (z.B. mit sichtbaren Zugriffsmethoden oder per Parameterübergabe) und dass jeder Beobachter geeignet auf die Zustandsänderung reagieren muss. In einer weiteren Entwurfsaufgabe habe ich in der Musterspezifikation die aus der erläuternden Beschreibung des Musters entnommene Information ergänzt, dass die Beobachter bei den Subjekten über die observers-Referenz registriert werden müssen.

ungewollte Abhängigkeiten erfasst

Eine weitere der Entwurfsskizze fehlende Information ist die erwünschte Unabhängigkeit der ConcreteSubject-Klassen von den sie beobachtenden ConcreteObserver-Klassen. Diese Regel habe ich mit einer durchgestrichenen use-Kopplungskante zwischen den Klassen spezifiziert. Damit wird ausgedrückt, dass jede ConcreteSubject-Klasse keinerlei (Kompilier-)Abhängigkeit zu einer beliebigen ConcreteObserver-Klasse derselben Musterimplementierung haben darf.

Im direkten Vergleich wirkt die Musterspezifikation komplexer und umfangreicher als die Entwurfsskizze. Die Spezifikation ist jedoch wie beschrieben präziser und enthält mehr Informationen als die Skizze. Ein Teil des Verhaltens sowie die beliebig oft vervielfältigbaren Teile des Entwurfs wurden zusätzlich spezifiziert. Außerdem sind die Entwurfsaufgaben ausführlicher beschrieben, was den zusätzlichen Platzbedarf im Diagramm im Vergleich zur Entwurfsskizze hauptsächlich ausmacht. Bedingt durch die gewählte Notation (Operationen werden als Knoten dargestellt und mit zugehörigen Typen per Kante verbunden) enthält die Spezifikation mehr Knoten und Kanten als die UML-basierte Entwurfsskizze. Das erhöht den Platzbedarf im Diagramm zusätzlich und reduziert den Wiedererkennungswert für mit der UML vertraute Entwickler. Set Fragments bringen eine zusätzliche Komplexität mit sich (man muss sich die möglichen aus der Spezifikation ableitbaren Entwurfsvarianten vorstellen), können aber Vielfache von Entwurfsteilen kompakt darstellen.

Vergleich zur Musterbeschreibung

Strategy

Die Entwurfsstruktur zum Entwurfsmuster Strategy [GHJV95, S. 315 ff.] kann wie in der Abb. 8.3 skizziert werden. Eine Context-Klasse referenziert eine abstrakte Strategy-Klasse, zu welcher es beliebig viele konkrete Implementierungen / Spezialisierungen gibt. Ein Context-Objekt delegiert sein Verhalten an die referenzierte Strategie, indem es eine der in der abstrakten Strategy-Klasse definierten Operationen aufruft. Die Strategie-Implementierungen rufen bei Bedarf Operationen der Context-Klasse auf.

das Muster

Wie bei der Spezifikation des Observer-Musters habe ich auch hier Klassen, Methoden und Assoziationen aus der Entwurfsskizze in die Musterspezifikation übernommen (siehe Abb. 8.4). Vielfache von Klassen und Methoden habe ich durch Set Fragments spezifiziert. Das Set Fragment strategies definiert eine beliebige Anzahl von ConcreteStrategy-Klassen samt zugehöriger concreteAlgorithm-Operationen und Entwurfsaufgaben an. Das Set Fragment operations gibt eine beliebige Anzahl der Vorkommen von request-Operationen zusammen mit je einer zugehörigen abstrakten algorithm-Operation in der Strategy-Klasse sowie der concreteAlgorithm-Operation in allen Strategy-Unterklassen an.

Varianten

Im Gegensatz zu den Set Fragments in der Spezifikation des Observer-Musters überschneiden sich bei dieser Spezifikation die Set Fragments und sind somit nicht unabhängig voneinander. Der Inhalt eines Set Fragments wird nur als Ganzes vervielfältigt. Das heißt, dass mit jeder zusätzlichen algorithm-Operation auch eine zusätzliche Implementierung dieser Operation in einer ConcreteStrategy-Klasse vorhanden sein muss sowie eine zusätzliche request-Operation, welche ihr Verhalten an die algorithm-Operation delegiert. Gleichzeitig kommen mit jeder zusätzlichen ConcreteStrategy-Klasse auch die concreteAlgorithm-Operationen hinzu, welche sämtliche algorithm-Operationen implementieren.

überschneidende Set Fragments

Auch wenn dieser Zusammenhang seine Komplexität mit sich bringt und eine gewisse Vorstellungskraft verlangt, ermöglicht die gewählte Notation mit Set Fragments eine präzise und kompakte Spezifikation dieses Zusammenhangs. Im Vergleich zur Entwurfsskizze, in welcher sechs algorithmOperation-Operationen dargestellt sind, reduziert sich die Anzahl dieser Operationen in der Musterspezifikation

komplex vs. kompakt

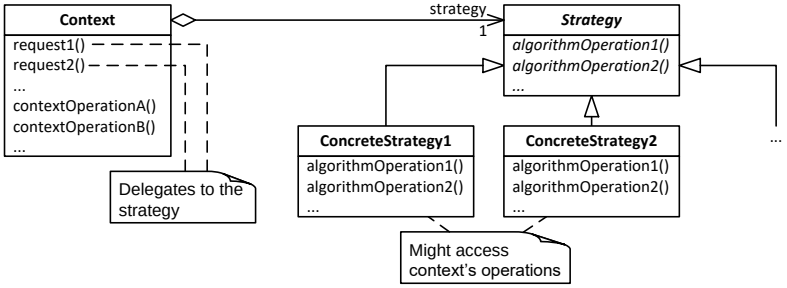


Abbildung 8.3: Grobe Idee der Entwurfslösung hinter dem Strategy-Muster

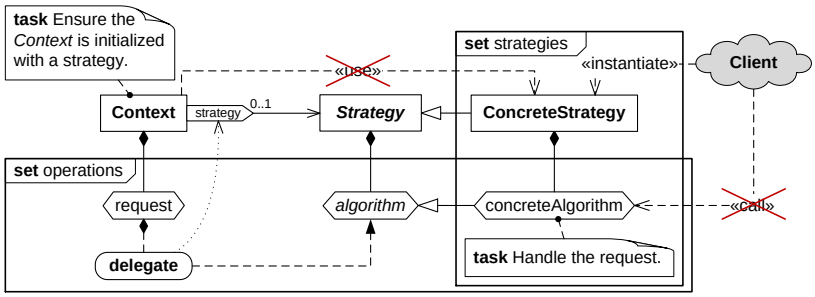


Abbildung 8.4: Spezifikation des GoF-Entwurfsmusters „Strategy“

auf nur zwei Operationen, algorithm und concreteAlgorithm. Analog verhält es sich bei den request-Operationen und den ConcreteStrategy-Klassen.

Die Gang of Four hat in ihrem OMT-Klassendiagramm zum Strategy-Muster die Schnittstelle der Context-Klasse mit einer ContextInterface-Methode angedeutet. In Wirklichkeit besteht diese Schnittstelle aus beliebig vielen Methoden, was ich in Abb. 8.3 als eine beliebige Anzahl von contextOperation-Methoden dargestellt habe. Laut Musterbeschreibung kann eine Strategie über diese optionale Schnittstelle auf Kontextinformationen zugreifen. Da der Strategie-Algorithmus vom Anwendungsfall abhängt, sind meiner Meinung nach die Methoden in dieser Schnittstelle für die Spezifikation des Strategy-Musters weitestgehend irrelevant. Diese Information kann weder für die Musteranwendung präzisiert, noch als Eigenschaft nach einer Musteranwendung automatisch geprüft werden. Darum habe ich die contextOperation-Methoden in meiner Spezifikation weggelassen.

für Entwurfslösung irrelevante Teile weglassen

Einen Teil des Verhaltens, nämlich die Delegation an die Strategie, habe ich jedoch explizit durch eine delegate-Aktion spezifiziert. Diese unterscheidet sich kaum von einer redirect-Aktion wie sie in der Spezifikation des Musters Observer vorkommt (vgl. Abb. 8.2, S. 171). Laut Spezifikation des Musters Strategy delegiert die request-Operation ihr gesamtes Verhalten an die algorithm-Operation und übergibt dabei sämtliche Argumente an die aufgerufene algorithm-Operation. Der Aufruf erfolgt auf dem Strategy-Objekt, welches über die strategy-Referenz erreichbar ist. Im Gegensatz zu einer redirect-Aktion grenzt eine delegate-Aktion ein wie mit einem eventuell vorhanden Rückgabewert umgegangen wird. Sollte die algorithm-Operation einen Wert als Ergebnis zurückliefern, wird dieser unverändert als Ergebnis der request-Operation zurückgegeben. Wie delegiert wird, ist in der Spezifikation (Abb. 8.4) also präziser ausgedrückt als in der Entwurfsskizze (Abb. 8.3).

Verhalten präzisiert

Wie bei der Spezifikation des Observer-Musters werden auch bei der Strategy-

Musterspezifikation einige Entwurfsaufgaben spezifiziert, um Entwickler bei der Anwendung des Musters zu leiten. So wird z.B. in einer Entwurfsaufgabe spezifiziert, dass der Kontext mit einer Strategie initialisiert werden muss, was leicht vergessen werden könnte.

Entwurfsaufgaben als Hinweise

Neben zu implementierenden Entwurfsteilen werden auch Rahmenbedingungen für die Musterimplementierung spezifiziert. Zum einen wird gefordert, dass der Kontext von konkreten Strategie-Implementierungen unabhängig bleiben soll. Zum anderen wird spezifiziert, dass ein anderer Teil des Softwaresystems – ausgedrückt durch den Environment-Knoten Client – die konkreten Strategien instanziieren muss, jedoch nicht ihre Operationen direkt aufrufen darf. Durch diese zusätzlichen Bedingungen wird die Intention des Musters zumindest teilweise festgehalten und könnte bei einer Musterimplementierung auf Einhaltung geprüft werden.

Kopplungskanten erfassen Intention

Abstract Factory

Die Lösungsidee zu dem Entwurfsmuster Abstract Factory [GHJV95, S. 87 ff.] habe ich in der Abb. 8.5 skizziert. Das Muster beschreibt wie zu einer Gruppe von Klassen, den Produkten, beliebig viele zueinander passende Implementierungen, die Produktfamilien, instanziiert werden können. Jede der Produktfamilien wird über eine Fabrikklasse instanziiert. Die verschiedenen Produktarten werden durch eine beliebige Anzahl von AbstractProduct-Klassen definiert. Jede Fabrikklasse erhält zu jeder Produktart eine zugehörige Operation zur Erzeugung des Produkts, z.B. eine Operation createProductA() für die Produktart AbstractProductA. Außerdem gibt es beliebig viele konkrete Implementierungen der Produktarten. Diese werden in Produktfamilien zusammengefasst und durch je eine zugehörige Fabrik instanziiert, z.B. erzeugt die ConcreteFactory1-Klasse Instanzen sämtlicher Klassen der Produktfamilie bestehend aus ProductA1 und ProductB1.

das Muster

Bei Anwendung dieses Musters können die Anzahlen von Produktarten und der Produktfamilien und damit auch die der zugehörigen Fabrikklassen beliebig gewählt werden. Die Anzahl der die Produkte erzeugenden Operationen hängt aber stark von den anderen beiden ab. In der Entwurfsskizze wird das exemplarisch durch je zwei Produktarten und Produktfamilien ausgedrückt, was den Entwurf in der Entwurfsskizze bereits relativ umfangreich und komplex macht. Der Zusammenhang kann in der Skizze nur erahnt werden und kann nur im das Muster beschreibenden Text genauer erläutert werden.

abhängige Variationspunkte

Bei diesem Muster zeigt sich die Stärke von Set Fragments in Musterspezifikationen ganz besonders. Bei der Spezifikation des Musters (siehe Abb. 8.6) muss jede Rolle der Entwurfslösung, hier jede abstrakte und jede konkrete Klasse und Methode, nur je ein Mal spezifiziert werden. Die Spezifikation wird dadurch deutlich kompakter und übersichtlicher. Alle Produktarten werden in dem Set Fragment products, alle Produktfamilien in dem Set Fragment product families spezifiziert. Es muss nur eine abstrakte Product-Klasse und nur eine konkrete Unterklasse davon, ConreteProduct, spezifiziert werden. Das Set Fragment products drückt aus, dass diese beiden Klassen in Kombination mit den sie erzeugenden Operationen create und concreteCreate bei einer Musteranwendung beliebig häufig vorkommen können. Analog dazu besagt das Set Fragment product families, dass es bei einer

Varianten kompakt beschrieben

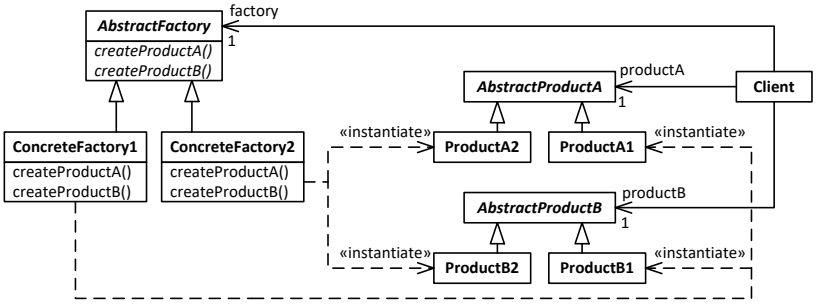


Abbildung 8.5: Grobe Idee der Entwurfslösung hinter dem Abstract-Factory-Muster

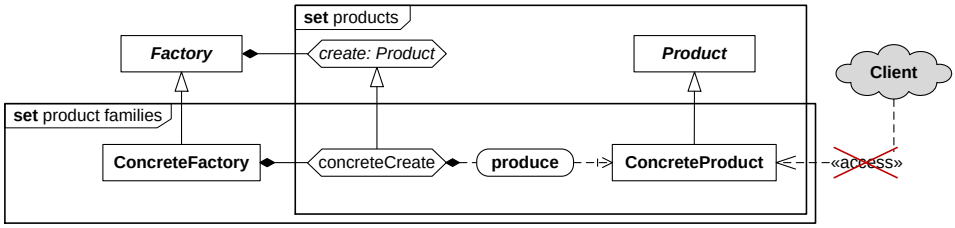


Abbildung 8.6: Spezifikation des GoF-Entwurfsmusters „Abstract Factory“

Abhängigkeiten der Variationspunkte verdeutlicht

Musteranwendung beliebig viele Produktfamilien bestehend aus eine ConcreteFactory-Klasse, den concreteCreate-Operationen sowie den ConcreteProduct-Klassen geben kann. Die Set Fragments machen die Spezifikation also nicht nur kompakter, sondern drücken vor allem präzise aus, wie Teile des Entwurfs bei einer Musteranwendung vervielfältigt werden können und insbesondere, welche Teile nur zusammen mit anderen als Ganzes vervielfältigt werden müssen.

Verhalten

Das Instanziieren einer ConcreteProduct-Klasse und das Zurückgeben der erzeugten Instanz als Ergebnis wird durch die produce-Aktion ausgedrückt.

Signatur nur einmalig angeben

Da bereits die abstrakte Operation create einen Rückgabetyp Product definiert, muss bei den diese Operation implementierenden konkreten Operationen concreteCreate kein Rückgabetyp mehr spezifiziert werden. Er wird aufgrund der Spezialisierungsbeziehung zwischen create und concreteCreate zusammen mit der Signatur von create übernommen.

ungewollte Kopplung verdeutlichen

Wie bei den vorhergehenden Musterspezifikationen wird auch hier eine Bedingung zur einer ungewollten Kopplung spezifiziert. Bis auf die konkreten Fabrikklassen soll keine Klasse außerhalb des Musters (ausgedrückt durch den Environment-Knoten Client) auf die ConcreteProduct-Klassen zugreifen, also diese Klassen nicht instanziieren, nicht auf Attribute der Klasse zugreifen und keine Operation der Klasse aufrufen (siehe Kopplungsarten in Abb. 3.39, S. 72).

Facade

das Muster

Die Entwurfslösung zu dem Entwurfsmuster Facade [GHJV95, S. 185 ff.] habe ich in der Abb. 8.7 in Anlehnung an die Darstellung der Gang of Four skizziert. Die Entwurfsidee besteht im Wesentlichen aus einer Klasse, der Fassade, welche eine Reihe von Operationen bereitstellt, welche das Benutzen eines Subsystems vereinfachen und die Implementierung im Subsystem verbergen sollen. Die Operationen der Fassade delegieren ihr Verhalten an das Subsystem, indem Operationen der

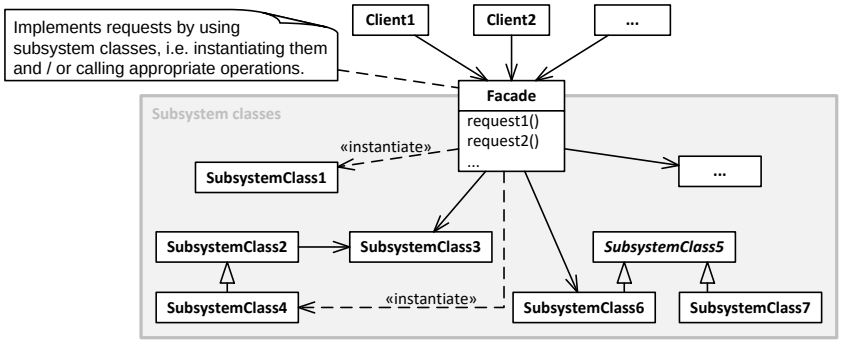


Abbildung 8.7: Grobe Idee der Entwurfslösung hinter dem Facade-Muster

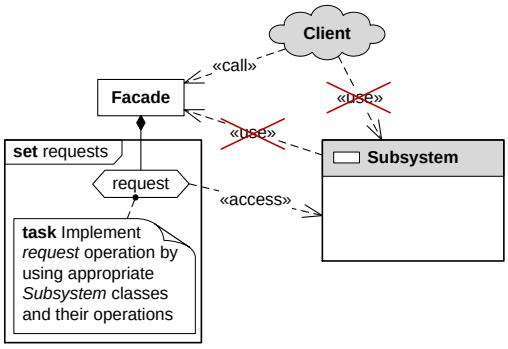


Abbildung 8.8: Spezifikation des GoF-Entwurfsmusters „Facade“

Klassen im Subsystem aufgerufen oder bei Bedarf Klassen im Subsystem instanziiert werden. Client-Klassen greifen nur über die Fassade auf das Subsystem zu.

Was genau die Operationen der Fassade tun, hängt extrem vom konkreten Anwendungsfall ab und kann in der Musterbeschreibung oder einer Spezifikation nicht weiter präzisiert werden. Bei der Spezifikation des Musters (Abb. 8.8) habe ich mich deswegen auf die Fassade selbst und die Kopplungsrestriktionen fokussiert. Diese lassen sich hinreichend genau spezifizieren. Die Operationen der Fassade (Facade) habe ich mit der request-Operation in einem Set Fragment requests spezifiziert. Die Implementierung dieser Operationen wird nur durch eine Entwurfsaufgabe spezifiziert.

Verhalten unbekannt, Fokus auf Struktur

Die Klassen des Subsystems werden durch einen Subsystem-Knoten repräsentiert. Bei Anwendung des Musters werden diesem Knoten die konkreten Klassen des Subsystems zugeordnet, sodass die Bedingungen bzgl. der Kopplung geprüft werden können.

Subsystem = Gruppe von Klassen

Die Implementierung der request-Operationen verlangt irgendeine Art des Zugriffs auf das Subsystem, was ich durch eine access-Kopplungskante von der request-Operation zum Subsystem spezifiziert habe. Das Subsystem soll nur über die Fassade verwendet werden. Den oder die Nutzer des Subsystems habe ich mit einem Environment-Knoten Client beschrieben. Die durchgestrichene use-Kopplungskante von Client zum Subsystem drückt den Verbot sämtlicher Abhängigkeiten zum Subsystem aus. Stattdessen sollen Clients die Fassade nutzen, was ich durch die call-Kopplungskante vom Client zur Fassade ausgedrückt habe. Die Subsystemklassen sollen laut Musterbeschreibung nichts von der Fassade wissen.

diverse gewollte & ungewollte Abhängigkeiten erfasst

Das habe ich durch die durchgestrichene use-Kopplungskante vom Subsystem zur Facade-Klasse ausgedrückt.

Auch wenn bei diesem Muster nicht viel über einen konkreten Softwareentwurf ausgesagt werden kann (konkrete Klassen und Operationen nicht bekannt), lässt sich ein Teil des Musters und seiner Intention relativ genau in meiner Spezifikation ausdrücken. Insbesondere die Regeln zu gewollten und ungewollten Abhängigkeiten könnten nach Anwendung des Musters automatisch geprüft werden.

Validierbarkeit

8.1.2 Spezifikation anderer Muster

Um die Ausdrucksfähigkeit meiner Musterspezifikationssprache zu evaluieren, habe ich neben den Mustern der Gang of Four auch andere Muster untersucht und exemplarisch spezifiziert. Dazu gehören je zwei Architekturmuster, Entwurfsmuster und Idiome.

Architekturmuster

Abgrenzung zu Entwurfsmustern

Architekturmuster (oft auch Architekturstil genannt) werden auf einem höheren Abstraktionslevel beschrieben als die meisten der GoF-Entwurfsmuster. Architekturmuster beschreiben insbesondere Teilsysteme und ihre Abhängigkeiten untereinander.

MVC-Muster

Eines der bekanntesten Architekturmuster ist Model-View-Controller (MVC) [BMR+96, Fow02]. Dieses habe ich spezifiziert wie in der Abb. 8.9 dargestellt. Die Spezifikation beschreibt, dass ein Softwaresystem u.a. aus den drei Teilsystemen Model, View und Controller besteht.

Teilsysteme & diverse Abhängigkeiten erfasst

Was sich hinter den jeweiligen Teilsystemen genau verbirgt, lässt sich in meiner Spezifikationssprache nicht ausdrücken. Die gewollten und ungewollten Abhängigkeiten lassen sich jedoch beschreiben. So habe ich spezifiziert, dass die Klassen im Subsystem Model weder Abhängigkeiten zum Subsystem View noch zum Subsystem Controller haben sollen (durchgestrichene use-Kopplungskanten). Analog dazu soll das Subsystem View keine Abhängigkeiten zum Subsystem Controller haben. Auf das Subsystem Model soll die View nur lesend, aber nicht schreibend zugreifen (read- und write-Kopplungskanten). Zur Instanziierung und Verknüpfung der Subsysteme Model und View soll das Subsystem Controller auf die beiden anderen Subsysteme Zugriff haben (access-Kopplungskanten). Außer der Klassen im Subsystem Controller soll keine Klasse schreibend auf Model-Klassen zugreifen, was ich durch eine durchgestrichene write-Kopplungskante vom Environment-Knoten System zum Subsystem Model spezifiziert habe.

Schichten als Muster

Analog dazu lässt sich auch eine Schichtenarchitektur [BMR+96], bzw. eine Ausprägung davon spezifizieren (Abb. 8.10). Hier werden mehrere Architekturschichten als Subsysteme spezifiziert. Jede Architekturschicht darf nur von der direkt darunter liegenden Schicht abhängig sein, aber nicht von einer darüber liegenden. Diese Regeln lassen sich gut mit Kopplungskanten ausdrücken. Die Schichten können sogar hierarchisch organisiert werden, indem Subsysteme ineinander geschachtelt werden (z.B. wie in Abb. B.29, S. 300).

Bei der Schichtenarchitektur bzw. dem Layers-Architekturstil wird eigentlich nicht von bestimmten Architekturschichten gesprochen wie in meiner Musterspezi-

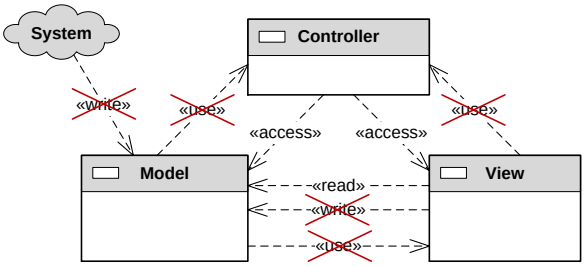


Abbildung 8.9: Spezifikation des Architekturmusters „Model-View-Controller“ (MVC) [BMR+96, Fow02]

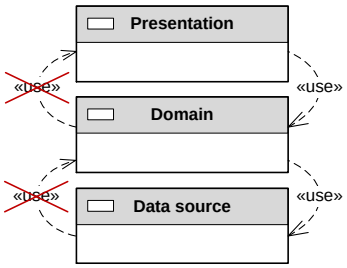


Abbildung 8.10: Spezifikation einer Ausprägung des „Layers“-Architekturstils [BMR+96] nach Fowler [Fow02, Kap. 8]

fikation, bei der ich mich auf die Schichtenarchitektur nach Fowler beziehe [Fow02, Kap. 8]. Die dazu nötige Verallgemeinerung (eine beliebige Anzahl von Schichten, mit Regeln zu ihren Abhängigkeiten und ohne ihrer Aufzählung / Nennung) lässt sich in meiner Spezifikationssprache nicht ausdrücken. Würde man eine beliebige Menge von Schichten durch ein Set Fragment und ein darin platziertes Subsystem (die Schicht) spezifizieren, so ließen sich nur Abhängigkeiten einer Schicht zu sich selbst ausdrücken (es gäbe nur einen Subsystem-Knoten). Darum müssen die Schichten wie in der Abb. 8.10 explizit genannt / aufgezählt werden, auch wenn es sich hierbei nur um eine von vielen möglichen Schichtenarchitekturen handelt. In diesem Fall muss man also für jede Variante der Schichtenarchitekturen eine eigene Musterspezifikation erstellen.

variable Anzahl von Schichten nicht ausdrückbar

Entwurfsmuster

Neben den GoF-Entwurfsmustern lassen sich in meiner Spezifikationssprache auch andere objektorientierte Entwurfsmuster beschreiben. So lässt sich z.B. das Muster Data Transfer Object (DTO) [Fow02, S. 401 ff.] wie in der Abb. 8.11 spezifizieren. Hierbei werden Domänenobjekte (DomainObject) in sogenannte Data Transfer Ob-

DTO-Muster

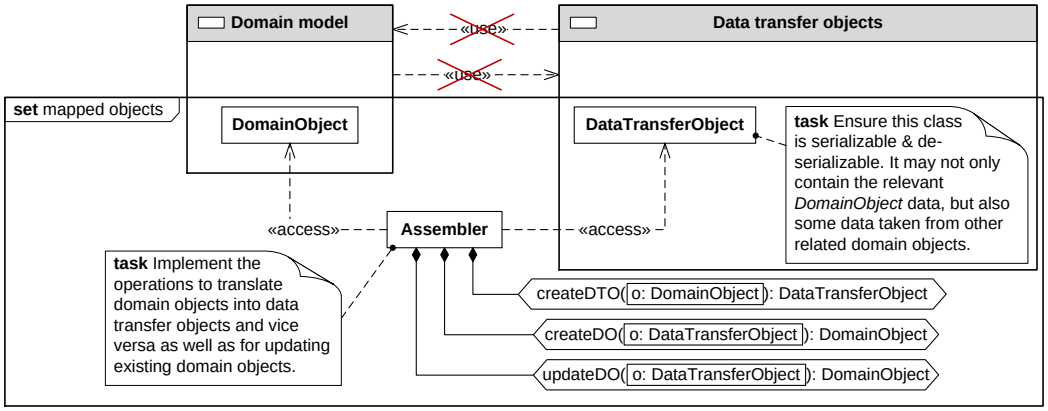


Abbildung 8.11: Spezifikation des Enterprise Application Architecture Patterns „Data Transfer Object“ (DTO) [Fow02]

jects (DataTransferObject) übersetzt und umgekehrt. Die Übersetzung übernimmt eine Assembler-Klasse, welche die Übersetzungsoperationen für beide Richtungen und eine Operation zur Aktualisierung der Attributwerte eines Domänenobjekts bereitstellt. Die Domänenobjekte gehören zu einem Subsystem namens Domain model, die Data Transfer Objects gehören zu einem weiteren Subsystem. Beide sollen voneinander unabhängig sein, was durch entsprechende Kopplungskanten ausgedrückt wird. Es gibt beliebig viele Domänenobjekte, zu welchen je ein Data Transfer Object und ein Assembler gehört. Darum werden diese drei Klassen in einem Set Fragment mapped objects spezifiziert.

Layer-Supertype-Muster

Ein weiteres von mir spezifiziertes Entwurfsmuster aus dem gleichen Buch ist Layer Supertype [Fow02, S. 475]. Die Spezifikation dieses Musters ist im Anhang in der Abb. B.30 (S. 300) zu finden.

Grenzen durch Sprachumfang

Die hier genannten Entwurfsmuster lassen sich gut in meiner Musterspezifikationssprache beschreiben. Das gilt insbesondere für Entwurfsmuster, welche im Wesentlichen Klassenstrukturen – also Klassen, Methoden, Assoziationen und Vererbungsbeziehungen – beschreiben. Es gibt jedoch auch Entwurfsmuster, welche sich in meiner Musterspezifikationssprache nicht oder nicht gut spezifizieren lassen. In den meisten Fällen liegt das entweder an der zu vagen, nicht formalisierbaren Beschreibung eines groben Softwareentwurfs oder an fehlenden Sprachkonstrukten in meiner Spezifikationssprache, z.B. zur Beschreibung parallelen Verhaltens, der Zuordnung von Softwareteilen zu Rechnerknoten (Deployment-Sicht) oder zur Repräsentation von Datenbanktabellen (z.B. bei OR-Mapping-Mustern[3]). Während sich Ersteres kaum ändern lassen wird, können fehlende Sprachkonstrukte in der Musterspezifikationssprache durch Erweiterung der DAL ergänzt werden (siehe Abschn. 3.6, S. 75).

Idiome

Abgrenzung zu Entwurfsmustern

Idiome sind Entwurfs- oder Programmiermuster auf niedrigem Abstraktionslevel. Sie waren nicht im Fokus meiner Arbeit, werden aber der Vollständigkeit halber mitbetrachtet. Darum habe ich zwei Idiome exemplarisch in meiner Musterspezifikationssprache erfasst. Dabei handelt es sich um SmallTalk Best Practice Patterns [Bec96], welche unter anderem in der Implementierung des JUnit-Frameworks verwendet wurden [Gam01].

Collecting-Parameter-Muster

Das Muster Collecting Parameter [Bec96] beschreibt einen Parameter einer Methode, welcher zum Sammeln von Zwischenergebnissen mehrerer aufeinander folgender Methodenaufrufe in einem bei jedem Aufruf weitergereichten Objekt genutzt wird. Das Muster wird anhand von SmallTalk-Programmbeispielen erläutert, lässt sich aber auch auf Java oder andere Programmiersprachen übertragen.

Struktur erfasst

Eine von Programmiersprachen unabhängige Spezifikation des Musters ist in der Abbilding 8.12 dargestellt. Das Ergebnis sowie sämtliche Zwischenergebnisse werden in einem Result-Objekt festgehalten. Eine Klasse ResultProducer stellt die Operation request und beliebig viele Operationen contributeToRequest bereit. Die request-Operation erzeugt das Ergebnis und liefert es zurück. Darum hat sie den Rückgabetyp Result. Die contributeToResult-Operation erhält ein Result-Objekt als

[3] Muster für objektrelationale (OR) Abbildungen zwischen OO-Klassen und Datenbanktabellen, z.B. Single Table Inheritance oder Class Table Inheritance [Fow02]

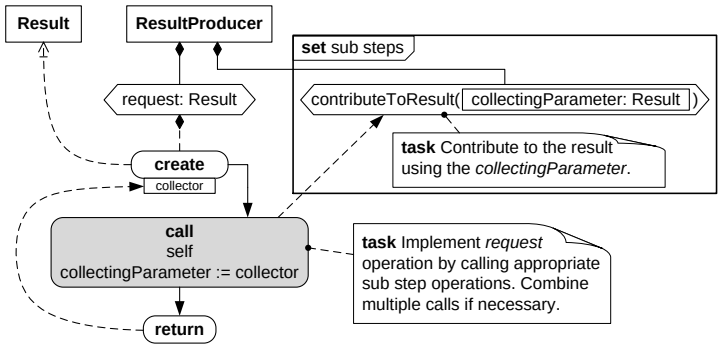


Abbildung 8.12: Spezifikation des SmallTalk Best Practice Patterns „Collecting Parameter“ [Bec96]

Parameter und befüllt dieses in seiner vom Anwendungsfall abhängigen Methodenimplementierung.

Die request-Operation enthält drei aufeinander folgende Aktionen. Die erste ist eine create-Aktion. Sie instanziiert die Klasse Result und stellt das erzeugte Objekt in der Variable collector für die folgenden Aktionen bereit. Die nächste Aktion ist eine call-Aktion, welche eine der contributeToResult-Operationen aufruft und das zuvor erzeugte Result-Objekt als Argument übergibt. Da der Aufruf der contributeToResult-Operation auf demselben Objekt erfolgt wie die aufgerufene request-Operation, wird als Zielobjekt self (in Java `this`) angegeben. Da zum Zeitpunkt der Musterspezifikation nicht bekannt ist, welche der contributeToResult-Operationen aufgerufen werden soll, und in vielen Fällen sogar eine Folge solcher Aufrufe nötig sein wird, wurde die call-Aktion als nicht generierbar markiert (grau hinterlegt). Eine entsprechende Entwurfsaufgabe weist auf diese Situation hin. Die letzte der drei Aktionen ist eine return-Aktion. Diese gibt das zuvor erzeugte und nun befüllte Result-Objekt als Ergebnis des request-Operationsaufrufs zurück.

Verhalten teilweise erfasst

Diese Spezifikation sowie die des Musters Pluggable Selector (siehe Abb. B.33, S. 301) zeigen, dass sich auch Idiome mit meiner Musterspezifikationssprache beschreiben lassen. Hierbei zeigt sich jedoch auch, dass gerade beim unvollständig beschreibbaren Verhalten – hier der nicht näher beschreibbare Aufruf einer der contributeToResult-Operationen – Einschränkungen bei der Spezifizierbarkeit von Mustern entstehen. Es stellt sich die Frage wie präzise Muster spezifiziert werden können und sollen, damit sie automatisiert zur Anwendung gebracht werden können und Anwendungsstellen geprüft werden können. Sollte trotz präziser Spezifikation keine Werkzeugunterstützung bei der Anwendung oder Validierung möglich sein, ist der Nutzen der detaillierten Spezifikation fraglich. Der Nutzen der detaillierten Spezifikation der hier betrachteten Idiome ist im Verhältnis zum Aufwand für das Erstellen der Spezifikation zumindest eingeschränkt.

begrenzter Nutzen

8.1.3 Fazit

Die 34 erstellten Musterspezifikationen (siehe Anhang B) und insb. die in diesem Kapitel vorgestellten und erläuterten Spezifikationen zeigen, dass sich mit der Musterspezifikationssprache zahlreiche Muster hinreichend präzise und vollständig spezifizieren lassen. Mit der vorgestellten Sprache lassen sich sowohl die Struktur

Sprache für betrachtete OO-Muster ausreichend

der durch Entwurfsmuster beschriebenen Entwurfslösung als auch ein Teil des zugehörigen Verhaltens beschreiben. Mit Hilfe von Sprachkonstrukten wie den Set Fragments lassen sich außerdem zahlreiche Varianten von Softwareentwürfen beschreiben. Mit Hilfe von Kopplungsrestriktionen lassen sich zusätzlich Bedingungen zu gewollten und ungewollten Abhängigkeiten zwischen Entwurfsteilen ausdrücken und so die Intention eines Musters besser erfassen als ausschließlich mit der Spezifikation der Klassenstruktur. Trotz einiger im Folgenden beschriebener Einschränkungen eignet sich die Sprache zur Beschreibung der im Rahmen dieser Arbeit spezifizierten Muster.

inhärente Vagheit von Mustern unumgehbar

Dennoch bleibt ein bei Architektur- und Entwurfsmustern sowie Idiomen inhärentes Problem der ungenauen, unvollständigen Beschreibung von vielen möglichen Implementierungen (und vieler Entwurfsvarianten) eines Musters. Dadurch lassen sich Muster nur bis zu einem bestimmten Grad präzise beschreiben oder spezifizieren. Je konkreter ein Muster beschrieben werden kann, desto einfacher lässt es sich in eine konkrete Entwurfslösung überführen, desto konkreter lässt sich das Muster spezifizieren, die Lösung lässt sich dann jedoch auch in umso weniger Fällen einsetzen (implementieren). Darum werden die meisten Muster eher vage, dafür aber mit vielen Beispielen für mögliche Implementierungen und ausführlichen Erläuterungen der Intention beschrieben, um so die Einsetzbarkeit der Muster nicht mehr einzuschränken als nötig. Diese Ungenauigkeit wiederum schränkt die Spezifizierbarkeit der Muster deutlich ein und lässt sich durch die Konzeption einer Musterspezifikationssprache nicht umgehen (man kann nur spezifizieren, was bekannt ist).

Spezifizierbare Teile eines Musters

Der Kern der in einem Muster beschriebenen Entwurfslösung konnte in allen erstellten Musterspezifikationen weitestgehend erfasst werden.

Klassenstruktur, wenn bekannt

Dabei lässt sich vor allem die Klassenstruktur (Klassen, Assoziationen, Vererbungsbeziehungen, Methoden, Attribute, etc.) gut beschreiben, sofern sie im zugehörigen Muster eindeutig und vollständig beschrieben wurde. Zum Beispiel sind bis auf die Klasse Client alle Teile des Entwurfs im Muster Abstract Factory eindeutig beschrieben und konnten ebenso präzise in die Spezifikation übernommen werden (siehe Abb. 8.6, S. 176). Bei dem Observer-Muster dagegen (siehe Abb. 8.2, S. 171) ist aufgrund der Abhängigkeit vom konkreten Anwendungsfall aus der Musterbeschreibung nicht eindeutig erkennbar, welche ConcreteObserver-Klasse mit welcher ConcreteSubject-Klasse über eine subject-Referenz verknüpft werden soll. Diese fehlende Eindeutigkeit resultiert darin, dass zumindest die subject-Referenz bei einer Musteranwendung nicht automatisch erzeugt werden kann.

keine Objektstruktur

Sobald in Mustern Aussagen über Objektstrukturen gemacht werden, wie z.B. bei dem Muster Chain of Responsibility [GHJV95, S. 223 ff.], wo von einer Kette von Objekten gesprochen wird, stößt die Musterspezifikationssprache an ihre Grenzen. Zur Laufzeit zu erzeugende Objektstrukturen oder Regeln für solche Strukturen lassen sich nicht beschreiben.

vordefinierte Aktionen

Verhalten wird in Musterspezifikationen nur sehr eingeschränkt in Form von einigen wenigen, vordefinierten Aktionen erfasst. Für die erstellen Musterspezifikationen waren die vordefinierten Aktionen ausreichend. Bei einigen Mustern (z.B.

Collecting Parameter, Abb. 8.12, S. 181) ist das Verhalten jedoch so vage durch die Muster beschrieben, dass die Spezifikationen sich kaum für eine automatische Musteranwendung eignen. In diesen Fällen könnte das spezifizierte Verhalten für eine Konsistenzprüfung von Musterimplementierungen genutzt werden.

Entkopplung & Kopplung

Weitere Konsistenzprüfungen werden durch Kopplungsregeln ermöglicht. Diese erfassen einen Teil der Intention hinter einem Muster, insb. kann erfasst werden, welche Teile des Entwurfs voneinander unabhängig gehalten werden sollen (z.B. bei Observer, Abb. 8.2, S. 171 und Facade, Abb. 8.8, S. 177).

Hinweise für Vages

Ergänzende, nicht eindeutig oder formal erfassbare Informationen können in Form von textuell und informell beschriebenen Entwurfsaufgaben erfasst werden. Diese können Entwickler bei der Musteranwendung anleiten, ersetzen jedoch nicht die umfangreichen Musterbeschreibungen in der Literatur.

begrenzter, erweiterbarer Sprachumfang

Die vorgestellte Sprache beschreibt Teile des Entwurfs durch objektorientierte Sprachkonstrukte, welche in der DAL, einem Teil der Musterspezifikationssprache, definiert sind. Dadurch ist die Ausdrucksfähigkeit der Sprache beschränkt auf die vordefinierten Sprachkonstrukte. Da die DAL sich um weitere Sprachkonstrukte erweitern lässt, könnte die Musterspezifikationssprache auch für andere Muster verwendet werden, indem sie z.B. um Komponenten, Rechnerknoten oder Konstrukte zur Beschreibung parallelen oder verzahnten Verhaltens erweitert wird.

Abdeckung von Entwurfs- und Implementierungsvarianten

Vielfache von Entwurfsteilen oft nützlich & kompakt

Durch den Einsatz von Set Fragments lassen sich Mehrfachvorkommen von zusammenhängenden Entwurfsteilen und damit zahlreiche Entwurfsvarianten gut beschreiben. Die gewählte Notation kann Spezifikationen wie die des Musters Abstract Factory erstaunlich kompakt machen (siehe Abb. 8.6, S. 176). Set Fragments haben sich bei den erstellten Spezifikationen als nützlich erwiesen. Mit Ausnahme von einer Spezifikation enthalten alle GoF-Musterspezifikationen mindestens ein Set Fragment, was ebenso für einige der anderen Spezifikationen gilt.

Varianten nicht direkt sichtbar

Obwohl durch Set Fragments die Spezifikationen kompakter werden, bringen sie eine gewisse Komplexität mit sich. Man muss sich die daraus ableitbaren Entwurfsvarianten vorstellen.

Überschneidung beherrschbar

Die Einschränkung, dass sich maximal zwei Set Fragments überschneiden dürfen, hat die Komplexität der Spezifikationen beschränkt, allerdings ohne erkennbare Einschränkungen bei der Ausdrucksfähigkeit. Ein höherer Grad an Überschneidung war bei den erstellten Spezifikationen nicht erforderlich und würde sowohl die Darstellung von Musterspezifikationen verkomplizieren als auch das Verstehen des spezifizierten Entwurfs erschweren.

transitive Beziehungen vergrößern Lösungsraum

Weitere Entwurfsvarianten werden in Musterspezifikationen dadurch beschrieben, dass Beziehungen in Musterspezifikationen sowohl auf direkte als auch auf indirekte Beziehungen in Musterimplementierungen abgebildet werden können. Das gibt Entwicklern weiteren Spielraum beim Anpassen der Entwurfslösung an einen konkreten Anwendungsfall.

weniger Details ⇒ mehr Varianten

Auch das Weglassen einiger Details in der Spezifikation, z.B. durch das Offenlassen, ob eine Klasse abstrakt oder konkret sein muss oder welche Parameter eine Operation haben muss, lässt Raum für Anpassungen an einen Anwendungsfall und erhöht die Anzahl erfasster Implementierungsvarianten.

Grenzen erfassbarer Varianten

Trotz der vielen Möglichkeiten, Entwurfsvarianten zu beschreiben, lassen sich einige Muster und die zugehörigen Entwurfsvarianten nicht allgemein genug oder zumindest nicht in nur einer Spezifikation ausdrücken. Zum Beispiel kann eine Schichtenarchitektur nicht spezifiziert werden, ohne bestimmte Schichten aufzuzählen (siehe Abb. 8.10, S. 179). Bei den Mustern Composite, Observer und State mussten aufgrund zu unterschiedlicher Entwurfsvarianten jeweils mehrere Musterspezifikationen erstellt werden (siehe Anhang B.1).

Eignung für unterschiedliche Entwurfsmuster

Sprache erweiterbar

Die Musterspezifikationssprache wurde mit der Wahl der DAL-Sprachkonstrukte (siehe Abschn. 3.4) auf Elemente objektorientierter Sprachen beschränkt, vor allem auf Klassenstrukturen, aber auch auf einige für Entwurfsmuster typische Interaktionen (z.B. Delegation oder Methodenaufrufe). Die DAL lässt sich prinzipiell um weitere Sprachkonstrukte erweitern, womit die Eignung der Musterspezifikationssprache auf nicht objektorientierte Muster oder auf andere Aspekte solcher Muster ausgedehnt werden könnte. Zum Beispiel können Muster für objektrelationale Abbildungen (OR-Mapping-Muster) oder parallele bzw. verzahnte Programme aufgrund fehlender Sprachkonstrukte bisher nicht spezifiziert werden.

nur zur Entwurfszeit Bekanntes

Da eine Musterspezifikation in meinem Ansatz Teile eines Softwareentwurfs beschreibt, kann damit (nach entsprechender Ergänzung fehlender Sprachkonstrukte) prinzipiell jede zur Entwurfszeit verfügbare Modellstruktur (Klassenstrukturmodelle, Verhaltensmodelle, Deployment-Modelle, etc.) beschrieben werden. Zur Laufzeit zu erzeugende Objektstrukturen wie sie z.B. bei dem Muster Chain of Responsibility [GHJV95] beschrieben werden lassen sich jedoch nicht in einer Musterspezifikation ausdrücken[4]. Stattdessen kann bestenfalls das Verhalten beschrieben werden, welches die gewünschten Objektstrukturen erzeugt.

fehlender Sprachumfang umgehbar

für Vages weniger geeignet

Die spezifizierten Muster konnten mit Hilfe der gewählten Sprachkonstrukte mit kleineren Einschränkungen komplett erfasst werden. Einschränkungen sind z.B. bei der Spezifikation des Musters Singleton (siehe Abb. B.5, S. 292) in Form von fehlenden Sprachkonstrukten für Konstruktoren, statische Attribute und Operationen sowie ihre Sichtbarkeiten zu beobachten. Die fehlenden Sprachkonstrukte wurden durch entsprechende Hinweise in Entwurfsaufgaben umgangen. Die meisten Einschränkungen bei der Musterspezifikation rührten jedoch daher, dass Muster bereits in ihrer informellen Beschreibung nicht weit genug konkretisiert werden konnten. Bei dem Singleton-Muster ist z.B. offen gelassen, wie dafür gesorgt werden soll, dass die Singleton-Klasse nur höchstens einmal instanziiert wird. Bei dem Facade-Muster (siehe Abb. 8.8, S. 177) ist ebenso offen, welches Verhalten die Operationen der Fassade implementieren sollen. Dies ist extrem vom Anwendungsfall abhängig und kann weder in einer informellen Musterbeschreibung, noch in einer formalen Musterspezifikation erfasst werden. Bei einigen Musterspezifikationen wie z.B. bei dem Flyweight-Muster (Abb. B.12, S. 294) deuten vergleichsweise viele Entwurfsaufgaben (Kommentare) auf eine beschränkte Eignung der Musterspezifikationssprache zur formalen Beschreibung der Muster hin.

[4] Bei dem Muster Chain of Responsibility wird eine Ketten-artige Objektstruktur verlangt. Die in der Musterspezifikation (Abb. B.14, S. 295) definierte Klassenstruktur lässt jedoch auch andere Objektstrukturen zu.

Die Spezifikationen einiger Architekturmuster und Idiome hat gezeigt, dass die Musterspezifikationssprache u.a. auch für einige solcher Muster geeignet ist. Insbesondere bei Idiomen sind jedoch der Nutzen ihrer Spezifikation und die Eignung der Musterspezifikationssprache eingeschränkt. Die beiden Idiom-Spezifikationen (Abb. B.32, B.33, S. 301) sind relativ komplex, bieten jedoch im Vergleich zum Aufwand für ihre Spezifikation und für eine Werkzeug-gestützte Anwendung der Muster einen relativ geringen Nutzen. Das Verhalten, welches einen wesentlichen Teil des Musters ausmacht, lässt sich bei Musteranwendung nicht automatisch herleiten und nur schwer auf Konsistenz mit der Musterspezifikation prüfen. Das Verhalten ist in den Mustern nur partiell beschrieben und lässt sich darum nur bedingt spezifizieren. Z.B. kann bei den genannten Musterspezifikationen bedingt durch die vage Musterbeschreibung zwar ausgedrückt werden, dass auf ein Attribut lesend zugegriffen wird, jedoch nicht wie. Es kann ausgedrückt werden, dass einige Methoden aufgerufen werden, aber nicht welche. Der Hauptnutzen der Spezifikation verbleibt in der Beschreibung der Klassenstruktur, welche in diesen Fällen meist sehr simpel ausfällt.

geringer Nutzen bei unklarem Verhalten

Notation

Zwecks Wiedererkennung, Einhaltung von Konventionen und einfacher Erlernbarkeit orientiert sich die Notation in Musterspezifikationen an der in der objektorientierten Softwareentwicklung weit verbreiteten und bekannten UML [OMG11a, OMG11b]. Typen bzw. Klassen und Vererbungsbeziehungen werden genauso wie in der UML dargestellt. Das gleiche gilt für die Darstellung abstrakter bzw. konkreter Klassen und Methoden sowie für Kardinalitäten von Assoziationen.

Notation von UML abgeleitet

Bedingt durch Set Fragments und die damit verbundene Darstellung von sich wiederholenden Entwurfsteilen musste jedoch von der UML-Notation deutlich abgewichen werden. Damit z.B. eine beliebige Anzahl von Methoden zu einer Klasse mit Set Fragments spezifiziert werden kann, mussten die Methoden als eigene Knoten abgebildet werden. Das gleiche gilt für Attribute, aber auch für Assoziationen (bzw. Referenzen). So wird abweichend von der UML die Kardinalität und der Rollenname einer Assoziation nicht auf der Seite der referenzierten Klasse, sondern auf der Seite der referenzierenden Klasse dargestellt. Damit bei Assoziationen (analog zu Klassen, Methoden und Attributen) dargestellt werden kann, ob daraus bei Musteranwendung automatisch Teile des Entwurfs abgeleitet werden können (Element wird grau hinterlegt, falls das nicht möglich ist), musste jede referenzierende Seite der Assoziationsdarstellung mit einem einfärbbaren Label (dargestellt als eigener Knoten) versehen werden.

notwendige Anpassungen der Notation

Aufgrund der gewählten Notation, Attribute, Methoden und Referenzen als eigene Knoten darzustellen, nimmt der Platzbedarf für ihre Darstellung im Vergleich zu UML-Klassendiagrammen zu. Die zusätzlichen Kanten zur Darstellung der Zugehörigkeit von Methoden und Attributen zur ihren Elternklassen erhöhen die Anzahl dargestellter Kanten und steigern damit die Komplexität der Darstellung.

komplexere Darstellung

Für die Darstellung der unterschiedlichen Kanten und Knoten wurden verschiedene Formen, Strich-Arten und Pfeilspitzen verwendet. Auch die Hintergrundfarbe wurde zur Unterscheidung von Eigenschaften genutzt. Diese Maßnahmen erhöhen die kognitive Effektivität der Sprache und orientieren sich an den Prin-

erhöhte kognitive Effektivität

zipien für effektive grafische Notation [Moo09]: semiotische Klarheit (eindeutige Zuordnung von semantischen Konstrukten zu grafischen Symbolen), perzeptuelle Unterscheidbarkeit (klar von einander unterscheidbare Symbole) und visuelle Ausdrucksfähigkeit (hohe Ausnutzung von visuellen Variablen wie Position, Form, Größe, Farbe, Helligkeit, Ausrichtung und Textur). Farben wurden in der Spezifikationssprache bis auf eine Ausnahme (durchgestrichene Kopplungskanten) nicht eingesetzt, weil diese Ausdrucksfähigkeit in der Visualisierung von Anwendungsstellen – einer Erweiterung der Notation von Musterspezifikationen für die Musteranwendungssicht (Abschn. 4.3) – zur Unterscheidung zusätzlicher Eigenschaften benötigt wird.

keine Vielfachen einer Referenz je Klasse

Die verwendete Notation für Assoziationen (bzw. Referenzen) bringt eine Einschränkung mit sich. Dadurch, dass Assoziationsenden in Form eines Knotens direkt an Klassen angeordnet werden und nicht analog zu Operationen und Attributen mit einer Zugehörigkeitskante (Containment) mit der Klasse verbunden werden, kann mit Set Fragments keine Menge von Referenzen in einer Klasse beschrieben werden wie z.B. Mengen von Operationen in einer Klasse beschrieben werden können. In den meisten der 34 erstellten Musterspezifikationen war diese Ausdrucksmöglichkeit unnötig. Bei der Spezifikation des Observer-Musters in Abb. 8.2 (S. 171) hätte diese Notation eine eindeutige Spezifikation der subject-Referenz ermöglicht, wo jeder ConcreteObserver für jedes ConcreteSubject eine Referenz erhalten hätte. (Die Referenz hätte in demselben Set Fragment wie die Klasse ConcreteSubject spezifiziert werden können, wäre damit eindeutig und könnte bei einer Musteranwendung vollständig generiert werden.) Ob diese Variante wirklich dem Muster entspricht, ist allerdings eher fraglich, da nicht jeder Beobachter jedes Subjekt beobachten können muss.

8.2 Werkzeug-gestützte Musteranwendung

Anwendungsbeispiele

Zur Einschätzung wie gut sich Muster mit meinem Verfahren anwenden lassen und wie Praxis-tauglich das vorgestellte Verfahren ist habe ich zahlreiche Musteranwendungen mit meinem Prototypen und den im Anhang B dokumentierten Musterspezifikationen vorgenommen (d.h. mehrere kleine Fallstudien durchgeführt). Im Folgenden stelle ich einige dieser Musteranwendungen vor und bespreche sie.

Vorgehen

Dabei untersuche ich wie flexibel Muster angewendet werden können, indem ich ein Muster in unterschiedlichen Ausgangssituationen anwende und eine Anwendungsstelle nachträglich dokumentiere oder erweitere. Zusätzlich betrachte ich diverse andere, verschiedenartige Muster, wende sie exemplarisch mit meinem Verfahren an und gehe auf die Besonderheiten bei ihrer Anwendung ein. Außerdem führe ich schrittweise mehrere Musteranwendungen in einem realistischen, Praxisnahen Beispiel durch und orientiere mich dabei an einer Dokumentation der Musteranwendungen in dem JUnit-Framework[5]. Abschließend beurteile ich die Eignung meines Ansatzes für den praktischen Einsatz.

[5] http://junit.org/

8.2.1 Eignung des Verfahrens für unterschiedliche Einsatzszenarien

durchgehendes Bsp.: Observer

Eine exemplarische Anwendung des Observer-Musters ist in Abschnitt 7.2 (S. 158 ff.) beschrieben. Hier zeigt sich, dass das Muster abhängig von bereits vor der Musteranwendung existierenden und den Musterrollen zugeordneten Entwurfsteilen im Klassendiagramm flexibel angewendet werden kann. Dabei wird sowohl das Klassenmodell vervollständigt als auch ein Verhaltensmodell erstellt, sofern es noch nicht existiert. Teile der Musterspezifikation, welche nicht eindeutig und vollständig spezifiziert werden können (diese sind in Spezifikationen grau hinterlegt), müssen bei der Musteranwendung manuell ergänzt werden. Ein nach der Musteranwendung erstelltes Modell der Anwendungsstelle kann zur Dokumentation der Musterimplementierung verwendet werden und könnte zur Validierung der Musterimplementierung nach Entwurfsanpassungen verwendet werden (was jedoch im vorliegenden Prototypen nicht vollständig implementiert wurde).

Gebotene Unterstützung

Anwendungsszenarien

Die Werkzeug-gestützte Anwendung von Mustern ist so konzipiert, dass sie in folgenden vier Szenarien eingesetzt werden kann (vgl. Szenarien in Abschn. 5.1). Anhand des Observer-Musters möchte ich auf die Eignung meines Ansatzes für den Einsatz in diesen Szenarien eingehen.

1. Eine Musterimplementierung und alle Korrespondenzen (bzw. Rollenzuordnungen, siehe Abb. 4.3, S. 83) vollständig neu generieren — alles generiert
2. Eine partielle Musterimplementierung durch Ergänzen fehlender Entwurfsteile und Korrespondenzen vervollständigen — Teile generiert
3. Eine vollständig existierende Musterimplementierung nachträglich um Korrespondenzen erweitern (den Musterrollen und Set Fragments zuordnen) — nur Korrespondenzen
4. Eine vollständig existierende Musterimplementierung nachträglich um zusätzliche Implementierungen von Set Fragments (Set-Fragment-Instanzen) und zugehörige Korrespondenzen erweitern — nachträgliche Anpassung

Szenario 1: Musterimplementierung komplett neu generieren

Musteranwendung vorbereitet

Das erste Szenario wird in der Abb. 8.13 (S. 188) behandelt. Im linken Teil des Bildes wird die Situation vor der Musteranwendung und nach der Festlegung zu verwendender Namen für die zu erzeugenden Entwurfselemente dargestellt. Das Klassendiagramm (oben links) ist bis auf die Markierung einer Anwendungsstelle völlig leer. Ein Modell der Anwendungsstelle wurde bei der Initialisierung der Musteranwendungsansicht (Mitte links) automatisch erzeugt. Darin sind die vom Entwickler vor der Musteranwendung festgelegten Namen für zu erzeugende Elemente bereits erfasst. Zur Veranschaulichung aller Ausprägungen der Set Fragments samt der vergebenen Namen wurde unten links im Bild die Visualisierung der jeweils zweiten Ausprägung der beiden Set Fragments ergänzt (normalerweise ist nur eine der Set-Fragment-Instanzen sichtbar).

nach autom. Musteranwendung

Im rechten Teil des Bildes wird die Situation nach der automatischen Musteranwendung dargestellt. Das Klassendiagramm (oben rechts) wurde mit allen automatisch erzeugbaren Entwurfselementen gefüllt und es wurde ein Story-Diagramm (Mitte rechts) zur Modellierung des Verhaltens der Methode firePropertyChange erstellt. Wie in der Musteranwendungsansicht (unten rechts) dargestellt, wurde

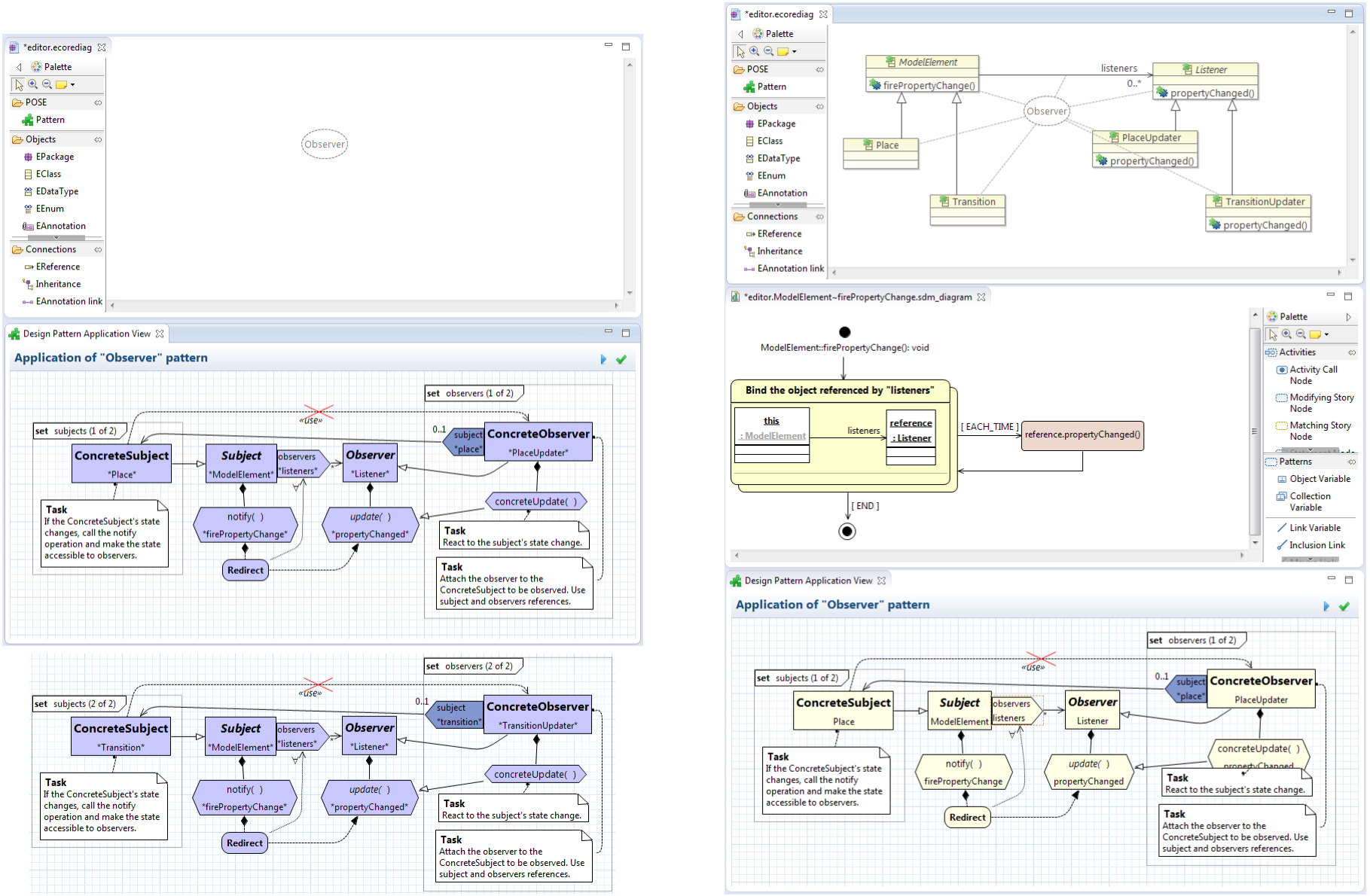


Abbildung 8.13: Vollständige Generierung einer Observer-Musterimplementierung. Links die Situation vor der Anwendung, rechts nach der automatischen Anwendung.

nur die subject-Referenz nicht in das Klassendiagramm übertragen, was hier allein an der fehlenden Eindeutigkeit der Spezifikation liegt (darum ist die Referenz als nicht generierbar markiert, vgl. Abb. 8.2, S. 171).

Machbarkeit gezeigt

Der vorgestellte Ansatz zur automatischen Musteranwendung ist also (mit Ausnahme der grundsätzlich nicht automatisch generierbaren Entwurfsteile) im ersten der vier Szenarien einsetzbar. Diese Eignung lässt sich ohne Einschränkungen auf alle Musterspezifikationen übertragen.

Szenario 2: Vervollständigen einer Musterimplementierung

6 Ausgangssituationen

Zur Betrachtung des zweiten Szenarios, eine Musterimplementierung durch Ergänzen fehlender Entwurfsteile zu vervollständigen, habe ich exemplarisch vier verschiedene Ausgangssituationen betrachtet und in diesen Fällen die automatische Musteranwendung angestoßen. Die vier Fälle sind in den Abb. 8.14 bis 8.17 jeweils mit den Situationen vor und nach der Musteranwendung dargestellt. Streng genommen stellt das Szenario 1 nur einen Spezialfall des Szenarios 2 dar und beschreibt damit einen fünften Fall. Ein sechster Anwendungsfall wurde bereits in Abschnitt 7.2 (S. 158 ff.) betrachtet.

Machbarkeit bei Klassen gezeigt, kleine Abstriche

In allen sechs Fällen war eine automatische Musteranwendung möglich. Eine kleine Einschränkung bei der automatischen Musteranwendung gibt es im Fall 4 (Abb. 8.17). Die concreteUpdate-Operation stellt laut Musterspezifikation eine Implementierung der abstrakten update-Operation dar. Die zugehörigen Methoden im Entwurf müssen also identische Methodensignaturen besitzen. In diesem Fall ist vor der Musteranwendung die schon existierende Methode propertyChanged der Rolle concreteUpdate zugeordnet, während der Rolle update noch keine Methode im Klassenmodell zugeordnet ist. Die Signatur der zu erzeugenden, der update-Rolle zugeordneten Methode kann hier nicht automatisch ermittelt werden, weil der Musteranwendungsalgorithmus die Methodensignaturen entlang der Vererbungshierarchie nur nach unten, jedoch nicht – wie hier nötig wäre – nach oben propagiert. Diese Einschränkung wurde umgangen, indem vor der Musteranwen-

Signatur wird nur nach unten propagiert

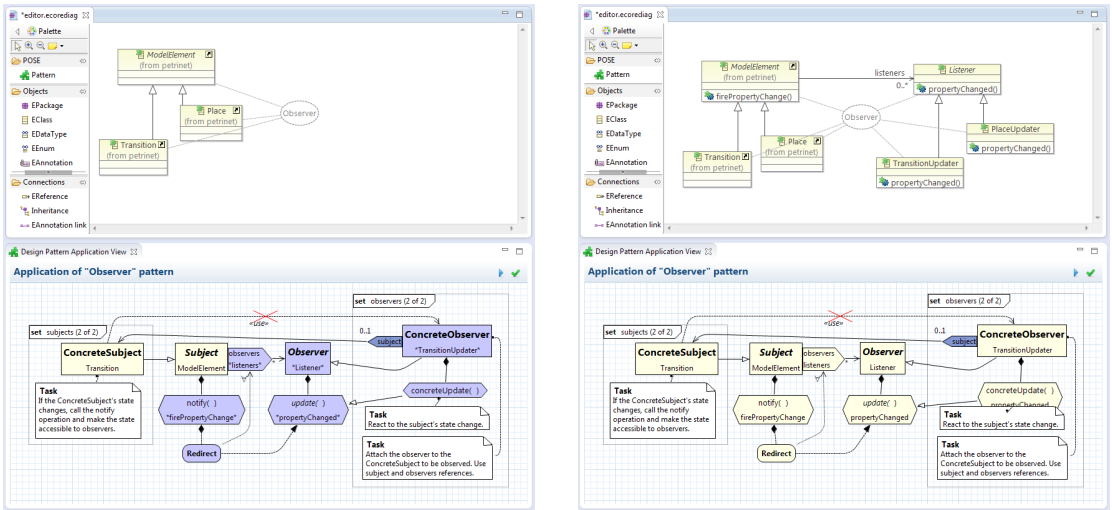


Abbildung 8.14: Vervollständigung einer Observer-Musterimplementierung – Fall 1

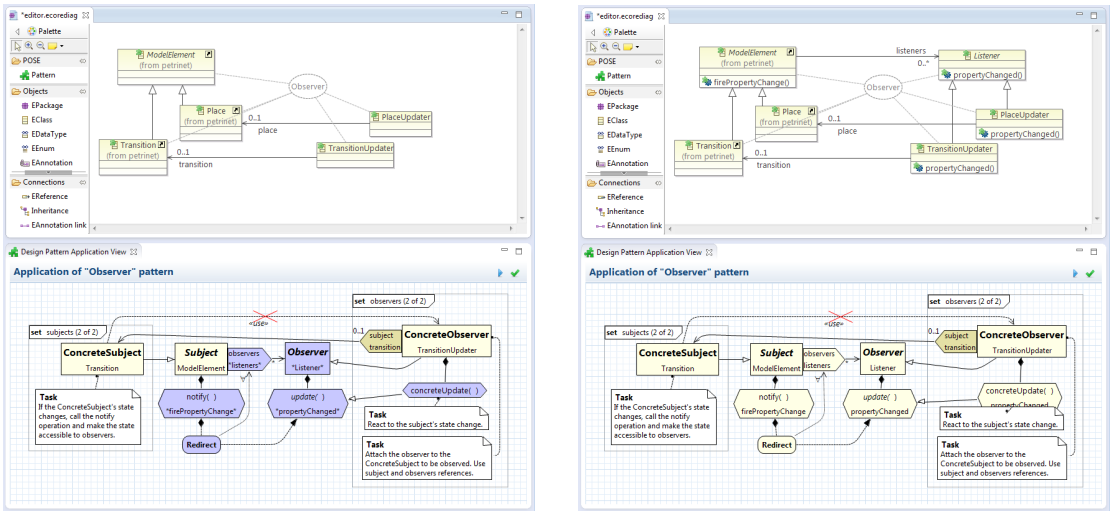


Abbildung 8.15: Vervollständigung einer Observer-Musterimplementierung – Fall 2

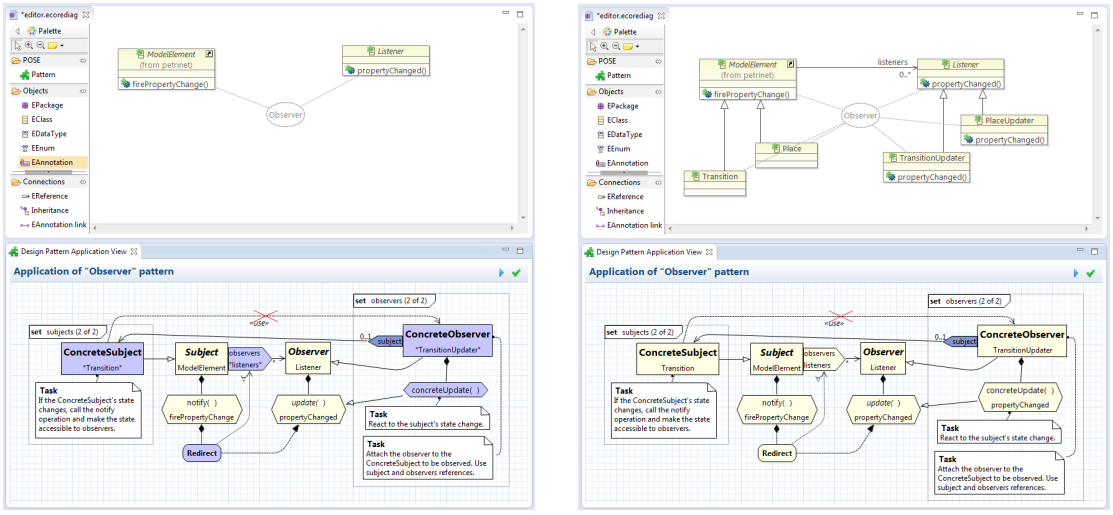


Abbildung 8.16: Vervollständigung einer Observer-Musterimplementierung – Fall 3

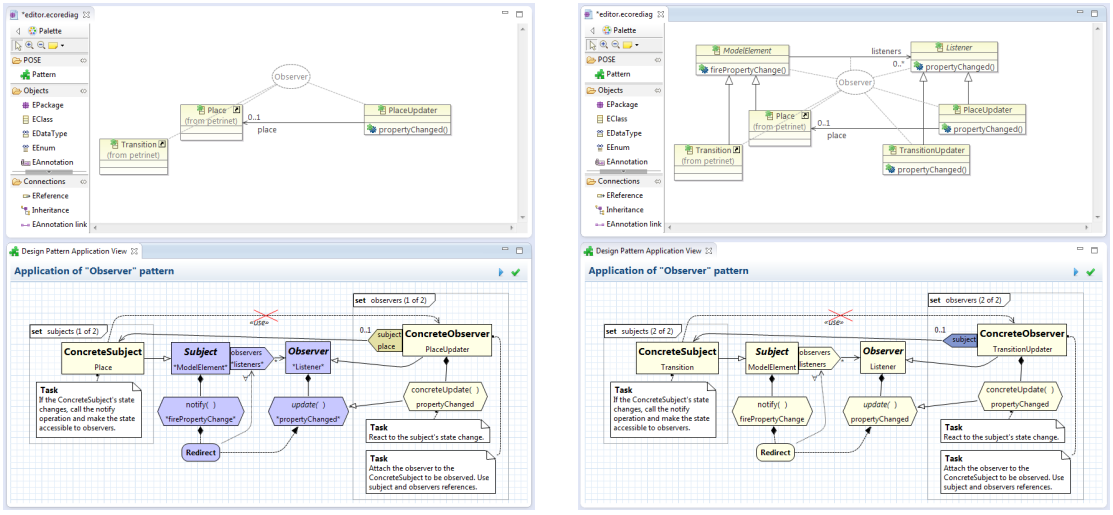


Abbildung 8.17: Vervollständigung einer Observer-Musterimplementierung – Fall 4

dung der Name für die zu erzeugende Methode zur Rolle update auf den Namen propertyChanged gesetzt wurde. So wird bei der Musteranwendung eine Methode mit passender Signatur (hier nur aus dem Methodennamen bestehend) erzeugt. Hätte die schon existierende propertyChanged-Methode der Klasse PlaceUpdater Parameter, müssten diese bei den generierten propertyChanged-Methoden in anderen Klassen nach der Musteranwendung manuell ergänzt werden. Bis auf diese waren keine Einschränkungen des Ansatzes bei diesem Szenario festzustellen.

kein Vervollständigen des Verhaltens

Das Vervollständigen partieller Verhaltensmodelle ist aufgrund der komplexen Korrespondenzen und der Übersetzung (siehe Anhang C.3.3, S. 328 ff.) konzeptionell schwierig und bisher nicht möglich. Bei den betrachteten Mustern und ihren Anwendungen kam diese Einschränkung jedoch nicht zum Tragen, weil eine Musteranwendung bei diesen Mustern typischerweise auf Klassenebene beginnt und zu diesem Zeitpunkt normalerweise noch keine Verhaltensmodelle existieren.

Szenario 3: Musterimplementierung um Korrespondenzen ergänzen

Das dritte Szenario, bei welchem eine bereits vollständige Implementierung eines Musters nachträglich durch Rollenzuordnung mit einer Musterspezifikation verknüpft wird, wird in den vier Teilen der Abb. 8.18 betrachtet. Oben links ist die Ausgangssituation dargestellt: ein Klassendiagramm ohne jegliche Verknüpfung zu einer Musterspezifikation. Trotz vollständig modellierter Implementierung des Observer-Musters im Klassendiagramm ist die Anwendungsstelle nicht direkt erkennbar und könnte leicht übersehen werden.

undokumentierte Musterimplementierung

Oben rechts im Bild wird die Situation nach Auswahl und Initialisierung einer Anwendungsstelle des Observer-Musters dargestellt. Zu diesem Zeitpunkt wurde noch keine Rollenzuordnung gemacht, was sowohl das Klassendiagramm als auch die Musteranwendungsansicht veranschaulichen.

Anwendungsstelle angelegt

Unten links im Bild werden das Klassendiagramm und die Musteranwendungsansicht nach Zuordnung fast aller hier sichtbarer Musterrollen dargestellt. Zu diesem Zeitpunkt existiert zu den beiden Set Fragments subjects und observer nur jeweils eine Ausprägung, was in der Musteranwendungsansicht durch die Labels „(1 of 1)“ ausgedrückt wird.

erste Rollenzuordnungen

Die redirect-Aktion kann allerdings den sie implementierenden Teilen in einem Story-Diagramm nicht nachträglich zugeordnet werden. Das gilt für sämtliche Aktionen und stellt eine Einschränkung des Verfahrens dar. Zum einen fehlt eine geeignete Möglichkeit in der Benutzungsschnittstelle der prototypischen Werkzeuge, zum anderen wäre eine solche Zuordnung zu kompliziert, um sie manuell durchzuführen. Es müssten relativ viele, teilweise nicht in konkreter Syntax der Story-Diagramme sichtbare Teile eines Story-Diagramm-Modells unter Verwendung von Tokens und passender Schlüsselwörter mit einer Aktion in der Musterspezifikation verknüpft werden. Die Komplexität dieser Zuordnung wird aus der Beschreibung der automatischen Musteranwendung (im Anhang C.3.3, S. 328 ff.) erkennbar.

Zuordnung bei Verhalten nicht praktikabel

Unten rechts in Abb. 8.18 wird die Situation nach (bis auf Aktionen) vollständiger Rollenzuordnung dargestellt. Hier wurde je eine zusätzliche Instanz der beiden Set Fragments ergänzt und entsprechende Rollenzuordnungen gemacht.

nach Rollenzuordnung

Das vorgestellte Verfahren zur Werkzeug-gestützten Musteranwendung eignet sich also auch im dritten Szenario. Eine im Entwurfsmodell vollständig model-

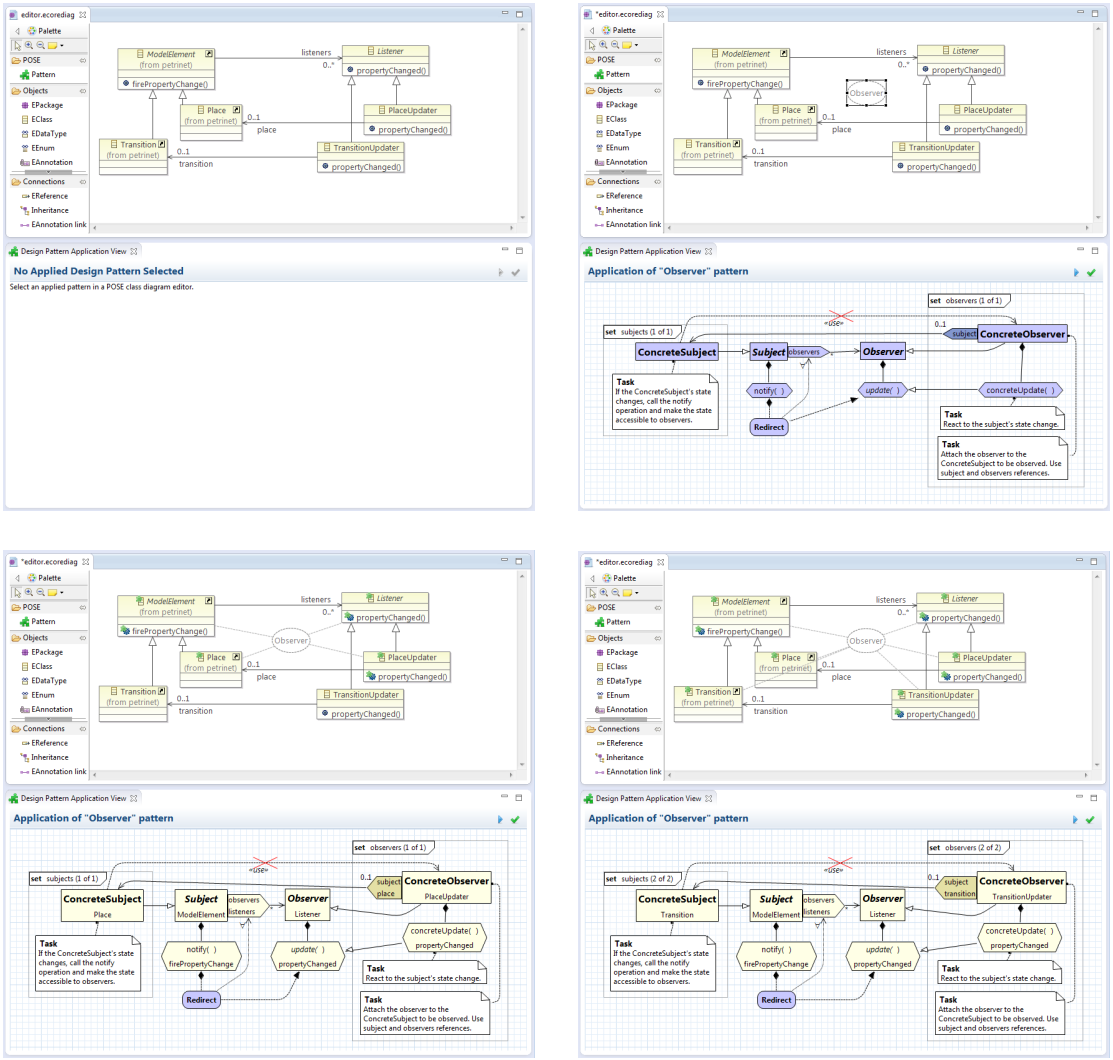


Abbildung 8.18: Nachträgliche Rollenzuordnung bei einer bereits existierenden Observer-Musterimplementierung

Machbarkeit bei Klassen gezeigt, nicht bei Verhalten

lierte Musterimplementierung kann mit Ausnahme des Verhaltensmodells auch nachträglich mit einer zugehörigen Musterspezifikation verknüpft werden. So kann sowohl die Anwendungsstelle dokumentiert als auch eine nachträgliche Validierung der Musterimplementierung ermöglicht werden. Aufgrund der fehlenden Rollenzuordnung beim Verhaltensmodell würde eine Validierung jedoch stets auf ein angeblich fehlendes Verhalten in der Musterimplementierung hinweisen.

Szenario 4: Musterimplementierung um Set-Fragment-Instanzen erweitern

mind. 1 Set-Fragment-Instanz ergänzen

Das vierte Szenario beschreibt einen Fall, bei dem eine Musterimplementierung bereits vollständig vorliegt und regulär mit einer zugehörigen Musterspezifikation verknüpft ist, die Anwendungsstelle also ebenfalls modelliert ist. Anschließend soll diese Musterimplementierung erweitert werden, indem zu mindestens einem der Set Fragments in der Musterspezifikation eine zusätzliche Ausprägung der im Set Fragment spezifizierten Entwurfsstruktur im Entwurfsmodell ergänzt wird.

Auch dieses Szenario stellt nur einen Spezialfall von Szenario 2 dar, da durch Ergänzen weiterer Set-Fragment-Instanzen in der Musteranwendungssicht eine partielle Rollenzuordnung entsteht – die vorher existierenden Ausprägungen der Rollen wurden vollständig zugeordnet, die neuen, nachträglich hinzugefügten Ausprägungen der Rollen noch nicht. Durch erneutes Anwenden des Musters werden die fehlenden Teile des Entwurfsmodells einfach ergänzt.

analog zu Szenario 2

Dieser Fall wurde in mehreren Situationen erprobt und funktioniert mit dem vorgestellten Ansatz mit den gleichen, bereits genannten Einschränkungen. Zum Beispiel lässt sich der Fall aus Abb. 8.16 (S. 190) in zwei Musteranwendungsschritte unterteilen. Zuerst wird das Observer-Muster mit nur je einer Ausprägung der Rollen in den Set Fragments subjects und observers angewandt, sodass im ersten Schritt nur die Klassen Place und PlaceUpdater generiert werden. Anschließend werden in einem zweiten Schritt in der Musteranwendungsansicht weitere Ausprägungen dieser Set Fragments ergänzt, um auch die Klassen Transition und TransitionUpdater erzeugen zu lassen. Durch erneute Anwendung des Musters werden diese neuen Ausprägungen der Set Fragments im Entwurfsmodell ergänzt, ohne die schon existierende und den Rollen zugeordnete Teillösung der Musterimplementierung zu ändern. Das Ergebnis dieser beiden Musteranwendungsschritte entspricht exakt dem Ergebnis in Abb. 8.16.

Machbarkeit gezeigt

Das vorgestellte Verfahren zur Werkzeug-gestützten Musteranwendung ist somit für alle vier genannten Szenarien geeignet. Die signifikanteste Einschränkung dabei stellt die fehlende Möglichkeit dar, Verhaltensmodelle nachträglich mit Musterrollen zu verknüpfen oder partielle Verhaltensmodelle zu vervollständigen.

Hauptmanko bei Verhalten

8.2.2 Beispiele für Musteranwendungen bei der Entwicklung eines grafischen Petrinetz-Editors

In dem vorhergehenden Abschnitt wurden unterschiedliche Einsatzszenarien der entwickelten Werkzeug-gestützten Musteranwendung am Beispiel des Observer-Musters diskutiert. In diesem Abschnitt gehe ich auf die Anwendbarkeit des Verfahrens anhand weiterer Musterspezifikationen ein und diskutiere die Besonderheiten der gewählten Anwendungsbeispiele. Im Folgenden gehe ich davon aus, dass ein grafischer Petrinetz-Editor basierend auf dem Graphical Editing Framework[6] für Eclipse[7] modellgetrieben entwickelt wird. Das Programmiermodell sowie die Klassen- und Methodennamen in den Beispielen stammen aus diesem Framework. Die bisher betrachteten Observer-Musteranwendungen wurden unter gleichen Rahmenbedingungen durchgeführt.

diverse, Praxis-nahe Musteranwendungen

Strategy

Das Strategy-Muster [GHJV95, S. 315 ff.] lässt sich bei der Entwicklung des grafischen Editors dazu einsetzen, verschiedene Layout-Algorithmen für die grafischen Repräsentationen der Modellelemente eines Petrinetzes zu implementieren, diese mit einheitlicher Schnittstelle auszustatten und austauschbar zu machen.

Anwendungsfall

[6] GEF: https://eclipse.org/gef/
[7] Eclipse: https://eclipse.org/

Situation vor Musteranwendung

In der Abb. 8.19 ist ein mögliches Anwendungsszenario dargestellt. Auf einer Zeichenfläche (Canvas) werden grafische Elemente (Figure) platziert. Ihre teilweise hierarchische Anordnung soll durch unterschiedliche Layout-Algorithmen bestimmt werden. Alle Algorithmen sollen mit einer einheitlichen Schnittstelle (Layout) ausgestattet werden. Diese besteht aus einer layout-Methode mit einem Parameter vom Typ Figure. Ein erster Algorithmus wird bereits durch die Klasse RowLayout repräsentiert. Eine zweite Algorithmus-Klasse (XYLayout) soll bei der Musteranwendung ergänzt werden. Die Klasse Figure soll beim Zeichnen der grafischen Elemente durch die Methode draw ihr Verhalten an den zuständigen Layout-Algorithmus delegieren. Die Musteranwendungsansicht drückt diese Rollenverteilung durch Zuordnung der Rollen request und algorithm entsprechend aus.

für Anwendungsfall unpassende Aktion

Leider lässt sich das Muster in diesem Fall nicht ohne Anpassung der Spezifikation anwenden. Die Semantik der delegate-Aktion ist so festgelegt, dass die delegierende Methode – hier draw – sämtliche erhaltenen Argumente unverändert an die aufgerufene Methode – hier layout – weitergibt. Aus diesem Grund muss die delegierende Methode über dieselben Parameter verfügen wie die aufgerufene Methode. In diesem Beispiel unterscheiden sich jedoch die Parameter der Methoden draw und layout, weswegen eine automatische Musteranwendung mit einem Fehler abgebrochen wird.

allgemeinere Aktion ermöglicht Anwendung

Passt man die Spezifikation des Strategy-Musters an, indem die delegate-Aktion durch eine call-Aktion ersetzt wird, lässt sich das Muster wie in der Abb. 8.20 dargestellt anwenden. Bei der Anwendung wird dann u.a. ein Story-Diagramm für die draw-Methode mit einem Aufruf der layout-Methode generiert (siehe Abb. 8.21 links). Bei diesem Aufruf fehlt ein Argument für den Figure-Parameter der layout-Methode und muss nachträglich manuell ergänzt werden (this in Abb. 8.21 rechts).

Aktionen-Semantik anpassen?

Dieser Anwendungsfall deutet darauf hin, dass die Semantik der delegate-Aktionen möglicherweise gelockert werden sollte, damit die Anwendbarkeit der damit spezifizierten Muster nicht unnötig eingeschränkt wird.

Abstract Factory

Anwendungsfall

Nehmen wir für das nächste Anwendungsbeispiel an, dass das in der Abb. 8.22 links dargestellte Petrinetz-Meta-Modell nicht nur zur Modellierung und Darstellung von Petrinetzen in einem Editor verwendet werden soll, sondern auch zur Repräsentation von Petrinetzen in einem Petrinetz-Simulator während ihrer Ausführung. Die Repräsentationen der grafischen Elemente im Editor und im Simulator würden sich voneinander unterscheiden und benötigen je eine eigene Implementierung. In diesem Fall kann das Muster Abstract Factory [GHJV95, S. 87 ff.] dazu verwendet werden, die für diese beiden Zwecke nötigen Implementierungen einheitlich zu behandeln. Dazu sollen Schnittstellen für alle grafischen Elemente definiert werden, je grafisches Element zwei unterschiedliche Implementierungen dieser Schnittstellen für die beiden Einsatzzwecke Editor und Simulator erstellt werden und zwei sogenannte Fabrikklassen zur Instanziierung dieser Implementierungen für den Editor bzw. für den Simulator bereitgestellt werden.

Situation vor Musteranwendung

Die beiden Implementierungen der grafischen Repräsentationen des Modellelements Place sind in der Abb. 8.22 bereits in Form der Klassen PlaceEditorFigure (Repräsentation im Editor) und PlaceSimulatorFigure (Repräsentation im Simu-

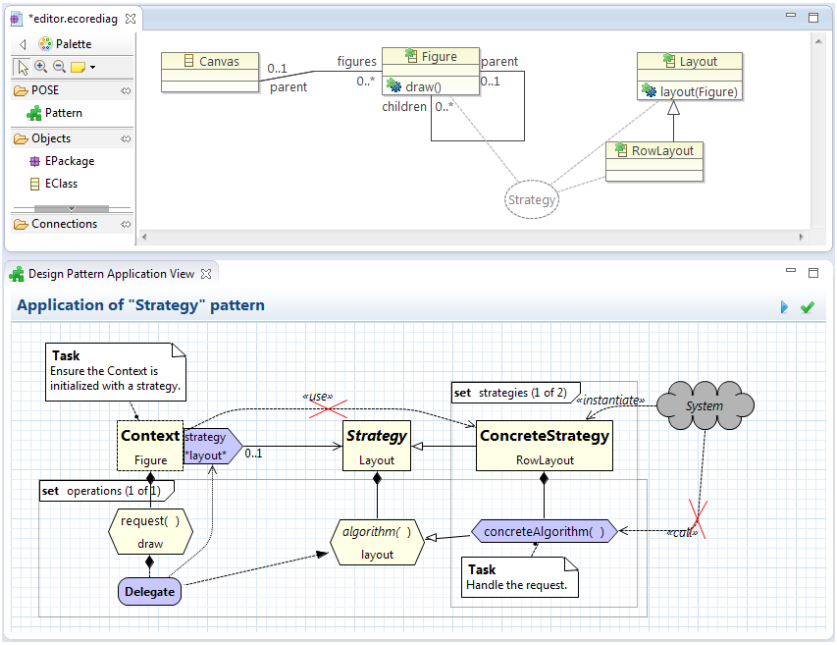


Abbildung 8.19: Problematischer Fall bei der Anwendung des Strategy-Musters

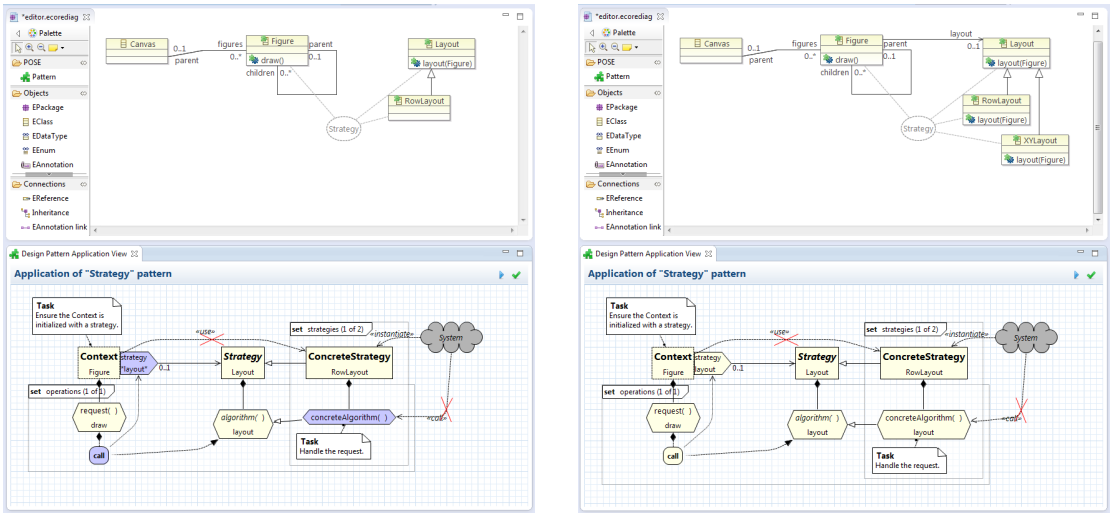


Abbildung 8.20: Anwendung des Strategy-Musters nach Anpassung der Spezifikation

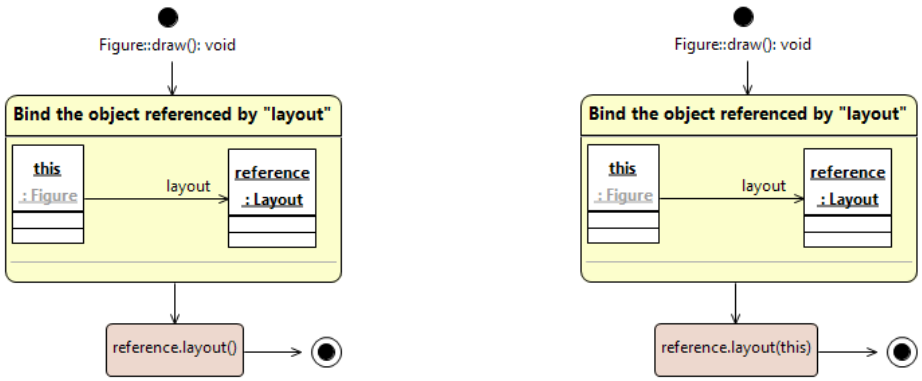


Abbildung 8.21: Vor & nach manueller Anpassung des generierten Story-Diagramms

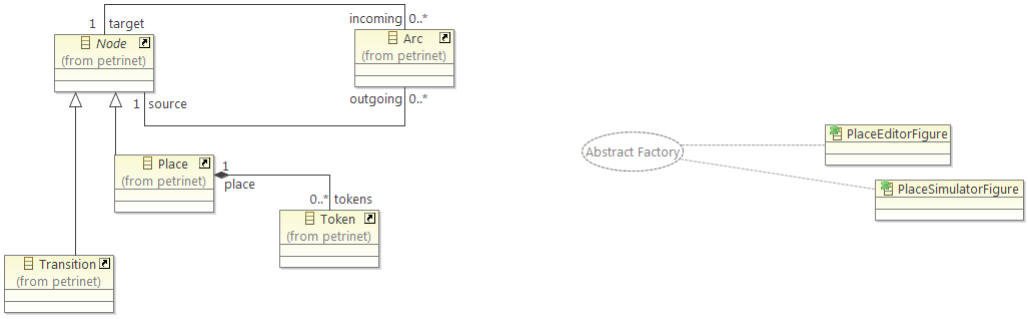


Abbildung 8.22: Klassendiagramm vor Anwendung des Musters Abstract Factory

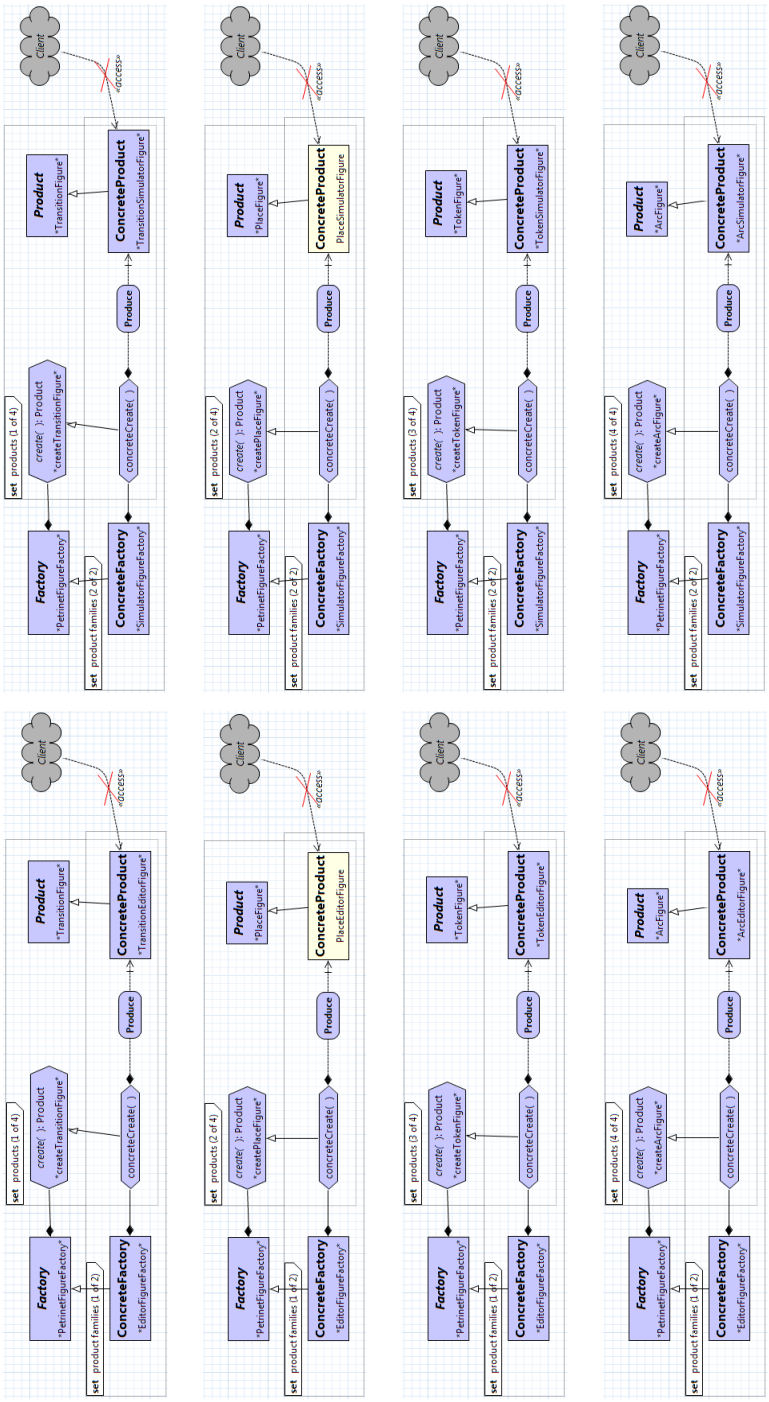


Abbildung 8.23: Rollenzuordnungen vor Anwendung des Musters Abstract Factory

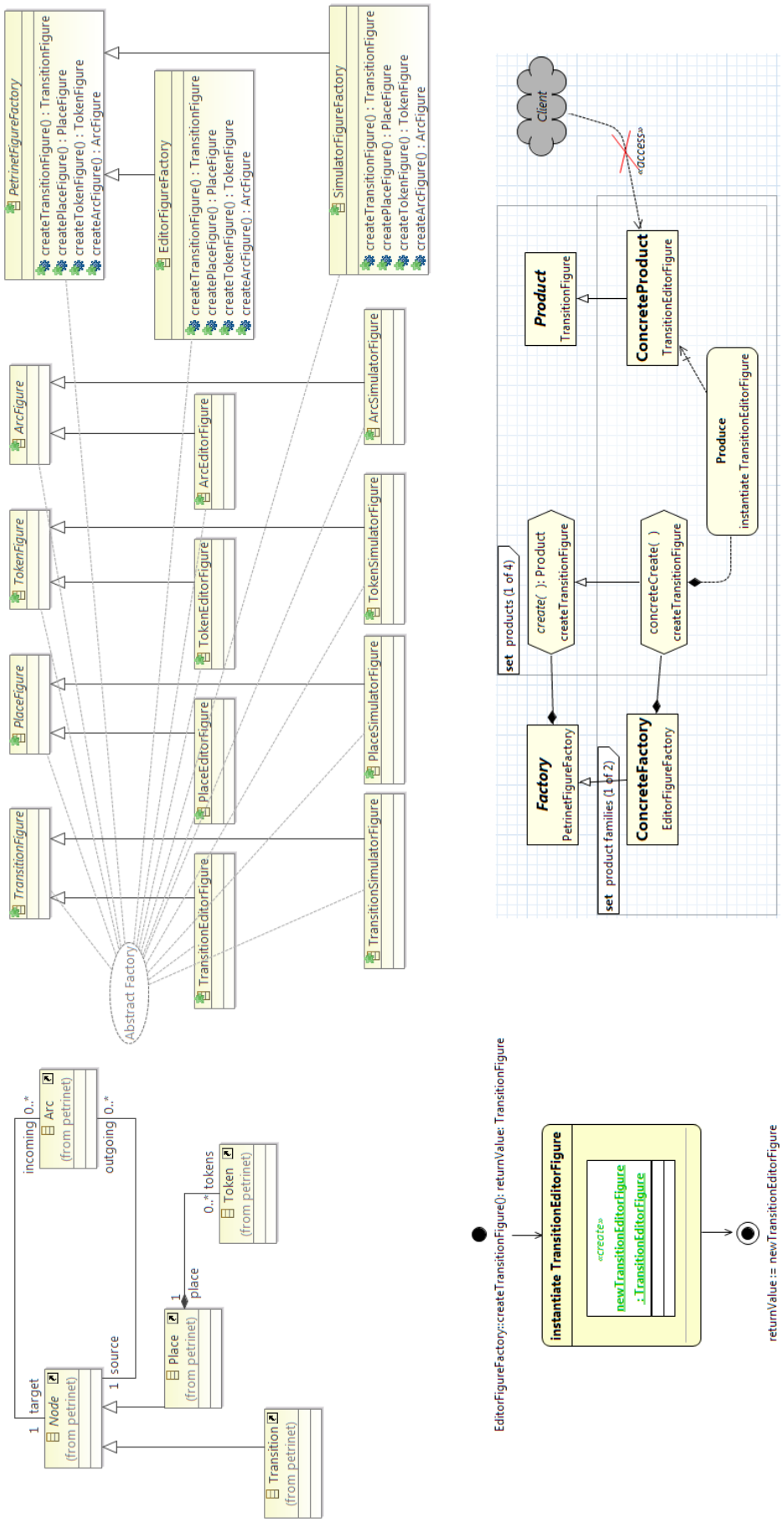


Abbildung 8.24: Modelle nach Anwendung des Musters Abstract Factory

lator) modelliert und zwecks Werkzeug-gestützter Anwendung des Musters ihren Rollen zugeordnet. Alle für die Musteranwendung erstellten Rollenzuordnungen und alle vordefinierten Namen für zu erzeugende Entwurfselemente sind den acht Musteranwendungsansichten in der Abb. 8.23 zu entnehmen. Hier werden alle Ausprägungen zu den beiden Set Fragments products und product families der Musterspezifikation dargestellt: zwei Fabrikklassen EditorFigureFactory und SimulatorFigureFactory (Rolle ConcreteFactory) zur Verwendung mit dem Editor bzw. dem Simulator sowie vier Produktschnittstellen (Rolle Product) und ihre acht Implementierungen (Rolle ConcreteProduct).

Ergebnis der vollautomatischen Anwendung

Nach der Rollenzuordnung und Festlegung der Namen für zu erzeugende Elemente kann das Entwurfsmodell vollautomatisch vervollständigt werden. Das Ergebnis ist in der Abb. 8.24 dargestellt. Das Klassendiagramm stellt alle erzeugten Klassen und Methoden dar. Zu jeder der acht create...-Methoden in den beiden nicht abstrakten Fabrikklassen wurde je ein Story-Diagramm generiert, welches exemplarisch für die Methode createTransitionFigure der Klasse EditorFigureFactory in der Abb. 8.24 aufgeführt ist. Ebenso ist eine der acht Sichten auf die Rollenzuordnungen nach Musteranwendung abgebildet. Alle Rollen sind nach dieser Musteranwendung zugeordnet, es sind keine manuellen Schritte nach der Musteranwendung nötig, es sind keine Entwurfsaufgaben offen.

zahlreiche Vorteile durch Set Fragments

Dieses Anwendungsbeispiel zeigt eindrucksvoll die Vorteile von Set Fragments. Trotz der komplexen Abhängigkeiten der in der Musterspezifikation definierten Entwurfsstrukturen und ihrer zahlreichen Ausprägungen bleibt die Musteranwendungsansicht stets übersichtlich und kompakt. Zwischen den Ansichten verschiedener Set-Fragment-Instanzen kann jederzeit gewechselt werden. Die Lösungsidee und insb. die Beziehungen der Entwurfsteile untereinander bleiben stets gut sichtbar. Bei derart vielen Ausprägungen der Set Fragments wie in diesem Beispiel (in der Praxis dürfte die Anzahl zumindest bei diesem Muster sogar deutlich höher ausfallen) kann ein erstaunlich großer Teil des Softwareentwurfs inkl. der Verhaltensmodelle automatisch erzeugt werden. So kann nicht nur der Aufwand bei der Musteranwendung reduziert, sondern auch Flüchtigkeitsfehler vermieden werden.

Model-View-Controller

Anwendungsfall

Bei dem Muster bzw. Architekturstil Model-View-Controller (MVC) [BMR+96, Fow02] ist die Intention, drei Teile eines Softwaresystems klar voneinander zu trennen (Prinzip: separation of concerns) und weitestgehend unabhängig voneinander zu machen. Diese Teile sind das Domänenmodell (Model), seine grafische Repräsentation (View) sowie der für Modelländerungen, Benutzerinteraktionen und Repräsentationsaktualisierungen zuständige Teil (Controller). Bei dem Anwendungsbeispiel Petrinetz-Editor (siehe Abb. 8.25) sollen die Domänenklassen (z.B. Place), die zur grafischen Repräsentation verwendeten Figure-Klassen und die für die Benutzerinteraktionen zuständigen EditPart-Klassen nach dem MVC-Prinzip separiert werden.

keine Generierung möglich

Im Gegensatz zu den anderen bisher betrachteten Musterspezifikationen können bei einer Musteranwendung in diesem Fall keine Entwurfsteile aus der Spezifikation generiert werden. Stattdessen wird nur eine Rollenzuordnung zwecks späterer Validierung des Softwareentwurfs oder schlicht zu Dokumentationszwecken vorge-

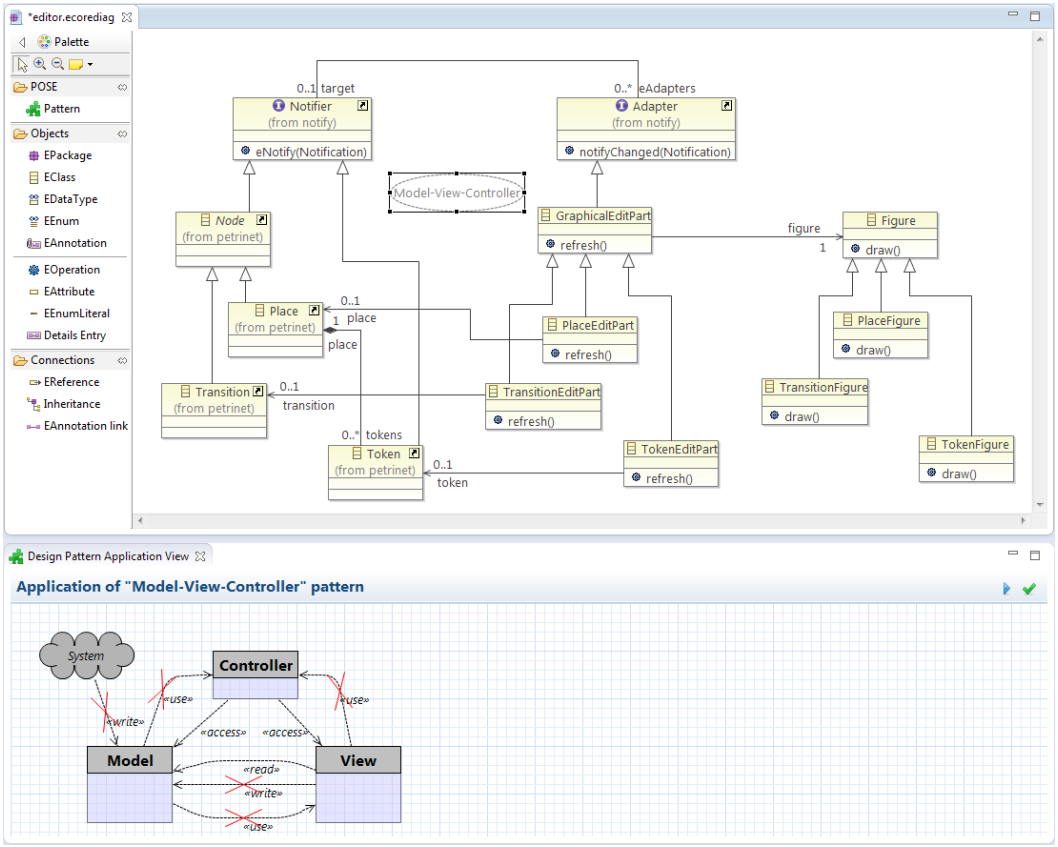


Abbildung 8.25: Ansicht vor Anwendung des Musters Model-View-Controller

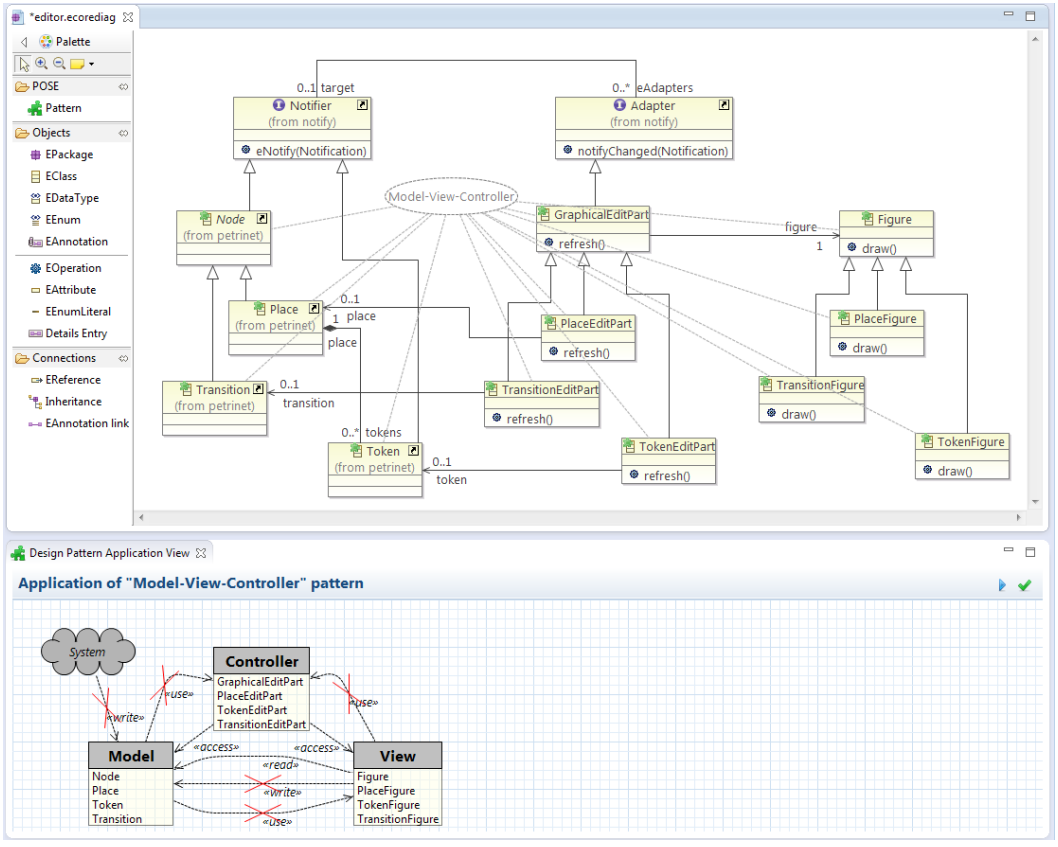


Abbildung 8.26: Ansicht nach Anwendung des Musters Model-View-Controller

nommen. Dazu werden den spezifizierten Subsystemen Model, View und Controller (siehe Abb. 8.25, hier noch ohne Rollenzuordnung) Klassen aus dem Softwareentwurf zugeordnet und damit die Zuordnung der Klassen zu dem jeweiligen Subsystem ausgedrückt. Nach ihrer Zuordnung werden die Klassen in der Musteranwendungsansicht bei dem entsprechenden Subsystem aufgelistet (siehe Abb. 8.26). Die Darstellung der Zuordnung im Klassendiagramm erfolgt analog zu anderen Mustern in Form von Kollaborationskanten.

Machbarkeit nur in Teilen gezeigt

Da beim vorliegenden Prototypen eine Validierung von Musterimplementierungen nicht vollständig implementiert wurde, kann eine explizit modellierte Musterimplementierung wie in der Abb. 8.26 derzeit nur als Dokumentation dienen. Die Einsetzbarkeit des Verfahrens zur Validierung des Entwurfs erscheint aber prinzipiell möglich und vielversprechend.

8.2.3 Modellgetriebene Entwicklung des JUnit-Frameworks durch inkrementelle Anwendung diverser Muster

Bsp. aus der Praxis

Zu dem bekannten und vielfach eingesetzten Test-Framework JUnit[8] gibt es eine von Erich Gamma erstellte Entwicklerdokumentation der Entstehungshistorie des Frameworks in der Version 3.8.x [Gam01]. Darin sind die bei der Entwicklung des Frameworks eingesetzten Muster und Entwurfsentscheidungen in chronologischer Reihenfolge und mit Zwischenständen des Entwurfs beschrieben. Der Entwurf ist mit nur ca. vier beteiligten Hauptklassen vergleichsweise klein (siehe Abb. 8.27).

Zur Einschätzung der Praxistauglichkeit meines Ansatzes suchte ich ein realistisches Beispiel für einen durch mehrere Musteranwendungen entstandenen Softwareentwurf, um diesen Entwurf mit meinem Verfahren schrittweise nachzuentwickeln. Das JUnit-Framework stellt ein solches Beispiel dar und ist dabei verglichen mit anderen Frameworks relativ überschaubar. Dank seiner detaillierten Dokumentation eignet es sich gut für meine Evaluation.

Vorgehen

Ursprünglich ist JUnit nach meinem Kenntnisstand nicht modellgetrieben oder modellbasiert entstanden, sondern wurde direkt in Java implementiert. Ich habe die dokumentierten Entwicklungsschritte modellgetrieben nachempfunden. Der so erstellte Softwareentwurf ist dabei analog zu den bisher betrachteten Musteranwendungen in einem Klassen- und Story-Diagramm-Modell entstanden.

angewandte Muster

Laut Gamma wurden bei der Entwicklung des JUnit-Frameworks die folgenden sechs Muster in angegebener Reihenfolge angewandt: Command, Template Method, Collecting Parameter, (Class) Adapter, Pluggable Selector, Composite [GHJV95, Bec96]. Der fertige Softwareentwurf ist in der Abb. 8.27 dokumentiert.

Ergebnis reproduzierter Musteranwendungen

Mit Ausnahme des Adapter-Musters habe ich alle dokumentierten Musteranwendungen mit meinem Ansatz nachempfunden. Das Ergebnis dieser Musteranwendungen ist in der Abb. 8.28 dargestellt. Darin sind alle Klassen und Methoden aus der Abb. 8.27 zusammen mit weiteren von Gamma in seiner Dokumentation genannten Klassen und Methoden aufgeführt. Die Methode addTest der Klasse TestSuite aus der Abb. 8.27 ist im Klassendiagramm in der Abb. 8.28 implizit in Form der Assoziation fTests enthalten. Für Assoziationen werden immer derartige Zugriffsmethoden generiert, weswegen sie in Ecore-Klassendiagrammen nicht ex-

[8] http://junit.org/

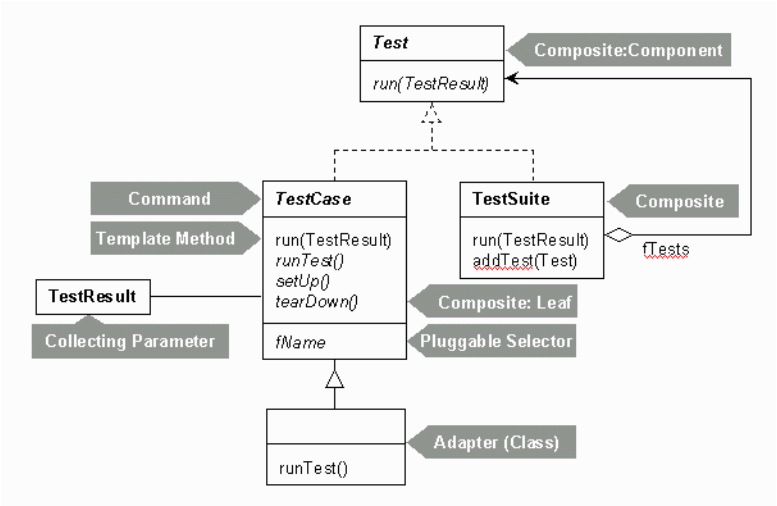


Abbildung 8.27: Entwurf des JUnit-Frameworks mit eingesetzten Mustern (Darstellung aus der JUnit-Dokumentation entnommen [Gam01])

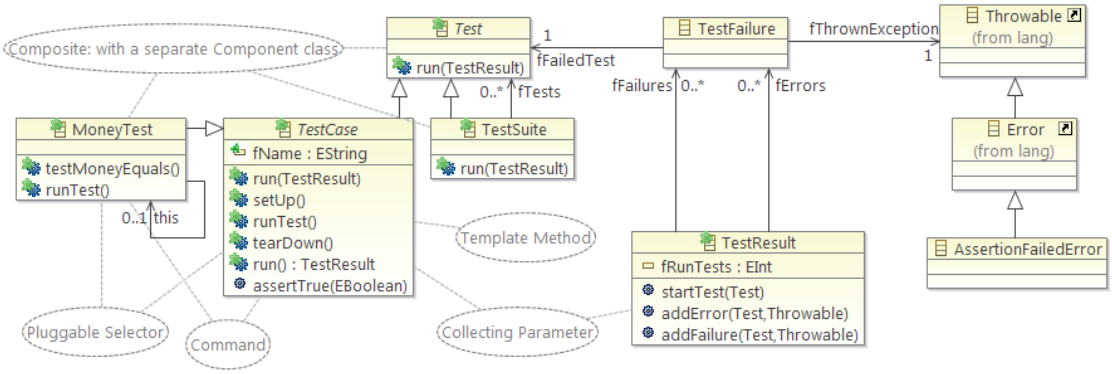


Abbildung 8.28: Nachempfundener Entwurf des JUnit-Frameworks mit eingesetzten Mustern

Grenzen von Ecore

plizit dargestellt werden. Die anonyme Klasse[9] aus der Abb. 8.27 – diese nimmt eine Rolle des Musters (Class) Adapter ein – habe ich jedoch nicht übernommen, weil sie sich im Ecore-Klassenmodell nicht modellieren lässt. Aus demselben Grund lässt sich das Adapter-Muster nicht anwenden.

Ansatz weitestgehend anwendbar

Die einzelnen Schritte bei der Anwendung der Muster und die entstandenen Zwischenergebnisse sind im Anhang D dokumentiert. Die Anwendung des hier vorgestellten Verfahrens zur modellgetriebenen Anwendung von Mustern am Beispiel von JUnit lässt sich zusammenfassend als weitestgehend erfolgreich beschreiben.

fast alle Musteranwendungen reproduziert

Mit Ausnahme des Adapter-Musters ließen sich alle Muster spezifizieren und semiautomatisch anwenden. Die Anwendungsstellen waren nach der Musteranwendung und manueller Anpassungen vollständig in einem dafür vorgesehenen Modell erfasst und konnten visualisiert werden. Die explizit modellierten Anwendungsstellen (Rollenzuordnungen) sind prinzipiell auch auf Konsistenz zu den Musterspe-

[9] Anonyme Klassen sind ein Java-Sprachkonstrukt zur Implementierung einer Schnittstelle (`interface`) direkt im Rumpf einer Methode, ohne die Implementierung mit einem Klassennamen zu versehen.

zifikationen prüfbar, was insb. aufgrund der gemachten manuellen Anpassungen und der vielen überlappenden Musteranwendungen nützlich wäre. Diese Funktionalität stellt der von mir entwickelte Prototyp jedoch nicht zur Verfügung. Das gleiche gilt für die in Abschnitt 4.4 (S. 89 ff.) vorgestellte, detaillierte Visualisierung von Anwendungsstellen in Klassen- und Story-Diagrammen, weswegen die eingenommenen Rollen in diesen Diagrammen vom Prototypen nicht dargestellt werden (vgl. Abb. 4.11 und 4.14, S. 91, 92).

eingeschränkte Evaluationsmöglichkeiten

Bei der Anwendung der Muster in diesem Szenario waren einige Besonderheiten zu beobachten. Das Adapter-Muster wurde bei JUnit auf eine untypische Weise angewandt (in anonymen Klassen) und ließ sich aufgrund begrenzter Ausdrucksmöglichkeiten der verwendeten Ecore-Modelle (Klassendiagramme) nicht anwenden, jedoch nicht aufgrund von Einschränkungen des vorgestellten Ansatzes zur Musteranwendung.

Musteranwendung nicht durch den Ansatz verhindert

In Einzelfällen musste der Entwurf vor einer Musteranwendung manuell angepasst werden, z.B. zur Vermeidung von Dateinamenskonflikten aufgrund sich ändernder Methodensignaturen (Collecting Parameter, siehe S. 347) oder zur nachträglichen Spezifikation einer Schnittstelle zu einer existierenden Methodensignatur (Composite, siehe S. 355).

Eingriffe bei Refactoring

In einem Fall musste eine durch automatische Musteranwendung generierte Methode wieder entfernt werden (Pluggable Selector, siehe S. 353). Der Grund dafür ist, dass die betroffene Methode zum Zeitpunkt der Musteranwendung in JUnit nicht existieren sollte (sie entsteht erst durch Implementierung konkreter Tests außerhalb des Frameworks) und per Reflection aufgerufen wird. Reflection ist bei der Modellierung eines Softwareentwurfs eher untypisch. Es handelt sich hier also um einen Sonderfall.

Sonderfall: Reflection

Bei der Anwendung des Musters Command wurde im Klassendiagramm eine eigentlich unnötige Referenz this modelliert (siehe S. 360). In Java ist `this` ein Schlüsselwort, der einen Selbstverweis ausdrückt. Als Alternative zur Spezifikation einer zusätzlichen Variante des Musters, wo die Rollen ConcreteCommand und Receiver in nur einer Klasse verschmelzen, habe ich im Entwurf die implizit immer vorhandene Selbstreferenz als Assoziation modelliert. Dadurch konnte in einem Story-Diagramm modelliert werden, dass ein Objekt eine Methode auf sich selbst aufruft. Außerdem konnte diese Referenz der zugehörigen Rolle in der Musterspezifikation zugeordnet werden.

Eingriff zur Vermeidung weiterer Spezifikationen

Durch die Konzipierung von JUnit als Framework kommt eine weitere Besonderheit hinzu: Die Klassen im Framework decken bei einigen Mustern nur einen Teil der in Musterspezifikationen erfassten Entwurfslösungen ab. Einige Rollen werden erst durch Implementierung konkreter, auf dem JUnit-Framework basierender Test-Klassen wie MoneyTest in der Abb. 8.28 eingenommen. Während der Entwicklung eines Frameworks würden solche, das Framework nutzenden Klassen normalerweise fehlen. Das würde zu einer unvollständigen Anwendung einiger Muster führen. Bei JUnit sind die Muster Command, Pluggable Selector und Composite betroffen (siehe Abb. 8.28). Die Klasse MoneyTest ist streng genommen nicht Teil des JUnit-Frameworks, wurde aber zu Testzwecken mit dem Framework zusammen entwickelt und in der Entwicklerdokumentation beschrieben. Darum habe ich die Klasse in den Entwurf des Frameworks übernommen und damit die Anwendung der Muster vervollständigt (Details in Anhang D.8, S. 358 ff.).

Sonderfall: partielle Musterimplementierung

8.2.4 Fazit

Der in dieser Arbeit vorgestellte Ansatz zur modellgetriebenen Anwendung von Entwurfsmustern wurde durch seinen prototypischen Einsatz in unterschiedlichsten Szenarien erprobt und in den vorhergehenden Abschnitten diskutiert. Der Ansatz konnte mit einigen Einschränkungen erfolgreich angewendet werden.

Ansatz erfolgreich angewandt

Eignung für unterschiedliche Anwendungsszenarien

Das Verfahren kann in den folgenden vier Anwendungsszenarien eingesetzt werden (vgl. Abschn. 8.2.1). Partiell modelliertes Verhalten lässt sich nicht vervollständigen. Ebenso lässt sich modelliertes Verhalten nicht nachträglich zugehörigen Aktionen zuordnen. In allen anderen Fällen ist die automatische Musteranwendung bis auf wenige Einschränkungen problemlos möglich.

Mankos bei Verhalten

1. Eine Musterimplementierung vollständig generieren
2. Beliebige, fehlende Teile einer Musterimplementierung ergänzen
3. Einer Musterimplementierung nachträglich Musterrollen zuordnen
4. In einer Musterimplementierung Set-Fragment-Instanzen ergänzen

Bei den betrachteten Musteranwendungen wurden vor allem Teile des Klassenmodells generiert. Verhalten konnte in diversen Fällen nicht genau genug spezifiziert und dadurch nicht oder nur teilweise generiert werden. Die generierten Verhaltensmodelle sind eher klein ausgefallen. Sie beschreiben meist einzelne Methodenaufrufe, mehrere Aufrufe in einer Schleife oder Klasseninstanziierungen. Aus Subsystemen und Kopplungsregeln wurden keine Entwurfsteile generiert.

Stärken insb. im Klassenmodell

Set Fragments haben sich als gutes Hilfsmittel erwiesen, sich wiederholende Teile des Softwareentwurfs kompakt zu beschreiben und bei der Musteranwendung zu generieren (z.B. bei Abstract Factory, S. 194 ff.).

Set Fragments nützlich

Muster können in nahezu jeder Ausgangssituation angewendet werden, denn der Entwurf wird bezogen auf eine Musterspezifikation und eine Rollenzuordnung grundsätzlich nur vervollständigt. Es werden immer nur diejenigen Musterrollen in den Entwurf übertragen, zu denen noch keine Entsprechungen im Entwurf existieren. Das bietet Entwicklern eine enorme Flexibilität bei der Musteranwendung. Sie entscheiden selbst, wann sie ein Muster anwenden und mit welchen Rollen eines Musters sie beginnen.

flexibel anwendbar, nicht destruktiv

Indem Beziehungen zwischen Musterrollen keine direkten Beziehungen im Entwurf verlangen, sondern u.a. indirekte Beziehungen gleicher Art zulassen, wird Entwicklern weiterer Spielraum bei der Anwendung und Implementierung von Mustern geboten. Es werden z.B. indirekte Vererbungs- und Kompositionsbeziehungen zugelassen, sodass eine laut Muster in einer Klasse geforderte Operation in der Klasse selbst oder in einer ihrer Oberklassen liegen darf.

Raum für Anpassungen

Leiten von Entwicklern bei der Musteranwendung

Die vorgestellten Musterspezifikationen helfen dabei, Teile einer Musterimplementierung in den Entwurf zu übertragen. Das kann durch automatische Musteranwendung in einem Schritt geschehen oder schrittweise, indem der Entwurf in Kom-

bination mit manuellen Anpassungen, geleitet durch die Musterspezifikation inkrementell vervollständigt wird.

Kopplungsregeln als Leitplanken

Durch Kopplungsregeln wird ein wichtiger Teil der Intention eines Musters in der Musterspezifikation erfasst. Diese zeigen auf, welche Abhängigkeiten unerwünscht und welche erwartet werden, und leiten auf diese Weise Entwickler bei ihrer Musteranwendung.

Entwurfsaufgaben kaum bewertbar

Der Nutzen von Entwurfsaufgaben konnte in den betrachteten Szenarien nicht evaluiert werden. Sie können vermutlich dabei helfen, Entwickler bei der Musteranwendung zu leiten und Flüchtigkeitsfehler zu vermeiden. Da ich alle Muster selbst spezifiziert habe, die Muster und Entwurfsaufgaben also gut kenne, konnte ich den Nutzen der Entwurfsaufgaben nicht objektiv bewerten.

Validierung kaum bewertbar

Aufgrund unvollständiger Implementierung der Validierung von Musterimplementierungen im vorliegenden Prototypen kann auch der Nutzen dieser Funktionalität nicht evaluiert werden. Es ist jedoch zu erwarten, dass diese Funktionalität gerade bei vielen manuellen Entwurfsanpassungen und überlappenden Musteranwendungen wie im Beispiel des JUnit-Frameworks (S. 200 ff.) sehr hilfreich wäre. Außerdem ließe sich diese Funktionalität dazu nutzen, Architekturkonformität zu prüfen wie z.B. bei dem Model-View-Controller-Architekturstil (S. 198 ff.).

Aufwandsbetrachtungen

Musterspezifikation: vernachlässigbarer Aufwand

Die Modellierung von Musterspezifikationen und die Pflege von Musteranwendungsstellen (der Rollenzuordnung) in Entwurfsmodellen erfordern gewissen Aufwand. Ein erfahrener Entwickler oder Architekt kann nach meiner Einschätzung ein ihm bekanntes Muster abhängig von seiner Komplexität und zugehöriger Varianten in ca. 15 bis 60 Minuten spezifizieren. Da die Spezifikation nur einmalig erfolgen muss, ist dieser Aufwand vernachlässigbar, wenn das Muster anschließend häufig eingesetzt und der damit verbundene Nutzen hoch genug ist.

Musteranwendung: signifikanter, variierender Aufwand

Bei der Musteranwendung muss bei meinem Ansatz eine relativ detaillierte Zuordnung der spezifizierten Rollen erfolgen, was ebenfalls seine Zeit erfordert. Wie hoch der Zeitaufwand ist, hängt stark von der Musterspezifikation und dem Anwendungsfall ab. Bei Mustern wie Facade (S. 176 ff.) und MVC (S. 198 ff.) werden relativ viele Klassen im Entwurf den Subsystemen in der Spezifikation zugeordnet. Im Vergleich zu den wenigen bzw. gar keinen bei Musteranwendung generierten Klassen und Methoden ist das relativ aufwändig. Anders verhält es sich bei Mustern wie Abstract Factory (S. 194 ff.), wo zwar eine relativ umfangreiche Rollenzuordnung nötig ist, dafür aber ein großer Teil des Entwurfs inklusive Verhalten generiert wird und während der automatischen Musteranwendung im Vergleich zu einer manuellen Musteranwendung Zeit gespart wird. Bei den meisten der hier betrachteten Muster und ihren Anwendungsfällen habe ich für eine Rollenzuordnung ca. 1 bis 5 Minuten gebraucht, abhängig von der Anzahl der schon vorhandenen Entwurfsteile und den vor Musteranwendung festzulegenden Namen für zu erzeugende Entwurfsteile. Dieser Mehraufwand ist nicht zu unterschätzen, könnte sich aber aufgrund des damit verbundenen Nutzens lohnen. Höhere Benutzbarkeit der bisher nur prototypischen Werkzeuge könnte die Aufwände etwas reduzieren.

Den Hauptnutzen meines Ansatzes sehe ich weniger in der automatischen Musteranwendung und der damit verbundenen potentiellen Zeiteinsparung, sondern

(a) in der genauen Dokumentation und Visualisierung des angewandten Musters und noch mehr (b) in der Möglichkeit, die Anwendungsstellen auf Konsistenz mit der Musterspezifikation zu prüfen. Ersteres könnte übersehene oder missverstandene Musteranwendungen vermeiden. Letzteres könnte Falschimplementierungen und versehentliche Abweichungen von Mustern reduzieren. Ob sich der Mehraufwand lohnt, müsste man jedoch in einer bisher fehlenden, empirischen Studie prüfen. Da mein Prototyp die Möglichkeit zur Validierung von Anwendungsstellen bisher nicht bietet, lässt sich der Nutzen einer Validierung derzeit nicht empirisch untersuchen.

Vermeiden von Fehlern rechtfertigt vermutlich den Aufwand

8.3 Modellierung und Visualisierung von Anwendungsstellen

Die im Rahmen dieser Arbeit in Kapitel 4 vorgestellte, explizite Modellierung von Anwendungsstellen ist für den dafür vorgesehenen Einsatzzweck ausreichend. Vor der Musteranwendung konnten die Rollenzuordnungen in vom Entwurf separaten Modellen erfasst werden. Bei der Musteranwendung konnte die Übersetzung der Spezifikation in den Entwurf in zwei Schritten erfolgen. Die durch Auffalten der Set Fragments erzeugte 1-zu-1-Repräsentation des Entwurfs (siehe Abschn. 5.4) konnte im Anwendungsmodell erfasst werden, bevor sie in einem weiteren Schritt in den Entwurf übersetzt wurde (siehe Abschn. 5.5). Gleichzeitig konnte das Modell zur Visualisierung der Anwendungsstellen in Klassendiagrammen und in der Musteranwendungsansicht verwendet werden.

Modell erfüllt seinen Zweck

Die prototypische Visualisierung von Anwendungsstellen in Klassendiagrammen wurde nur teilweise, die in Story-Diagrammen gar nicht implementiert. Der Aufwand der dazu nötigen Anpassungen an existierenden Editoren ist zu hoch im Vergleich zum damit verbundenen wissenschaftlichen Beitrag, welcher sich bereits durch das vorgestellte Konzept in Abschnitt 4.4 (S. 89 ff.) abschätzen lässt. Die vielen Beteiligungen der Klassen im JUnit-Framework an unterschiedlichen, eingesetzten Mustern und die fehlende Information im Klassendiagramm über eingenommene Musterrollen (siehe Abb. 8.28, S. 201) sprechen klar für eine detailliertere Visualisierung wie sie in Abschnitt 4.4 vorgestellt wurde (vgl. Abb. 4.11 und 4.14, S. 91, 92).

Darstellung scheint sinnvoll, derzeit kaum bewertbar

In einem der betrachteten Szenarien für eine Musteranwendung lässt sich das Verhalten einer vorliegenden Musterimplementierung nicht nachträglich mit der Musterspezifikation verknüpfen. Während Methoden ihren Rollen zugeordnet werden können, können Teile von Story-Diagrammen nicht manuell ihrer Entsprechung in einer Musterspezifikation, einer Aktion, zugeordnet werden (Szenario 3, S. 191). Möglicherweise ist ein anderes, praktikableres Konzept zur Dokumentation von Anwendungsstellen in Verhaltensmodellen nötig.

Verknüpfen bei Verhalten unpraktisch

8.4 Validierung von Anwendungsstellen

Der im Kapitel 6 vorgestellte Ansatz zur automatischen Validierung von Anwendungsstellen bzw. Musterimplementierungen, womit die Konsistenz zur Musterspezifikation sichergestellt werden soll, wurde im vorliegenden Prototypen nicht

vollständig implementiert. Der Aufwand der dazu nötigen prototypischen Implementierung ist sehr hoch im Vergleich zum damit verbundenen wissenschaftlichen Beitrag, welcher sich aus dem vorgestellten Konzept grob abschätzen lässt.

Evaluierung mit dem Prototypen nicht möglich

Wie schon in den Aufwandsbetrachtungen beschrieben (S. 204), sehe ich den Hauptnutzen des in dieser Arbeit vorgestellten Ansatzes in der stets gepflegten, detaillierten Dokumentation aller Anwendungsstellen und insb. in der Möglichkeit, Anwendungsstellen jederzeit auf Konsistenz mit der Musterspezifikation prüfen zu können. Wie hilfreich genau so eine Validierung wäre, konnte ich aufgrund der fehlenden Funktionalität im Prototypen nicht evaluieren. Welche Merkmale einer Musterimplementierung automatisch geprüft werden können, wird jedoch im Kapitel 6 beschrieben.

8.5 Zusammenfassung

Eignung der Sprache & Spezifikationen gezeigt

Es wurde gezeigt, dass sich mit der vorgestellten Spezifikationssprache zahlreiche Entwurfsmuster beschreiben lassen und sich die Spezifikationen für eine semiautomatische Musteranwendung eignen. Objektorientierte Entwurfsstrukturen und sich wiederholende Teile des Entwurfs lassen sich besonders gut beschreiben, während sich die Verhaltensbeschreibung auf einfache Interaktionen wie Methodenaufrufe beschränkt. Subsysteme und Kopplungsregeln ergänzen die Spezifikationen und könnten für Architekturkonsistenzprüfungen genutzt werden. Die Spezifikationssprache könnte um zusätzliche Sprachkonstrukte zur Beschreibung anderer Muster erweitert werden.

flexible Anwendung machbar, Einschränkungen bei Verhalten

Spezifizierte Muster lassen sich flexibel anwenden. Es kann die komplette Entwurfslösung – die Klassenstruktur und zugehöriges Verhalten – generiert werden oder abhängig von einer nahezu beliebigen Ausgangssituation nur fehlende Teile der Entwurfslösung generiert werden. Verhalten kann jedoch nur generiert werden, wenn es vor Musteranwendung komplett fehlt. Wurde ein Muster bereits manuell modelliert / angewendet, so lässt sich die Anwendungsstelle auch nachträglich dokumentieren, also in einem separaten Modell der Anwendungsstelle erfassen und mit einer Musterspezifikation verknüpfen. Verhaltensmodelle können bisher nicht nachträglich dokumentiert werden.

teils keine Evaluation möglich

Aufgrund fehlender Funktionalität im Prototypen konnten die automatische Validierung sowie die detaillierte Visualisierung von Anwendungsstellen in Klassen- und Story-Diagrammen bisher nicht erprobt und evaluiert werden.

empirische Studien benötigt

Wie gut Entwickler mit Hilfe der Musterspezifikationen und der darin enthaltenen Entwurfsaufgaben bei der Musteranwendung geleitet werden können und ob sich der Mehraufwand im Vergleich zum Nutzen lohnt, kann bisher nicht abschließend bewertet werden. Dazu wären Untersuchungen mit anderen Entwicklern als mir selbst nötig. Das gleiche gilt für die Frage, ob mein Ansatz versehentliche Abweichungen von zuvor angewandten Mustern reduziert. Diese lässt sich vermutlich mit kontrollierten Experimenten oder anderen empirischen Studien beantworten, deren Durchführung im Rahmen dieser Arbeit zu aufwändig gewesen wäre.

9 Verwandte Arbeiten

Die Hauptbeiträge der vorliegenden Arbeit lassen sich genauso wie verwandte Arbeiten in vier Forschungsgebiete einteilen. Dazu gehören die Spezifikation von Software-Entwurfsmustern (Kap. 3), die Dokumentation (Persistierung und Visualisierung) von Musteranwendungsstellen (Kap. 4), die Werkzeug-gestützte (semi-automatische) Musteranwendung (Kap. 5) und die automatische Prüfung der Konformität einer Architektur oder eines Softwareentwurfs in Bezug auf Architektur- und Entwurfsvorgaben (z.B. durch Architekturstile und Entwurfsmuster) (Kap. 6). In den folgenden vier Abschnitten werden die verwandten Arbeiten zu diesen Gebieten separat betrachtet (siehe Abb. 9.1).

Forschungsgebiete

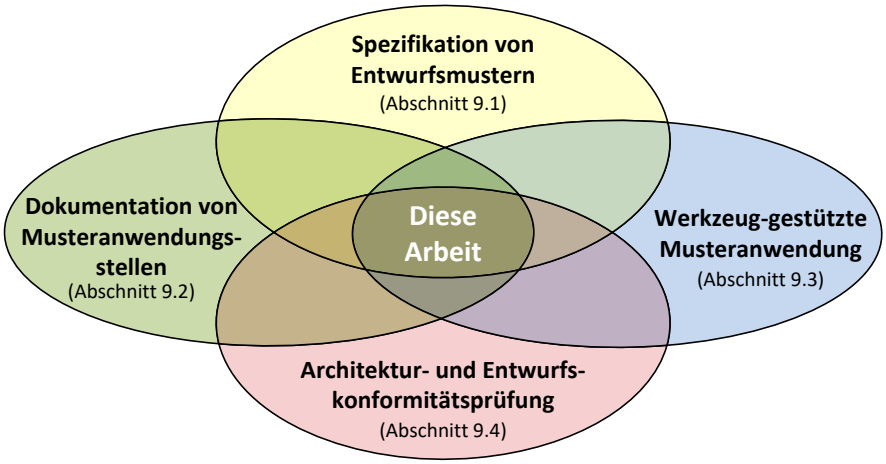


Abbildung 9.1: Übersicht der Themengebiete verwandter Arbeiten (Euler-Diagramm)

Neben den verwandten Arbeiten gibt es auch einige Vorarbeiten. Die ersten Ideen zu meinem Dissertationsvorhaben habe ich im Rahmen eines Doktorandensymposiums beschrieben [Tra10]. Einige dieser Ideen (insb. Musterspezifikation, Modellierung von Anwendungsstellen, automatische Musteranwendung und Konsistenzprüfung) wurden in einem studentischen, von mir initiierten Projekt namens *PG POSE* [1] [BBF+11] weiter ausgearbeitet und teilweise in einem ersten Prototypen implementiert. Im Rahmen des Projekts wurde auch ein Reverse-Engineering-Verfahren zur Erkennung von Musteranwendungsstellen realisiert, bei welchem die für das Forward Engineering entwickelten Musterspezifikationen genutzt wurden. Aufbauend auf den Ergebnissen der PG POSE wurden im Rahmen einer von mir

Vorarbeiten

[1] Projektgruppe *Pattern-Oriented Software Engineering (POSE)*: Eine 2-semestrige Veranstaltung der Universität Paderborn im SS 2010 und WS 2010/2011 unter der Leitung von Dietrich Travkin, Markus von Detten und Prof. Dr. Wilhelm Schäfer, Web-Seite der Veranstaltung: http://www-old.cs.uni-paderborn.de/no_cache/fachgebiete/fachgebiet-softwaretechnik/lehre/lehrveranstaltungen/projektgruppen/pg-pose.html

betreuten Masterarbeit erste Ideen zur Generierung von Verhaltensmodellen aus Musterspezifikationen erarbeitet [Bac11]. Die Ideen aus diesen Vorarbeiten habe ich grundlegend überarbeitet und erweitert.

9.1 Spezifikation von Entwurfsmustern

Die zahlreichen, über Jahrzehnte entstandenen Beschreibungen diverser Softwareentwurfsmuster in der Literatur haben zum Ziel, bewährte Lösungen möglichst allgemein wiederverwendbar zu dokumentieren. Aus diesem Grund sind sie meist informell, in natürlicher Sprache, ggf. mit einigen Skizzen und Beispielen, jedoch absichtlich ungenau und unvollständig beschrieben (siehe Bsp. in Abb. 1.1, S. 4). Es wird die grobe Idee vorgestellt, aber keine konkrete, direkt verwendbare Lösung und keine Auflistung aller Möglichkeiten. Bei Anwendung eines Musters muss die umrissene Lösung konkretisiert und vervollständigt werden. Genau das erschwert jedoch die Entwicklung von Werkzeugen, welche Entwickler bei der Anwendung von Mustern und der Überprüfung von Musterimplementierungen unterstützen.

Unpräzises schwer formalisierbar

Inzwischen sind zahlreiche Versuche unternommen worden, Entwurfsmuster formal und damit präzise und vollständig zu spezifizieren, um so den Weg für bessere Werkzeugunterstützung und formale Analysen ihrer Eigenschaften zu ebnen. Es hat sich gezeigt, dass solche Formalisierungen immer auf Kosten der Generalisierbarkeit erfolgen [BHS07b, S. 84 ff.]. Für bestimmte, eingeschränkte Anwendungsgebiete erfüllen sie jedoch ihren Zweck. Zu den Anwendungsgebieten gehören insb. das Beweisen bestimmter Mustereigenschaften wie die korrekte Komposition mehrerer Muster, das semiautomatische Übertragen einer Entwurfslösung in ein Entwurfsmodell oder in Quellcode, das automatische Auffinden von Musterimplementierungen in Entwurfsmodellen oder Quellcode sowie das Prüfen der korrekten Implementierung eines oder mehrerer Muster.

Formales eingeschränkt nutzbar

Im Folgenden stelle ich einige solcher Arbeiten vor und betrachte dabei vor allem den Aspekt der Musterspezifikation. Ich gruppiere die Arbeiten nach ihrem Hauptanwendungszweck für Beweise (9.1.1), für Musteranwendungen oder Konsistenzprüfungen (9.1.2) und für Reverse Engineering (9.1.3). Informelle Beschreibungen von Entwurfsmustern betrachte ich dabei nicht, sie werden in Abschnitt 2.1.2 behandelt.

Arbeiten gruppiert nach Einsatzzweck

9.1.1 Mathematisch formale Musterspezifikationen für Beweise

Ein Großteil der verwandten Arbeiten auf dem Gebiet der Musterspezifikation baut auf mathematische, oft prädikatenlogische Formalisierungen von Entwurfsmustern und ihren Eigenschaften. Taibi hat 16 Entwurfsmuster-Formalisierungsansätze diverser Wissenschaftler zusammengetragen [Tai07a], wovon 12 eine derartig strikte, mathematische Formalisierung verwenden[2]. Dabei werden z.B. mathematische Relationen [FCA07, SS07, RCOH07, Gas07], Prädikatenlogik und tem-

zahlreiche Formalisierungsansätze

[2]Die anderen 4 verwenden UML-Erweiterungen [MHG07, Kim07], eigene grafische Sprachen zur Beschreibung von Zielen [MAW07] oder spezielle Java-Code-Annotationen zur Markierung und Prüfung von Musterimplementierungen mit Hilfe von Java Reflection und selbst implementierten Prüfungen in einem speziellen Compiler [LSV07].

porale Logik [Tai07b, HKM07, DAC07, Ble07, Gas07] oder Kalküle [SS07, Lan07, Gas07] zur eindeutigen Beschreibung der Eigenschaften von Entwurfsmustern verwendet. Solche Formalisierungen eignen sich aufgrund ihrer Präzision, Eindeutigkeit und mathematischen Grundlage insbesondere für Beweise und ermöglichen z.B. den Einsatz von Theorembeweisern [HKM07, DAC07, SS07].

Taibi

Taibi kombiniert in seiner Musterspezifikationssprache Prädikatenlogik erster Ordnung zur Spezifikation der Entwurfsstruktur eines Musters und temporale Logik (temporal logic of actions) zur Verhaltensspezifikation [Tai07b]. Die Spezifikationen nutzt er, um zu beweisen, dass zwei Muster wie Observer und Mediator ihre Eigenschaften auch in einer Komposition beider Muster erhalten. Die Beweise werden manuell geführt.

Mikkonen et al.

Mikkonen et al. verfolgen einen ähnlichen Ansatz, um den Erhalt bestimmter Eigenschaften und bestimmten Verhaltens in Musterkompositionen zu beweisen [Mik98, HKM07]. Sie setzen Theorembeweiser ein [HKM07]. Ist der Beweis auf einer Komposition von Musterspezifikationen erfolgt, kann die geprüfte Komposition wie ein neues Muster immer wieder angewandt und das Ergebnis mit Hilfe eigenschaftserhaltender Verfeinerungsregeln angepasst werden.

Dong et al., Smith et al.

Analog zu Mikkonen et al. setzen auch Dong et al. [DAC07, DACY07] und Smith et al. [SS07, Smi12] Theorembeweiser zum Beweisen korrekter Musterkompositionen ein. Smith und Stotts definieren zusätzlich sogenannte *Elemental Design Patterns* [Smi12] – die Grundbausteine der Entwurfsmuster – und nutzen diese zur Spezifikation komplexerer Entwurfsmustern wie z.B. des GoF-Musters *Decorator* [GHJV95]. Sie verwenden das ρ-Kalkül, eine Erweiterung des ς-Kalküls, zur Kombination von Elemental Design Patterns zu komplexeren Entwurfsmustern und zur Beweisführung [SS07, Smi12].

Blewitt

Blewitt führt seine Beweise im Gegensatz zu den vorgenannten Ansätzen nicht auf Ebene der Musterspezifikationen, sondern auf Ebene der Musterimplementierungen. Er beweist oder widerlegt die korrekte Implementierung eines Musters im Java-Code [Ble07]. Dabei setzt er eine Proof Engine namens HEDGEHOG ein.

Soundarajan & Hallstrom

Auch Soundarajan und Hallstrom stellen sicher, dass ein Muster korrekt implementiert ist [SH04, SH07]. Dazu setzen sie neben ihrer Formalisierung und geeigneter Werkzeuge ein Laufzeitmonitoring und Aspekt-orientierte Programmierung zur Prüfung des Verhaltens einer Musterimplementierung ein.

Lano, Herranz & Moreno-Navarro

Lano [Lan07] sowie Herranz und Moreno-Navarro [HMN07] wählen noch einen anderen Weg. Sie beweisen, dass Musteranwendungstransformationen oder -operationen bestimmte Eigenschaften erfüllen und somit ein Muster immer korrekt anwenden.

Bayley & Zhu

Bayley und Zhu setzen eine an die UML angelehnte auf Prädikatenlogik basierende Musterspezifikation ein und komplementieren diese mit einer Algebra, welche Operatoren zur Musteranwendung und -komposition bietet [BZ07, ZB13]. Mit Hilfe dieser Algebra können Musterimplementierungen auf Korrektheit geprüft werden.

Eden et al.

Eden, Gasparis et al. setzen eine Kombination aus einer formalen, mathematisch fundierten, objektorientierten Sprache Class-Z und einer grafischen, relativ übersichtlichen und lesbaren, objektorientierten Sprache LePUS3 ein [Gas07, GENK08, NGEK09, EN11]. Beide Sprachen lassen sich aufeinander abbilden und beschreiben objektorientierte Softwareentwürfe (Programmstruktur). Class-Z ba-

siert hauptsächlich auf Prädikatenlogik erster Ordnung und kann für Beweise verwendet werden, während LePUS3 eine grafische Repräsentation von Class-Z-Spezifikationen darstellt und insbesondere zur Visualisierung vorhandener Softwarestrukturen [GENK08, EN11] oder von Entwurfsmustern genutzt werden kann. Der Ansatz wird vor allem zur Prüfung von Quellcode auf Konformität mit zuvor spezifizierten Mustern verwendet [NGEK09, EN11]. Auf diesen bemerkenswerten Ansatz gehe ich in den Abschnitten 9.1.2 und 9.4.1 näher ein.

Mak et al.

Mak et al. [MCL03, Mak04] erweitern den Ansatz von Eden et al. und ermöglichen ähnlich wie Smith et al. die formale Spezifikation von Musterkompositionen.

Die hier genannten Formalisierungen ergänzen die informellen Beschreibungen von Entwurfsmustern aus der Literatur und ermöglichen genaue Analysen und Beweisführungen. Gleichzeitig sind sie aufgrund ihrer Komplexität und des notwendigen Theoriewissens für die meisten Softwareentwickler und Architekten schwer lesbar und in der Praxis kaum einsetzbar. Die Formel-basierte, textuelle Notation kann insb. bei der Repräsentation von Beziehungen wie Assoziationen und Vererbung zwischen Klassen relativ schnell zu unübersichtlichen Darstellungen im Vergleich zu grafischen Notationen führen. Beweisführungen sind bei meinem Ansatz nicht im Fokus. Stattdessen wird eine möglichst einfach zu handhabende, verständliche Notation für den praktischen Einsatz bei Musteranwendungen benötigt. Darum verwende ich in meinen Musterspezifikationen eine grafische, UML-ähnliche Notation, welche aufgrund der Bekanntheit der UML den meisten Softwareentwicklern und Architekten relativ leicht zugänglich sein sollte.

nicht Beweise, sondern Praxistauglichkeit im Fokus

9.1.2 Musterspezifikation zwecks Werkzeug-Unterstützung im Forward Engineering

Für den Einsatz von Entwurfsmustern in Werkzeugen wird nicht zwingend eine mathematische Formalisierung von Entwurfsmustern benötigt. Es gibt diverse Ansätze, in denen z.B. UML-Erweiterungen oder eigene grafische Modellierungssprachen (Domänen-spezifische Sprachen) zur eindeutigen, Werkzeug-verarbeitbaren Spezifikation von Entwurfsmustern verwendet werden.

Florijn et al.

Florijn et al. stellten als einige der ersten einen Ansatz vor, bei welchem zum einen Entwurfsmuster modelliert und semiautomatisch angewandt und zum anderen Musterimplementierungen auf Korrektheit geprüft werden können [FMvW97]. Sie verwenden eine Art erweiterbares Meta-Modell für objektorientierte Programmstrukturen und beschreiben damit Entwurfsmuster durch eine Art Programmstruktur-Schablone.

Ein Muster wird durch einen Graphen spezifiziert (siehe Abb. 9.2 rechts). Die Knoten bilden eine Art Platzhalter für Elemente eines Programms wie z.B. eine Klasse, eine Methode oder ein Stück Programmcode. Weitere, beliebig ergänzbare Platzhalter repräsentieren allgemeinere Konzepte wie eine Musterimplementierung. Beziehungen zwischen den Platzhaltern werden durch benannte Kanten eines bestimmten Typs repräsentiert. Eine method-Kante beschreibt z.B., dass eine Observer-Klasse eine update-Methode enthält.

generischer Platzhalter-Graph

Im Gegensatz zu meinem Ansatz können Implementierungsvarianten wie z.B. unterschiedliche Anzahlen an ConcreteObserver-Klassen im Observer-Muster (vgl. Set Fragments in Abb. 8.2, S. 171) nicht spezifiziert werden. Verhalten kann nur

keine Varianten

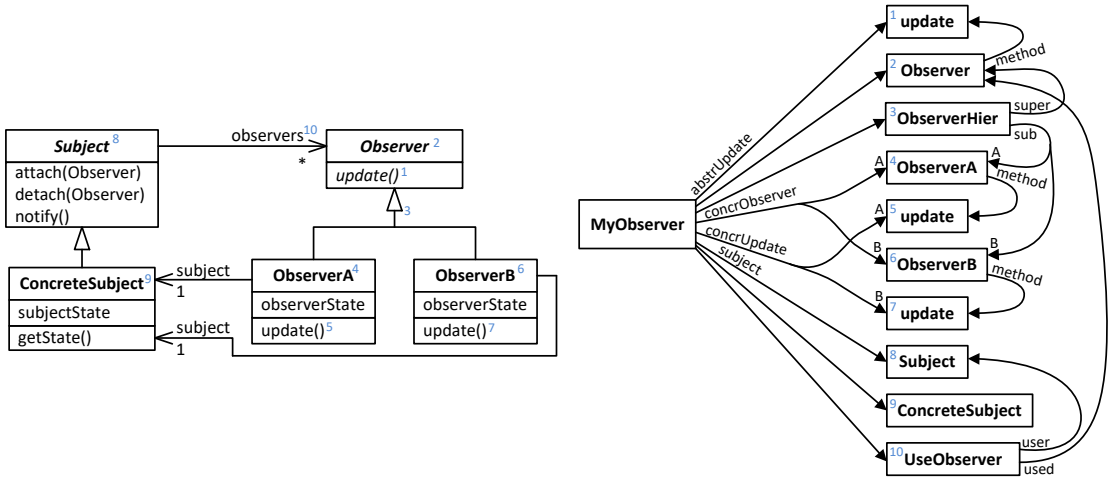


Abbildung 9.2: Spezifikation des Observer-Musters nach Florijn et al. [FMvW97], *links:* Muster in UML-Notation (aus OMT-Notation abgeleitet), *rechts:* Musterspezifikation in der Notation von Florijn et al.

in Form von fertigem Quellcode spezifiziert werden, was den Ansatz Plattformabhängig macht. Außerdem stellen Florijn et al. alle Knoten und Kanten in ihrer Notation gleich dar, was die Übersichtlichkeit im Vergleich zu anderen Musterspezifikationssprachen deutlich einschränkt.

Eden et al.

Eine der frühesten und einflussreichsten wissenschaftlichen Arbeiten auf dem Gebiet der Musterspezifikation ist die von Eden, Gasparis, Nicholson et al. [Ede00, ENG07, NGEK09, EN11]. Wie in Abschnitt 9.1.1 beschrieben, setzen sie eine grafische Musterspezifikationssprache LePUS3 in Kombination mit einer mathematisch formalen Musterspezifikationssprache Class-Z basierend auf prädikatenlogischen Formeln erster Ordnung ein. LePUS3-Musterspezifikationen lassen sich in Class-Z-Spezifikationen übersetzen und umgekehrt (siehe Bsp. in Abb. 9.3, 9.4).

Formelbasierte und grafische Notation von Constraints

Der Ansatz von Eden et al. ist auf zweierlei Weise besonders. Zum einen ist die Semantik der grafischen Musterspezifikationen mathematisch formal und eindeutig definiert. Zum anderen können zahlreiche Implementierungsvarianten eines Musters mit Hilfe spezieller Sprachkonstrukte besonders kompakt in Musterspezifikationen erfasst werden.

Zum Beispiel lässt sich eine ganze Klassenhierarchie bei der Spezifikation des Musters Abstract Factory [GHJV95] (siehe Abb. 8.5, S. 176) durch nur ein einziges Sprachkonstrukt, ein Dreieck, repräsentiert werden (siehe Abb. 9.3). Mit einem Schatten kann spezifiziert werden, dass es von einem Element wie einer PRODUCTS-Klassenhierarchie oder einer Factory-Methods-Operation beliebig viele in einer Musterimplementierung geben kann (entspricht einer Potenzmenge $\mathcal{P}$ in Abb. 9.4). Beziehungen werden als totale oder isomorphe Abbildungen zwischen Mengen von Entwurfselementen modelliert, sodass die Elemente in den Mengen nicht aufgezählt werden müssen. Die Produce-Kante in Abb. 9.3 repräsentiert z.B. das Instanziieren von Klassen in der PRODUCTS-Hierarchie durch Factory Methods. Alle Kanten repräsentieren sowohl direkte als auch indirekte Beziehungen, wodurch weitere Implementierungsvarianten abgedeckt werden. Einige dieser Ideen zur Repräsentation von Implementierungsvarianten wie die Spezifikation von

Ideengeber für Set Fragments & transitive Relationen

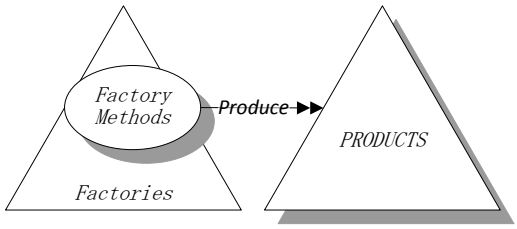


Abbildung 9.3: LePUS3-Spezifikation des Musters Abstract Factory nach Eden et al. [ENG07] (Legende: http://lepus.org.uk/ref/legend/legend.xml)

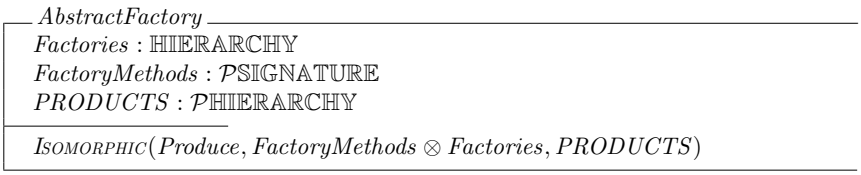


Abbildung 9.4: Class-Z-Spezifikation des Musters Abstract Factory nach Eden et al. [ENG07] (Legende: http://lepus.org.uk/ref/legend/legend.xml)

Mengen (Set Fragments, Abschn. 3.5.2) und die Berücksichtigung indirekter Beziehungen (Abschn. A.2.1) greife ich in meiner Musterspezifikationssprache auf.

Der Hauptanwendungszweck von LePUS3 ist die Überprüfung existierenden Programmcodes auf Konformität mit einer Musterspezifikation in einer Forward- oder Re-Engineering-Phase. Dafür wird neben einem Muster auch der Quellcode in LePUS3-Diagrammen (sogenannten Codecharts) erfasst [GENK08, EN11] und auf Basis von prädikatenlogischen Formeln auf Konformität zur Musterspezifikation geprüft [NGEK09].

für Musteranwendungen ungeeignet

Zum Herleiten einer Musterimplementierung aus einer Musterspezifikation ist der Ansatz jedoch nicht geeignet. Zum Beispiel lassen Klassenhierarchien wie in Abb. 9.3 völlig offen, welche Klassen mit welchen Vererbungsbeziehungen aus der Spezifikation abgeleitet werden sollen. Ebenso ist unklar, welche Operationen hergeleitet werden sollen und welche Klassen sie instanziieren sollen, sodass jede Factory-Klasse nur Produkte aus einer Produktfamilie instanziiert wie es laut Musterbeschreibung [GHJV95] gefordert wird (vgl. Abb. 8.5, S. 176). Erschwerend kommt hinzu, dass nur wenige[3] Spezifikationen so kompakt sind wie die in Abb. 9.3. Der Großteil der Musterspezifikationen ist eher mit der des Observer-Musters in Abb. 9.5 vergleichbar. Die Notation von Operationen in Form von Ellipsen benötigt unnötig viel Platz. Unterschiedliche Arten von Beziehungen wie *erben von* (Inherit), *Aufruf von* (Call) und *Referenz auf* (Member) lassen sich nur anhand ihrer Labels unterscheiden.

Le Guennec et al. UML-Erweiterung

Le Guennec et al. stellen eine UML-Erweiterung zur Spezifikation von Entwurfsmustern auf Ebene des UML-Meta-Modells vor [LGSJ00, SLGJ00]. In Anlehnung an die Arbeit von Eden [Ede00] erweitern sie das UML-Meta-Modell um das Konzept der Meta-Level-Kollaborationen und Mustervorkommen. In einer Meta-Level-Kollaboration wird ein Muster als Kombination aus Platzhaltern für Klassen

[3] Nach meiner Einschätzung sind nur 5 der 23 GoF-Muster ähnlich kompakt spezifiziert wie in Abb. 9.3: Abstract Factory, Factory Method, Bridge, Flyweight und Iterator [ENG07]

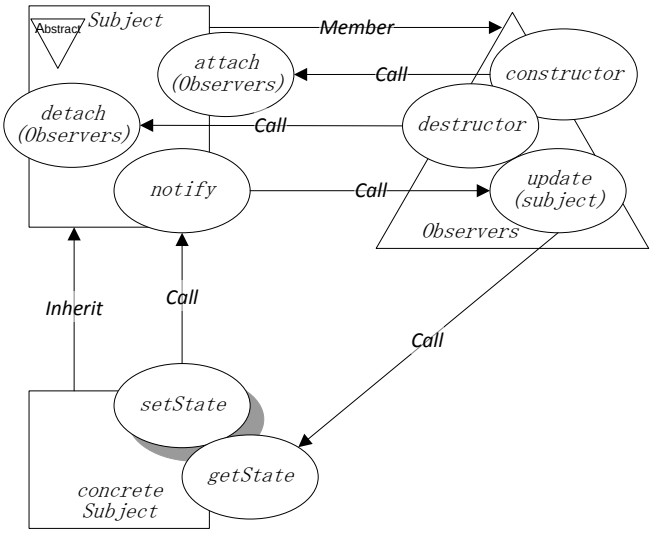


Abbildung 9.5: LePUS3-Spezifikation des Observer-Musters nach Eden et al. [ENG07] (Legende: http://lepus.org.uk/ref/legend/legend.xml)

und ihre Features (Methoden oder Attribute) modelliert. Zur Konkretisierung der Struktur und des Verhaltens schlagen Le Guennec et al. vor, OCL-Constraints in Kombination mit Temporallogik zur Beschreibung von Wohlgeformtheitsregeln sowie von Vor- und Nachbedingungen für Operationen zu verwenden. Es bleibt jedoch bei der groben Idee ohne Ausarbeitung eines umsetzbaren Konzepts.

nicht umgesetzt

Maplesden et al. modellieren Entwurfsmuster und Anwendungsstellen in einer die UML erweiternden Sprache, der Design Pattern Modeling Language (DPML) [MHG02, MHG07]. Die Struktur eines Musters wird in einer Art Klassendiagramm mit eigener Notation modelliert (siehe Abb. 9.6). Die Musterspezifikationen dienen einer Werkzeug-gestützten Musteranwendung in UML-Modellen. Implementierungsvarianten werden durch sogenannte Dimensionen spezifiziert. Damit lässt sich ausdrücken, dass es zu einem Element in einer Musterspezifikation mehrere Entwurfselemente in einer Musterimplementierung geben kann. Damit sind Dimensionen mit Set Fragments (siehe Abschn. 3.5.2) vergleichbar. Bei der Spezifi-

Maplesden et al.

UML-Erweiterung

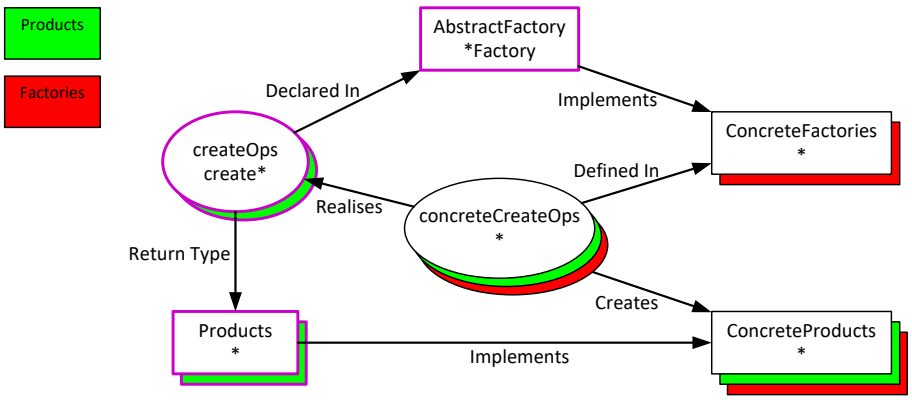


Abbildung 9.6: Spezifikation des Musters Abstract Factory nach Maplesden et al. [MHG07]

kation der Musters Abstract Factory in Abb. 9.6 werden die beiden Dimensionen Products und Factories verwendet, welche durch eine grüne bzw. rote Schattierung dargestellt werden.

Abgesehen von der Notation unterscheidet sich die von Maplesden et al. vorgestellte Musterspezifikationssprache von meiner insb. durch folgende Punkte.

Einschränkungen bei Verhalten, keine transitiven Relationen

Bei der Spezifikation von Entwurfsstruktur können weder Kopplungsrestriktionen (Abschn. 3.5.3) noch Subsysteme (Abschn. 3.4.3) modelliert werden. Die Transitivität von Beziehungen (Abschn. A.2.1) wird nicht berücksichtigt (d.h. eine Kante repräsentiert eine direkte, aber keine indirekte z.B. Vererbungsbeziehung). Verhalten kann nur unzureichend spezifiziert werden. Bisher werden spezielle Relationen wie Creates (Abb. 9.6) oder Invokes verwendet. Daraus lässt sich jedoch, im Gegensatz zu Aktionen bei meinem Ansatz (Abschn. 3.4.2), kein ausführbares Verhalten ableiten. Solche Relationen werden nur zu Konsistenzprüfungen verwendet. Es gibt erste Überlegungen, das Verhalten in erweiterten UML-Sequenzdiagrammen zu modellieren, die Ideen dazu sind jedoch bisher nur skizziert worden.

Mak et al.

Mak et al. greifen Ideen von Le Guennec et al. auf und modellieren Entwurfsmuster mit Meta-Level-Kollaborationen und speziellen Stereotypen [MCL04, Mak04].

UML-Erweiterung

Sie greifen auch die Idee von Dimensionen von Maplesden et al. auf, um eine variable Anzahl von Vorkommen bestimmter Teile eines Musters in Musterimplementierungen zu spezifizieren. Beziehungen und Verhalten werden durch spezielle UML-Relationen, markiert durch einen Stereotypen wie «invoke», modelliert. Analog zu Eden et al. und den darauf aufbauenden Vorarbeiten von Mak et al. [MCL03] wird bei diesen Relationen Transitivität berücksichtigt.

Einschränkungen bei Verhalten

Der Ansatz von Mak et al. unterliegt jedoch ähnlichen Einschränkungen wie die vorgenannten Arbeiten. Das Verhalten ist zu vage spezifiziert, um daraus ausführbare Verhaltensmodelle oder Quellcode im Rahmen einer Musteranwendung herzuleiten. Eine Abbildung auf ausführbare Modelle wie bei meinem Ansatz fehlt (siehe Abschn. A.3.2). Kopplungsrestriktionen und Subsysteme können nicht modelliert werden.

Kim et al.

Kim et al. setzen ebenfalls eine Erweiterung des UML-Meta-Modells zur Modellierung von Entwurfsmustern ein [FKGS04, KW05, Kim07, KS08]. Die Spezifikationen werden zur Musteranwendung [KW05] und zur Prüfung der Konsistenz von Anwendungsstellen eingesetzt [KS08].

UML-Erweiterung

Muster werden in speziellen Klassen-, Sequenz- und Zustandsmaschinen-Diagrammen spezifiziert. Alle UML-Modellelemente einer Musterspezifikation werden durch spezielle Varianten der UML-Meta-Modell-Klassen modelliert. So wird z.B. beim Observer-Muster die Rolle *Observer* als spezielle Klasse namens Observer (Unterklasse der UML-Meta-Modell-Klasse Class) modelliert (siehe Abb. 9.7 (b)). Analog dazu werden auch Assoziationen in der Spezifikation als spezielle Varianten der UML-Assoziationen modelliert. Eine Musterimplementierung besteht damit ausschließlich aus Instanzen dieser speziellen UML-Meta-Modell-Klassen. Alle Elemente einer Musterspezifikation (Klassen, Attribute, etc.) werden mit einer Kardinalität versehen (siehe Abb. 9.7 (a)). Diese gibt an wie viele Instanzen dieses Elements in einer Musterimplementierung vorkommen dürfen. Somit lassen die Spezifikationen einige Implementierungsvarianten je Muster zu. Verhalten von Methoden wird durch Vor- und Nachbedinungen konkretisiert, ausgedrückt in einer OCL-Erweiterung.

Der Ansatz von Kim et al. macht überlappende Musteranwendungen unmöglich. Durch den Einsatz von speziellen Meta-Modell-Klassen je Musterrolle kann z.B.

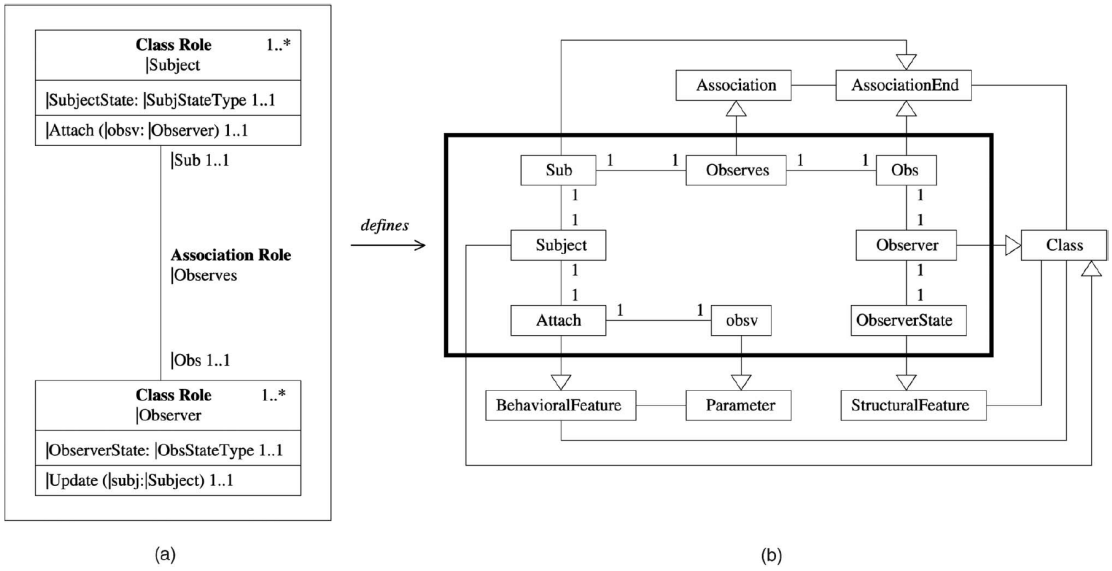


Abbildung 9.7: (a) Spezifikation des Observer-Musters und (b) zugehörige UML-Meta-Modell-Klassen nach Kim et al. [FKGS04]

überlappende Musteranwendungen unmöglich

eine Klasse in einem UML-Klassendiagramm nur eine Rolle eines Musters einnehmen, da sie Instanz nur einer UML-Meta-Modell-Klasse sein kann. Kardinalitäten an einzelnen Modellelementen lassen zwar eine variable Anzahl solcher Elemente in einer Musterimplementierung zu, allerdings kann im Gegensatz zu meinem und dem Ansatz von Maplesden et al. nicht spezifiziert werden, dass bestimmte Modellelemente nur zusammen mit anderen mehrfach vorkommen können (z.B. eine abstrakte und konkrete *create*-Methode und die zugehörige abstrakte und konkrete Produktklasse bei dem Muster Abstract Factory, vgl. Abb. 8.6, S. 176 und Abb. 9.6). Wie bei Maplesden et al. können auch bei Kim et al. keine Kopplungsrestriktionen und keine Subsysteme spezifiziert werden, ebenso wird die Transitivität von Beziehungen nicht berücksichtigt.

Park et al.

UML-Erweiterung, kein Verhalten

Park et al. verwenden eine zu Kim et al. sehr ähnliche Musterspezifikationssprache auf Basis einer Erweiterung der UML mit geringfügig anderer Notation [PRW08, Par07]. Im Gegensatz zu Kim et al. werden hier allerdings nicht Meta-Klassen, sondern Stereotypen zur Spezifikation von Musterrollen verwendet, was überlappende Musteranwendungen möglich macht. Musterverhalten wird nicht spezifiziert. Ebenso fehlt die Spezifikation zu vermeidender Abhängigkeiten.

Mili & El-Boussaidi

UML-Erweiterung

Limitierung wie bei Kim et al.

Mili und El-Boussaidi unterscheiden bei der Musterspezifikation das von einem Muster gelöste Problem von der vorgeschlagenen Lösung und ergänzen diese durch Modelltransformationen zur automatischen Musteranwendung [MEB05, EBM07]. Sowohl das Problem als auch die Lösung zu einem Muster werden separat mit Hilfe eines erweiterten UML-Meta-Modells modelliert. Ähnlich wie bei Kim et al. werden auch hier mit jedem zu spezifizierenden Entwurfsmuster eigene Meta-Modell-Klassen definiert. Das als „Problem“ bezeichnete Konzept scheint hier jedoch bloß eine Art abstraktere Repräsentation der Entwurfslösung zu einem Muster zu sein. Bis auf die zweigeteilte Spezifikation eines Musters auf unterschiedlichen Abstraktionsebenen und die zusätzliche Spezifikation von Modelltransformationen unterscheidet sich der Ansatz kaum von dem Ansatz von Kim et al.

Yacoub & Ammar

Komponenten

nicht umgesetzt, widerspricht Zitat, S. 23

Einen ganz anderen Ansatz zur Spezifikation von Entwurfsmustern verfolgen Yacoub und Ammar [YA03]. Sie spezifizieren die Entwurfsstruktur in klassischen UML-Klassendiagrammen in Form eines exemplarischen Entwurfs. Diesen Entwurf eines Musters kapseln sie in einer Komponente mit spezieller Schnittstelle, welche das Kombinieren mehrerer so spezifizierter Muster zu einem Gesamtsystem ermöglichen soll. Es gibt jedoch scheinbar keinerlei Werkzeuge zwecks Evaluation des Ansatzes. Die Spezifikation und Kombination der Muster zu einem Gesamtsystem erfolgt nach meinem Verständnis bisher komplett auf Papier und manuell. Es gibt keine Musterspezifikationssprache, sondern vielmehr eine Beschreibung der groben Vorgehensweise bei der Spezifikation der Muster, deren Kombination und Anwendung durch schrittweise Konkretisierung zu Musterimplementierungen.

Enterprise Architect & Together

Neben den zahlreichen wissenschaftlichen Arbeiten gibt es auch einige kommerzielle Werkzeuge, welche das Spezifizieren von Entwurfsmustern ermöglichen, um sie später in UML-Modellen anwenden zu können. Zu diesen gehören insbesondere Enterprise Architect[4] von Sparx Systems und Borland Together[5].

UML-Modell-Schablonen

Enterprise Architect ermöglicht seinen Benutzern das Modellieren von UML-Diagrammen, welche anschließend als „Muster“ gespeichert werden können, um später von weniger erfahrenen Entwicklern in UML-Modellen angewandt zu werden [Spab]. Dabei werden die Namen der Modellelemente (z.B. der Klassen und Methoden in einem Klassendiagramm) zu Variablen gemacht und es wird je Modellelement angegeben, ob sie bei einer Musteranwendung immer neu erzeugt oder mit einem existierenden Element im Entwurf gemischt (merge) werden können.

Analog dazu bietet auch Together seinen Benutzern die Möglichkeit, Muster in UML-Klassendiagrammen zu spezifizieren [Bor11]. Die so spezifizierten Klassendiagrammstrukturen können analog zu Enterprise Architect mit minimalen Anpassungen in ein existierendes UML-Klassendiagramm übertragen werden. Außerdem kann in Klassendiagrammen nach so spezifizierten Strukturen gesucht werden, z.B. zur nachträglichen Dokumentation der angewandten Muster.

keine Varianten, unflexibel

Bei beiden Werkzeugen werden Entwurfsmuster jedoch eher als Modell-Schablonen verstanden. Es können keine Implementierungsvarianten eines Musters spezifiziert werden. Im Wesentlichen werden nur die Namen der Klassen und Methoden bei Musteranwendung angepasst. Die spezifizierte Struktur wird bei Musteranwendung 1-zu-1 in den Entwurf übernommen. Verhalten wird entweder in Form von UML-Diagramm-Schablonen oder überhaupt nicht spezifiziert. Kopplungsrestriktionen und Subsysteme werden nicht unterstützt.

Wenzel

Platzhalter-Graph

Wenzel spezifiziert Entwurfsmuster, indem er die Entwurfsstruktur basierend auf einem einfachen, allgemeinen Rollenmodell beschreibt [Wen05b, Wen05a], vergleichbar mit dem Ansatz von Florijn et al. Wenzel verwendet seine Musterspezifikationen jedoch nicht für Musteranwendungen, sondern für die Erkennung und Prüfung von ggf. unvollständigen Musterimplementierungen während des initialen Entwurfs oder nachträglich bei Re-Engineering-Tätigkeiten. Rollen repräsentieren beliebige Elemente in UML-Klassendiagrammen. Sie lassen sich hierarchisch strukturieren. Ihr Typ gibt an, welchen Elementen sie zugeordnet werden können, z.B. einer Klasse oder einer Assoziation. Eine Multiplizität gibt an, ob die Rolle

[4] Enterprise Architect: http://www.sparxsystems.de
[5] Together: http://www.borland.com/de-DE/Products/Requirements-Management/Together

von einem oder mehreren UML-Elementen eingenommen werden können. OCL-Constraints ergänzen weitere Bedingungen, welche bei einer Rollenzuordnung erfüllt sein müssen, z.B. dass eine Rolle nur von abstrakten Klassen eingenommen werden kann oder eine Assoziation auf einen bestimmten Typen verweisen muss.

Wenzels Ansatz ist vor allem auf das Erkennen von Musterimplementierungen in UML-Klassendiagrammen ausgerichtet. Zum einen berücksichtigt er kein Verhalten. Zum anderen deckt er in seinen Spezifikationen Implementierungsvarianten nur durch Angabe einer Rollen-Multiplizität ab. Weitere Varianten werden nur durch den Erkennungsalgorithmus berücksichtigt, indem auch unvollständige Rollenzuordnungen als Fundstellen erkannt werden. Ähnlich wie bei den zuvor genannten kommerziellen Werkzeugen könnten Wenzels Musterspezifikationen bei einer automatisierten Musteranwendung nur als Entwurfsstrukturschablone dienen. Die Entwurfsstruktur würde 1-zu-1 übertragen, Elemente mit Multiplizität * würden ggf. vervielfältigt werden. OCL-Constraints, welche Einschränkungen auf UML-Klassendiagramm-Ebene beschreiben, machen die Spezifikationen abhängig von der verwendeten Modellierungssprache, einer bestimmten Version des UML-Klassendiagramm-Meta-Modells. Kopplungsrestriktionen könnten vermutlich mit OCL beschrieben werden, fehlen jedoch in seinen Spezifikationen.

Modell-Schablonen, kaum Flexibilität, kein Verhalten

9.1.3 Musterspezifikation zwecks Werkzeug-Unterstützung im Reverse Engineering

Während Wenzel das Erkennen von partiellen Musterimplementierungen vor allem in der Forward-Engineering-Phase verfolgt, existieren diverse Ansätze, welche die nachträgliche Erkennung von Musterimplementierungen im Rahmen eines Reverse oder Re-Engineerings zum Ziel haben. Die Musterspezifikationen bei diesen Ansätzen haben jedoch ähnliche Anforderungen zu erfüllen wie die von Wenzel, nämlich unterschiedliche Musterimplementierungen in möglichst wenigen, kompakten Spezifikation zu erfassen. Da Reverse Engineering (insb. aufgrund abweichender Anforderungen[6] und nicht vielversprechender Ergebnisse aus den Vorarbeiten [BBF+11]) nicht im Fokus meiner Arbeit liegt, gehe ich nur auf einige wenige Arbeiten in dieser Kategorie ein.

Reverse Engineering nicht im Fokus

Guéhéneuc et al. spezifizieren Entwurfsmuster sowohl zum Zweck der Musteranwendung als auch zur nachträglichen Erkennung von Musterimplementierungen [AACGJ01, AAG01], der Fokus ihrer Forschung liegt jedoch auf dem Reverse Engineering. Sie modellieren Entwurfsmuster mit Hilfe einer eigenen Musterspezifikationssprache. Diese entspricht im Prinzip einem vereinfachten, minimal erweiterten UML-Klassendiagramm-Meta-Modell. Für jede Musterspezifikation wird eine das Muster repräsentierende Klasse in dem Meta-Modell der Musterspezifikationssprache ergänzt. Sollten die bisherigen Sprachkonstrukte zur Beschreibung eines Musters nicht reichen, werden weitere Klassen bzw. Sprachkonstrukte ergänzt. Die Klassen werden so implementiert, dass sie u.a. Code zur Anwendung des Musters enthalten [AAG01]. Bei Anwendung eines Musters wird Java-Quellcode ge-

Guéhéneuc et al.

UML-Erweiterung

[6] Während beim Forward Engineering zwecks Musteranwendung vollständige Spezifikationen von Musterimplementierungen benötigt werden, ist es beim Reverse Engineering geschickter, für das automatische Erkennen von Musterimplementierungen irrelevante Details wegzulassen [Tra06, Kap. 2.2, 2.3] [vDMT10a].

neriert. Neben dem Prototypen für Musteranwendungen, PatternsBox, wird auch ein Werkzeug zur Erkennung von Musterimplementierungen vorgestellt, Ptidej. Dieses verwendet dieselben Musterspezifikationen.

umständliche Spezifikation, keine Varianten

Der Ansatz ist am ehesten mit dem von Kim et al. vergleichbar, verwendet jedoch keine Kardinalitäten zur Erfassung diverser Implementierungsvarianten. Verhalten wird nicht deklarativ beschrieben, sondern operational mit Hilfe explizit ausimplementierter Musteranwendungsalgorithmen in den Meta-Modell-Klassen der Musterspezifikationssprache. Mit jedem neuen Muster müssen weitere Klassen in diesem Meta-Modell implementiert werden. Die Notwendigkeit, für jede Musterspezifikation weitere Meta-Modell-Klassen zu implementieren, machen das Spezifizieren von Entwurfsmustern deutlich aufwändiger als bei Ansätzen mit einer vordefinierten, abgeschlossenen Musterspezifikationssprache.

Niere et al.

abstrakter Syntaxgraph & Graphtransformationsregel

Niere et al. spezifizieren Entwurfsmuster, indem sie die unterschiedlichen Implementierungsvarianten eines Musters als Objektstrukturen in abstrakter Syntax spezifizieren [NSW+02, Nie04, Wen07, vDMT10a, vDMT10b]. Die Spezifikationen entsprechen im Prinzip Graphtransformationsregeln, welche die spezifizierte Objektstruktur in einem Modell eines UML-Diagramms oder in einem Java-Quellcode-Modell (genauer: in einem abstrakten Syntaxgraphen) suchen und bei Erfolg die Fundstelle markieren.

zu detailliert

Die Spezifikationen sind über ein bestimmtes Meta-Modell typisiert, z.B. über Klassen eines UML-Klassendiagramm-Meta-Modells. Das macht Spezifikationen relativ detailliert und umfangreich. Während dieser Detailgrad für die Suche nach Musterimplementierungen notwendig ist, lassen sich Entwurfsmuster zum Zweck der Musteranwendung wie bei meinem Ansatz kompakter, auf einem höheren Abstraktionslevel beschreiben. Bei Niere et al. werden diverse Implementierungsvarianten eines Musters geschickt mit Sprachkonstrukten wie Mengen (vergleichbar mit Kardinalitäten einzelner Rollen), Pfaden (indirekte Beziehungen) sowie Vererbungsbeziehungen und Kompositionen von Musterspezifikationen modelliert. Das Konzept der Mengen habe ich bei meinem Ansatz zu Set Fragments weiterentwickelt (siehe Abschn. 3.5.2), sodass ganze Teilgraphen statt einzelner Elemente in Musterimplementierungen mehrfach vorkommen können.

Mengen = Vorstufe zu Set Fragments

Backofen et al.

komplexe Übersetzung, weniger Treffer

Im Rahmen einer Vorarbeit wurde ein Vorgänger meiner Musterspezifikationssprache sowohl zur Musteranwendung als auch zur Erkennung von Musterimplementierungen verwendet. Dabei wurden die abstrakten Forward-Engineering-Musterspezifikationen zwecks Anwendung des Mustererkennungsverfahrens von Niere et al. in detailliertere Reverse-Engineering-Musterspezifikationen übersetzt. Die Übersetzung war komplex, unvollständig und führte im Vergleich zu Niere et al. zu weniger erkannten Musterimplementierungen [BBF+11, Kap. 6.3], weil die generierten Spezifikationen zu wenige Varianten abdeckten. Die Spezifikationen mussten verallgemeinert werden. Aufgrund dieser nicht vielversprechenden Ergebnisse habe ich den Reverse-Engineering-Aspekt in meiner Arbeit ausgeklammert.

9.2 Dokumentation von Musteranwendungsstellen

Nachdem ein Muster angewandt wurde ist die zugehörige Anwendungsstelle im Entwurf, also alle das Muster implementierenden Teile, nicht leicht als solche zu

erkennen. Darum wird empfohlen, die Anwendungsstellen in irgendeiner Form zu dokumentieren. Das kann z.B. in Form von Kommentaren oder speziellen Markierungen, Modellen und Visualisierungen erfolgen. Im Folgenden stelle ich Arbeiten vor, welche automatisiert verarbeitbare Repräsentationen von Musteranwendungsstellen (Modelle) einsetzen (Abschn. 9.2.1) oder menschenlesbare Visualisierungen von Anwendungsstellen bereitstellen (Abschn. 9.2.2). Auf Ansätze zur informellen Dokumentation von Anwendungsstellen, z.B. durch Kommentare, gehe ich nicht ein, weil sie sich nicht für eine Werkzeug-gestützte Musteranwendung sowie zur Visualisierung und Konsistenzprüfung von Musterimplementierungen eignen.

Arbeiten gruppiert nach Einsatzzweck

9.2.1 Modellierung von Anwendungsstellen

Damit einst angewandte Muster leicht erkannt und die entstandenen Musterimplementierungen geeignet visualisiert und überprüft oder nachträglich angepasst werden können, werden die Anwendungsstellen oder die Musterimplementierungen explizit in Modellen erfasst und ggf. persistiert.

Maplesden et al.

Maplesden et al. modellieren Musterimplementierungen in Musterinstanziierungsdiagrammen (DPML Instantiation Diagrams) [MHG02, MHG07]. Diese haben die gleiche Struktur und Notation wie Musterspezifikationsdiagramme (DPML Specification Diagrams) (siehe Abb. 9.6, S. 213), verknüpfen (binding) jedoch zusätzlich die spezifizierten Musterrollen mit Elementen eines UML-Klassendiagramms, z.B. mit Klassen oder Methoden. Wenn es die Musterspezifikation zulässt, können einer Rolle auch mehrere UML-Modell-Elemente zugeordnet werden. Jede Anwendungsstelle kann durch Ergänzen zusätzlicher Elemente in einem Musterinstanziierungsdiagramm angepasst werden. Die aus der Musterspezifikation übernommenen Elemente sind dabei unveränderbar.

Rollenzuordnungen & Musterinstanzmodell

Dieser Ansatz ähnelt meinem am meisten. Auch ich verwende eine Art Musterinstanziierungsmodell, welches die Rollenzuordnungen und eine abstrakte Repräsentation einer Musterimplementierung enthält (grau in Abb. 4.3, S. 83). Bei meinem Ansatz lässt sich die Musterimplementierung jedoch nicht im Anwendungsmodell erweitern, sondern direkt im Entwurfsmodell (z.B. im Klassendiagramm). Zusätzlich zu einfachen Rollenzuordnungen werden bei mir auch Zuordnungen auf Ebene der Set Fragments erfasst (bei Maplesden et al. entsprechen diese den Dimensionen, S. 213 ff.), sodass das Mehrfachvorkommen mehrerer zusammenhängender Musterrollen gemeinsam und nicht einzeln erfasst werden (siehe SetFragmentBinding und SetFragmentInstance in Abb. 4.4, S. 85 und 4.5, S. 86).

Korrespondenzen je Element statt je Teilgraph

Außerdem unterstützt mein Ansatz Rollenzuordnungen auf Ebene von Verhaltensmodellen, während Maplesden et al. bisher nur erste Ideen zur Spezifikation von Verhalten in Sequenzdiagrammen, aber nicht zur Anwendung oder Rollenzuordnung in Verhaltensmodellen entwickelt haben. Mein Ansatz setzt aufgrund der unterschiedlichen Detailgrade von Musterspezifikationen und Verhaltensmodellen Tokens ein, mit denen eine komplexe 1-zu-n-Rollenzuordnung[7] in mehrere hierarchische Teilzuordnungen zerlegt werden kann (siehe Token in Abb. 4.5, S. 86).

keine Korrespondenzen bei Verhalten

Kim & Shen

Kim und Shen [KS08] modellieren Musteranwendungsstellen auf ähnliche Weise wie Maplesden et al. Es werden Rollenzuordnungen in Modellen erfasst, welche

[7] D.h., eine Musterrolle wird mehreren Elementen im Entwurfsmodell zugeordnet.

die Musterspezifikationen auf UML-Meta-Modell-Ebene mit Musterimplementierungen in UML-Klassendiagrammen verknüpfen (siehe S. 214).

Le Guennec et al. und Mak et al.

Le Guennec et al. [LGSJ00] sowie Mak et al. [MCL04] erweitern das UML-Meta-Modell um spezielle Abhängigkeiten, welche eine Musterimplementierung repräsentieren. Beide Ansätze halten in Modellen fest, welches Element eines Klassendiagramms welche Rolle einer Musterspezifikation einnimmt.

Wenzel

Wenzel verwendet ein einfaches Modell für Rollenzuordnungen, mit welchem er einer Rolle beliebig viele Elemente eines UML-Klassendiagramms zuordnen kann [Wen05a, Kap. 6.1]. Da nach Musterimplementierungen automatisch gesucht wird und u.a. unvollständige Musterimplementierungen erkannt werden, wird jede Rollenzuordnung auch mit einer Qualitätsbewertung in Prozent versehen, einer Art Vollständigkeitsgrad.

Together, Rational Software Architect, ModelMaker

Kommerzielle Werkzeuge bieten ebenfalls Unterstützung bei der Anwendung von Entwurfsmustern und halten die Anwendungsstellen in Modellen fest. Borland Together[8] verwendet die in der UML propagierte Modellierung von Anwendungsstellen: Kollaborationen [OMG11b]. Die Anwendungsstellen werden in UML-Klassendiagrammen durch einen Kollaborationsknoten mit benannten Kanten zu den die Musterrollen einnehmenden Klassen erfasst [Bor11]. Auf ähnliche Weise werden Anwendungsstellen, *Pattern Instance* genannt, auch in IBM Rational Software Architect[9] modelliert [LB05]. ModelMaker[10] erfasst Anwendungsstellen ebenfalls auf Modell-Ebene, jedoch nur auf Wunsch [BP99]. Hier werden Verweise auf die generierten Quellcode-Stellen in einem Modell erfasst. Details dazu konnte ich jedoch nicht in Erfahrung bringen.

gleiche Einschränkungen

Für die Arbeiten von Kim & Shen, Le Guennec et al., Mak et al., Wenzel sowie die kommerziellen Werkzeuge gelten dieselben Einschränkungen wie für Maplesden et al.: detaillierte Korrespondenzen auf wiederholbaren Teilgraphen (vergleichbar mit Set-Fragment-Instanzen) und auf Verhaltensmodellen sind nicht möglich, Entwurfsaufgaben mit Erledigungsstatus sind nicht vorgesehen.

Eden et al.

Eden et al. setzen sogenannte Codecharts bzw. LePUS3-Diagramme zur Visualisierung und Analyse von Musterimplementierungen ein [NGEK09, EN11]. Im Gegensatz zu mir dokumentieren Eden et al. die Anwendungsstellen jedoch

Anwendungsstellen nicht persistent

nicht dauerhaft, sondern erzeugen sie immer wieder zur Validierung bestimmter Programmstellen. Eine Anwendungsstelle besteht aus einer Repräsentation des Quellcodes als LePUS3-Diagramm (einem Codechart, Abb. 9.12, S. 224) und einer Zuordnung der Elemente im Codechart zu den Elementen einer Musterspezifikation (Abb. 9.5, S. 213). Die Zuordnung der Elemente erfolgt jedes Mal manuell.

Florijn et al.

Florijn et al. modellieren Musterimplementierungen auf die gleiche Weise wie Musterspezifikationen (siehe Abb. 9.2, S. 211), nämlich mit einer Art Platzhalter für Elemente eines Smalltalk-Programms [FMvW97]. Die Platzhalter werden

Platzhalter-Graph, Kopie der Musterspezifikation

Fragments genannt. Eine Struktur aus miteinander verbundenen Fragments repräsentiert ein Muster. Bei einer Musteranwendung wird so eine Struktur geklont, um von nun an eine Musterimplementierung zu repräsentieren. Ein Fragment-Graph kann also sowohl ein Muster als auch eine Musterimplementierung repräsentieren. Ein Fragment repräsentiert z.B. ein Konzept wie eine Musterimple-

[8] Together: http://www.borland.com/de-DE/Products/Requirements-Management/Together
[9] Rational Software Architect: http://www-03.ibm.com/software/products/de/ratsadesigner
[10] ModelMaker: http://www.modelmakertools.com/modelmaker/index.html

mentierung, ein Smalltalk-Objekt oder Quellcode. Außerdem enthalten die Fragments Smalltalk-Quellcode zur Validierung von Musterimplementierungen.

Im Gegensatz dazu werden Musteranwendungen bei meinem Ansatz schon vor der Implementierung während des Systementwurfs erfasst. Anwendungsstellen werden separat von den Spezifikationen in einem eigenen Modell erfasst, welches als Verknüpfung zwischen Musterimplementierungen im Entwurfsmodell und den Musterspezifikationen dient (siehe Abb. 4.3, S. 83). So können in Musterspezifikationen andere, z.B. verallgemeinernde Sprachkonstrukte wie Set Fragments verwendet werden als in Musterimplementierungen bzw. Modellen davon.

keine Teilgraph-Korrespondenzen

Baniassad et al. verfolgen einen deutlich anderen Ansatz zur Kennzeichnung von Musteranwendungsstellen als in den bisher genannten Arbeiten [BMS03]. Sie spezifizieren nicht die Entwurfsstruktur zu einem Entwurfsmuster, sondern bilden eine Graph-basierte Repräsentation der das Muster beschreibenden Sätze, wo insbesondere Entwurfsalternativen und Argumente für Entwurfsentscheidungen enthalten sind. Substantive in diesen Sätzen, welche den Rollen des Musters entsprechen, werden mit den die Rollen einnehmen Implementierungsstellen verknüpft. (Der Code wird dazu in einen abstrakten Syntaxgraphen überführt.) Die Anwendungsstellen werden durch solche Verknüpfungen repräsentiert und eine Navigation zwischen dem Code und den das Muster beschreibenden Sätzen (ihrer Graph-Repräsentation) wird ermöglicht. Das Erstellen der Verknüpfungen erfolgt manuell und ist relativ aufwändig. Die Verknüpfungen werden insb. zur Dokumentation von Entwurfsentscheidungen verwendet, was nicht im Fokus meiner Arbeit liegt. Automatische Konsistenzprüfungen scheinen bei diesem Ansatz nicht möglich zu sein.

Baniassad et al.

textuelle Musterbeschreibung mit Implementierung verknüpft

aufwändig & keine Validierung

9.2.2 Visualisierung von Anwendungsstellen

Musteranwendungsstellen lassen sich unabhängig von ihrer in Abschnitt 9.2.1 angesprochenen abstrakten Syntax auf unterschiedliche Weisen in konkreter Syntax darstellen. Es gibt diverse Notationen, sowohl zur menschenlesbaren Dokumentation (teilweise vollkommen ohne dahinterliegende Modelle oder Werkzeuge) als auch für den Einsatz in Werkzeugen. Am häufigsten verbreitet sind Darstellungen von Musteranwendungsstellen in UML-Klassendiagrammen, bei welchen die an einer Musterimplementierung beteiligten Klassen markiert und in den meisten Fällen auch ihre eingenommenen Musterrollen angegeben werden.

Notation meist Klassen-basiert

In der UML werden Kollaborationen dazu verwendet, die Teile eines Klassendiagramms zu markieren, welche zu einer Musteranwendungsstelle gehören [OMG11b]. In der Abb. 9.8 sind Kollaborationen als gestrichelte Ellipsen mit zugehörigen Kanten dargestellt und blau hervorgehoben. Die Ellipsen sind mit dem Namen eines angewandten Musters beschriftet, die Kanten mit dem Rollennamen einer beteiligten Klasse. Die gleiche Notation kann prinzipiell auch in anderen Diagrammen als den Klassendiagrammen verwendet werden, z.B. in Komponenten- oder Kommunikationsdiagrammen, ist jedoch weniger verbreitet.

OMG (UML)

Kollaborationen

Ein Nachteil dieser Notation sind die relativ vielen Kanten zu den beteiligten Klassen, was bei besonders vielen oder besonders weit verteilten Klassen im Diagramm die Übersicht einschränkt. Außerdem werden nur an einer Musterimplementierung beteiligte *Klassen* markiert. Methoden, Parameter, Attribute und

unübersichtlich, einige Rollen fehlen

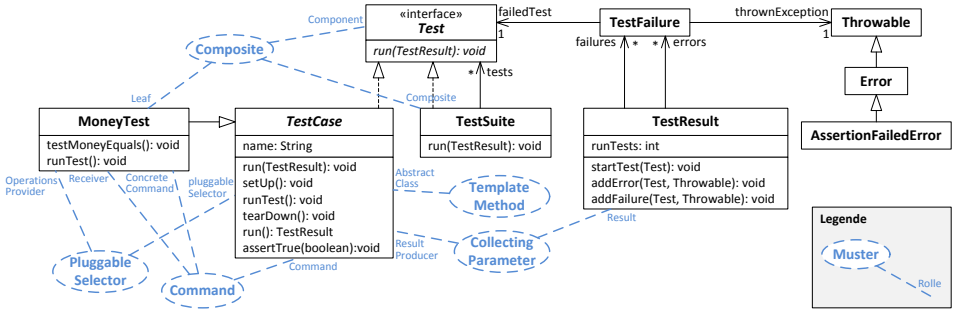


Abbildung 9.8: Visualisierung von Anwendungsstellen in der UML [OMG11b]

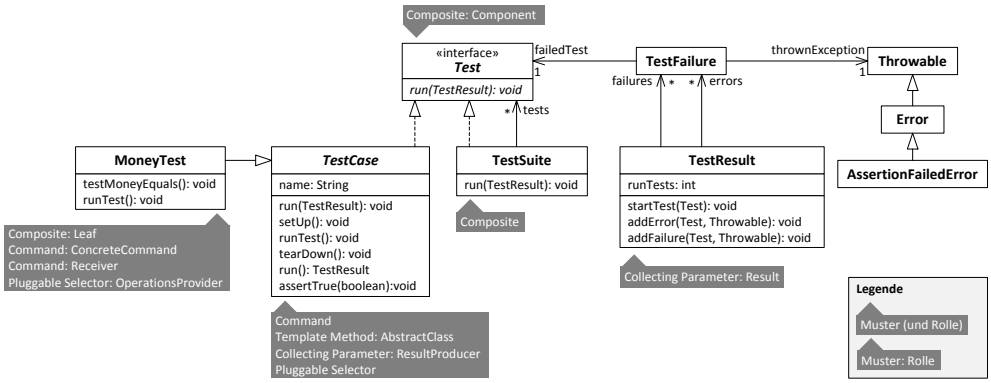


Abbildung 9.9: Visualisierung von Anwendungsstellen nach Gamma [Gam96]

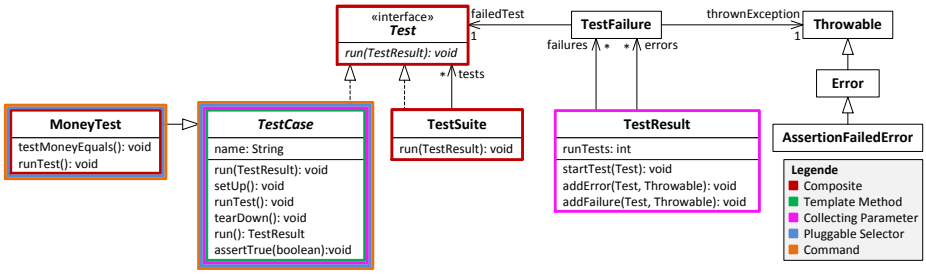


Abbildung 9.10: Visualisierung von Anwendungsstellen nach Schauer & Keller [SK98]

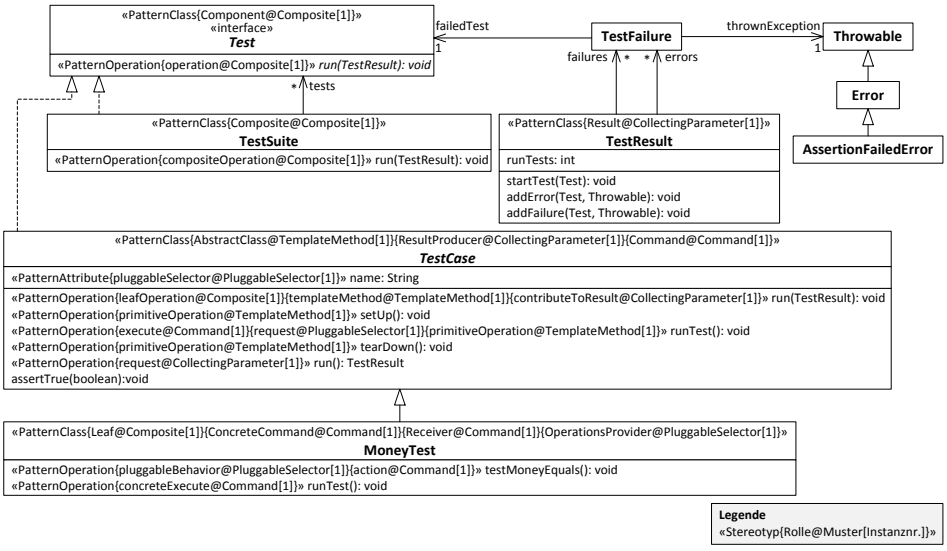


Abbildung 9.11: Visualisierung von Anwendungsstellen nach Dong et al. [DYZ07]

Assoziationen werden normalerweise nicht markiert, wenn sie eine Musterrolle einnehmen. Auch kommerzielle Software-Modellierungswerkzeuge wie Rational Software Architect[11] und Together[12] nutzen UML-Kollaborationen zur Darstellung von Musteranwendungsstellen. In meiner Arbeit greife ich diese Notation auf, erweitere sie jedoch um fehlende Informationen über eingenommene Rollen, zugehörige Set Fragments und ergänze alternative Ansichten mit unterschiedlichem Detailgrad oder einem bestimmten Modellausschnitt (Abschn. 4.3, 4.4).

Rational Software Architect & Together

Noch vor der Etablierung der UML und der UML-Kollaborationen hat Erich Gamma zur Kennzeichnung von Entwurfsmusterimplementierungen in OMT[13]-Klassendiagrammen Markierungen eingeführt [Gam96, Gam01], welche von anderen aufgegriffen wurden [Vli98a]. Diese Markierungen werden in der Literatur als *pattern:role*-Annotationen bezeichnet und sind exemplarisch in der Abb. 9.9 dargestellt. Abgesehen von der Notation (insb. gibt es keine Kanten) unterscheidet sich dieser Ansatz kaum von UML-Kollaborationen. Der Informationsgehalt ist der gleiche und ist damit zu gering.

Gamma

einige Rollen fehlen

Schauer und Keller verwenden eine deutlich einfachere Notation zur Hervorhebung von Musteranwendungsstellen in Klassendiagrammen [SK98]. Sie markieren alle an einer Musterimplementierung beteiligten Klassen durch einen Rahmen und verwenden je Musterimplementierung eine eigene Rahmenfarbe (siehe Abb. 9.10). Bei dieser Notation fehlen die eingenommenen Musterrollen. Zur Unterscheidung der eingesetzten Muster ist eine Legende notwendig. Außerdem schränkt die Anzahl klar unterscheidbarer Farben die Anzahl der darstellbaren Musterimplementierungen deutlich ein. Eine sehr ähnliche Notation setzt auch Wenzel in seiner Arbeit ein. Er macht den Rahmen jedoch nicht um einzelne Klassen, sondern um mehrere zu einer Musterimplementierung gehörende Klassen [Wen05b, Abb. 5].

Schauer & Keller

unübersichtlich, einige Rollen fehlen

Wenzel

Dong et al. schlagen eine Notation vor, bei welcher im Gegensatz zu den bisher genannten Notationen auch von Methoden und Attributen eingenommene Musterrollen in einem Klassendiagramm angegeben werden [DYZ07]. Dazu erweitern sie die UML durch Stereotypen und Tagged Values und geben die Muster und Rollen wie in der Abb. 9.11 an. Im Vergleich mit den Abb. 9.8 bis 9.10 ist diese Notation deutlich umfangreicher, textlastiger und unübersichtlicher.

Dong et al.

detailliert, unübersichtlich

Porras und Guéhéneuc stellen die vier Notationen aus den Abb. 9.8 bis 9.11 in einer empirischen Studie einander gegenüber und vergleichen ihre Lesbarkeit [PG10]. Dabei zeigte sich kein klarer Favorit. Die Ansätze haben jeweils ihre Schwächen und Stärken, abhängig von der verfolgten Aufgabe bei der Betrachtung der Musteranwendungsstellen. Während sich eine Darstellung besser zur Visualisierung von Musterkompositionen und der von Klassen eingenommenen Rollen eignet, eignen sich zwei andere besser zur Veranschaulichung der Stellen (Klassen), an denen bestimmte Muster angewandt wurden.

Vergleich in empirischer Studie

Mit meinem Ansatz (Abschn. 4.4) versuche ich, die Stärken dieser und anderer Ansätze zu kombinieren. Um den unterschiedlichen Einsatzzwecken gerecht zu werden, biete ich nicht eine, sondern mehrere auf die Einsatzzwecke zugeschnittene Darstellungen an (vgl. Abb. auf S. 88, 95, 97). Porras und Guéhéneuc beschränken sich bei ihrer Studie auf Ansätze zur Darstellung von Musteranwendungsstellen in

Stärken kombinieren

[11] Rational Software Architect: http://www-03.ibm.com/software/products/de/ratsadesigner
[12] Together: http://www.borland.com/de-DE/Products/Requirements-Management/Together
[13] OMT: Object-Modeling Technique, einer der Vorgänger der UML

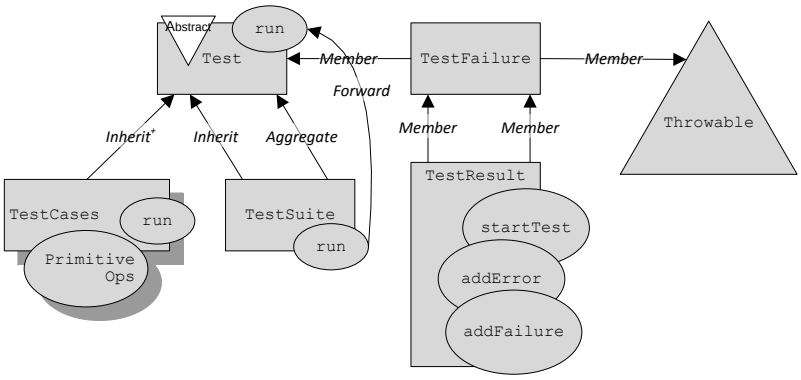


Abbildung 9.12: Visualisierung von Anwendungsstellen nach Eden et al. [EN11, Ede00] (Legende: http://lepus.org.uk/ref/legend/legend.xml)

Verhalten berücksichtigen

Klassendiagrammen. Andere Diagramm- oder Modellarten, insb. für Verhalten, bleiben unberücksichtigt. Bei meinem Ansatz sehe ich auch die Visualisierung von Anwendungsstellen in Verhaltensdiagrammen (Story-Diagrammen) vor (siehe Abb. auf S. 92 und Abb. 4.17 auf S. 94).

Meijler et al.

Meijler, Demeyer und Engel haben schon 1997 eine den UML-Kollaborationen sehr ähnliche Erweiterung der OMT vorgestellt [MDE97]. Zusätzlich zu eingesetzten Mustern und Musterrollen können hier auch einige Muster-spezifische Relationen angegeben werden (z.B. eine *creates*-Beziehung). Die Vor- und Nachteile dieser Notation ähneln denen von UML-Kollaborationen.

Eden et al.

Eden et al. verwenden ihre LePUS3-Notation nicht nur zur Spezifikation von Entwurfsmustern (siehe Abb. 9.3, S. 212 und 9.5, S. 213), sondern auch zur Repräsentation von Musterimplementierungen [EN11, Ede00]. Im Gegensatz zu den bisher genannten Ansätzen werden hier nicht die Stellen einer Musteranwendung im Entwurf markiert, sondern eine separate, isolierte, abstrakte Sicht auf eine Musterimplementierung geboten. Dazu werden die an einer Musterimplementierung beteiligten Entwurfsteile als LePUS3-Diagramm, als sogenanntes Codechart, wie in der Abb. 9.12 dargestellt. Dadurch, dass sowohl für die Spezifikation eines Musters als auch für die Musterimplementierung die gleiche Notation verwendet werden, können beide gut verglichen und Abweichungen von der Spezifikation erkannt werden. Die isolierte Darstellung einer Musterimplementierung ohne den Ballast von am Muster unbeteiligten Klassen ermöglicht einen besseren Fokus auf eine bestimmte Musterimplementierung. Die Codecharts lassen allerdings den Namen des verwendeten Musters und die Musterrollen vermissen. Die Notation von Methoden als Ellipsen benötigt relativ viel Platz in einem Diagramm.

separate, abstrakte Sicht

Rollen fehlen, hoher Platzbedarf

Diese Notation ist mit der von mir eingeführten Musteranwendungssicht vergleichbar (siehe Abb. 4.6, S. 88), wo eine Musterimplementierung ebenfalls unter Verwendung der gleichen Notation wie für Musterspezifikationen dargestellt wird. Im Gegensatz zu Eden et al. stelle ich jedoch u.a. den Musternamen und die Musterrollen dar. Ergänzend dazu kann eine Musterimplementierung bei meinem Ansatz auch in Klassendiagramm-Notation isoliert vom übrigen Entwurf betrachtet werden (siehe Abb. auf S. 97).

Smith

Smith schlägt eine flexibel anpassbare Notation mit unterschiedlichen Detailgraden vor [Smi11] [Smi12, Kap. 3] und nennt sie Pattern Instance Notation (PIN). In

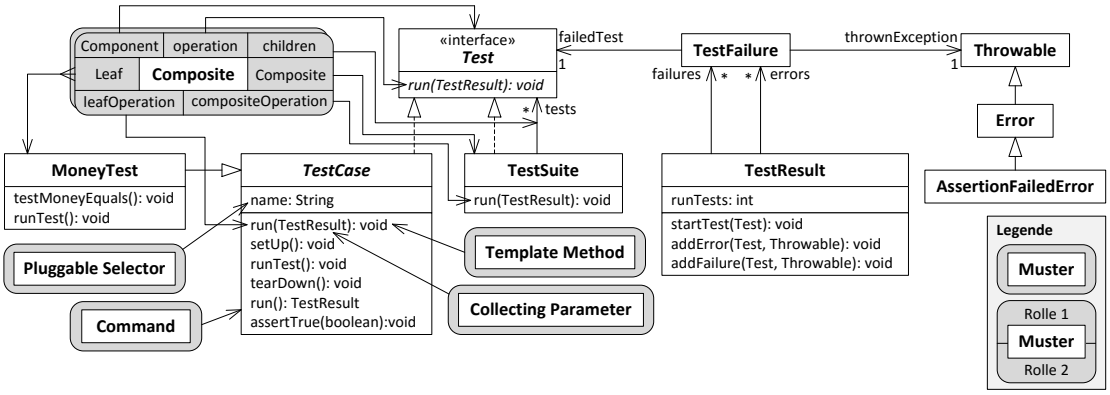


Abbildung 9.13: Visualisierung von Anwendungsstellen nach Smith [Smi11, Smi12]

Markierungen ähnlich zu Kollaborationen

ihrer einfachsten Form ähnelt die Notation den UML-Kollaborationen. Mit einer sogenannten PINbox, einem grau umrahmten Rechteck mit Beschriftung, wird das Entwurfselement, welches die Hauptrolle einer Musterimplementierung einnimmt, mit dem Musternamen beschriftet. Zum Beispiel wird die Klasse TestCase in der Abb. 9.13 durch eine PINbox als Implementierung der Rolle *Command* des gleichnamigen Musters gekennzeichnet.

Detailgrad anpassbar

Eine PINbox lässt sich bei Bedarf wie bei der Composite-PINbox in der Abb. 9.13 aufklappen. Dadurch werden alle Rollen des Musters innerhalb des grauen PINbox-Rahmens sichtbar und sie werden mit den sie einnehmenden Entwurfsteilen durch eine Kante verbunden. Kann eine Rolle mehrfach eingenommen werden, so wird die PINbox als Stapel dargestellt (bekommt einen weiteren Rahmen), die Kante zur Rollenimplementierung erhält eine spezielle Markierung auf der Seite der Rolle (siehe Leaf-Rolle) und kann auf mehrere Entwurfselemente verweisen.

Musterkomposition

Die PINbox-Notation lässt sich außerdem dazu nutzen, Muster zu spezifizieren, indem ein Muster als Komposition anderer, insb. der von Smith definierten Elementarmuster (Elemental Design Patterns, EDPs)[14]) spezifiziert wird, z.B. wie das Decorator-Muster [GHJV95] in Abb. 9.14. Zur Beschreibung eines Musters kann innerhalb der PINbox auch ein UML-Klassendiagramm verwendet werden, z.B. wie in der Abb. 9.15 dargestellt. Die Notation aus den Abb. 9.14 und 9.15 lässt sich nicht nur zur Spezifikation eines Musters, sondern ebenso zur Darstellung von Anwendungsstellen eines Musters eingesetzt und mit der Notation aus Abb. 9.13 kombiniert werden. Die Notation aus Abb. 9.13 ist nicht auf Klassendiagramme beschränkt. Smith setzt sie z.B. u.a. in UML-Sequenzdiagrammen ein oder kombiniert ein Sequenzdiagramm mit einer PINbox mit darin enthaltenem Klassendiagramm wie in der Abb. 9.15.

Musterspezifikationen & Anwendungsstellen

Details und Übersicht ausbaufähig

Der flexibel wählbare Detailgrad, die Komponierbarkeit von Mustern und die Kombinierbarkeit mit diversen UML-Diagrammen sind die Stärken des Ansatzes. Im Vergleich zu anderen Ansätzen ist der Detailgrad in Abb. 9.13 jedoch zu gering (je Anwendungsstelle wir nur eine Rolle markiert), während er bei einer Darstellung mit allen beteiligten Rollen wie bei der Composite-Anwendungsstelle

[14] Ursprünglich zu Reverse-Engineering-Zwecken und das Analysewerkzeug SPQR (System for Pattern Query and Recognition) entwickelt. https://www.cs.unc.edu/∼smithja/SPQR.html [SS03]

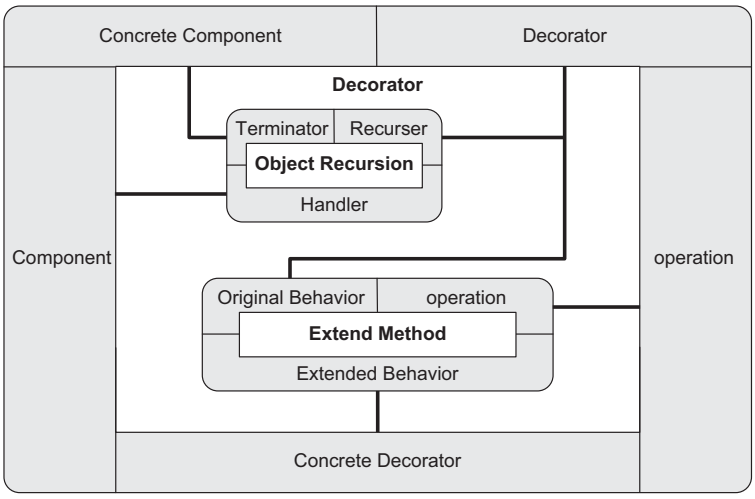


Abbildung 9.14: Visualisierung der Komposition des Decorator-Musters aus Elementarmustern nach Smith [Smi11]

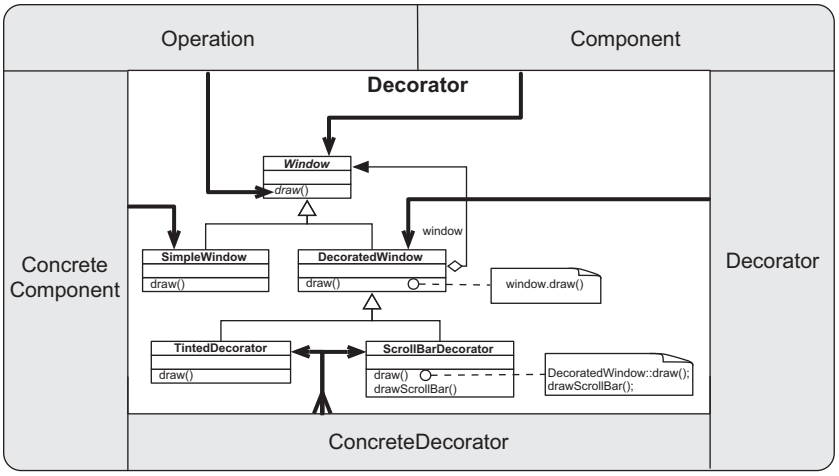


Abbildung 9.15: Visualisierung einer Decorator-Musterimplementierung mit Klassendiagrammen nach Smith [Smi11]

in Abb. 9.13 zu viele Kanten involvieren und damit vergleichsweise unübersichtlich werden würde. Viele Kanten entstehen auch bei der Darstellung aus Abb. 9.15, wenn eine Rolle von vielen Klassen eingenommen wird (z.B. beim Muster Abstract Factory sowie vielen Produkten und Fabriken in der Anwendungsstelle). Auch ich verwende mehrere Detailgrade für die Darstellung von Anwendungsstellen. Im Gegensatz zu Smith verwende ich bei meiner Musteranwendungssicht (siehe Abb. 4.6, S. 88) und meiner Musterrollen-im-Entwurf-Ausschnittssicht (siehe Abb. 4.22 bis 4.24, S. 97) weniger Kanten. Musterkompositionen wie in Abb. 9.14 betrachte ich bei meinem Ansatz jedoch nicht.

Platenius, von Detten, Travkin

In einer meiner früheren Arbeiten an einem Reverse-Engineering-Werkzeug namens Reclipse [vDMT10a, vDMT10b] stellen wir drei Darstellungen von Musteranwendungsstellen vor (genauer: von Funden von Musterimplementierungen) [PvDT11]. Neben der Verwendung von UML-Kollaborationen analog zu Abb. 9.8 (S. 222) [PvDT11, Abb. 3] nutzen wir zwei weitere Sichten auf eine Anwendungs-

stelle. Da die Spezifikation eines gesuchten Musters in abstrakter Syntax erfolgt, stellen wir neben den Musterspezifikationen auch die gefundenen Anwendungsstellen als annotiertes Objektdiagramm dar. In der einen Darstellung wird die Anwendungsstelle analog zur Musterspezifikation als Objektdiagramm visualisiert und gibt die konkreten Eigenschaften (insb. Namen) der beteiligten Entwurfselemente an [PvDT11, Abb. 5]. In der anderen Darstellung wird die Musterspezifikation um die Information angereichert, ob eine spezifizierte Bedingung erfüllt oder ein gesuchtes Element in der Anwendungsstelle vorhanden ist [PvDT11, Abb. 4].

Matching visualisiert

Die Musteranwendungssicht in dieser Arbeit (siehe Abb. 4.6, S. 88) ist eine Art Kombination dieser beiden Sichten auf eine Anwendungsstelle. Sie greift die Notation für Musterspezifikationen auf, gibt Auskunft darüber, welche Musterrollen Entwurfselementen zugeordnet wurden (farbliche Markierung) und welche Entwurfselemente an der Musterimplementierung beteiligt sind (ihre Namen).

Ähnlichkeit

Kajsa et al. stellen Anwendungsstellen sowohl in UML-Klassendiagrammen als auch in Java-Code dar [KM10, KN12]. In Klassendiagrammen nutzen sie mit Stereotypen versehene Relationen zwischen Klassen zur Beschreibung einer Musteranwendungsstelle (z.B. eine ≪Observes≫-Relation zur Beschreibung einer Klasse, welche eine andere im Rahmen des Observer-Musters beobachtet [KN12, Abb. 8]). Zusätzlich markieren sie jede an einer Anwendungsstelle beteiligte Klasse mit einem Stereotypen, welcher die Musterrolle angibt. Im Java-Code setzen sie selbst definierte Java-Annotationen ein, mit denen sie Klassen, Attribute und Methoden markieren. Zum Beispiel markieren sie eine an einer Observer-Musterimplementierung beteiligte Klasse in der Rolle *Subject* wie folgt mit einer Annotation.

Kajsa et al.

Stereotypen & Java-Annotationen

```
@DesignPattern(patterName = PatterNames.Observer, instanceAlias = "obs1",
        roleName = RoleNames.Subject)
```

[sic] Quelle: [KN12, Abb. 2]

Die Notation in Klassendiagrammen lässt unklar, welche mit Stereotypen markierten Klassen zur selben Musterimplementierung gehören. Die Kennzeichnung von Anwendungsstellen im Code ist in meinem modellgetriebenen Ansatz nicht einsetzbar.

Details fehlen, nicht anwendbar

9.3 Werkzeug-gestützte Musteranwendung

Seit der Einführung von Entwurfsmustern in der Softwareentwicklung gibt es Bestrebungen, die Anwendung von Entwurfsmustern mit Hilfe von Werkzeugen zu vereinfachen. So sind zahlreiche Werkzeuge und Ansätze entstanden, die ich im Folgenden meiner Arbeit gegenüberstelle. Die vielen, teils sehr unterschiedlichen Arbeiten unterteile ich in diejenigen, die Entwurfsmuster mit Hilfe von Werkzeugen im Quellcode anwenden (Abschn. 9.3.1) und diejenigen, die Muster in Entwurfsmodellen (meist UML) anwenden (Abschn. 9.3.2).

Arbeiten gruppiert nach Einsatzzweck

9.3.1 Musteranwendung im Quellcode

Eine der frühesten Arbeiten zum Thema der Werkzeug-gestützten Musteranwen-

Budinsky et al.

dung entstand bereits kurz nach Veröffentlichung des Gang-of-Four-Musterkatalogs [GHJV95] und stammt von Budinsky et al.[15] [BFYV96]. In dieser Arbeit wurde ein Web-basiertes Werkzeug vorgestellt, welches basierend auf einigen Benutzereingaben, insb. zur Festlegung von Klassennamen, Anwendungsfall-spezifische Implementierungen eines ausgewählten Entwurfsmusters in C++ generiert hat. Neben der fehlenden IDE-Integration hat dieser Ansatz einen entscheidenden Nachteil: Der generierte Code muss manuell in den Code der sich in Entwicklung befindenden Anwendung kopiert und an die konkrete Situation angepasst werden, was aufwändig und fehleranfällig ist. Außerdem können die so entstandenen Anwendungsstellen aufgrund fehlender Dokumentation oder Kennzeichnung nur schwer wiedererkannt und angepasst werden.

generierte Code-Schnipsel; copy, paste, adapt

Florijn et al.

Florijn et al. treiben die Werkzeugunterstützung bei ihrem Ansatz deutlich weiter [FMvW97]. Ihr Werkzeug generiert Musterimplementierungen in bestehenden Smalltalk-Code einer Anwendung, setzt eine Art Modell des Quellcodes und der Entwurfsmuster ein, einen sogenannten Fragment-Graphen (siehe Musterspezifikation nach Florijn et al. auf S. 210) und persistiert darin die Anwendungsstellen (siehe S. 220), um sie visualisieren und automatisiert prüfen zu können. Die Anwendung eines Musters erfolgt durch Replizieren des Fragment-Graphen, welcher ein Muster repräsentiert. Die Repräsentation eines Musters unterscheidet sich dabei nicht von der Repräsentation einer Musterimplementierung / Anwendungsstelle. Die als Muster spezifizierte Struktur wird also 1-zu-1 kopiert. Fragments können u.a. Smalltalk-Code enthalten, sodass bei der Musteranwendung sogar Verhalten (Methodenrümpfe) mitgeneriert werden kann.

Musterimplementierung kopieren & Anwendungsstellen erfassen

Trotz der vielen, richtungsweisenden Ideen hat diese Arbeit ihre Grenzen. Mit jedem neuen Muster müssen Hilfsoperationen programmiert werden, welche das Anwenden des Musters und Prüfen der Anwendungsstellen überhaupt möglich machen. Das Implementieren dieser Operationen ist aufwändig, fehleranfällig und erfordert Expertenwissen. Die Musteranwendung ist aufgrund des Klonens einer Musterspezifikation sehr unflexibel, z.B. kann die Anzahl der konkreten Observer-Klassen des Observer-Musters nicht situationsabhängig gewählt werden.

aufwändige Spezifikation, keine Varianten

Eden et al.

Eden et al. setzen Programme höherer Ordnung bzw. Meta-Programme (Programme, welche andere Programme modifizieren) zur Anwendung von Entwurfsmustern in bestehendem Code ein [EYG97]. Die Anwendung eines Musters wird in mehrere Schritte unterteilt, welche durch je ein wiederverwendbares Hilfsprogramm implementiert werden. Es gibt z.B. Operationen zum Ergänzen einer Referenz zwischen zwei existierenden Klassen oder zur Implementierung eines Methodenaufrufs mit Argumentweiterreichung (einer Delegation). Im Gegensatz zu meinem Ansatz werden Anwendungsstellen bei Eden et al. nicht als solche gekennzeichnet und können daher nach der Musteranwendung weder visualisiert noch analysiert werden. Wie bei Florijn et al. müssen auch hier mit jedem neuen Muster Hilfsoperationen aufwändig programmiert und getestet werden.

modulare Code-Modifikationen, Anwendungsstellen nicht persistent

Cinnéide & Nixon

Refactoring

Wie bei Eden et al. werden auch bei Cinnéide und Nixon Entwurfsmuster mit Hilfe von diversen, kombinierten Programm-modifizierenden Operationen (und einiger Hilfsoperationen wie benötigter Prädikate) angewandt [CN99, Cin00]. Das Besondere dabei ist, dass es sich bei diesen Operationen um Refactorings han-

[15] u.a. von einem der Gang-of-Four-Autoren, John M. Vlissides

delt, also ausschließlich um verhaltenserhaltende Code-Modifikationen. Die Verhaltenserhaltung wird mit Hilfe von Vor- und Nachbedingungen und zugehöriger Beweise sichergestellt. Der Nachteil, diese Operationen mit jedem neuen Muster programmieren zu müssen, bleibt auch hier. Außerdem wird hier vorausgesetzt, dass Musteranwendungen nur durch Refactoring erfolgen, der Quellcode der Anwendung also weitestgehend vollständig und konsistent (kompilierbar) ist, bevor ein Muster angewandt wird. Diese Annahme trifft meiner Meinung nach nur in seltenen Fällen zu, insb. nicht bei der initialen Entwicklung oder der Erweiterung einer Anwendung.

Entwurf erweitern unmöglich

Auch Guéhéneuc et al. stellen einen Ansatz vor, bei dem Entwurfsmuster mit Hilfe selbst programmierter Operationen direkt im Quellcode angewandt werden [AACGJ01, AAG01]. Zur Repräsentation der Anwendungsstellen implementieren sie ein Meta-Modell für Entwurfsmuster und Anwendungsstellen. Dieses enthält Klassen für die Muster selbst sowie für einige objektorientierte Softwarekonstrukte, hauptsächlich Klassen, Methoden, Attribute, Assoziationen und Delegationen. Bei Anwendung eines Musters wird dieses Meta-Modell instanziiert, das allgemeine Modell einer Musterimplementierung an eine konkrete Musteranwendung angepasst, indem Klassennamen angegeben und Elemente ergänzt werden (z.B. Leaf-Klassen beim Composite-Muster), und schließlich mit Hilfe je Muster selbst implementierter Operationen in Quellcode übersetzt. Das Ergänzen von Entwurfsmustern ist bei diesem Ansatz aufgrund der notwendigen Implementierung zusätzlicher Meta-Modell-Klassen und Musteranwendungsoperationen aufwändig.

Guéhéneuc et al.

selbst programmierte Übersetzung je Muster, aufwändig

Bei dem Ansatz von Kajsa und Návrat werden Entwurfsmuster direkt im Quellcode angewandt [KN12] und die Anwendungsstellen mit Hilfe von Java-Annotationen markiert. Eine Musteranwendung erfolgt durch eine Art Code-Vervollständigung. Dazu markiert ein Entwickler eine Java-Klasse mit einer `@DesignPattern`-Annotation und gibt mit Hilfe eines Code-Assistenten einige Details wie das gewünschte Muster, einen eindeutigen Namen für die Anwendungsstelle und die Rolle der Klasse in dem Muster an (siehe Bsp. auf S. 227). Anschließend wird der vorliegende Code automatisch ergänzt und damit die gewünschte Musterimplementierung vervollständigt. Die Implementierungsalternativen eines Musters werden in einem Feature-Modell bestehend aus Klassen, Methoden und Attributen erfasst und als Vorlage für die Template-basierte Code-Generierung verwendet.

Kajsa & Návrat

Templates & Feature-Modell, Annotieren von Anwendungsstellen

Die Verwendung von Java-Annotationen bringt eine signifikante Einschränkung mit sich: Je Klasse kann nur eine Annotation gleichen Namens und damit auch nur ein Muster angewandt werden. Feature-Modelle sind eine interessante Möglichkeit, die Varianten einer Musterimplementierung zu spezifizieren, scheinen jedoch im Vergleich zu meiner Notation nicht sehr übersichtlich zu sein. Mit Set Fragments lassen sich in der Anzahl variierende Entwurfsteile kompakter spezifizieren. Meine UML-ähnliche Notation verwendet eine konkrete, statt einer abstrakten Syntax.

überlappende Musteranwendungen unmöglich, Varianten unübersichtlich

Alternativ zum beschriebenen Vorgehen kann die Musteranwendung in UML-Klassendiagrammen erfolgen. Darauf gehe ich in Abschnitt 9.3.2 (S. 231) ein.

Ehms hat ein einfaches Werkzeug namens PatternBox entwickelt, welches die Anwendung von 16 der 23 GoF-Entwurfsmuster [GHJV95] ermöglicht, indem je Muster Java-Code generiert wird [Ehm13]. Es werden anpassbare Code-Templates verwendet. Vor Musteranwendung werden im Wesentlichen nur Klassennamen festgelegt, bevor Code generiert wird. Nach der Generierung muss der Code manu-

Ehms

generierte Code-Schnipsel; copy, paste, adapt

ell in den Anwendungscode integriert werden. Damit ist der Nutzen des Werkzeugs gering und vergleichbar mit dem Ansatz von Budinsky et al. [BFYV96].

ModelMaker Tools

ModelMaker[16] ist ein weiteres Werkzeug, welches die Anwendung von Entwurfsmustern im Quellcode unterstützt [BP99]. ModelMaker generiert Delphi- oder C#-Code aus einem UML-Entwurfsmodell und sieht explizit die Verwendung von Entwurfsmustern vor – hier eher einer Art Code-Template-Anwendungen. Entwickler legen bei der Anwendung eines Musters die Anwendungsstelle und zu verwendende Namen fest und bekommen Code-Schnipsel in ihren vorliegenden Code generiert. Die Anwendungsstellen können auf Wunsch in einem Modell persistiert werden, damit der generierte Code bei bestimmten Änderungen (z.B. Namensänderungen) automatisch angepasst werden kann. Im Code sind die Musteranwendungen nicht als solche erkennbar. Das Werkzeug unterstützt nur acht Muster. Weitere Muster können entweder nur eingeschränkt mit Hilfe einfacher, selbst geschriebener Code-Templates oder aufwändig durch Implementieren eigener Werkzeugerweiterungen (Wizards) ergänzt werden. Im Vergleich zu meinem Ansatz ist die Musteranwendung unflexibel, weil die Existenz bestimmter Teile einer Musterimplementierung vor der Anwendung vorausgesetzt wird.

Templates

intransparent, aufwändig, festgelegte Anwendungssituationen

9.3.2 Musteranwendung in Entwurfsmodellen

Maplesden et al.

Maplesden et al. stellen einen Modell-basierten Ansatz vor, wo Entwurfsmuster in UML-Klassen-Modellen angewandt werden [MHG02, MHG07]. Bei der Anwendung eines Musters wird aus einer Musterspezifikation (Abb. 9.6, S. 213) ähnlich wie bei meinem Ansatz ein Modell der Musteranwendungsstelle erstellt (siehe S. 219). Anschließend wird zu jeder Musterrolle entweder eine existierende Klasse, Methode oder ein Attribut des UML-Modells zugeordnet oder im UML-Modell ergänzt und der Rolle zugeordnet. Das Modell einer Anwendungsstelle verknüpft somit eine Musterspezifikation mit den das Muster implementierenden Teilen des UML-Modells und kann analog zu meinem Ansatz als Dokumentation oder zur Prüfung der Konsistenz einer Musterimplementierung genutzt werden.

flexible Anwendung & Rollenzuordnung

Im Gegensatz zu meiner Arbeit wird hier zu einem Muster gehörendes Verhalten nicht berücksichtigt, weil sich die Musteranwendung bisher auf UML-Klassendiagramme beschränkt. Außerdem werden bei meinem Ansatz zusammengehörende Teile eines Musters, welche bei der Musteranwendung mehrfach vorkommen dürfen (z.B. ConcreteFactory-, ConcreteProduct-Klassen und zugehörige create-Methoden des Musters Abstract Factory), mit Hilfe von Set Fragments ausschließlich gemeinsam erzeugt und visualisiert, während sie bei Maplesden et al. auch einzeln erzeugt werden können, was zu Inkonsistenzen zum Muster führen kann.

Verhalten unberücksichtigt, ggf. Inkonsistenzen

Kim & Whittle

Kim und Whittle haben ein Verfahren zur Anwendung von Entwurfsmustern in UML-Klassen- und Sequenzdiagrammen entwickelt [KW05]. Bei einer Musteranwendung wird die Musterspezifikation bestehend aus der Struktur eines Klassen- (Abb. 9.7, S. 215) und eines Sequenzdiagramms in ein UML-Modell übertragen. Dabei wird zu jeder Musterrolle ein Element im UML-Modell erzeugt, sodass zu jeder in der Musterspezifikation angegebenen Kardinalität die minimale Anzahl geforderter Elemente erzeugt wird. Die minimale Anzahl kann auf Wunsch vor

Generieren & Markieren ganzer Musterimplementierung

[16] ModelMaker: http://www.modelmakertools.com/modelmaker/index.html

der Musteranwendung erhöht werden. Die Rollennamen werden durch generierte Namen ausgetauscht. Die generierten Elemente, z.B. Klassen, erhalten je einen Stereotypen mit dem Namen der eingenommenen Musterrolle. Anschließend wird das UML-Modell manuell an die konkrete Situation angepasst.

kein Überlappen mit vorhandenem Entwurf, Modelle nicht ausführbar

Im Gegensatz zu meinem Ansatz können bei diesem Verfahren bereits vorhandene Teile einer Musterimplementierung nicht den Rollen eines anzuwendenden Musters zugeordnet und wiederverwendet werden. Es wird immer eine komplette Musterimplementierung ohne jeden Bezug zu bereits existierenden Klassen generiert, was die Flexibilität bei der Musteranwendung erheblich einschränkt. Das zu einem Muster gehörende Verhalten wird zwar in Sequenzdiagrammen modelliert, lässt sich jedoch nach meinem Kenntnisstand nicht mit Hilfe eines Interpreters oder eines Code-Generators ausführen (d.h. modellgetriebene Entwicklung ist somit nicht möglich), ganz im Gegensatz zu den in meiner Arbeit verwendeten Story-Diagrammen.

Kajsa et al.

Bei dem Ansatz von Kajsa et al. werden Entwurfsmuster in UML-Klassendiagrammen angewandt und anschließend in Quellcode übersetzt [KM10, KN12]. In einem ersten Schritt wird die Anwendung eines Musters vorgesehen, indem eine spezielle, mit einem Stereotypen markierte Assoziation modelliert wird. Zum Beispiel gibt der Stereotyp «Observes» an einer solchen Assoziation an, dass eine Klasse von einer anderen mit Hilfe des Observer-Musters beobachtet werden soll. Im nächsten Schritt wird das Muster im Klassendiagramm angewandt. Dabei wird der vorliegende Entwurf angepasst, indem mit Hilfe selbst programmierter Transformationen und eines Feature-Modells als Repräsentation aller möglichen Musterimplementierungsvarianten die fehlenden Klassen, Methoden, Attribute und Assoziationen ergänzt werden. Mit Hilfe von Stereotypen und Tagged Values wird die Anwendungsstelle markiert. In einem letzten Schritt wird das UML-Modell in Quellcode übersetzt (siehe S. 9.3.1).

Relationen zwischen Klassen übersetzt zu Musterimplementierungen

nur wenige Muster beschreibbar, kein Verhalten

Das Modellieren von Musteranwendungen als Relationen zwischen nur zwei Klassen (z.B. «Observes») schränkt diesen Ansatz stark ein. Entwurfsmuster wie z.B. Abstract Factory erfordern mehr Relationen und nicht nur zwischen Klassen: Je konkreter Fabrik (Klasse) muss beschrieben werden, welche Produktgruppe (Gruppe von Produktklassen) sie erzeugt und je konkretem Produkt (Klasse) muss beschrieben werden, zu welcher Produktgruppe es gehört. Zu einem Entwurfsmuster gehörendes Verhalten wird auch hier nicht berücksichtigt.

Mili & El-Boussaidi

Mili und El-Boussaidi wenden Entwurfsmuster in UML-Klassendiagrammen mit Hilfe von implizit spezifizierten, endogenen Modelltransformation an [MEB05, EBM07]. Dazu spezifizieren sie „das von einem Entwurfsmuster gelöste Problem“, wie sie es beschreiben, durch das Modellieren einer Klassenstruktur (problem model), welche durch Anwenden eines Entwurfsmusters durch eine bessere Klassenstruktur (solution model) ersetzt werden soll. Die bessere Klassenstruktur wird auf die gleiche Weise spezifiziert (siehe S. 215) und entspricht der Situation nach einer Musteranwendung. Durch Zuordnen der Elemente beider Modelle zueinander wird deklarativ eine Modelltransformation beschrieben[17]. Eine Musteranwendung erfolgt durch Identifizieren (matching) der problematischen Struktur (problem mo-

Ersetzungen bestimmter Klassenstrukturen

[17] Bei deklarativen Modelltransformationssprachen spricht man von einer Left-hand Side (LHS) und einer Right-hand Side (RHS). Hier entspricht das problem model der LHS und das solution model der RHS. Mehr zu LHS & RHS in Abschnitt 2.3.3, S. 31 ff.

del) in einem UML-Modell (input model) und anschließendes Übersetzen dieser in eine Klassenstruktur mit angewandtem Entwurfsmuster (solution model).

Wie bei Cinnéide und Nixon (siehe S. 228) wird hier vorausgesetzt, dass es einen weitestgehend vollständigen Softwareentwurf gibt, bevor ein Muster angewandt wird, sodass eine Musteranwendung nur eine Transformation bzw. ein Refactoring wie bei Kerievsky [Ker04] darstellt. Da Muster häufig schon beim Erstellen eines Softwareentwurfs angewandt werden und es in diesem Fall noch nichts zu transformieren gibt, habe ich meinen Ansatz darauf ausgelegt, den Entwurf durch Musteranwendungen zu vervollständigen, anstatt Teile davon zu ersetzen. Meiner Meinung nach eignen sich Ersetzungen und Refactoring mehr für das Beseitigen von Entwurfsproblemen wie z.B. beim Ansatz von Matthias Meyer [Mey09]. Zu Mustern gehörendes Verhalten wird bei Mili und El-Boussaidi nicht berücksichtigt.

Entwurf erweitern unmöglich

Könemann

Könemann hat ein Verfahren entwickelt, mit welchem wiederkehrende Modelländerungen, u.a. Musteranwendungen, in Transformationen erfasst und an anderer Stelle wiederholt werden können [Kön11]. Dazu kombiniert er Methoden zur Bestimmung von Unterschieden zwischen zwei Modellen mit Modelltransformationsansätzen und nennt das resultierende Konzept „Model-Independent Differences“. Wiederkehrende Modelländerungen wie das Ergänzen eines ID-Attributs in einer Klasse oder das Anwenden eines bestimmten Entwurfsmusters werden exemplarisch an einer Stelle in einem Entwurfsmodell vorgenommen. Die ursprüngliche, unveränderte Version des Modells wird mit der geänderten Version verglichen (z.B. mit EMF Compare[18]) und als Modelldifferenz in einem speziellen Modell erfasst. Anschließend wird diese Differenz durch eine spezielle Transformation verallgemeinert und zu einer Modell-unabhängigen Differenz (Model-Independent Difference) gewandelt. So eine Differenz stellt eine Art Modelltransformation dar, weil sie an beliebigen Stellen eines gleichartigen Modells angewandt werden kann, um die gleiche Modelländerung vorzunehmen. Bei Anwendung einer Modell-unabhängigen Differenz werden die geänderten Entwurfsmodellelemente mit den Elementen und Änderungsoperationen in der Modell-unabhängigen Differenz verknüpft und damit die Anwendungsstelle explizit erfasst. Außerdem wird jede angewandte Modellunabhängige Differenz mit einem Modell der getroffenen Entwurfsentscheidung verknüpft, aus welcher u.a. Entwurfsalternativen hervorgehen.

Herleiten von Transformationen aus exemplarischen Entwurfsänderungen

Im Gegensatz zu meinem Ansatz lassen sich Modelländerungen bei diesem Vorgehen in nahezu beliebigen Modellen anwenden, nicht zwingend auf Softwareentwurfsmodelle beschränkt[19] Der Preis dafür ist jedoch eine gewisse Unschärfe bei der Verallgemeinerung der Modelländerungen (bedingt durch angewandte Heuristiken) und damit verbunden nicht immer die gewünschte Modelländerung bei der Wiederanwendung einer Modell-unabhängigen Differenz. Soll z.B. eine Typanpassung bei allen ID-Attributen in einem Modell vorgenommen werden und wurde zwecks Erfassung der Modelländerung der Typ eines Attributs namens „id“ exemplarisch geändert, so wird die verallgemeinerte Modelländerung auch für Attribute *ähnlichen* Namens, z.B. „mid“ oder „cont_id“ vorgeschlagen. Solche Modelländerungen müssen also vor ihrer Anwendung immer geprüft und ggf. korrigiert werden. Dadurch wird das Verfahren fehleranfällig und aufwändig.

fehleranfällig, erfordert Prüfungen und Korrekturen

[18] EMF Compare: https://www.eclipse.org/emf/compare/
[19] Technisch bedingt muss es ein EMF-Modell sein [SBPM08].

Außerdem hat Könemanns Evaluation des Verfahrens gezeigt, dass bei 14 der 23 GoF-Entwurfsmuster bei Anwendung des Musters mit einer Modell-unabhängigen Differenz nur die exakt gleiche Entwurfsstruktur erzeugt werden konnte, welche ursprünglich zwecks Erfassung der Musteranwendung als Modelländerung erstellt wurde [Kön11, Kap. 8.1.1]. Das entspricht im Wesentlichen dem Kopieren eines exemplarischen Entwurfs zu einem Muster (analog zu Kim & Whittle, S. 230) und macht die Musteranwendung sehr unflexibel. Eines der GoF-Muster konnte gar nicht als Modelländerung erfasst werden. Es ist zu beachten, dass die von Könemann erfassten Modelltransformationen nicht Entwurfsmuster repräsentieren, sondern bestenfalls wiederholbare Modelländerungen, welche zu einer ganz bestimmten Implementierungsvariante eines Musters führen. Die von Könemann zusätzlich betrachteten Entwurfsentscheidungen (Begriffsabgrenzung auf S. 23) sind nicht im Fokus meiner Arbeit.

zu genaue Kopie einer Musteranwendung

keine Varianten

Le Guennec et al. verfolgen in ihrer Arbeit ebenfalls das Ziel, Muster mit Hilfe von Modelltransformationen in UML-Modellen anzuwenden [SLGJ00] (bzgl. Musterspezifikation siehe S. 212). Die Transformationen sollen hierbei explizit in einer OCL-ähnlichen Notation spezifiziert werden. Dazu sollen elementare Transformationen durch algebraische Komposition zu komplexeren Transformationen kombiniert werden. Die Ideen zu diesem Ansatz sind nach meinem Kenntnisstand nie prototypisch implementiert und evaluiert worden. Der verwendete Werkzeug-Prototyp UMLAUT bietet zwar die Möglichkeit, UML-Modell-Transformationen zu spezifizieren und auszuführen, die vorgeschlagene OCL-ähnliche Transformationssprache wurde jedoch nie implementiert, die Transformationen zur Anwendung von Entwurfsmustern nie implementiert.

Le Guennec et al.

Transformationen

nie implementiert

In kommerziellen Werkzeugen wie Enterprise Architect[20] und Together[21] werden Entwurfsmuster in Form von vorgefertigten UML-Diagrammen spezifiziert und in ein existierendes UML-Modell übertragen (siehe S. 216). Es wird ein Minimum an Flexibilität geboten, indem Teile des Musters nur dann erzeugt werden, wenn sie nicht bereits existierenden Teilen des Entwurfsmodells zugeordnet wurden. Außerdem können Namen abweichend von der Musterspezifikation gewählt werden. Während bei Together ausschließlich Klassendiagramme verwendet werden [Bor11], können bei Enterprise Architect beliebige UML-Diagramme zur Modellierung von Mustern genutzt werden [Spab]. Wie bei Together werden auch bei dem freien, Werkzeug-Prototypen Design Pattern Automation Toolkit (DPATool-kit)[22] Entwurfsmuster in Form von Klassendiagrammen spezifiziert und durch Vervollständigen in ein UML-Klassendiagramm im Entwurfsmodell übertragen [NSN].

Enterprise Architect, Together, DPAToolkit

Kopieren von Entwurfsbeispielen

Alle drei Werkzeuge haben gemeinsam, dass Entwurfsmuster als UML-Diagramm-Schablonen betrachtet und 1-zu-1 in ein Entwurfsmodell übertragen werden. Mit Ausnahme von Together (siehe S. 220) sind die Anwendungsstellen bei diesen Werkzeugen nach der Musteranwendung nicht mehr als solche erkennbar. Sie werden nicht in einem Modell erfasst oder anderweitig gekennzeichnet. Zu einem Muster gehörendes Verhalten wird nur von Enterprise Architect bei der Musteranwendung mitberücksichtigt, da z.B. Sequenzdiagramme oder State Machines in Musterspezifikationen verwendet werden können. Keines dieser Werkzeuge bie-

keine Varianten, Anwendungsstellen nicht persistent, kein Verhalten

[20] Enterprise Architect: http://www.sparxsystems.de
[21] Together: http://www.borland.com/de-DE/Products/Requirements-Management/Together
[22] Design Pattern Automation Toolkit: http://dpatoolkit.sourceforge.net/

tet ein Konzept, welches die variablen Teile eines Musters bei der Musteranwendung berücksichtigt, wie z.B. eine wählbare Anzahl von konkreten Produkten und Produktgruppen bei dem Muster Abstract Factory. Eine nachträgliche Anpassung einer Anwendungsstelle ist bei den Werkzeugen nicht vorgesehen.

Rational Software Architect & MagicDraw

Rational Software Architect[23] und MagicDraw[24] – ebenfalls kommerzielle Werkzeuge – stellen spezielle, selbst implementierte Modelltransformationen bereit, welche je ein bestimmtes Entwurfsmuster in einem UML-Modell anwenden können. Bei Rational Software Architect wird neben der Musteranwendung im UML-Modell auch eine Template-basierte Codegenerierung je Muster implementiert [SCG+05, Kap. 9.4] [LB05]. Bei MagicDraw ist die Musteranwendung auf das Modifizieren eines UML-Modells beschränkt [Vai15]. Das Anwenden eines Musters erfolgt analog zu den zuvor genannten Werkzeugen Enterprise Architect, Together und DPAToolkit. Rational Software Architect markiert die Anwendungsstelle zusätzlich mit einer UML-Kollaboration, analog zu Together (siehe S. 220).

Transformationen pro Muster

zu unflexibel, keine Varianten

Durch die explizite Implementierung von Musteranwendungstranformationen speziell für ein bestimmtes Entwurfsmuster, lassen sich bei Rational Software Architect und MagicDraw auch komplexere Operationen als eine 1-zu-1-Übertragung von als Schablonen dienender UML-Modelle implementieren. Dennoch sind solche Transformationen durch eine feste Anzahl ihrer vordefinierten Parameter[25] in ihrer Flexibilität beschränkt. Die Parameter legen u.a. fest, wie viele Klassen die Rollen eines Entwurfsmusters einnehmen können, sodass z.B. bei dem Muster Abstract Factory im Gegensatz zu meinem Ansatz nur eine feste Anzahl an Fabrik- und Produkt-Klassen verwendet werden kann. Bei MagicDraw ist die Einschränkung sogar noch größer, weil dort ein Muster grundsätzlich auf genau einer Klasse (a pattern's target) im UML-Modell angewandt wird [Vai15].

Yacoub & Ammar

Muster als Komponenten, keine Werkzeuge

Yacoub und Ammar beschreiben einen Ansatz, bei dem Entwurfsmuster als Komponenten mit speziellen Schnittstellen spezifiziert und miteinander zu kompletten Entwürfen kombiniert werden [YA03] (siehe S. 216). Sie beschreiben ein Vorgehen, welches bisher scheinbar aus ausschließlich manuellen Schritten besteht. Renommierte Autoren zahlreicher Bücher zum Thema Entwurfsmuster schreiben, dass Muster keine wiederverwendbaren Komponenten sind (siehe Zitat auf S. 23). Es ist somit fraglich, ob der von Yacoub und Ammar beschriebene Ansatz praktikabel ist. Ein Nachweis in Form eines Werkzeugprototypen fehlt jedenfalls bisher.

Briand et al.

Anwendungsstellen vorschlagen

Briand et al. gehen davon aus, dass heutige IDEs[26] die Anwendung eines Musters bereits ausreichend unterstützen. Sie stellen ein Verfahren vor, bei welchem mit Hilfe von Modellanalysen und Benutzerbefragungen Stellen in UML-Klassendiagrammen identifiziert werden, an denen die Anwendung eines Entwurfsmusters Sinn machen könnte [BLS06]. Die anschließende Anwendung des vorgeschlagenen Musters wird den IDEs und dem Entwickler überlassen.

nicht im Fokus

Der Fokus meiner Arbeit liegt auf der Musteranwendung und nicht auf der Auswahl der Anwendungsstelle. Das Verfahren von Briand et al. könnte meinen Ansatz jedoch ergänzen.

[23] Rational Software Architect: http://www-03.ibm.com/software/products/de/ratsadesigner
[24] MagicDraw: http://www.nomagic.com/products/magicdraw.html
[25] Bedingt durch den Erweiterungsmechanismus von MagicDraw & Rational Software Architect
[26] IDE: Integrated Development Environment

9.4 Architektur- und Design-Konformitätsprüfung

Ist ein Entwurfsmuster einmal angewandt und der Entwurf oder die Implementierung einer Software danach mehrfach angepasst worden, ist es schwer zu entscheiden, ob die ursprüngliche, inzwischen modifizierte Musterimplementierung weiterhin dem angewandten Entwurfsmuster entspricht und der ursprünglichen Intention dient. Zur Beantwortung dieser Frage wurden diverse Ansätze entwickelt, die ich im Folgenden vorstelle und meiner Arbeit gegenüberstelle. Bei einem Teil der Ansätze erfolgt die Konformitätsprüfung einer Musterimplementierung im Quellcode (Abschn. 9.4.1), während die Mehrheit der Ansätze die Konformität von Musterimplementierungen in UML-Entwurfsmodellen prüft (Abschn. 9.4.2). Ergänzend zu diesen gehe ich auf entfernt verwandte Arbeiten mit dem Ziel einer Entwurfs- oder Architekturkonformitätsprüfung ein (Abschn. 9.4.3).

Arbeiten gruppiert nach Einsatzzweck

9.4.1 Konsistenz von Musterimplementierungen im Quellcode

Florijn et al. beschreiben Musterspezifikationen, Musterimplementierungen und Anwendungsstellen durch sogenannte Fragment-Graphen, wo die Fragmente als eine Art Platzhalter für den Namen eines Musters, für ein Smalltalk-Objekt oder für ein Stück Quellcode (z.B. eine Methodenrumpfimplementierung) verwendet werden [FMvW97] (siehe Seiten 210, 220, 228). Fragments enthalten u.a. manuell implementierte Operationen zur Prüfung einer Musteranwendungsstelle. Solche Prüfoperationen werden komplett frei oder aus Basis prädikatenlogischer Formeln implementiert. In den implementierten Formeln können die logischen Operatoren $\wedge$, $\vee$, $\neg$, $\forall$ und $\exists$ sowie folgende Prädikate verwendet werden: eine Klasse enthält die Implementierung einer konkreten oder die Deklaration einer abstrakten Methode, eine Klasse implementiert alle in einer anderen Klasse deklarierten Methoden, eine Methode instanziiert eine Klasse, eine Methode ruft direkt oder indirekt eine andere Methode auf, eine Klasse enthält eine andere Klasse.

Florijn et al.

programmierte Prüfoperationen, Prädikatenlogik

Im Vergleich zu meinem Ansatz ist die Menge der prüfbaren Eingenschaften bei Florijn et al. relativ eingeschränkt, z.B. werden anders als bei meiner Arbeit ungewünschte Abhängigkeiten im Entwurf (siehe Abschn. 6.2.5) und gefordertes Verhalten nicht geprüft (siehe Abschn. 6.2.4). Die manuelle Implementierung der Prüfungen je Muster ist aufwändig. Im Gegensatz dazu sieht mein Ansatz vordefinierte Prüfungen für jedes Sprachkonstrukt der Musterspezifikationssprache vor, sodass bei hinzukommenden Musterspezifikationen keine neuen Prüfungen implementiert werden müssen. Der Konsistenzprüfungsmechanismus von Florijn et al. ist im Gegensatz zu meinem prototypisch implementiert.

Kopplung & Verhalten nicht validiert

Eden et al. spezifizieren sowohl Entwurfsmuster (siehe S. 211) als auch Implementierungen von Entwurfsmustern (siehe S. 224) in einer formal definierten, abstrakten Sprache namens LePUS3. Die Konsistenz einer Musterimplementierung mit der zugehörigen Musterspezifikation wird durch Vergleichen der beiden LePUS3-Repräsentationen geprüft. Die Prüfung entspricht dabei dem Auswerten einer prädikatenlogischen Formel, welche ausdrückt, dass eine Musterimplementierung alle Eigenschaften der Musterspezifikation erfüllt [NGEK09, Ede00, EN11]. Dazu wird zunächst eine abstrakte Repräsentation des vorliegenden Quellcodes, sogenannte Codecharts [EN11, GENK08], mit Hilfe einer statischen Quellcode-

Eden et al.

Vergleich zweier Abstraktionen, Prädikatenlogik

analyse erstellt, vergleichbar mit dem Erstellen eines abstrakten Syntaxgraphen. Anschließend wird manuell eine Rollenzuordnung vorgenommen, indem Elemente im Quellcode (im Codechart), z.B. Klassen, Elementen einer Musterspezifikation, z.B. zu einer Klassenhierarchie, zugeordnet werden [EN11, Kap. 15.4]. Erst dann kann eine automatische Validierung der Musterimplementierung erfolgen.

Die Repräsentation von Musterspezifikationen und des Programmquellcodes in einer formalen, entscheidbaren Sprache (Teilmenge der Prädikatenlogik erster Ordnung [NGEK09]) stellt eine hervorragende Grundlage für Verifikationen und Analysen dar. Der Ansatz hat allerdings einige signifikante Nachteile: Für eine Prüfung der Konsistenz muss (a) jede zu prüfende Musterimplementierungsstelle bekannt sein und (b) die Rollenzuordnung vor jeder Prüfung manuell erstellt (oder zumindest aktualisiert) werden. Außerdem kann nicht der gesamte Quellcode auf Konformität mit allen spezifizierten Mustern geprüft werden. Es scheint so, als müsste die Prüfung für jede Musterimplementierung einzeln vorgenommen werden. Im Gegensatz dazu werden Anwendungsstellen bzw. Rollenzuordnungen bei meinem Ansatz schon bei der Musteranwendung in einem separaten Modell erfasst. Mit Hilfe dieser Rollenzuordnung kann die Konsistenzprüfung bei meinem Ansatz auf dem gesamten Entwurfsmodell für alle Musteranwendungsstellen erfolgen. Der Aufwand für die Rollenzuordnung entfällt. Musterimplementierungen werden nicht vergessen oder übersehen, womit das Risiko für versehentliche Entwurfsabweichungen reduziert wird. Ergänzend zu den bei Eden et al. prüfbaren Eigenschaften einer Musterimplementierung kann bei meinem Ansatz auch die Abwesenheit ungewollter Abhängigkeiten im Entwurf geprüft werden (siehe Abschn. 6.2.5).

Anwendungsstellen unbekannt, hoher manueller Aufwand, punktuelle Prüfungen

9.4.2 Konsistenz von Musterimplementierungen in Entwurfsmodellen

Maplesden et al. nutzen das bei einer semiautomatischen Musteranwendung (siehe S. 230) erstellte Musterinstanziierungsmodell (siehe S. 219) zur anschließenden Prüfung der Musterimplementierungen auf Konsistenz mit den zugehörigen Musterspezifikationen (siehe S. 213) [MHG02, MHG07]. Geprüft werden einfache Bedingungen wie die Vollständigkeit der Rollenzuordnung, passende Typen der zugeordneten Modellelemente und die Einhaltung von Namenskonventionen. Die Prüfungen sind beschränkt auf Klassendiagramme.

Maplesden et al.

Prüfung auf Klassenebene

Alle genannten Prüfungen mit Ausnahme der Namenskonventionen bietet auch mein Ansatz. Wie bei der Musteranwendung und der Rollenzuordnung werden bei Maplesden et al. im Gegensatz zu meinem Ansatz Verhaltensmodelle bei der Konsistenzprüfung nicht berücksichtigt. Darüber hinaus bietet mein Ansatz Entwicklern mehr Flexibilität bei der Musteranwendung, da meine Konsistenzprüfung auch indirekte Beziehungen wie z.B. indirekte Vererbungsbeziehungen oder indirektes Bereitstellen von Attributen, Methoden oder Assoziationen über Oberklassen erlaubt. Ergänzend dazu werden bei meinem Ansatz ungewollte Abhängigkeiten bei der Konsistenzprüfung berücksichtigt (siehe Abschn. 6.2.5), was dem Ansatz von Maplesden et al. fehlt.

Kopplung & Verhalten nicht validiert, keine Transitivität

Kim und Shen führen eine Konsistenzprüfung in UML-Modellen durch, indem sie mit Hilfe eines Reverse-Engineering-Algorithmus nach einer zur einer Musterspezifikation konsistenten Musteranwendungsstelle suchen. Wird so eine Stelle gefunden, erklären sie die Anwendung des Musters für korrekt [KS08]. Als Konsis-

Kim & Shen

Anwendungsstellensuche

tenzregeln werden ähnlich einfache Bedingungen wie bei Maplesden et al. geprüft, nämlich die Vollständigkeit der Rollenzuordnung, Typkonformität, die Einhaltung von Kardinalitäten sowie der Verfügbarkeit von spezifizierten Beziehungen (Vererbung, Assoziationen, Abhängigkeiten). Einige der Konsistenzregeln werden in Form von OCL-Ausdrücken in den Musterspezifikationen angegeben (siehe S. 214).

Dieser Ansatz hat nach meiner Einschätzung gravierende Nachteile, welche ihn unpraktikabel machen. Im Gegensatz zu meinem Ansatz sind hier die zu prüfenden Anwendungsstellen unbekannt und müssen daher per Reverse Engineering gefunden werden. Automatische Reverse-Engineering-Verfahren sind unpräzise und liefern u.a. False Positives[27], sodass ggf. nicht eine Anwendungsstelle eines Musters, sondern bloß eine strukturell ähnliche Stelle auf Konsistenz zu einer Musterspezifikation geprüft wird. Außerdem kann man sich bei Reverse-Engineering-Verfahren nicht darauf verlassen, dass alle Musteranwendungsstellen gefunden werden (Stichwort: False Negatives[28]). Der beschriebene Reverse-Engineering-Algorithmus sucht nach genau *einer* Musterimplementierung genau *eines* Musters in nur *einem* Klassendiagramm. Es ist jedoch unrealistisch, anzunehmen, in einem Softwareentwurf oder einem Klassendiagramm gäbe es nur eine Anwendungsstelle für ein bestimmtes Muster. Da man nicht weiß, welche Muster angewandt wurden, muss man die Konsistenzprüfung bei diesem Ansatz mit jeder Kombination aus Musterspezifikation und Klassendiagramm wiederholen, was unnötig aufwändig ist. Ist ein Muster nie angewandt worden, erhält man als Ergebnis die Aussage, die Musteranwendung bzw. das Klassendiagramm wäre nicht konform zur Musterspezifikation, was irreführend ist. Aufgrund der genannten Einschränkungen des Ansatzes ist die Aussagekraft der geprüften Konformität sehr beschränkt.

Anwendungsstellen unbekannt, ihre Suche unzuverlässig, Annahmen unrealistisch

Park et al. setzen auf eine zu Kim et al. sehr ähnliche, UML-basierte Musterspezifikationssprache (siehe S. 215), aber auf eine andere Konsistenzprüfung [Par07, PRW08]. Sie gehen davon aus, dass Musteranwendungsstellen in UML-Klassendiagrammen mit Hilfe von Stereotypen markiert und bestimmte Namenskonventionen eingehalten sind. Mit Hilfe dieser Markierungen werden die Anwendungsstellen schrittweise auf Konsistenz zu zugehörigen Musterspezifikationen geprüft. Es wird die Vollständigkeit und korrekte Kardinalität der Rollenzuordnungen sowie das Vorhandensein und passende Kardinalität der spezifizierten Beziehungen geprüft.

Park et al.

Prüfung auf Klassenebene

Da die Anwendungsstellen bekannt sind, werden die Schwächen des Konsistenzprüfungsalgorithmus von Kim et al. vermieden. Dagegen weist dieser Ansatz die gleichen Schwächen auf wie der Ansatz von Maplesden et al. Verhalten und zu vermeidende Abhängigkeiten werden bei der Konsistenzprüfung nicht beachtet. Indirekte statt direkter Beziehungen werden nicht als konform zu einer Musterspezifikation angesehen.

Kopplung & Verhalten nicht validiert, keine Transitivität

Borland Together[29] scheint das einzige kommerzielle Werkzeug zu sein, welches eine Konsistenzprüfung für Musteranwendungsstellen anbietet [Bor11, S. 556]. Aus der Benutzeranleitung lässt sich jedoch nicht ergründen, was genau an einer Mus-

Together

[27] False Positive im Allgemeinen: ein falscher Fund, hier: eine fälschlicherweise als Musteranwendungsstelle erkannte Stelle im UML-Modell

[28] False Negative im Allgemeinen: ein fehlender Fund, hier: eine fälschlicherweise nicht als Musteranwendungsstelle erkannte Stelle im UML-Modell

[29] Together: http://www.borland.com/de-DE/Products/Requirements-Management/Together

terimplementierung geprüft wird und wie. Detailliertere Informationsquellen zu diesem Thema habe ich nicht gefunden. Vermutlich wird auch von Together analog zu Maplesden et al. nur die Vollständigkeit einer Rollenzuordnung sowie die Verfügbarkeit aller spezifizierten Beziehungen geprüft.

unbekanntes Vorgehen

9.4.3 Weitere Architektur- und Entwurfskonformitätsprüfungen

Wenzel

Wenzel bietet eigentlich kein Verfahren zur Prüfung der Konsistenz von Musteranwendungsstellen zur ihren Musterspezifikationen. Stattdessen unterstützt er Entwickler beim Erstellen oder Erweitern eines Softwareentwurfs, indem sogar unvollständige Musterimplementierungen in UML-Klassendiagrammen automatisch identifiziert und visualisiert werden [Wen05a, Wen05b]. Damit ist die Arbeit eher im Gebiet des Reverse Engineerings anzusiedeln. Dennoch kann sein Verfahren u.a. dazu genutzt werden, die Korrektheit von Musteranwendungsstellen zu prüfen, indem Entwickler auf unvollständige Anwendungsstellen hingewiesen werden. Für jedes Element bzw. jede Bedingung seiner hierarchischen Musterspezifikation (siehe S. 216) wird ein Übereinstimmungsgrad zwischen 0 % und 100 % berechnet. Daraus lässt sich ableiten, welche Teile eines Musters bei einer Anwendungsstelle fehlen oder nicht hinreichend mit der Musterspezifikation übereinstimmen.

Erkennen unvollständiger Anwendungsstellen

Reclipse & Fujaba

Die Werkzeuge Reclipse und Fujaba[30] implementieren einen weiteren Ansatz aus dem Reverse Engineering [vDMT10a, vDMT10b, Wen07, NSW+02]. Auch hier können Musterspezifikationen dazu genutzt werden, Anwendungsstellen eines Musters automatisch in Modellen oder im Code zu finden. Mit Hilfe einer Laufzeitanalyse kann bei der Erkennung u.a. das Verhalten einer Software analysiert werden [Wen07]. Ähnlich wie bei Wenzel lassen sich für die gefundenen Teile einer Anwendungsstelle Übereinstimmungsgrade mit der Musterspezifikation bestimmen [Tra06] und damit problematische, ggf. unvollständige oder falsche Musterimplementierungen identifizieren. Außerdem lassen sich damit nicht nur Musteranwendungsstellen, sondern auch Entwurfsmängel wie Anti-Patterns [BMMM98] und Code Smells [Fow99, Kap. 3] erkennen, um anschließend per Modelltransformationen in bessere Entwurfslösungen gewandelt zu werden [Mey09]. Mit Archimetrix, einer Erweiterung von Reclipse, lässt sich eine Komponenten-basierte Architektur einer als Quellcode vorliegenden Software herleiten und Architekturmängel aufdecken [PvDB12, vD13]. Alle diese Arbeiten können indirekt dazu verwendet werden, Entwurfs- bzw. Architekturkonformität einer Software zu prüfen.

Erkennen & Beseitigen fehlerhafter Musterimplementierungen & Entwurfsmängeln

Dong et al.

Dong et al. nutzen Theorembeweiser für ihre Konsistenzprüfungen und beweisen damit die Korrektheit der Komposition mehrerer Muster oder die korrekte Implementierung eines Musters im Entwurf [DACY07, DAC07]. Damit die Beweisführung weitestgehend automatisch erfolgt, wird eine Prolog Inference Engine eingesetzt und sowohl die Musterspezifikationen, als auch Musterkompositionen und Musteranwendungsstellen formal in Prolog-Ausdrücken und Prolog-Regeln spezifiziert. Die Prolog-Spezifikationen sind nicht nur schwer zu lesen, sondern müssen bei jeder Entwurfsänderung bei betroffenen Anwendungsstellen angepasst werden, bevor eine erneute Konsistenzprüfung erfolgen kann.

Beweise auf Prolog-Repräsentationen, aufwändig

Neben der Prüfung der Konformität von Entwurfsmusterimplementierungen

[30] Fujaba: http://www.fujaba.de

kann auch die Einhaltung von Architektur- oder Entwurfsvorgaben geprüft werden. So gibt es einige Werkzeuge, welche sich der statischen Codeanalyse bedienen, um die Einhaltung explizit definierter Architekturregeln im Quellcode zu prüfen. Zu solchen Werkzeugen gehören z.B. SonarQube[31], Structure101[32] und Teamscale[33]. Die Architekturregeln beschreiben insbesondere Architekturschichten oder Softwarekomponenten sowie gewünschte und unerwünschte Abhängigkeiten dazwischen, was ich in meiner Arbeit teilweise aufgreife. Teile des Quellcodes wie Klassen und Pakete werden den Schichten oder Komponenten einer hierarchischen Architekturvorgabe zugeordnet. Mit Hilfe statischer Codeanalyse werden Abhängigkeiten zwischen den Methoden, Klassen und Paketen im Code bestimmt und damit auf die Abhängigkeiten zwischen den Schichten und Komponenten geschlossen. Weichen die so ermittelten Abhängigkeiten von den Architekturvorgaben ab, wird eine Architektur- oder Regelverletzung gemeldet. Neben den individuell definierten Architekturregeln werden auch allgemeine Regeln, z.B. zur Vermeidung zyklischer Abhängigkeiten geprüft.

SonarQube, Structure-101, Teamscale

statische Codeanalyse, ungewollte Kopplung aufzeigen

Bidirektionale Modelltransformationssprachen wie TGG [Sch94, HLG+13] und QVT-R [OMG11d] können nicht nur zur Beschreibung von Übersetzungen zwischen Modellen, sondern u.a. zur Prüfung oder Wiederherstellung der Konsistenz zwischen den Modellen genutzt werden [KW07, MGC13, LAS17]. Das klappt sogar, wenn die in Konsistenz gehaltenen Modelle sich auf unterschiedlichen Abstraktions- und Detailleveln befinden und nur bestimmte Modelländerungen in das jeweils andere Modell propagiert werden sollen [Anj14, Rie14]. Prinzipiell könnten solche Verfahren dazu genutzt werden, eine Musterspezifikation in eine Musterimplementierung zu übersetzen und diese anschließend nach jeder Modifikation auf Konsistenz zur Musterspezifikation zu prüfen, indem Konsistenzrelationen [Anj14, LAS17] oder Verfeinerungsregeln [Rie14] beschrieben werden.

bidirektionale Transformationen

Konsistenzprüfung, Modellsynchronisation

Dagegen sprechen nach meiner Einschätzung die in Abschnitt 5.5.2 angesprochenen Einschränkungen solcher Ansätze (vgl. Herausforderungen in Abschn. 5.5.1). Bidirektionale Transformationen übersetzen normalerweise komplette Modelle ineinander, während bei meinem Anwendungsszenario (a) mehrere Musterspezifikationen unabhängig voneinander und ggf. im Entwurfsmodell überlappend übersetzt werden, (b) zu einer Musterspezifikation ggf. mehrere Übersetzungen im Entwurfsmodell entstehen, (c) der Startpunkt für die Übersetzung im Zielmodell (Entwurfsmodell) aus mehreren vom Entwickler vorgegebenen Korrespondenzen (Rollenzuordnungen) besteht und nicht an der Wurzel des baumartigen Modells beginnt. Außerdem reicht für meinen Anwendungsfall eine unidirektionale Übersetzung aus, eine Rückpropagierung von Änderungen an einer Musterimplementierung zurück in die Musterspezifikation ist sogar unerwünscht (die empfohlene Entwurfslösung ändert sich nicht). Ob die Auffaltung von insb. sich überschneidenden Set Fragments (Abschn. 3.5.2 und 5.4) sich mit bidirektionalen Transformationen, selbst mit fortschrittlichen Konzepten wie Amalgamierung [LAST16] vollständig und Semantik-konform (siehe Anhang A.3.3) beschreiben lässt, ist bisher unklar und muss noch untersucht werden.

diverse Einschränkungen aufgrund untypischen Anwendungsfalls

[31] Architekturregeln in SonarQube: http://docs.sonarqube.org/display/SONAR/Rules#Rules-RuleTemplatesandCustomRules

[32] Architekturregeln in Structure101: http://structure101.com/products/#products=2

[33] Architekturregeln in Teamscale: https://www.cqse.eu/de/produkte/teamscale/features/#Analyses

Lee & Kruchten, Könemann, Durdik
Entscheidungen dokumentieren

Im Prinzip können auch dokumentierte Entwurfs- und Architekturentscheidungen dabei helfen, die Konformität einer Softwareimplementierung oder eines Softwareentwurfs zu Entwurfs- oder Architekturvorgaben zu prüfen. Arbeiten wie die von Lee und Kruchten [LK08], Könemann [Kön11] oder Durdik [Dur14] bieten Verfahren zur Erfassung solcher Entscheidungen auf Entwurfs- oder Architekturebene. Eine automatische Konformitätsprüfung bieten die Ansätze jedoch nicht.

9.5 Zusammenfassung und Fazit

In diesem Kapitel werden diverse Arbeiten betrachtet, welche sich mit ähnlichen Forschungsfragen beschäftigen wie die vorliegende Arbeit. Dazu zählen die vier folgenden Forschungsgebiete.

- Spezifikation von Entwurfsmustern (Spezifikation der Entwurfslösung eines Musters, d.h. typischer Musterimplementierungen)
- Semiautomatische Musteranwendung (Synthese einer Musterimplementierung aus einer Musterspezifikation)
- Modellierung & Visualisierung von Musteranwendungsstellen (Verknüpfung von Musterimplementierungen mit zugehörigen Musterspezifikationen sowie ihre Darstellung)
- Validierung von Musteranwendungsstellen (Prüfung der Konsistenz einer Musterimplementierung zur zugehörigen Musterspezifikation)

Unter den zahlreichen Arbeiten auf dem Gebiet der Musterspezifikation befinden sich Arbeiten mit mathematisch formalen, aber dafür für Entwickler ohne mathematisches Hintergrundwissen kaum lesbaren Spezifikationen. Arbeiten mit besser lesbaren Musterspezifikationen vernachlässigen die Spezifikation des zu einem Muster gehörenden Verhaltens und bieten – wenn überhaupt – nur beschränkte Möglichkeiten bei der Beschreibung der zu einem Muster konformen Implementierungsvarianten. Insbesondere fehlt allen Arbeiten ein mit Set Fragments (Abschn. 3.5.2) vergleichbares Konzept zur Beschreibung zusammenhängender, sich als Ganzes wiederholender, ggf. überschneidender Entwurfsstrukturen. In einer Entwurfslösung ungewünschte Abhängigkeiten (Abschn. 3.5.3) werden ebenfalls nicht berücksichtigt.

Spezifikationen komplex, zu wenige Varianten, kein Verhalten

Musteranwendung unflexibel, schwer zu erweitern, ignoriert Verhalten, primitive Anwendungsstellenerfassung

Die Ansätze zur semiautomatischen Anwendung von Entwurfsmustern sind entweder auf simple Replikation vorgefertigter Entwurfsstrukturen bzw. die Generierung einfacher Klassenhülsen beschränkt oder erfordern das aufwändige Programmieren von Musteranwendungsoperationen für jedes neue Muster. Teile des bereits existierenden Entwurfs oder Codes lassen sich bei vielen Ansätzen bei der Musteranwendung nicht wiederverwenden. Zu Mustern gehörendes Verhalten wird kaum berücksichtigt. Bestenfalls wird vordefinierter Code für Methodenrümpfe oder nicht ausführbare Sequenzdiagramme generiert. Nur wenige Arbeiten erfassen Anwendungsstellen bei Musteranwendung automatisch und dauerhaft, um eine Visualisierung oder Validierung der Stellen zu ermöglichen. Anwendungsstellen werden in Form einer Rollenzuordnung auf Ebene der Klassendiagramme erfasst. Verhalten wird dabei bei keinem der Ansätze berücksichtigt. Ebenso fehlt ein

Konzept, solche Rollenzuordnungen auch zwischen sehr unterschiedlich detaillierten Modellen für Musterspezifikationen und den Entwurf zu erstellen wie es in meiner Arbeit mit Hilfe von Tokens möglich ist (Abschn. 4.1, 4.2, C.3.3).

unvollständige, unübersichtliche Darstellung von Anwendungsstellen

Unabhängig von der Erfassung von Anwendungsstellen gibt es einige Ansätze zur Visualisierung von Anwendungsstellen in Entwurfsmodellen. Die meisten davon verfolgen den Ansatz, Musterimplementierungen in UML-Klassendiagrammen zu markieren. Es gibt ähnliche Ansätze zur Markierung von Musterimplementierungen im Quellcode. Meist fehlen diesen Ansätzen Informationen über eingenommene Rollen auf Ebene von Methoden, Attributen, Assoziationen oder anderen an einer Musterimplementierung Beteiligten. Werden die Rollen detailliert dargestellt, nimmt die Komplexität der Darstellung deutlich zu und die Übersicht ab. Zu einer Musterimplementierung modelliertes oder implementiertes Verhalten wird bei der Visualisierung im Gegensatz zu meinem Ansatz gar nicht berücksichtigt.

einfache Konsistenzprüfung, kein Verhalten, keine Kopplung

Nur wenige Forscher beschäftigen sich mit der automatischen Prüfung der Konsistenz einer Musterimplementierung im Entwurfsmodell oder Quellcode mit einer zugehörigen Musterspezifikation. Die Prüfung beschränkt sich bei den existierenden Arbeiten auf die Vollständigkeit der Musteranwendung – zu jeder Musterrolle muss eine Entsprechung im Modell oder Code zu finden sein – und Einhaltung der spezifizierten Beziehungen (Vererbung, Assoziationen, etc.). Kopplungsrestriktionen wie bei meiner Arbeit werden nicht geprüft. Ebenso wird das Verhalten von Musterimplementierungen wie z.B. laut Musterspezifikationen geforderte Delegationen oder Methodenaufrufe bis auf wenige Ausnahmen nicht geprüft.

Eine Arbeit, welche alle vier Forschungsgebiete in einem ähnlichen Umfang gemeinsam betrachtet und einen zu meiner Arbeit vergleichbaren, ganzheitlichen Ansatz zur Spezifikation, Anwendung, Dokumentation und Validierung von Entwurfsmustern bietet, gibt es bisher nicht.

10 Zusammenfassung und Ausblick

Zur Vermeidung von versehentlichen Abweichungen von angewandten Softwareentwurfsmustern wird in dieser Arbeit ein Verfahren vorgestellt, welches die Erfassung, Visualisierung und Validierung von Musterimplementierungen vereinfachen und Entwickler bei der Anwendung von Entwurfsmustern unterstützen soll. Im Folgenden fasse ich den entwickelten Ansatz sowie die Ergebnisse der Arbeit zusammen und gebe anschließend einen Ausblick auf potentielle zukünftige Forschungsarbeiten.

10.1 Zusammenfassung

Damit einst angewandte, jedoch nicht kenntlich gemachte Entwurfsmusterimplementierungen nicht übersehen und Entwurfsänderungen nicht ohne Kenntnis über diese Musterimplementierungen gemacht werden, stelle ich in dieser Arbeit ein Verfahren vor, welches Softwareentwicklern dabei helfen soll, Musterimplementierungen zu dokumentieren, geeignet zu visualisieren (Ziel 2, S. 9) und sogar nach Entwurfsänderungen auf versehentliche Abweichungen von Entwurfsmustern zu prüfen (Ziel 3, S. 9). Eine der Kernideen dabei ist es, Musterimplementierungen schon bei der Anwendung eines Musters halbautomatisch zu erfassen und Entwickler durch die Musteranwendung zu führen, um auch bei diesem Schritt das Risiko für Entwurfsfehler zu reduzieren (Ziel 1, S. 9).

Nachvollziehbarkeit angewandter Muster

Das vorgestellte Verfahren erfordert zunächst die Spezifikation der Entwurfslösung zu einem anzuwendenden Entwurfsmuster. Zu diesem Zweck wurde eine Musterspezifikationssprache für objektorientierte Entwurfsmuster entwickelt, mit welcher die Klassenstruktur und das Verhalten einer Entwurfslösung definiert und verschiedene Implementierungsvarianten eines Musters kompakt in bloß einer Spezifikation erfasst werden können (Antwort auf Forschungsfrage (i), S. 10). Ergänzend dazu können beabsichtigte und unerwünschte Abhängigkeiten im Entwurf sowie Freitexthinweise für Entwickler (Entwurfsaufgaben) angegeben werden.

Musterspezifikation

Spezifizierte Entwurfsmuster können Werkzeug-gestützt angewandt werden, indem Entwickler den Rollen eines Entwurfsmusters bereits existierende Teile des Entwurfs (z.B. Klassen oder Methoden) zuordnen und alle im Entwurf fehlenden Teile der gewünschten Musterimplementierung automatisch ergänzen lassen (Antwort auf Forschungsfrage (ii), S. 10). Neben der dabei generierten, auf die vorliegende Situation zugeschnittenen Musterimplementierung – bestehend aus einem Klassen- und Verhaltensmodell – wird auch ein Modell der Anwendungsstelle generiert. Liegt eine Musterimplementierung vollständig vor, kann sie auf die gleiche Weise nachträglich mit den Musterrollen verknüpft und das Modell der Anwendungsstelle generiert werden.

Musteranwendung

Durch das Generieren eines Modells der Anwendungsstelle wird dem Anwen-

der eines Musters Aufwand für die Dokumentation der Anwendungsstelle (zumindest teilweise) erspart. Die Anwendungsstelle wird als detaillierte Zuordnung aller Musterrollen zu den sie einnehmenden Entwurfsteilen erfasst und kann auf verschiedene Arten mit diversen Detailgraden visualisiert werden (Antwort auf Forschungsfrage (iii), S. 10). Die Darstellungsmöglichkeiten reichen von der Markierung beteiligter Elemente in Entwurfsdiagrammen über die detaillierte Angabe aller ihrer eingenommen Rollen bis zur isolierten Darstellung der an einer bestimmten Musterimplementierung beteiligten Entwurfsteile in einem generierten Entwurfsdiagramm oder einer an die Musterspezifikation angelehnten Darstellung einer Musterimplementierung.

Erfassung & Visualisierung von Anwendungsstellen

Muss der Entwurf an der Stelle einer Musterimplementierung angepasst werden, kann mit Hilfe einer automatischen Validierung die Konsistenz der geänderten Musterimplementierung mit der zugehörigen Musterspezifikation geprüft werden. Versehentliche Abweichungen vom Entwurfsmuster werden dabei aufgedeckt, z.B. fehlende Klassen, Methoden, Beziehungen, Methodenaufrufe oder aufgetauchte ungewollte Abhängigkeiten (Antwort auf Forschungsfrage (iv), S. 10).

Validierung von Anwendungsstellen

Als Proof of Concept wurde das Verfahren prototypisch implementiert und anhand diverser Entwurfsmuster und Anwendungsszenarien evaluiert. Dabei sind diverse Werkzeuge und Editoren in einer Entwurfsumgebung namens *Pattern-Oriented Software Engineering Tool Suite* (POSE) entstanden.

Machbarkeit gezeigt

Zur Evaluation der Musterspezifikationssprache wurden 34 Musterspezifikationen erstellt, worunter die 23 Gang-of-Four-Entwurfsmuster [GHJV95] sowie einige andere Entwurfsmuster, Architekturmuster und Idiome fallen. Eine heterogene Auswahl der spezifizierten Muster wurde mit dem entwickelten Verfahren in verschiedenen Ausgangssituationen angewandt. Außerdem wurde zwecks Evaluation ein realistisches Anwendungsszenario mehrerer überlappend angewandter Muster mit dem vorgestellten Verfahren modellgetrieben nachempfunden. Als Beispiel diente das JUnit-Framework, dessen Musteranwendungen detailliert und chronologisch dokumentiert sind [Gam01] (siehe Anhang D).

Evaluation

Das Verfahren geht der Einfachheit halber von einer modellgetriebenen Softwareentwicklung aus, ist aber prinzipiell nicht darauf beschränkt[1]. Bei dem Prototypen werden Muster in Entwurfsmodellen bestehend aus einem Ecore-Klassendiagramm-Modell und einem Story-Diagramm-Modell (einer speziellen Art von Aktivitätendiagrammen) angewandt. Für Ecore-Modelle gibt es eine Codegenerierung im Eclipse Modeling Framework [SBPM08]. Story-Diagramme haben eine Ausführungssemantik [Zün01, vDHH+12] und können direkt per Interpreter [GHS09] oder nach ihrer Übersetzung in Quellcode [GSR05] ausgeführt werden.

Annahme MDSD

Dank sogenannter Set Fragments lassen sich zahlreiche Varianten von Musterimplementierungen kompakt und übersichtlich in nur einer Musterspezifikation erfassen. Ihre formal definierte Semantik, die Auffaltung, beschreibt wie ein Teil einer Musterspezifikation (ein Teilgraph) beliebig häufig so repliziert werden kann, dass das Ergebnis weiterhin konform zur Musterspezifikation bleibt, selbst bei sich

Varianten durch Set Fragments

[1] Die Musteranwendung könnte technisch auch auf einer abstrakten Coderepräsentation, einem abstrakten Syntaxgraphen erfolgen, was analog für die Modellierung von Anwendungsstellen und die Validierung gilt. Die Darstellungen müssten auf eine Code-basierte Notation umgestellt werden. Die Umsetzung wäre aber aufgrund detaillierterer Code-Repräsentationen und zusätzlicher Implementierungsvarianten komplizierter.

überschneidenden oder ineinander geschachtelten Set Fragments. Dieses Konzept hat sich beim Großteil der betrachteten Muster als sehr nützlich erwiesen.

Synthese einer Musterimplementierung, Auffaltung & Übersetzung

Zwecks Anwendung eines Musters wurde ein zweistufiger Mechanismus entwickelt. Im ersten, interaktiven Schritt erfolgt die Rollenzuordnung manuell, während im Hintergrund automatisch ein Modell der Anwendungsstelle und eine abstrakte Repräsentation der gewünschten Musterimplementierung entsteht. Letztere entsteht u.a. durch Auffaltung der Set Fragments, d.h. es werden Vielfache der in Set Fragments spezifizierten Entwurfsmodellstrukturen erzeugt. Im zweiten, automatischen Schritt wird das abstrakte Modell der Musterimplementierung in eine konkrete Musterimplementierung im Entwurfsmodell übersetzt. Für diesen Schritt wurde eine spezielle Modelltransformation entwickelt. Zu den Herausforderungen dieser Transformation gehören insb. Folgende: interaktiv festgelegte Korrespondenzen und damit mehrere frei wählbare Startpunkte für die Übersetzung (Rollenzuordnung), die Übersetzung bildet nur einen Teil des Zielmodells (Teil des Entwurfs), die Übersetzung hängt nicht nur vom Quell-, sondern auch von der Umgebung im Zielmodell ab (z.B. Methodensignatur erben), die Übersetzung eines Musters kann beliebig häufig auf unterschiedliche Weise im Zielmodell vorkommen (mehrere verschiedene Implementierungen eines Musters) und die Detailgrade von Quell- und Zielmodell variieren stark (insb. bei Verhalten).

Spezifikationssprache für den Ansatz geeignet

Es konnte gezeigt werden, dass mit der entwickelten Musterspezifikationssprache die betrachteten Entwurfsmuster samt Klassenstruktur und zugehörigem Verhalten hinreichend genau und mit zahlreichen Implementierungsvarianten kompakt beschrieben werden können. Die Spezifikationen lassen sich für eine semiautomatische Musteranwendung, für die Erfassung von Anwendungsstellen und für eine vollautomatische Validierung von Musterimplementierungen nutzen.

flexibel bei und nach der Musteranwendung

Die vorgestellte Anwendung von Entwurfsmustern bietet viel Flexibilität. Die Ausgangssituation kann nahezu beliebig gewählt werden, da die Rollenzuordnung bis auf wenige Ausnahmen frei wählbar ist und zu jeder nicht zugeordneten Musterrolle Entwurfsteile ergänzt werden. Nach einer Musteranwendung lässt sich die entstandene Musterimplementierung bis zu einem bestimmten Grad anpassen, ohne von der Spezifikation abzuweichen, da Musterspezifikationen durch Weglassen von Details, durch Ausnutzen der Transitivität von Beziehungen und durch Set Fragments viel Spielraum bei der Musterimplementierung bieten.

Anwendungsstellen komplett erfasst

Es wurde außerdem gezeigt, dass das Generieren eines detaillierten Modells einer Anwendungsstelle bei der Musteranwendung möglich ist und sich das Anwendungsstellenmodell nicht auf die Struktur beschränkt, sondern dank kompositionaler Zerlegung von Korrespondenzen auch über das Verhaltensmodell erstreckt.

Einschränkungen

Einschränkungen gibt es bei dem vorgestellten Ansatz beim Verhalten. Ist es komplett oder teilweise modelliert, kann es weder mit einer Spezifikation verknüpft, noch automatisch vervollständigt werden. Für die Visualisierung und Validierung von Anwendungsstellen wurden Konzepte vorgestellt und Argumente für ihre Vorteile geliefert, eine empirische Evaluierung war jedoch aufgrund fehlender Funktionalität im Prototypen bisher nicht möglich. Einige manuell erstellte Visualisierungsbeispiele liefern jedoch zumindest Hinweise auf ihre Stärken gegenüber bisherigen Ansätzen.

Insgesamt erscheint der Ansatz vielversprechend, seine Machbarkeit wurde gezeigt und die aufgestellten Forschungsfragen beantwortet.

10.2 Offene Fragen und Ausblick

Trotz der Ergebnisse bleiben bei der vorliegenden Arbeit noch einige Fragen offen. Diese bieten Potential für weitere Forschungsarbeiten, worauf ich im Folgenden eingehe.

Evaluation durch empirische Studien

Die vorliegende prototypische Implementierung des Ansatzes zeigt bereits die Umsetzbarkeit des vorgestellten Ansatzes. Bisher wurden allerdings relativ kleine Anwendungsbeispiele verwendet. Es wäre spannend, den Nutzen und die Einschränkungen des Verfahrens bei vollem Funktionsumfang in einem größeren Softwareprojekt unter realistischen Bedingungen und mit unbefangenen Anwendern empirisch zu evaluieren sowie bisherigen Entwicklungsverfahren gegenüberzustellen, z.B. in einer Feldstudie oder einem kontrollierten Experiment. Es sollte u.a. empirisch untersucht werden, ob der Nutzen der Visualisierung und der Validierung von Musteranwendungsstellen den Mehraufwand für den Einsatz der Werkzeuge zur Spezifikation und Anwendung von Entwurfsmustern rechtfertigt. Ebenso wäre ein Vergleich der vorgeschlagenen Visualisierungen von Anwendungsstellen im Vergleich zu anderen Ansätzen in Bezug auf Übersichtlichkeit, Informationsgehalt und Skalierbarkeit interessant. Schließlich wäre es nützlich zu untersuchen, ob Entwurfsfehler mit diesem Verfahren tatsächlich reduziert werden können.

Eigenschaften der Auffaltung beweisen

Die Einführung von Set Fragments hat sich als sehr nützlich erwiesen. Sie ermöglichen sehr kompakte Spezifikationen von zahlreichen Musterimplementierungsvarianten. Die Semantik von Set Fragments habe ich als Auffaltungsoperation formal definiert (Def. A.15, S. 289). Ergänzend dazu sollten die genannten Eigenschaften dieser Operation (siehe Anmerkung A.16, S. 289) bewiesen werden. Diese Eigenschaften sollen zur Spezifikation konsistentes Vervielfältigen des in einem Set Fragment enthaltenen Teilgraphen sicherstellen. Möglicherweise könnte ein Beweis auf Graphtransformationen und ihren bereits bewiesenen Eigenschaften basieren und die Auffaltung mit Hilfe sogenannter Multi-Amalgamierung (eine Art Schleifenoperation auf TGGs) [LAST16] ausgedrückt werden.

Konsistenzrelationen

Das bisherige Verfahren sieht für die Validierung von Anwendungsstellen programmierte Prüfungsoperationen für bestimmte Eigenschaften vor. Eine interessante Alternative dazu könnten möglicherweise Konsistenzrelationen [LAS17, Anj14] oder Verfeinerungsregeln [Rie14] sein. Ob und wenn ja wie damit die Konsistenz zwischen Musterspezifikationen und Musterimplementierungen sichergestellt werden kann und ob sich die Konsistenz auch über Verhaltensmodelle erstrecken kann, muss noch untersucht werden.

Modellsynchronisation

Dehnt man dieses Vorgehen auf Modellsynchronisation aus, könnte möglicherweise auch die Synthese einer Musterimplementierung als Modelltransformation zwischen einer Musterspezifikation und einer Musterimplementierung beschrieben werden (Schritte Auffaltung & Übersetzung in Abb. 5.1, S. 101). Dieses Vorgehen hätte den Vorteil, dass eine Musterimplementierung im Fall von Abweichungen von der Spezifikation automatisch „repariert“ werden könnte. Ein Nachteil könnte jedoch sein, dass manuell gemachte Änderungen an der Musterimplementierung bei so einer Reparatur unbeabsichtigt verloren gehen könnten. Es wäre vor allem spannend, zu untersuchen ob und wie solche Ansätze mit überlappenden Musteranwendungen und weiteren Entwurfsänderungen an derselben Stelle in Einklang gebracht werden können.

Verhaltensmodelle vervollständigen & verknüpfen

Bei dem vorgestellten Verfahren ist die Flexibilität bei der Musteranwendung in Verhaltensmodellen derzeit noch eingeschränkt. Das Verhalten zu einer Operation kann nur komplett neu generiert werden, nicht wie bei der Klassenstruktur vervollständigt werden. Auch das nachträgliche Verknüpfen eines modellierten Verhaltens im Entwurf mit einer Aktion in der Musterspezifikation ist zurzeit nicht möglich, weil die Korrespondenzen zu komplex sind, um die Verknüpfung benutzerfreundlich zu gestalten. Den vorgestellten Ansatz um diese Möglichkeiten zu erweitern, wäre eine sinnvolle und spannende Ergänzung der vorliegenden Arbeit.

Sprache für weitere Muster erweitern

Neben den bisher betrachteten Entwurfsmustern gibt es zahlreiche weitere. Es wäre sinnvoll zu untersuchen, welche Erweiterungen der Musterspezifikationssprache und des Anwendungsmechanismus nötig sind, um Entwurfsmuster für z.B. parallele und verteilte Anwendungen [SSRB00, MSM04, OA10], für Enterprise-Anwendungen [Fow02, HW03, Dai11, ACM13] oder Security [Fer13] spezifizieren zu können.

Komposition von Spezifikationen

In einigen Fällen lassen sich Entwurfsmuster als Komposition anderer Entwurfsmuster beschreiben (siehe Abb. 9.14, S. 226). Eine Erweiterung der Musterspezifikationssprache um ein Konzept zur Komposition von Musterspezifikationen könnte eine sinnvolle Ergänzung des Ansatzes sein und komplexe Spezifikationen ggf. vereinfachen.

Anhang A

Musterspezifikationsprache

Zur Vervollständigung der in Kapitel 3 vorgestellten Musterspezifikationssprache präsentiere ich im Folgenden zunächst die abstrakte Syntax (das Meta-Modell) der Sprache (Abschn. A.1), dann den vollen Sprachumfang mit grober Beschreibung der Sprachkonstrukte in konkreter Syntax (Abschn. A.2) und schließlich die Semantik der DAL-Sprachkonstrukte durch ihre Abbildung auf Ecore-Klassen- und Story-Diagramme sowie die Semantik von Set Fragments durch eine mathematische Definition der Auffaltungsoperation (Abschn. A.3).

A.1 Abstrakte Syntax

Die abstrakte Syntax der vorgestellten Musterspezifikationssprache habe ich durch ein Klassenmodell festgelegt.

Den Muster-spezifischen, Domänen-unabhägigen Kern der Sprache bilden die Klassen in Abb. A.1. Hier werden Muster (DesignPattern) und Musterspezifikationen (PatternSpecification), Musterkataloge (DesignPatternCatalog) und Kategorien (Category) definiert.

Eine Musterspezifikation besteht aus mehreren Musterelementen (PatternElement). Die wichtigsten darunter repräsentieren Teile eines Softwareentwurfsmodells (DesignElement) bzw. das Modell selbst (DesignModel) und entsprechen den Sprachkonstrukten der DAL. Jedes DesignElement-Objekt repräsentiert eine Musterrolle. Die Klasse DesignElement ist abstrakt. Die konkreten Unterklassen dazu sind in der Abb. A.2 aufgeführt. Alle anderen Unterklassen von PatternElement aus Abb. A.1 bilden Muster-spezifische Sprachkonstrukte wie Set Fragments (SetFragment) und die Elemente, die in Set Fragments enthalten sein können (SetFragmentElement), Aufgabenbeschreibungen (TaskDescription) sowie Abhängigkeits- bzw. Kopplungsregeln (AccessRule). Die verschiedenen Arten der Abhängigkeiten werden durch eine Hierarchie von Abhängigkeitstypen (AccessType) repräsentiert.

Das abstrakte Syntax der DAL ist in der Abb. A.2 in Form eines Klassendiagramms dargestellt. Alle hier dargestellten Klassen sind Unterklassen von DesignElement aus Abb. A.1. Durch Bilden weiterer Unterklassen von DesignElement kann die DAL erweitert werden.

Die Struktur eines objektorientierten Softwareentwurfs wird durch Typen (Type), Attribute (Attribute), Referenzen (Reference), Operationen (Operation), Parameter (Parameter) und primitive Typen (PrimitiveType und DataType) repräsentiert. Die subTypes- und superTypes-Beziehungen der Klasse Type repräsentiert Verer-

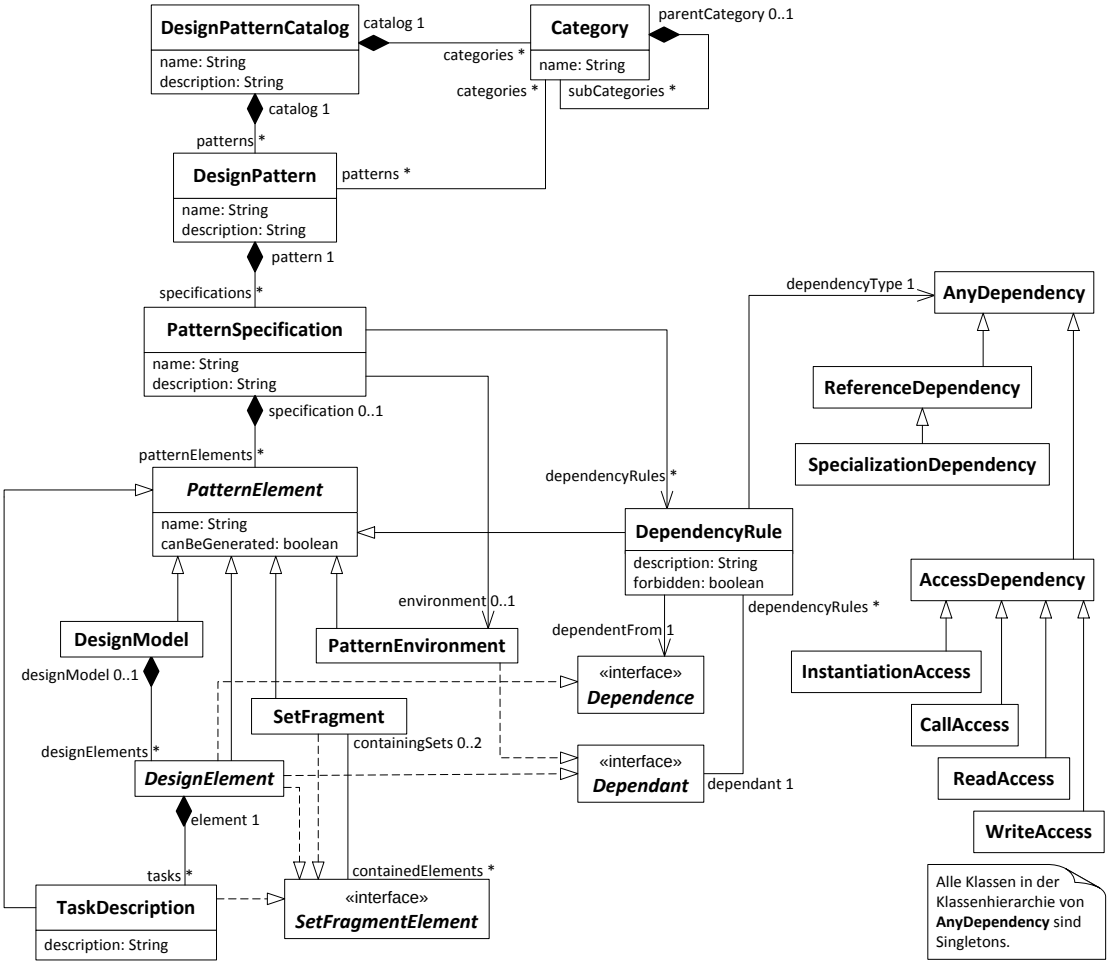


Abbildung A.1: Abstrakte Syntax der Musterspezifikationssprache (PSL) ohne den domänen-spezifischen Anteil (DAL)

bungsbeziehungen zwischen Typen. Die overriding- und overrides-Beziehungen der Klasse Operation repräsentiert Überschreib- bzw. Implementierungsbeziehungen zwischen Operationen. Unter den primitiven Datentypen befinden sich Bool'sche Werte (BOOLEAN), ganzzahlige (INTEGER) und reelle Zahlen (REAL) sowie Zeichenketten (STRING).

Referenzen können mit Hilfe der source- und target-Beziehungen paarweise miteinander verknüpft werden und zusammen eine Assoziation bilden. Die source- und target-Beziehungen drücken in diesem Fall die Leserichtung aus. Voraussetzung dafür ist, dass die Referenzen die gleichen Typen miteinander verbinden und in verschiedene Richtungen zeigen.

Typen und Operationen können als abstrakt (ABSTRACT), konkret (CONCRETE) oder beliebig (ANY) gekennzeichnet werden. Für die Kardinalität von Referenzen kann zwischen 0..1 (SINGLE), * (MULTIPLE) und beliebig (ANY, in konkreter Syntax als ? dargestellt) gewählt werden. Ein Attribut hat üblicherweise die Kardinalität 1..1 (SINGLE), was einem gewöhnlichen Attribut entspricht. Mit der Kardinalität * (MULTIPLE) repräsentiert das Attribut jedoch eine Menge von Attributen gleichen Typs, was einem Array oder einer Liste entspricht.

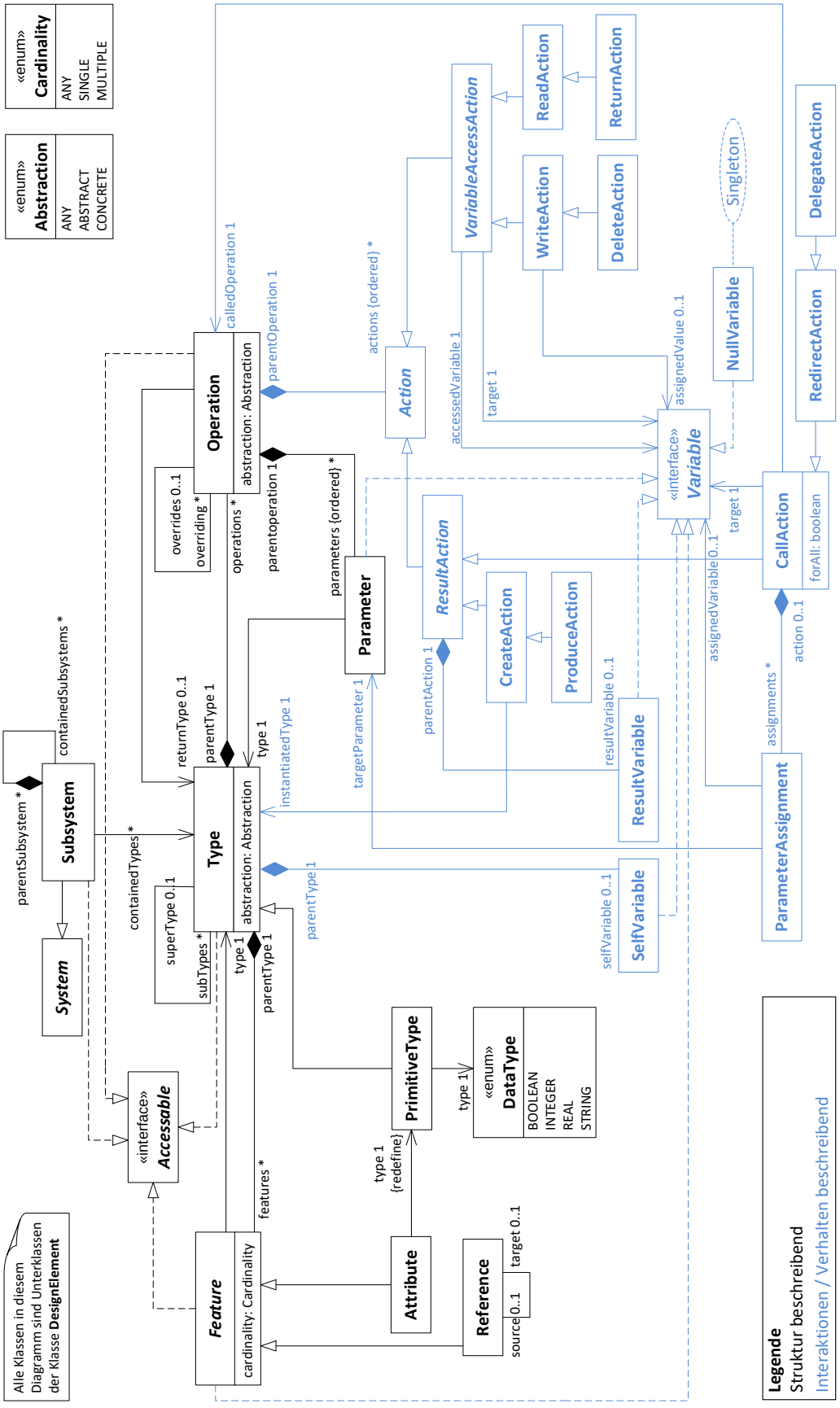


Abbildung A.2: Abstrakte Syntax des Domänen-spezifischen Anteils der Musterspezifikationssprache: der Design Abstraction Language

Zusätzlich zu den objektorientierten Konzepten steht das allgemeinere Konzept der Subsysteme (Subsystem) zur Verfügung. Diese können mit Hilfe der containedSubsystems-Beziehung ineinander geschachtelt werden und auf beliebig viele Typen verweisen, um einen Teil eines Softwaresystems zu beschreiben.

Alle Sprachkonstrukte zur Modellierung von Verhalten sind in der Abb. A.2 blau hervorgehoben. Das Verhalten wird durch verschiedene Aktionen (Action) modelliert. Jede Aktion wird einer Operation zugeordnet. Die Reihenfolge ihrer Ausführung wird durch die geordnete actions-Assoziation bestimmt.

Aktionen verweisen unter anderem auf Variablen (Variable), zu denen neben den Parametern einer Operation sowie den Attributen und Referenzen eines Typs auch der Selbstverweis (selfVariable), die Ergebnisvariable einer Aktion (ResultVariable) und der Wert null (modelliert als Nullvariable) zählen.

Eine der wichtigsten Aktionen ist die call-Aktion. Sie repräsentiert einen Operationenaufruf. Dazu verweist sie auf die aufzurufende Operation (calledOperation-Beziehung), auf eine das Zielobjekt repräsentierende Variable (target-Beziehung) und kann eine Ergebnisvariable (ResultVariable) deklarieren, um das Ergebnis des Aufrufs für folgende Aktionen bereitzustellen. Die Argumente eines Operationenaufrufs werden in Form von Zuweisungen (ParameterAssignment) modelliert, welche den Parametern der aufgerufenen Operation Variablen zuordnen. Wird das Zielojekt durch eine Referenz mit Kardinalität * bestimmt, wird durch das Attribut forAll zwischen einem Allquantor ∀ und einem Existenzquantor ∃ entschieden. Es wird also entweder beschrieben, dass auf allen referenzierten Objekten der Aufruf erfolgen soll oder nur auf mindestens einem der Objekte.

Eine create- oder produce-Aktion verweist auf den zu instanziierenden Typen (instantiatedType-Beziehung) und kann das erzeugte Objekt ebenfalls in einer Ergebnisvariable persistieren.

Die return-Aktion verweist auf die Variable, deren Wert zurückgegeben werden soll (accessedVariable-Beziehung). Analog dazu verweisen auch die anderen Lese- und Schreibaktionen auf die verwendete Variable und geben ggf. ein Objekt an, auf dessen Attribut oder Referenz zugegriffen wird (target-Beziehung).

A.2 Konkrete Syntax und Sprachumfang

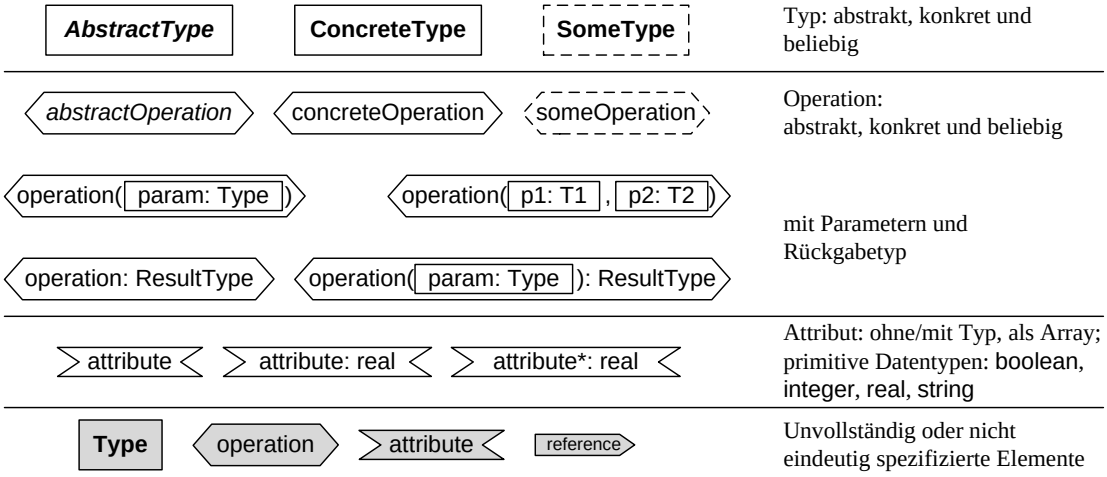
In den Abschnitten 3.4 und 3.5 wird nur ein Teil der Musterspezifikationssprache vorgestellt. Im Folgenden stelle ich den vollen Umfang der Sprache vor.

A.2.1 DAL: Klassenstrukturen

Zur Beschreibung der strukturellen Anteile von Entwurfsmustern für objektorientierte Softwaresysteme habe ich eine Auswahl von Konzepten aus objektorientierten Sprachen gewählt, die insbesondere in den Mustern von Gamma et al. [GHJV95] vorkommen. Dazu zählen vor allem Typen, Vererbung, Operationen und Attribute sowie Assoziationen bzw. Referenzen.

Die angebotenen Sprachkonstrukte sind in den Tabellen A.1 und A.2 zusammengefasst und bilden einen Großteil der DAL. Mit Hilfe dieser Sprachkonstrukte lassen sich Klassenstrukturen in objektorientierten Softwaresystemen auf hohem

Tabelle A.1: Typen, Operationen und Attribute



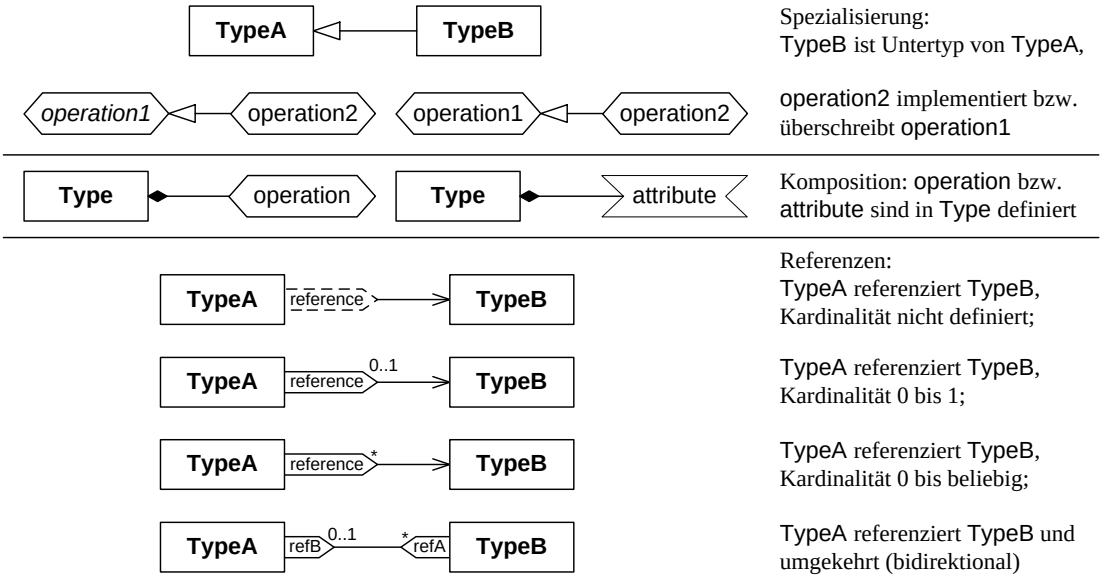
Abstraktionslevel beschreiben. Die hohe Abstraktion dient dazu, eine so beschriebene Entwurfsstruktur in möglichst viele objektorientierte Sprachen überführen zu können und Entwurfsmuster möglichst kompakt, implementierungsunabhängig und unabhängig von einer Modellierungssprache wie der UML oder EMOF / Ecore zu beschreiben.

Es wird von vielen zur Beschreibung von Entwurfsmustern irrelevanten Details wie sie in objektorientierten Modellierungs- und Programmiersprachen vorkommen abstrahiert. Zum Beispiel werden Assoziationseigenschaften wie *subset*, *redefine* oder *ordered*, die Navigierbarkeit von Assoziationsenden und Profile bzw. Annotationen aus Sprachen wie der UML oder Ecore in der DAL nicht mitmodelliert. Ebenso werden aus objektorientierten Programmiersprachen wie Java, Scala, Kotlin, C++, C#, Eiffel oder Smalltalk bekannte Details wie statische bzw. Klassenoperationen und -attribute, Zeiger, Sichtbarkeiten (*public*, *protected*, *private*, *friend*), Konstanten, generische Typen bzw. Templates und Ausnahmen (Exceptions) weggelassen.

Die Auswahl der Sprachkonstrukte orientiert sich insbesondere an den zur Beschreibung von Klassenstrukturen verwendeten Konzepten in Musterbeschreibungen von Gamma et al. [GHJV95]. So lassen sich wie in der Abb. A.1 dargestellt Typen (abstrakte, konkrete und abstrakte oder konkrete Klassen), Attribute sowie Operationen mit ihren Parametern und dem Rückgabetyp beschreiben. Für Attribute können zwecks Sprachunabhängigkeit nur die gebräuchlichen und vordefinierten, primitiven Datentypen boolean, integer, real und string verwendet werden. Außerdem lassen sich wie in der Abb. A.2 dargestellt Beziehungen wie Vererbung, Komposition (Zugehörigkeit von Operationen und Attributen zu einem Typen) und Referenzen (uni- und bidirektional) samt Kardinalität beschreiben.

Aufgrund der zuvor in Abschnitt 3.3.2 genannten Mehrdeutigkeiten bei der Spezifikation von sich wiederholenden Klassenstrukturen mit Hilfe von Set Fragments stelle ich Operationen und Attribute anders als in UML-Klassendiagrammen in eigenen Knoten und nicht zusammen mit der zugehörigen Klasse im selben Rechteck dar. Um dennoch die Zugehörigkeit einer Operation oder eines Attributs zu einem Typ visuell ausdrücken zu können, habe ich eine Kompositionskante ein-

Tabelle A.2: Beziehungen in Klassenstrukturen



geführt (siehe Tabelle A.2). Um ebenfalls zuvor genannte Mehrdeutigkeiten im Zusammenhang mit Assoziationen zu vermeiden, stelle ich auch die Rollen einer Referenz wie in der Tabelle A.2 dargestellt explizit in eigenen Knoten dar, die auf Seite der referenzierenden Klasse platziert werden und nicht wie bei der UML auf Seite der referenzierten Klasse.

Repräsentation von mehr als nur einer Klassenstruktur

Mit den bisher genannten Sprachkonstrukten lässt sich zum Beispiel die Klassenstruktur des Observer-Musters aus Abb. 1.2 (S. 5) wie in der Abb. A.3 dargestellt beschreiben. Es können die abstrakten Klassen (dargestellt durch den kursiv dargestellten Namen) Subject und Observer, die observers-Referenz (unidirektionale Assoziation) dazwischen, deren Operationen notify und update sowie je eine Unterklasse ConcreteSubject und ConcreteObserver mit einer subject-Referenz und einer concreteUpdate-Operation spezifiziert werden. Die Zugriffsoperationen und Variablen zur Implementierung der Referenzen wie sie in der Abb. 1.2 (S. 5) vorkommen wurden hier weggelassen, weil sie bereits durch die Referenzen implizit mitbeschrieben werden.

Bei der Beschreibung von Entwurfsmustern wird eine Klassenstruktur nur als Beispiel für eine mögliche Implementierung des Musters verwendet. Darum werden typischerweise einige Details wie Parameter und Rückgabetyp von Operationen, Attributtypen und ob es sich um unidirektionale oder bidirektionale Assoziationen handelt vernachlässigt (wie auch bei dem Observer-Muster in Abschnitt 3.3). Bei der Anwendung eines Musters benennt man die Klassen, Operationen, Attribute und Assoziationen ggf. um, ergänzt die fehlenden Details und definiert z.B. für den Anwendungsfall geeignete Methodensignaturen. Auch wenn sich damit die Klassenstruktur geringfügig ändert, widerspricht das nicht der im Muster beschriebenen Lösung.

Bei meiner Musterspezifikationssprache sehe ich solche Anpassungen bei der

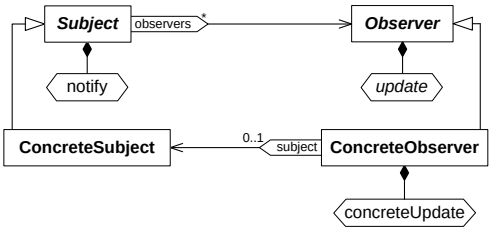


Abbildung A.3: Klassenstruktur des Observer-Musters in der DAL

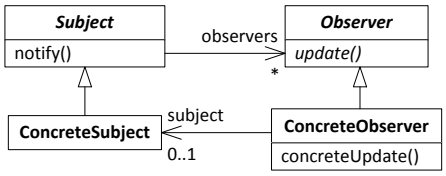


Abbildung A.4: a) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung

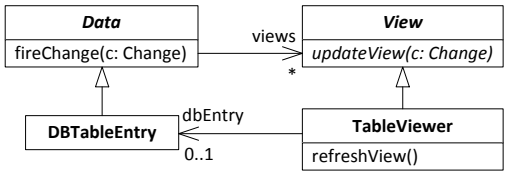


Abbildung A.5: b) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung

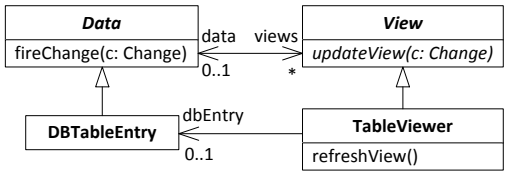


Abbildung A.6: c) Zur Spezifikation in Abb. A.3 passende Klassenstruktur in UML-Darstellung

Musteranwendung explizit vor und erlaube dazu auch teilweise unvollständige Beschreibungen von Klassenstrukturen in Musterspezifikationen. Zum Beispiel kann die Signatur einer Operation wie in Abb. A.3 weggelassen werden, wodurch ausgedrückt wird, dass es eine solche Operation geben muss, ihre Signatur für die Implementierung des Musters jedoch irrelevant ist. Damit passen die Klassenstrukturen aus den Abb. A.4, A.5 und A.6 zu der Spezifikation in Abb. A.3, auch wenn sich die Signatur der update und concreteUpdate-Operationen geändert hat oder die unidirektionale Assoziation zwischen den Klassen Subject und Observer zu einer bidirektionalen Assoziation geändert wurde. Analog dazu kann auch der Typ eines Attributs oder Parameters weggelassen werden. Werden Parameter einer Operation spezifiziert, so muss es diese auch bei der Implementierung des Musters geben, es können aber auch beliebig viele zusätzliche Parameter ergänzt werden.

Aufgrund der nicht angegebenen Signatur bei den Operationen update und concreteUpdate in Abb. A.3 ist ohne zusätzliche Angaben (auch bei gleichen Namen der Operationen) nicht ersichtlich, ob die Operation concreteUpdate die Operation update überschreibt, implementiert, überlädt oder völlig unabhängig von ihr ist. Damit man, wie bei dem Observer-Muster vorgesehen, spezifizieren kann, dass die update-Operation implementiert oder überschrieben werden soll, verwende ich auch bei Operationen eine Spezialisierungskante (siehe Tabelle A.1). Durch die zusätzliche Spezialisierungskante in Abb. A.7 zwischen den Operationen update und concreteUpdate wird klar, dass es sich um eine Implementierungsbeziehung handelt. Dadurch repräsentiert die Operation concreteUpdate aus Abb. A.7 nicht eine beliebige Operation mit evtl. anderer Signatur als bei der update-Operation der Oberklasse (wie in Abb. A.4 oder A.5), sondern unabhängig von der bei einer

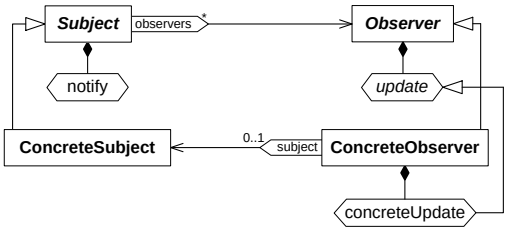


Abbildung A.7: Korrigierte Klassenstruktur des Observer-Musters in der DAL (vgl. Abb. A.3)

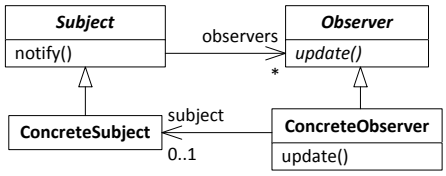


Abbildung A.8: a) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung

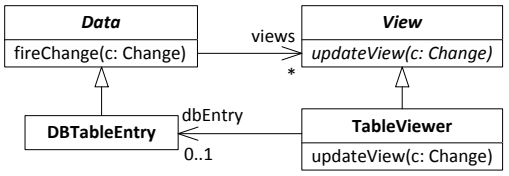


Abbildung A.9: b) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung

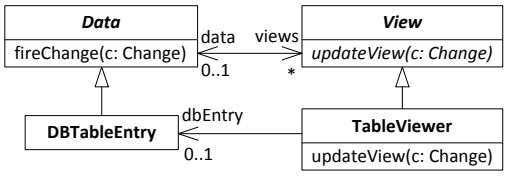


Abbildung A.10: c) Zur Spezifikation in Abb. A.7 passende Klassenstruktur in UML-Darstellung

Musteranwendung gewählten Sigantur auf jeden Fall eine Operation mit derselben Signatur wie bei der update-Operation der Oberklasse Observer. Dadurch passen die Klassenstrukturen aus den Abb. A.4, A.5 und A.6 nicht mehr zu der Spezifikation in Abb. A.7. Die Klassenstrukturen in den Abb. A.8, A.9 und A.10 entsprechen aber dieser Spezifikation.

In einigen Mustern ist es irrelevant, ob eine darin genannte Klasse abstrakt oder konkret ist. Zum Beispiel ist es bei der Context-Klasse im Strategy-Muster [GHJV95] egal, ob sie abstrakt oder konkret ist. Wichtig ist nur, dass sie eine Referenz zu einer Strategie hält und die Strategy-Schnittstelle nutzt. Darum erlaube ich bei der Musterspezifikation, bei Typen und Operationen offen zu lassen, ob sie abstrakt oder konkret sein sollen, was in diesem Fall durch gestrichelte Kanten dargestellt wird (siehe Tabelle A.1). Analog dazu kann auch die Kardinalität bei Referenzen weggelassen werden.

Durch die unvollständige Beschreibung von Klassenstrukturen kann aus einer Musterspezifikation nicht immer ein vollständiges Entwurfsmodell automatisch abgeleitet werden. Bei der Musteranwendung sind hier manuelle Schritte durch den Entwickler nötig. Ohne diese unvollständig beschriebenen Entwurfsteile wie z.B. die ohne Signatur beschriebenen Operationen update und notify des Observer-Musters wären die Musterspezifikationen jedoch unvollständig und das Muster dadurch kaum erkennbar. Außerdem könnten sie nicht zur Prüfung der Anwendungsstellen auf Konsistenz mit der Musterspezifikation genutzt werden, was eines der Ziele der Musterspezifikationen ist (siehe Abschnitt 3.1.1). Damit schon bei der Spezifikation eines Musters klar ist, welche Teile der Musterspezifikation nicht für eine automatische Anwendung des Musters ausreichen und somit manuelle

Schritte erfordern, hebe ich diese visuell hervor, indem ich die Elemente grau hinterlege wie in der Tabelle A.1 unten dargestellt. Welche Teile einer Musterspezifikation nicht für eine automatische Anwendung des Musters ausreichen, kann in den meisten Fällen automatisch bestimmt, aber auch explizit angegeben werden.

Transitive Beziehungen

Eine Musterspezifikation gibt im Wesentlichen Bedingungen an, die bei Musteranwendung und späteren Entwurfsänderungen eingehalten werden müssen, damit ein Muster korrekt angewendet wird und die Implementierung des Musters korrekt bleibt. Beziehungen wie Vererbung und Komposition in Musterspezifikationen gehören auch zu solchen Bedingungen.

Betrachtet man die Spezifikation des Observer-Musters (siehe Abb. 3.7 auf S. 52), so ist hier z.B. spezifiziert, dass ConcreteObserver ein Untertyp von Observer sein soll. Bei Anwendung eines Musters ist es häufig sinnvoll oder gar nötig, die in der Musterspezifikation angegebene Struktur zu erweitern und z.B. die direkte Vererbungsbeziehung in der Spezifikation des Observer-Musters durch eine indirekte in der Musterimplementierung zu ersetzen. Ausschließlich direkte Vererbungsbeziehungen zu verlangen, würde Entwickler unnötig einschränken.

Bei dem Beispiel in Abb. 3.11 auf S. 55 wurde zwischen der Observer-Klasse Listener und den ConcreteObserver-Klassen PlaceUpdater und TransitionUpdater eine weitere Klasse ViewUpdater ergänzt, um die gemeinsame Operation propertyChange nur ein Mal zu implementieren und somit Redundanz zu vermeiden (vgl. Abb. 3.10 auf S. 55). Trotz dieser Änderung und der nun von der Musterspezifikation abweichenden Struktur in der Musterimplementierung ist das Muster weiterhin korrekt implementiert. Weil die Vererbungsbeziehung transitiv ist[1], ist jede ConcreteObserver-Klasse in diesem Fall immer noch Untertyp der Observer-Klasse.

Um diese Art von strukturellen Abweichungen bei der Implementierung eines Musters zu ermöglichen, betrachte ich z.B. eine Vererbungsbeziehung in einer Musterspezifikation grundsätzlich als eine Forderung, sie bei Musteranwendung durch eine direkte *oder indirekte* Vererbungsbeziehung zu realisieren. Für das Implementieren oder Überschreiben von Operationen[2] gilt das Gleiche wie für Vererbungsbeziehungen. Eine Operation soll direkt oder indirekt eine andere Operation implementieren bzw. überschreiben. Folglich kann sich die implementierte bzw. überschriebene Operation in einer direkten oder indirekten Oberklasse der implementierenden bzw. überschreibenden Operation befinden.

Ähnlich dazu fasse ich auch Kompositionsbeziehungen in einer Musterspezifikation auf und erlaube u.a. indirekte Komposition in der Musterimplementierung. Wird z.B. in einer Observer-Musterspezifikation angegeben, dass der Typ ConcreteObserver über eine Operation concreteUpdate verfügen soll, so kann diese Operation auch in einer der Oberklassen einer ConcreteObserver-Klasse implementiert sein. So ist bei der Implementierung des Observer-Musters in der Abb. 3.11 auf S. 55 die Implementierung der Operation propertyChange von den ConcreteObserver-Klassen PlaceUpdater und TransitionUpdater in ihre Oberklasse ViewUpdater

[1] Eine Relation $r \subseteq \Sigma \times \Sigma$ ist transitiv, wenn für alle $a, b, c \in \Sigma$ gilt: $r(a,b) \wedge r(b,c) \Rightarrow r(a,c)$.

[2] Eine abstrakte Operation (ohne Verhalten) wird von einer anderen implementiert, während eine konkrete Operation (mit Verhalten) überschrieben wird.

ausgelagert worden, ohne damit die Musterspezifikation (siehe Abb. 3.7 auf S. 52) zu verletzen. Analog dazu kann auch ein Attribut oder eine Referenz in einer der Oberklassen deklariert sein.

Zusammenfassend betrachte ich folgende Arten von Beziehungen als transitiv:

- ein Typ ist Untertyp eines anderen Typs (Vererbung)
- eine Operation implementiert bzw. überschreibt eine andere Operation
- ein Typ stellt eine Operation, ein Attribut oder eine Referenz bereit (Komposition)

In meiner Musterspezifikationssprache gehe ich bei transitiven Beziehungen grundsätzlich davon aus, dass jede direkte Beziehung einer Musterspezifikation durch eine indirekte Beziehung in einer Musterimplementierung ersetzt werden kann. Bei den bisher spezifizierten Mustern (siehe Anhang B) waren nie ausschließlich direkte Beziehungen gefordert. Darum wird in Musterspezifikationen nicht zwischen direkten und indirekten Beziehungen unterschieden.

A.2.2 DAL: Interaktionen

Eine Aktion in einer Musterspezifikation wird durch ein Rechteck mit abgerundeten Ecken und einem Label repräsentiert (siehe Tabelle A.3 oben). Die bisher verfügbaren Aktionen sind im oberen Teil der Tabelle A.3 aufgelistet und sind in Abschnitt 3.4.2 (S. 64) beschrieben. Ihre Semantik ist durch Abbildung auf Story-Diagramme in Anhang A.3.2 definiert.

Parameter von Aktionsspezifikationen

Zur genauen Beschreibung des von einer Operation durchzuführenden Schritts werden neben der Art des Schritts (z.B. delegate oder write) auch Details (Parameter einer Aktion) wie die aufgerufene Operation oder die geschriebene Variable spezifiziert. Diese Details hängen von der Art der Aktion ab.

Komposition

Bei jeder Aktion wird angegeben, von welcher Operation sie durchgeführt bzw. realisiert wird. Dargestellt wird das durch eine gestrichelte Kante mit einer gefüllten Raute, ähnlich zu einer Kompositionsbeziehung (siehe Tabelle A.3).

Kontrollfluss

Die Hintereinanderausführung wird durch eine durchgezogene Kontrollflusskante zwischen zwei Aktionen mit Pfeilspitze in Richtung der folgenden Aktion dargestellt. Bei der folgenden Aktion entfällt dann die Realisierungskante zur Operation, weil implizit klar ist, dass diese Aktion von der gleichen Operation durchgeführt wird, wie die vorhergehende Aktion. Auf diese Weise wird aus Gründen der Übersichtlichkeit die Anzahl der Kanten in einer Musterspezifikation reduziert.

Bei den bisher spezifizierten Entwurfsmustern war die Möglichkeit, die Hintereinanderausführung von Aktionen zu beschreiben, ausreichend (siehe Musterspezifikationen im Anhang B). Komplexere Kontrollflussstrukturen wie Fallunterscheidungen und Schleifen waren bisher nicht nötig, könnten aber bei Bedarf in die Musterspezifikationssprache aufgenommen werden.

instanziierter Typ

Bei create- und produce-Aktionen wird durch eine gestrichelte Kante mit einem Plus und einer offenen Pfeilspitze in Richtung eines Typs spezifiziert, welcher Typ instanziiert wird (siehe Tabelle A.3).

Tabelle A.3: Interaktionen

call create read write redirect delegate produce return delete	Aktion
write	Unvollständig oder nicht eindeutig spezifizierte Aktion
operation — create	Realisierung von Verhalten: operation führt create-Aktion aus
write → return	Ausführungsreihenfolge: return folgt nach write
create - - ▷ Type	Instanziierung: hier wird ein Type-Objekt erzeugt
call - - - ▶ operation; write - - - ▶ reference (0..1)	Zugriff auf Operation bzw. Variable, d.h. Methodenaufruf oder Lese-/Schreibzugriff
call ∃ reference (*); call ∀ reference (*); write reference (0..1)	Zielobjekte: Angabe von Objekten, auf deren Methoden oder Variablen zugegriffen wird; eins von vielen, alle oder eins
call self; write self	self-Schlüsselwort bei Zugriff auf sich selbst
call result	Aktion mit Ergebnisvariable
call par := value; call par := null; write var := null	Aktion mit Parameterübergabe oder Zuweisung

Bei call- und delegate-Aktionen wird durch eine gestrichelte Kante mit gefüllter Pfeilspitze angegeben, welche Operation aufgerufen wird, also auf welche Operation zugegriffen wird. Analog dazu wird bei Lese- und Schreibzugriffsaktionen mit der gleichen Kantenart angegeben, welche Variable gelesen oder geschrieben wird, also auf welche Variable zugegriffen wird. Die Variable kann dabei eine Referenz, ein Attribut, ein Parameter oder das Ergebnis einer anderen Aktion sein.

aufgerufene Operation

Um Interaktionen zwischen Objekten, z.B. Operationenaufrufe und Variablenzugriffe, vollständig zu beschreiben, wird zusätzlich zur aufgerufenen Operation oder der gelesenen bzw. geschriebenen Variable auch spezifiziert, mit welchem Objekt interagiert wird, also auf welchem Objekt eine Operation aufgerufen oder die Variable welchen Objekts gelesen oder geschrieben wird. So ein Objekt nenne ich *Zielobjekt*. Zur Entwurfszeit lässt sich ein Zielobjekt (oder eine Menge von Zielobjekten) nur durch Angabe einer Variable spezifizieren, die zur Laufzeit auf ein Objekt (oder mehrere) verweisen wird. In meiner Musterspezifikationssprache spezifiziere ich ein oder mehrere Zielobjekte einer Aktion durch einen gepunkteten Pfeil auf eine Variable, z.B. auf eine Referenz oder ein Attribut (siehe Tabelle A.3). Die angegebene Variable muss sich im Sichtbarkeitsbereich der ausführenden Operation befinden, d.h. eine Referenz oder ein Attribut muss in der gleichen Klasse definiert sein wie die ausführende Operation oder in einer der Oberklassen. Wenn

Zielobjekt

mit dem Zielobjekt das die Aktion ausführende Objekt selbst gemeint ist, wird auf die Zielobjektkante verzichtet und stattdessen das Schlüsselwort self in der Aktion angegeben.

∀ und ∃

Verweist eine Zielobjektkante auf eine Variable mit Multiplizität *, repräsentiert die Variable also mehr als ein Objekt, so wird zusätzlich ein Quantor angegeben, der bestimmt, ob die Aktion auf allen Objekten ausgeführt werden soll (Allquantor ∀) oder nur auf einem noch durch einen Entwickler zu bestimmenden Objekt (Existenzquantor ∃). Im letzteren Fall ist die Aktion unvollständig spezifiziert und wird deswegen grau hinterlegt.

Ergebnisvariable

Damit das Ergebnis einer Aktion von einer folgenden Aktion weiterverwendet werden kann, lässt sich für die Aktionen call, delegate, create und produce eine Ergebnisvariable spezifizieren. Diese kann von einer nachfolgenden Aktion z.B. als Zielobjekt oder Argument eines Operationsaufrufs genutzt werden. Dargestellt wird die Variable in einem Kasten unterhalb der Aktion, wo der Name der Ergebnisvariable eingetragen wird (siehe Tabelle A.3).

Aufrufargumente

Bei Aktionen, die eine Operation aufrufen, werden Argumente explizit spezifiziert. Die Parameterübergabe wird dabei syntaktisch wie eine Zuweisung der Form parameter := variable angegeben. parameter ist dann der eindeutige Name eines der Parameter der aufgerufenen Operation, variable ist der Name einer Variable, die innerhalb der aufrufenden Operation sichtbar ist, und repräsentiert ein Argument des Aufrufs. Mehrere Parameterübergaben werden durch Kommata getrennt aufgelistet. Ein Sonderfall ist die Übergabe des leeren Werts, der durch das Schlüsselwort null repräsentiert wird.

Repräsentation von mehr als nur einem Verhalten

Wie die Klassenstruktur kann auch das Verhalten bei der Musterspezifikation unvollständig oder nicht eindeutig angegeben werden, um das spezifizierte Verhalten allgemein genung zu halten und bei Musteranwendung an die konkrete Situation anpassen zu können.

partielle Spezifikation und Konsistenz

Aktionenfolgen spezifizieren Schritte, die von einer Operation ausgeführt werden sollen. Sie stellen jedoch nicht zwingend das gesamte Verhalten einer Operation dar, sondern geben eine Art Minimum an gefordertem Verhalten an. So haben Entwickler nach Anwendung eines Musters den Freiraum, zusätzliches Verhalten wie Änderungsbenachrichtigungen, Persistierung, Logging und mehr zu ergänzen. In diesem Fall muss ein Entwickler allerdings selbst sicherstellen, dass die Intention des Entwurfsmusters erhalten bleibt.

Ebenso können auch die Details einzelner Aktionen offen gelassen werden. Zum Beispiel kann ein Schreibzugriff auf eine Variable spezifiziert werden, ohne anzugeben, welchen Wert die Variable erhält. Auch die Argumente eines Operationenaufrufs können weggelassen werden oder nur für einige der Parameter angegeben werden. In diesem Fall werden solche Aktionen analog zu anderen unvollständig oder nicht eindeutig spezifizierten Elementen grau hinterlegt[3] (vgl. Tabellen A.1 auf S. 253 und A.3).

[3] Ob etwas unvollständig spezifiziert ist, ist eine Einschätzung der Person, die ein Muster spezifiziert.

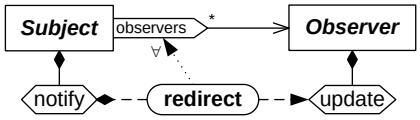


Abbildung A.11: Mit einer Aktion spezifizierte Umleitung im Observer-Muster

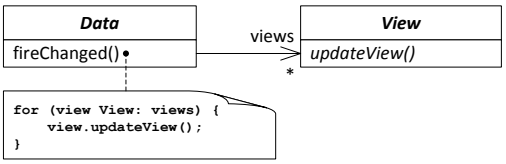


Abbildung A.12: a) Zur redirect-Aktion in Abb. A.11 passende Java-Implementierung

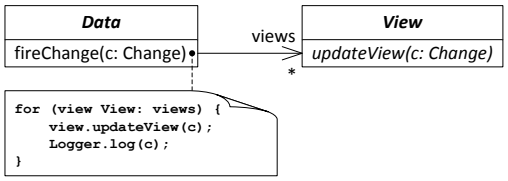


Abbildung A.13: b) Zur redirect-Aktion in Abb. A.11 passende Java-Implementierung

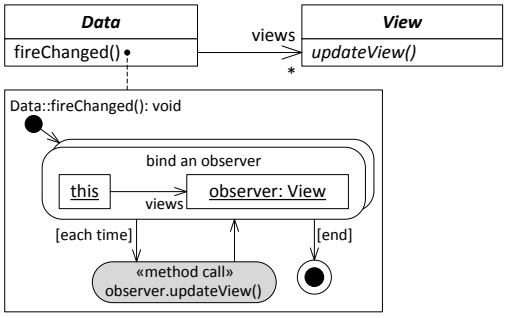


Abbildung A.14: c) Zur redirect-Aktion in Abb. A.11 passendes Story-Diagramm

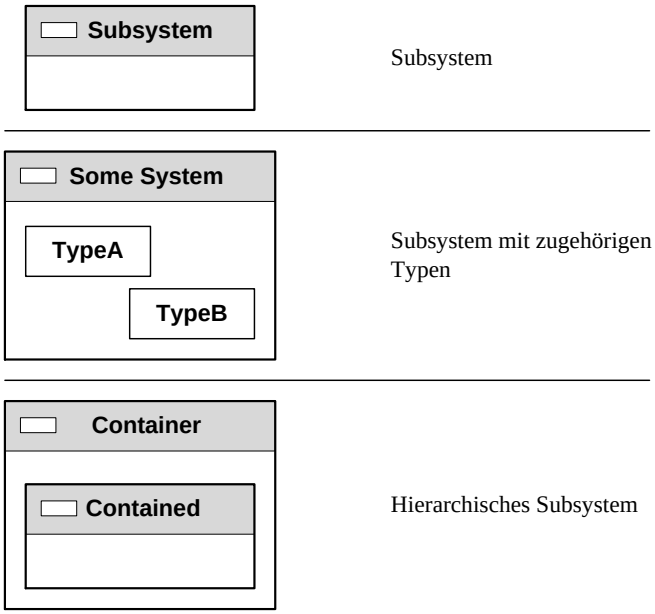
Ein Beispiel für die Verwendung einer Aktion in einer Musterspezifikation ist in der Abb. A.11 dargestellt. Hier ist ein Ausschnitt der Spezifikation des Observer-Musters abgebildet, wo eine redirect-Aktion zum Einsatz kommt. In dem Beispiel ist spezifiziert, dass ein Subject-Objekt bei Aufruf der notify-Operation den Aufruf an die update-Operation aller über die Referenz observers verknüpften Observer-Objekte umleitet. Die gepunktete Kante mit seinem Allquantor gibt an, dass alle verknüpften Observer-Objekte Zielobjekte des Aufrufs sind, während die gestrichelte mit Pfeilspitze auf die aufgerufene Operation zeigt. Beispiel

Die redirect-Aktion ist eine spezielle call-Aktion, bei der analog zur delegate-Aktion alle ggf. vorhandenen Argumente der aufrufenden Operation an die aufgerufene Operation als Argumente übergeben werden. Im Gegensatz zur delegate-Aktion wird hier jedoch nicht verlangt, ein ggf. vorhandenes Ergebnis des Aufrufs als Ergebnis der aufrufenden Operation zurückzugeben. Da in der Musterspezifikation keine Parameter angegeben sind, hängt es von der Musterimplementierung ab, ob evtl. vorhandene Argumente eines notify-Aufrufs an die update-Operation übergeben werden müssen.

Beispiele für gültige Implementierungen des durch die redirect-Aktion spezifizierten Verhaltens sind in den Abb. A.12, A.13 und A.14 dargestellt.

Während in Abb. A.12 eine Implementierung ohne Parameter dargestellt wird, enthalten die Operationen fireChange und updateView in Abb. A.13 einen Parameter. Im letzteren Fall wird das übergebene Change-Objekt, das als Argument an die fireChange-Operation übergeben wird, auch als Argument beim Aufruf der updateView-Operation verwendet. Wäre das Argument nicht weitergereicht werden, würde die Implementierung nicht der Musterspezifikation entsprechen.

Tabelle A.4: Subsysteme



Man beachte, dass die Anweisung `Logger.log(c);` bei der Implementierung in Abb. A.13 über das im Muster spezifizierte Verhalten hinausgeht, jedoch nicht der Musterspezifikation widerspricht.

Die Abb. A.14 stellt eine zur Abb. A.12 äquivalente Implementierung des spezifizierten Verhaltens als Story-Diagramm dar.

A.2.3 DAL: Subsysteme

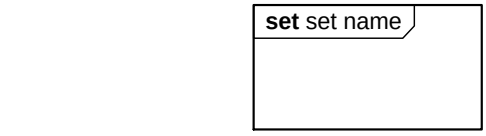
Ein Subsystem beschreibt eine Gruppe von Typen. Subsysteme sind hierarchisch und können ineinander geschachtelt werden. Dargestellt wird ein Subsystem als Rechteck mit einem grauen Label. Die darin enthaltenen Subsysteme und Typen werden innerhalb des Rechtecks dargestellt (siehe Tabelle A.4).

Die Bedeutung von Subsystemen wird in Abschnitt 3.4.3 beschrieben. Bei der Synthese einer Musterspezifikation werden sie nicht explizit übersetzt, weil sie in Ecore-Modellen keine Entsprechung haben. Sie repräsentieren eine Gruppe von Ecore-Klassen, die logisch zusammengehören, aber nicht zwingend z.B. in einem gemeinsamen Paket oder Ordner liegen müssen. Bei der Übersetzung einer Musterspezifikation mit Subsystemen wird nur die Zuordnung der Ecore-Klassen zum zugehörigen Subsystem der Musterspezifikation festgehalten.

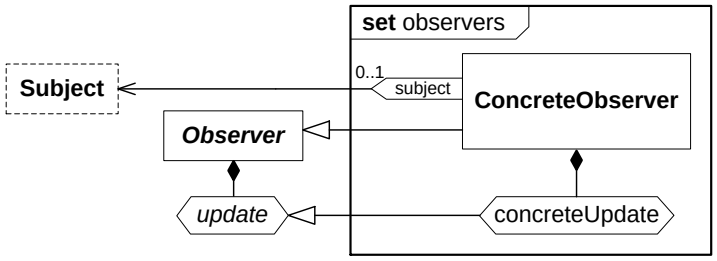
A.2.4 PSL: Set Fragments

Set Fragments markieren Teile einer Musterspezifikation, welche bei der Übertragung in den Entwurf mindestens ein Mal vorkommen müssen, aber beliebig häufig *als Ganzes* vervielfältigt werden können. Set Fragments können sich überschneiden und ineinander geschachtelt werden. Jedoch wird zwecks Reduktion der Komplexität von Musterspezifikationen nur erlaubt, dass sich maximal zwei Set Fragments überschneiden, ohne, dass eines davon komplett in dem anderen enthalten ist.

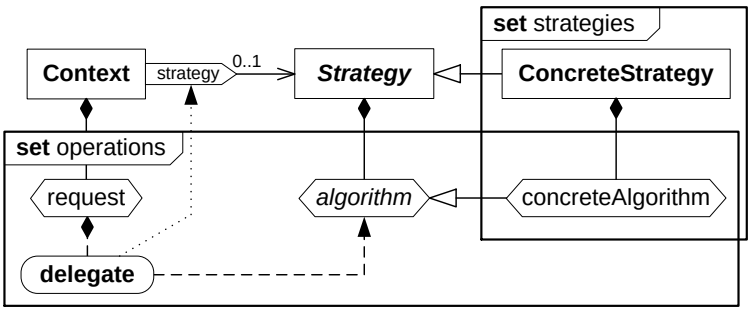
Tabelle A.5: Set Fragments – Wiederholbare Entwurfsstrukturen



Set Fragment



Set Fragment mit enthaltener Struktur:
Die Klassenstruktur bestehend aus einer ConcreteObserver-Klasse mit einer concreteUpdate-Operation und einer Referenz auf eine Subject-Klasse kann beliebig häufig vorkommen.



Überschneidende Set Fragments:
Es gibt beliebig viele ConcreteStrategy-Klassen und beliebig viele algorithm-Operationen, aber zu jeder algorithm-Operation gibt es auch eine entsprechende request-Operation in der Context-Klasse und eine implementierende concreteAlgorithm-Operation in jeder ConcreteStrategy-Klasse

Die Notation von Set Fragments wird in der Tabelle A.5 zusammengefasst. Die Bedeutung von Set Fragments wird in Abschnitt 3.5.2 erläutert, während ihre Semantik mathematisch formal in Anhang A.3.3 definiert wird.

A.2.5 PSL: Abhängigkeits- & Kopplungsregeln

Die PSL bietet Sprachkonstrukte zur Beschreibung gewollter und ungewollter Abhängigkeiten bzw. Kopplung (beide Begriffe verwende ich in diesem Zusammenhang synonym). Die Notation ist in der Tabelle A.6 zusammengefasst. Die Bedeutung der verschiedenen Kopplungskanten wird in Abschnitt 3.5.3 erläutert.

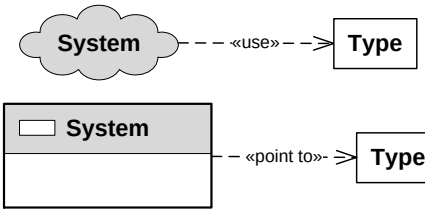
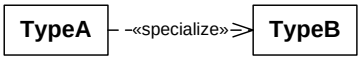
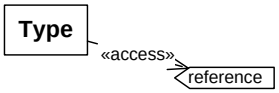
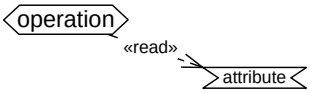
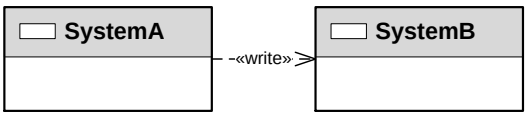
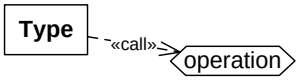
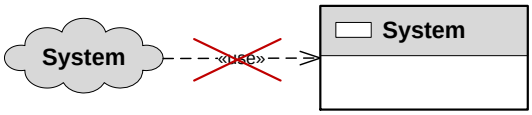
gültige Kombinationen

Die gültigen Kombinationen einer Kopplungskante mit anderen Sprachkonstrukten als Quelle und Ziel sind der Tabelle A.8 zu entnehmen. Im linken Teil der Tabelle sind die Abhängigkeiten aufgeführt, welche direkt zwischen Typen, Operationen, Attributen und Referenzen existieren können.

indirekte Abhängigkeiten

Damit z.B. Lesezugriffe u.a. zwischen Subsystemen oder Typen beschrieben werden können, ohne im Detail die zugreifenden Operationen und die Referenzen und Attribute, auf die zugegriffen wird, zu spezifizieren, können die Kopplungsbeziehungen auch auf höherer Abstraktionsebene zwischen Typen und Subsystemen beschrieben werden. Diese sind im rechten Teil der Tabelle bei den indirekten Beziehungen wiederzufinden. Eine Lesezugriffsabhängigkeit von einem Sybsystem S zu einem Typ T bedeutet dann, dass sich unter den im Subsystem S enthaltenen Typen und den darin enthaltenen Operationen sich mindestens eine Operation befindet, welche lesend auf mindestens ein Attribut oder eine Referenz des Typen T zugreift.

Tabelle A.6: Kopplungsrestriktionen

	Umgebung eines Musters (das übrige Softwaresystem)
	Kopplung: Kopplung oder Abhängigkeit beliebiger Natur
	Beliebige Verwendung eines Typen, z.B. als Parametertyp
	Verwendung eines Typen als Obertyp
	Verwendung eines Typen zur Instaziierung
	Beliebiger Zugriff auf Typen, Operationen, Referenzen oder Attribute
	Lesezugriff auf eine Referenz oder ein Attribut
	Schreibzugriff auf eine Referenz oder ein Attribut
	Aufruf einer Operation
	Verbot der Kopplung jeglicher Art
	Verbot jeglicher Schreibzugriffe

A.2.6 PSL: Entwurfsaufgaben

Entwurfsaufgaben wurden in Abschnitt 3.5.4 eingeführt und motiviert. Ihre Notation wird in der Tabelle A.7 zusammengefasst.

Tabelle A.7: Entwurfsaufgaben – nicht formal erfassbare Entwurfsinformationen

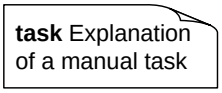
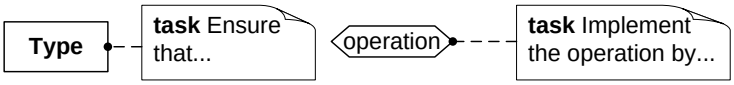
	Aufgabenbeschreibung (Task)
	Zuordnung einer Aufgabe zu einem Entwurfselement

Tabelle A.8: Gültige Kombinationen von Kopplungsarten und Entwurfsteilen (Environment ist hier ein spezielles Subsystem)

	direkt		indirekt	
	von	**zu**	**von**	**zu**
use	Typ, Operation	Typ, Operation, Referenz, Attribut	Subsystem, Typ, Operation	Subsystem, Typ, Operation, Referenz, Attribut
point to	Typ, Operation	Typ	Subsystem, Typ, Operation	Subsystem, Typ
specialize	Typ	Typ	Subsystem, Typ	Subsystem, Typ
access	Operation	Typ, Operation, Referenz, Attribut	Subsystem, Typ, Operation	Subsystem, Typ, Operation, Referenz, Attribut
instantiate	Operation	Typ	Subsystem, Typ, Operation	Subsystem, Typ
call	Operation	Operation	Subsystem, Typ, Operation	Subsystem, Typ, Operation
read	Operation	Referenz, Attribut	Subsystem, Typ, Operation	Subsystem, Typ, Referenz, Attribut
write	Operation	Referenz, Attribut	Subsystem, Typ, Operation	Subsystem, Typ, Referenz, Attribut

A.3 Semantik

In Kapitel 3 habe ich die Semantik der eingeführten Sprachkonstrukte PSL und der DAL nur informell beschrieben. Dort erkläre ich, welche Konzepte objektorientierter Sprachen die von mir eingeführten Sprachkonstrukte repräsentieren und wie Set Fragements veschiedene Implementierungsvarianten eines Musters repräsentieren. In Folgenden definiere ich die Semantik meiner DAL-Sprachkonstrukte durch exemplarische Abbildung auf eine Modellierungssprache für Softwareentwurfsmodelle. Anschließend definiere ich die Semantik von Set Fragments mit Hilfe mathematischer Abbildungen.

Semantik der DAL

Die Semantik der DAL-Sprachkonstrukte kann im Prinzip durch Abbildung auf eine beliebige objektorientierte Modellierungssprache erfolgen. Ich habe mich für die Kombination aus Ecore- und Story-Diagramm-Modellen entschieden und definiere die Semantik, indem ich jedes Sprachkonstrukt der DAL auf Sprachkonstrukte von Ecore- und Story-Diagramm-Modellen abbilde. Die zugehörige Übersetzungstransformation wird in Kapitel 5 beschrieben.

Die DAL ist so konzipiert, dass mit einer einzigen Spezifikation mehr als nur eine konkrete Entwurfsstruktur repräsentiert wird (siehe S. 254 ff.). Durch die hier vorgestellte Abbildung der Sprachkonstrukte aufeinander beschreibe ich die verschiedenen Möglichkeiten, einzelne DAL-Konstrukte in Teile von Ecore- und Story-Diagramm-Modellen zu übersetzen. Somit spanne ich den Lösungsraum auf, in dem sich alle zu einer Musterspezifikation konformen Ecore- und Story-Diagramm-Implementierungsvarianten befinden (siehe Abb. 3.12, S. 57 und vgl. Abb. 3.10 und 3.11, S. 55).

Die PSL erweitert die Ausdrucksmöglichkeiten der DAL. Insbesondere habe ich

Semantik der PSL / der Set Fragments

das PSL-Sprachkonstrukt der Set Fragments eingeführt, um mehrere Implementierungsvarianten in nur einer Musterspezifikation kompakt zu beschreiben. Welche Implementierungsvarianten eine Spezifikation mit Set Fragments repräsentiert, definiere ich in Abschnitt A.3.3 detailliert mit Hilfe mehrerer mathematischer Definitionen und Abbildungen. Bei der Semantik der Entwurfsaufgaben und Kopplungsrestriktionen belasse ich es bei der informellen Definition in den Abschnitten 3.5.4 und 3.5.3.

Im Folgenden beschreibe ich zunächst in Abschnitt A.3.1 die Abbildung von Struktur beschreibenden Sprachkonstrukten auf Ecore-Sprachkonstrukte. Anschließend erkläre ich in Abschnitt A.3.2 die Abbildung des durch Aktionen beschriebenen Verhaltens auf Story-Diagramme. Schließlich definiere ich in Abschnitt A.3.3 die Semantik von Set Fragements mathematisch.

A.3.1 DAL: Abbildung der Struktur auf Ecore

Die Struktur eines Softwareentwurfs wird in der DAL auf Basis von Typen, Attributen, Referenzen, Operationen und Beziehungen dazwischen beschrieben. In Ecore-Modellen werden sehr ähnliche Sprachkonstrukte zur Beschreibung von Klassenstrukturen verwendet wie in der DAL, wodurch die Abbildung auf den ersten Blick trivial erscheint. Typen werden auf Klassen (EClass) abgebildet, Attribute, Referenzen, Operationen und Parameter werden auf entsprechende Konstrukte in Ecore-Modellen abgebildet (EAttribute, EReference, EOperation, EParameter). Doch die DAL lässt nicht nur eine, sondern mehrere Möglichkeiten für die Abbildung einzelner Sprachkonstrukte zu. Das gibt Entwicklern mehr Flexibilität bei der Implementierung von Mustern, macht jedoch die Beschreibung der Abbildung und das automatische Prüfen der Konformität zur Musterspezifikation schwieriger. Bei mehreren Möglichkeiten führe ich die kanonische Abbildung (siehe Abb. 3.12, S. 57) als erste auf. Diese wird bei der Generierung einer Musterimplementierung verwendet (siehe Kapitel 5).

Subsysteme

Zusätzlich zu den genannten Sprachkonstrukten können in Musterspezifikationen auch Subsysteme zur Beschreibung der Softwarestruktur verwendet werden. Subsysteme spielen jedoch eine besondere Rolle. Für diese gibt es keine direkte Entsprechung im Entwurfsmodell – hier in Ecore-Modellen. Ein Subsystem könnte prinzipiell auf je ein Paket abgebildet werden, um mehrere Klassen logisch zusammenzufassen. Subsysteme sollen jedoch auch paketübergreifend zusammenhängende Teile eines Softwaresystems repräsentieren, so auch die Teilsysteme *Model*, *View* und *Controller* des MVC-Paradigmas, welche aus jeweils mehreren Paketen bestehen können. Darum übersetze ich Subsysteme bei der Generierung einer Musterimplementierung nicht in bestimmte Elemente in Ecore-Modellen. Stattdessen dienen Subsysteme lediglich der Dokumentation und Überprüfung von Musterimplementierungen. Dazu ordnet ein Entwickler nach einer Musteranwendung jedem Subsystem der Musterspezifikation eine Menge von Klassen und Paketen seines Softwaresystems zu und kann dadurch mit Hilfe des Validierungsmechanismus die Korrektheit seiner Musterimplementierung in Bezug auf spezifizierte Kopplungseigenschaften prüfen.

Typen

Typen werden auf Klassen (EClass) in Ecore-Modellen abgebildet (Abb. A.15). Sie werden nahezu eins zu eins übersetzt. Der Klassenname kann dabei beliebig

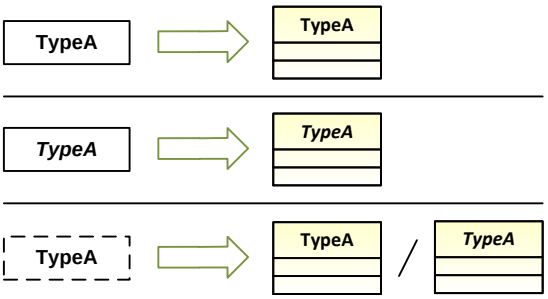


Abbildung A.15: Abbilden von Typen auf Klassen

vom Typnamen abweichen. Ein abstrakter Typ wird zu einer abstrakten Klasse, ein konkreter Typ zu einer konkreten Klasse, ist nichts von beidem angegeben, ist die entsprechende Klasse konkret oder abstrakt. Die resultierende Klasse kann beliebige zusätzliche Eigenschaften wie Attribute, Referenzen und Operationen haben.

Vererbung

Ist zwischen zwei Typen eine Vererbungsbeziehung spezifiziert und sind die beteiligten Typen bereits auf Klassen abgebildet worden (in Abb. A.16 grau dargestellt), so wird auch zwischen den Klassen eine Vererbungsbeziehung gefordert. Weil eine Vererbungsbeziehung in einer Musterspezifikation auch eine indirekte Vererbungsbeziehung im Entwurf zulässt, kann die Vererbungsbeziehung zwischen den Klassen eine direkte oder indirekte sein.

Attribute und primitive Datentypen

Attribute in der DAL werden auf Attribute in Ecore abgebildet. Sie werden bei den Klassen ergänzt, die dem Elterntyp des DAL-Attributs entsprechen (in Abb. A.17 entspricht die Klasse A dem Typ TypeA). Der Attributname ist dabei beliebig. Der primitive Typ des DAL-Attributs (PrimitiveType) wird in einen primitven Ecore-Datentyp (EDataType) übersetzt. Diese werden wie folgt abgebildet:

$$\mathsf{boolean} \rightarrow \mathsf{EBoolan}, \mathsf{integer} \rightarrow \mathsf{EInt}, \mathsf{real} \rightarrow \mathsf{EDouble}, \mathsf{string} \rightarrow \mathsf{EString}$$

Referenzen

Referenzen in der DAL werden auf Referenzen in Ecore abgebildet (Abb. A.18). Der Name einer Referenz (dieser entspricht dem Rollennamen einer UML-Assoziation) kann bei der Abbildung beliebig abweichen. Die Kardinalitäten werden nach dem Prinzip 0..1 → 0..1 und * → 0..* abgebildet. Bilden zwei DAL-Referenzen

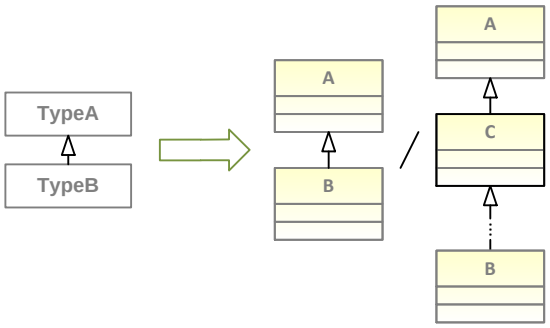


Abbildung A.16: Abbilden einer Vererbungsbeziehung

Abbildung A.17: Abbilden von Attributen

Abbildung A.18: Abbilden von Referenzen

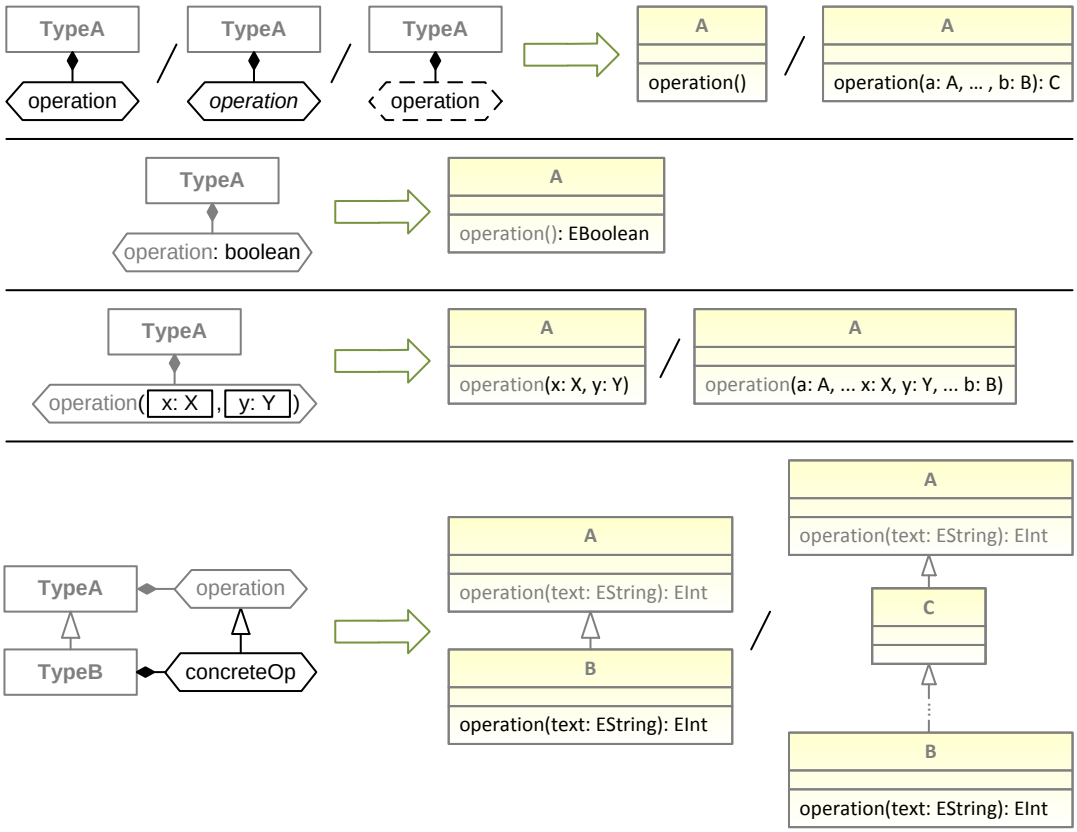


Abbildung A.19: Abbilden von Operationen und ihren Signaturen

zusammen eine bidirektionale Assoziation (in diesem Fall verweisen die Referenzen aufeinander, um die Zusammengehörigkeit auszudrücken), so werden sie analog dazu auf zwei zusammengehörige Referenzen in Ecore abgebildet (unten in der Abb. A.18).

Operationen

Operationen werden auf Methoden abgebildet (Abb. A.19). Da es in Ecore keine abstrakten Methoden gibt, werden alle Operationen – abstrakte, konkrete, beliebige – auf nicht abstrakte Methoden abgebildet. Der Name der Methode kann dabei beliebig von dem der Operation abweichen. Sind bei der Operation weder Parameter noch Rückgabetyp oder eine Spezialisierungsbeziehung zu einer anderen Operation definiert, ist die Signatur der Methode beliebig (Abb. A.19 oben).

Ist keine Spezialisierungsbeziehung zu einer anderen Operation definiert, wird der Rückgabetyp unabhängig von der Operation abgebildet. Handelt es sich dabei um einen nicht primitiven Typ, wird er analog zu Attributtypen auf primitive Ecore-Datentypen abgebildet.

Parameter von Operationen werden auf Methodenparameter abgebildet. Die Parameternamen können beliebig voneinander abweichen. Ihr Typ wird auf die gleiche Weise wie der Rückgabetyp einer Operation abgebildet. Zusätzlich zu den bei einer Operation definierten Parametern kann die zugehörige Methode beliebige weitere Parameter haben. Die Reihenfolge der abgebildeten Parameter bleibt erhalten.

Ist eine Operation concreteOp wie in der Abb. A.19 unten dargestellt eine Spezialisierung einer anderen Operation operation, so wird die Operation concreteOp auf eine Methode abgebildet, welche die gleiche Methodensignatur hat wie die zur Operation operation gehörende Methode. Man beachte, dass die Operation concreteOp in diesem Fall über keine Parameter und keinen Rückgabetypen verfügt. Andernfalls wäre die Musterspezifikation ungültig. Die zur Operation concreteOp gehörende Methode implementiert oder überschreibt die andere Methode, wobei die implementierte / überschriebene Methode auch in einer indirekten Oberklasse liegen kann.

A.3.2 DAL: Abbildung des Verhaltens auf Story-Diagramme

Aktionen

Das Verhalten von Operationen wird in der DAL durch Aktionen definiert (siehe Abschnitt 3.4.2, S. 61 ff.). Wurde mindestens eine Aktion zu einer Operation spezifiziert, wird das Verhalten der zur Operation gehördenden Methode durch ein Story-Diagramm beschrieben, welches mit Hilfe eines Annotationsmechanismus mit der Methode verknüpft wird (siehe Abb. A.20 links). Das Story-Diagramm erhält eine zur Methode passende Signatur. Parameter der Methode werden zu Eingabeparametern des Story-Diagramms. Ein Rückgabetyp wird zu einem Ausgabeparameter des Story-Diagramms.

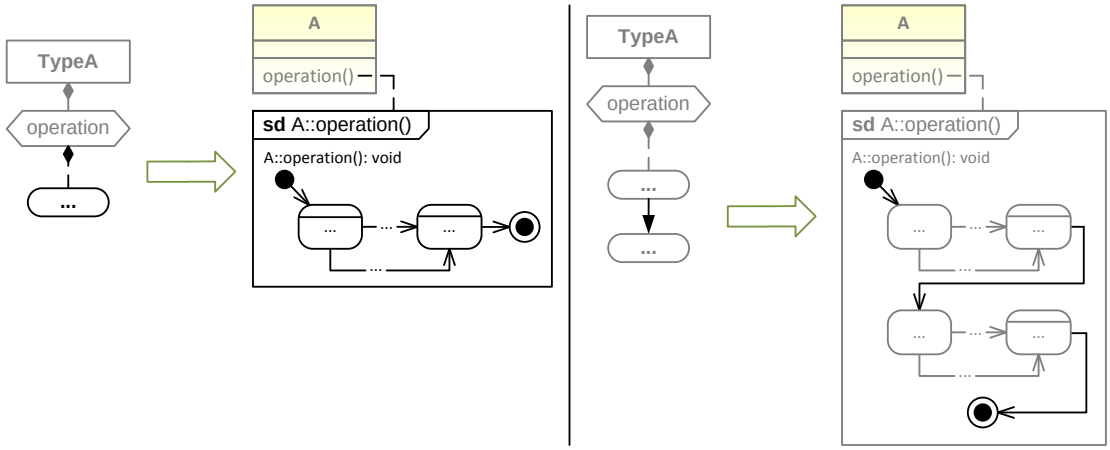


Abbildung A.20: Abbilden von Operationsverhalten auf ein Story-Diagramm

Jede Aktion wird einzeln auf einen Story-Diagramm-Ausschnitt abgebildet. Eine Folge von Aktionen wird zu einer Folge von aufeinander folgenden Kontrollflussausschnitten im Story-Diagramm. Dabei beginnt diese Folge immer mit einem Startknoten und endet mit einem Endknoten (siehe Abb. A.20 rechts).

Anders als bei den Softwarestruktur beschreibenden DAL-Sprachkonstrukten beschränke ich mich bei der Abbildung von Aktionen auf nur eine von theore-

tisch unendlich vielen Möglichkeiten, das durch eine Aktion repräsentierte Verhalten in einem Story-Diagramm zu modellieren. Der Grund dafür ist, dass sich das Verhalten einer Musterimplementierung im Gegensatz zu seiner Struktur nur sehr schwer automatisch auf Konformität mit der Musterspezifikation prüfen lässt. Zum einen ist es aufgrund der vielen Modellierungsmöglichkeiten schwierig, das durch eine Aktion beschriebene Verhalten (z.B. lesen einer Variable) in einem Story-Diagramm zu erkennen (Stichwort: Reverse Engineering). Zum anderen ist es im Allgemeinen nicht entscheidbar, ob eine bestimmte Anweisung (z.B. ein bestimmter Methodenaufruf) überhaupt ausgeführt wird (Stichwort: Halteproblem). Außerdem ist nicht immer klar, welches Verhalten zusätzlich zu dem in einer Musterspezifikation beschriebenen Verhalten in einer Musterimplementierung ergänzt werden darf. Während Logging-Anweisungen das gewünschte Verhalten erhalten, führen Kontrollflussänderungen und Aufrufe anderer Methoden ggf. zu völlig anderem Verhalten. Im Folgenden beschriebe ich nur, welche Story-Diagramm-Elemente bei der automatischen Synthese einer Musterimplementierung aus einer Aktion erzeugt werden. Weicht ein Entwickler aufgrund von notwendigen Entwurfsanpassungen von der generierten Variante ab, muss er die Konformität zum spezifizierten Verhalten selbst prüfen.

call-Aktionen

Eine call-Aktion wie sie in Abschnitt 3.4.2 (S. 61 ff.) definiert ist beschreibt einen Aufruf einer Operation. Neben der Operation werden auch das Zielobjekt – das Objekt, auf dem der Aufruf erfolgt – und ggf. Argumente spezifiziert. Außerdem kann eine Ergebnisvariable verwendet werden, um das Ergebnis des Aufrufs in einer folgenden Aktion weiterzuverwenden. Die Abbildung einer call-Aktion variiert abhängig von diesen Eigenschaften.

In einem Story-Diagramm wird der Aufruf einer Methode durch einen Aktivitätenknoten mit einem darin enthaltenen Ausdruck modelliert, welcher den Methodenaufruf beschreibt. Der Ausdruck beschreibt insbesondere das Zielobjekt und die aufzurufende Methode.

Zielobjekt

Das Zielobjekt wird in der Musterspezifikation durch Verweis auf eine Variable beschrieben. Gibt es keine entsprechende Variable im zugehörigen Story-Diagramm, muss sie deklariert und mit einem Wert belegt werden. Ob und ggf. wie das geschieht, ist in der Abb. A.21 (S. 271) dargestellt. Hier wird für alle Variablenarten, die ein Zielobjekt repräsentieren können, veranschaulicht, auf welchen Story-Diagramm-Ausschnitt eine call-Aktion jeweils abgebildet wird. Der Methodenaufruf befindet sich jeweils in einem Aktivitätenknoten (StatementNode) mit dem Label Call (im Bild grau hinterlegt). Außer der call-Aktion in der Musterspezifikation und den daraus resultierenden Story-Diagramm-Teilen (schwarz dargestellt) wird auch ihr Kontext gezeigt (grau dargestellt).

Verweist eine call-Aktion wie in der Abb. A.21 (a) auf eine Referenz zur Beschreibung des Zielobjekts, so wird im zugehörigen Story-Diagramm zunächst mit Hilfe eines Story Patterns in einem zusätzlichen Aktivitätenknoten (MatchingStoryNode) eine Objektvariable reference deklariert und an den aktuellen Wert der Ecore-Referenz ref gebunden. Der Name der Objektvariable entspricht dem

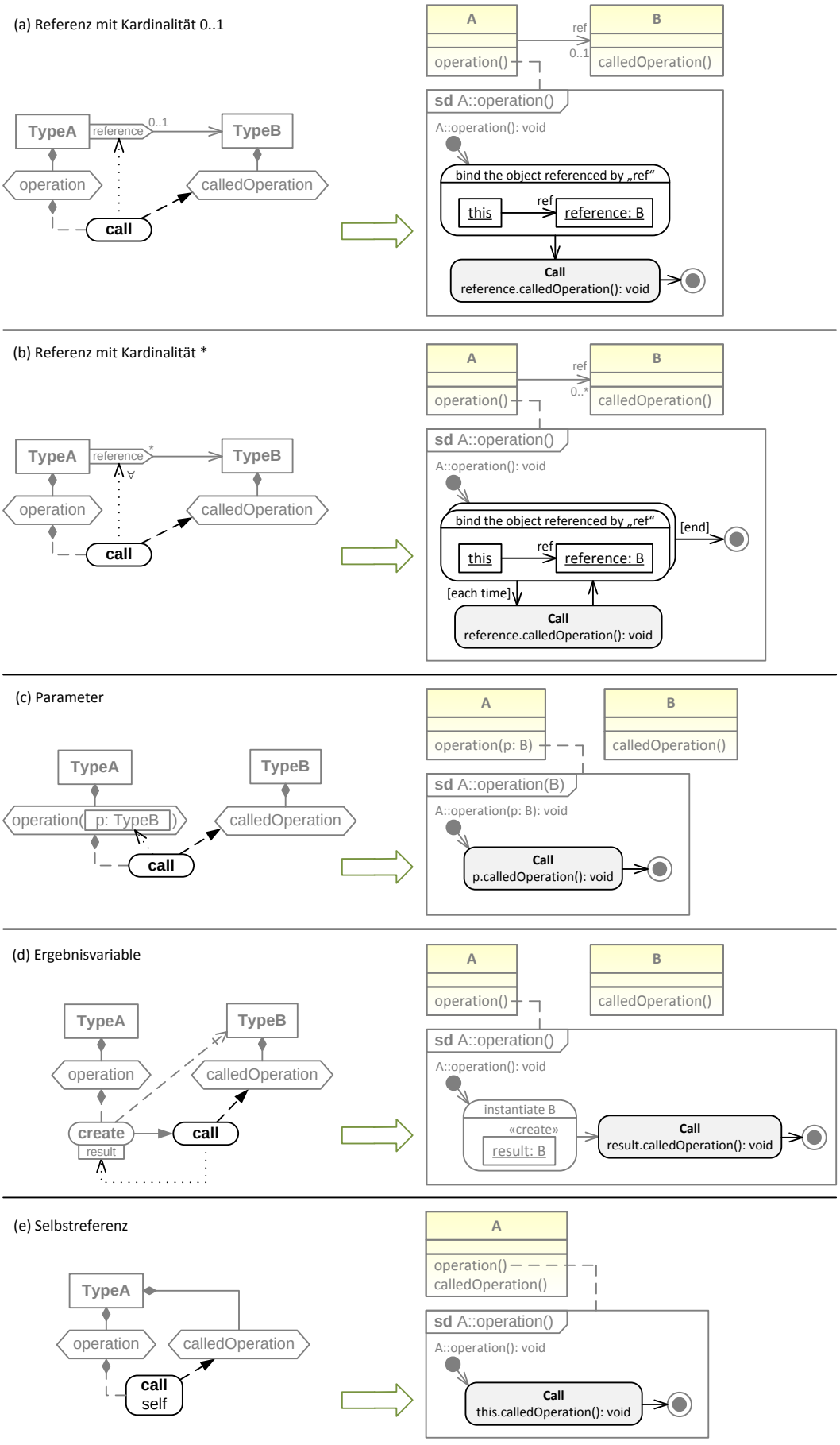


Abbildung A.21: Abbilden von call-Aktionen mit verschiedenen Zielobjekten

Namen der DAL-Referenz[4]. Anschließend erfolgt der Methodenaufruf auf dem referenzierten Objekt. In dem den Aufruf beschreibenden Ausdruck wird das Zielobjekt durch einen speziellen auf die zuvor deklarierte Objektvariable verweisenden Teilausdruck (ObjectVariableExpression) repräsentiert.

Hat die Referenz die Kardinalität * und ist der Verweis der call-Aktion darauf wie in Abb. A.21 (b) mit einem Allquantor versehen, so repräsentiert die call-Aktion nicht nur einen, sondern mehrere Methodenaufrufe; je einen Aufruf auf jedem der referenzierten Objekte. Im Story-Diagramm wird das durch eine Schleife ausgedrückt, welche durch einen iterierten Aktivitätenknoten (MatchingStoryNode mit entsprechendem Attributwert) modelliert wird. Dieser Knoten iteriert über alle durch die ref-Referenz verknüpften Objekte. Jedes Mal, wenn ein Objekt dereferenziert wird, wird anschließend die Methode auf dem Objekt aufgerufen und danach das Prozedere mit dem nächsten Objekt wiederholt. Erst wenn die Methode auf jedem referenzierten Objekt aufgerufen wurde, wird die Schleife beendet.

Wird das Zielobjekt wie in Abb. A.21 (c) durch einen Parameter der aufrufenden Operation beschrieben, ist kein zusätzlicher Aktivitätenknoten notwendig, weil die Variable, die das Zielobjekt beschreibt, auch im Story-Diagramm bereits deklariert und belegt ist. In diesem Fall ist es ein Eingabeparameter des Story-Diagramms (im Bild p). In dem Methodenaufruf im Story-Diagramm wird das Zielobjekt durch einen auf den Parameter verweisenden Ausdruck (ParameterExpression) repräsentiert.

Folgt eine Aktion einer anderen und verwendet sie wie in Abb. A.21 (d) die Ergebnisvariable der vorherigen Aktion zur Beschreibung des Zielobjekts, so wird im Story-Diagramm die Objektvariable weiterverwendet, welche der Abbildung der Ergebnisvariable der vorherigen Aktion entspricht. Im Bild ist es die Objektvariable result im Aktivitätenknoten vor dem Methodenaufruf. Im Methodenaufruf wird durch einen entsprechenden Ausdruck (ObjectVariableExpression) auf die Objektvariable verwiesen.

Soll ein Objekt eine der eigenen Operationen aufrufen, so wird das Zielobjekt in der DAL durch eine Selbstreferenz ausgedrückt, die durch das Schlüsselwort self gekennzeichnet wird (siehe Abb. A.21 (e)). In diesem Fall enthält der Methodenaufruf im Story-Diagramm zur Repräsentation des Zielobjekts einen speziellen Ausdruck, der auf das aufrufende Objekt verweist (ObjectVariableExpression mit dem Schlüsselwort this).

Die übrigen Variablenarten der DAL kommen zur Beschreibung eines Zielobjekts nicht in Frage. Attribute verwenden nur primitive Datentypen und diese haben keine Operationen. Die spezielle Variable, die den Wert null repräsentiert, verweist auf kein Objekt.

Aufrufargumente

Neben dem Zielobjekt und der aufgerufenen Methode muss der Ausdruck, der den Methodenaufruf beschreibt, auch zu überreichende Argumente beschreiben, wenn die aufgerufene Methode Parameter besitzt. Für jedes Argument enthält der Methodenaufrufausdruck einen das Argument repräsentierenden Teilausdruck.

Jedes Argument wird in der DAL analog zum Zielobjekt durch eine Variable be-

[4]Sollte der Name im Story-Diagramm nicht einmalig sein, wird die kleinste den Namen einmalig machende Zahl an den Namen gehängt.

schrieben. Ist eine entsprechende Variable im Story-Diagramm deklariert, kann sie im Methodenaufrufausdruck verwendet werden und muss andernfalls erst deklariert und belegt werden. Wie das Zielobjekt wird auch jedes Argument durch einen Ausdruck beschrieben, welcher auf die entsprechende Variable im Story-Diagramm verweist. Die das Argument beschreibenden Ausdrücke werden analog zu den zuvor beschriebenen Zielobjektausdrücken aufgebaut. Dabei können auch Verweise auf Attributwerte (AttributeExpression) und der Wert null (ObjectVariableExpression mit dem Schlüsselwort null) verwendet werden.

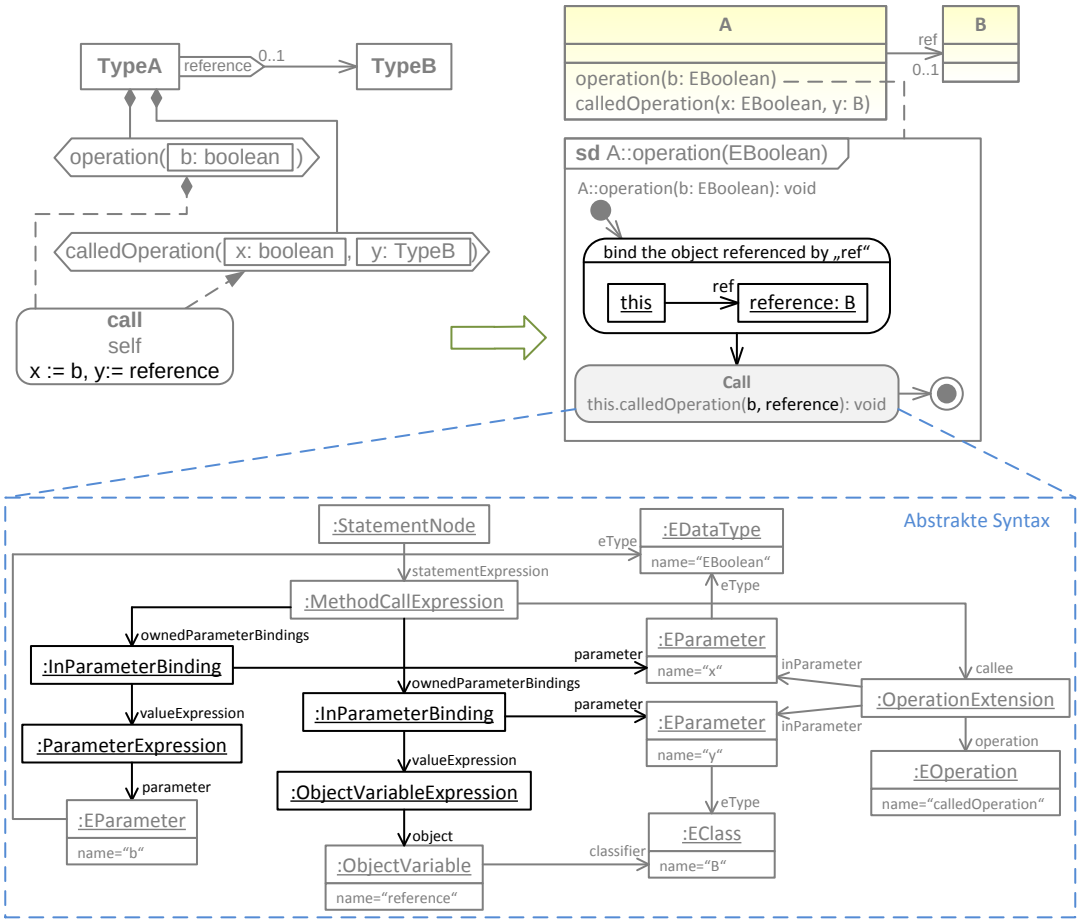


Abbildung A.22: Abbilden einer call-Aktion mit Argumenten

Im Gegensatz zum Vorgehen beim Zielobjekt wird jeder Argumentausdruck in einem Parameterbindungskonstrukt (InParameterBinding) eingebettet. Ein Beispiel mit zwei Argumenten ist in der Abb. A.22 dargestellt. Zusätzlich zur konkreten wird hier auch die abstrakte Syntax zur Darstellung des relevanten Ausschnitts des Methodenaufrufausdrucks verwendet. In dem Beispiel wird ein Argument durch einen Parameter b der aufrufenden Operation und ein Argument durch den Verweis auf eine Referenz reference in der DAL beschrieben. Die Werte der entsprechenden Variablen sollen an die Parameter x und y der aufgerufenen Operation gebunden werden. Im Story-Diagramm werden daraus zwei Parameterbindungen mit einem Parameterausdruck (ParameterExpression) und einem Objektvariablenausdruck (ObjectVariableExpression). Diese ordnen den Parametern der aufgerufe-

nen Methode die Werte der entsprechenden Variablen im Story-Diagramm zu. Im Fall der Referenz muss die Variable analog zum Vorgehen beim Zielobjekt vor dem Methodenaufruf mit Hilfe eines Story Patterns deklariert und gebunden werden. Man beachte, dass die Referenz in diesem Fall die Kardinalität 0..1 haben muss. Bei Argumenten, die durch eine Referenz mit Kardinalität * beschrieben werden, ist die Semantik der call-Aktion nicht definiert.

Ergebnisvariable

Soll das Ergebnis eines Aufrufs in einer Variable zwischengespeichert und ggf. von der nächsten Aktion weiterverwendet werden, wird eine Ergebnisvariable in der DAL verwendet. Im Story-Diagramm wird eine dazu entsprechende Variable deklariert und das Ergebnis des Methodenaufrufs an die Variable zugewiesen.

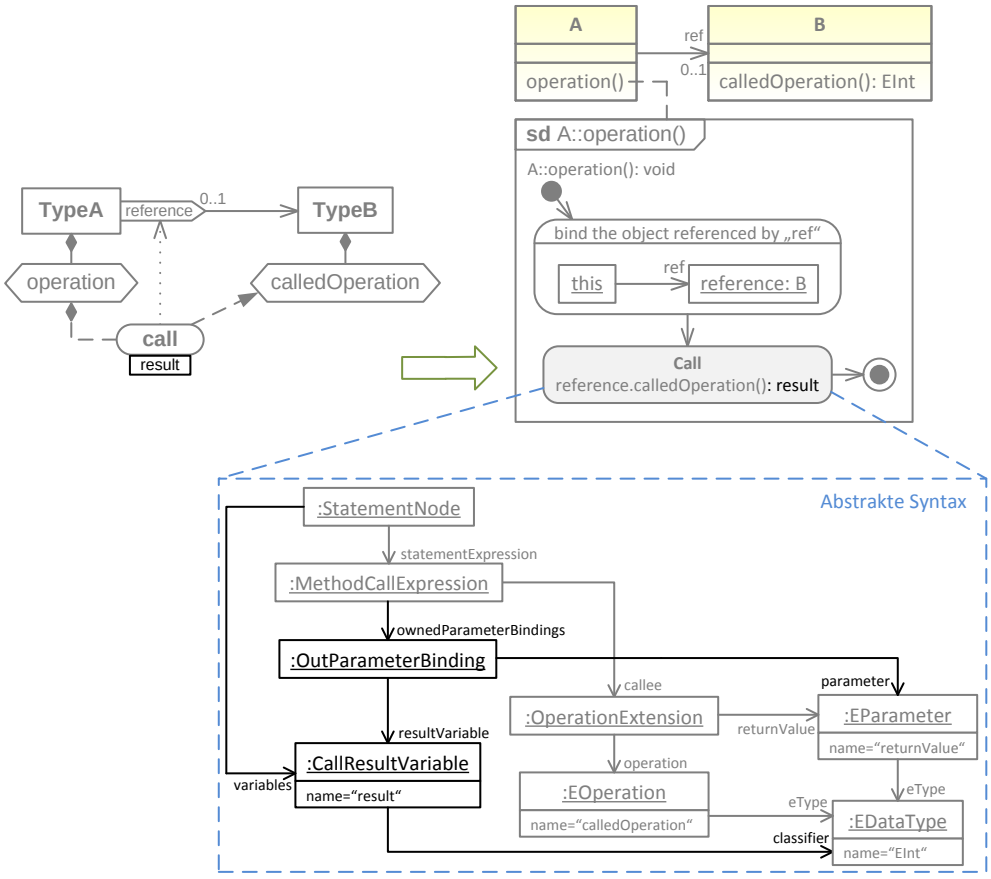


Abbildung A.23: Abbilden einer call-Aktion mit Aufrufergebnis

Die Abbildung einer Ergebnisvariable auf eine entsprechende Variable in einem Story-Diagramm wird durch die Abb. A.23 veranschaulicht. Der Name der Ergebnisvariable in der DAL wird für die Variable im Story-Diagramm übernommen (hier result). Die Variable wird im den Methodenaufruf enthaltenden Aktivitätenknoten deklariert, sodass die Variable in folgenden Aktivitätenknoten weitergenutzt werden kann. Dabei handelt es sich um eine spezielle Variable, die zum Methodenaufruf gehört (CallResultVariable). Analog zu Aufrufargumenten wird auch die Ergebnisvariable in einem Parameterbindungskonstrukt (OutParameterBinding) eingebettet. Darin wird neben der deklarierten Variable für das Ergebnis des Methodenaufrufs auch ein zur Verwendung in Story-Diagrammen angelegter

Ausgabeparameter der aufgerufenen Methode genutzt (returnValue, siehe abstrakte Syntax in Abb. A.23). Das Ergebnis des Methodenaufruf wird an diesen Ausgabeparameter und anschließend auch an die result-Variable gebunden.

redirect-Aktionen

Eine spezielle Form der call-Aktion bildet die redirect-Aktion. Während eine call-Aktion einen beliebigen Aufruf einer Operation beschreibt, steht eine redirect-Aktion für einen Aufruf, bei dem sämtliche Argumente der aufrufenden Operation unverändert an die aufgerufene Operation weitergereicht werden. Dazu müssen die Anzahl, die Reihenfolge und die Typen der Parameter beider Operationen zueinander passen. Im Gegensatz zu call-Aktionen werden bei redirect-Aktionen in Musterspezifikationen keine Parameterübergaben spezifiziert, weil diese implizit durch die Semantik von redirect-Aktionen gegeben sind.

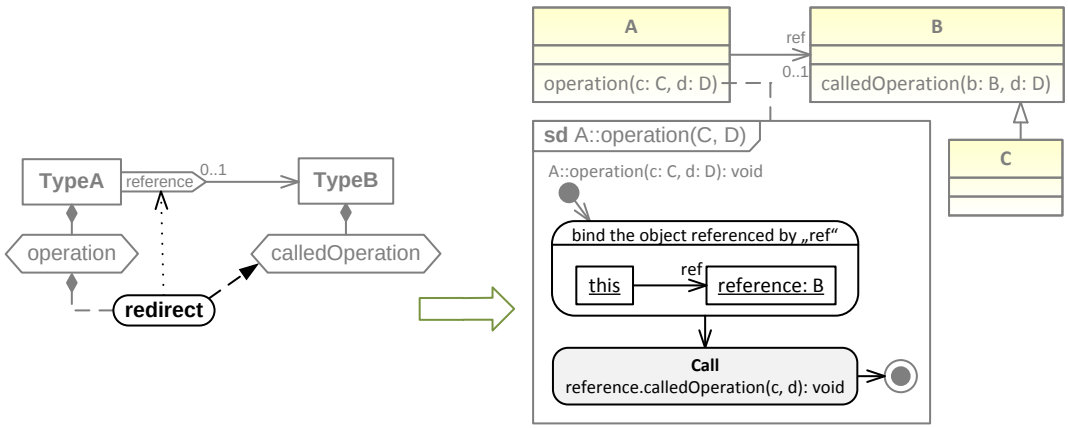


Abbildung A.24: Abbilden einer redirect-Aktion

Bis auf den Umgang mit Parameterübergaben ist das Abbilden einer redirect-Aktion auf Story-Diagramm-Teile identisch zu der einer call-Aktion. Zur Verdeutlichung des abweichenden Umgangs mit Argumenten ist das Abbilden einer redirect-Aktion auf einen Story-Diagramm-Ausschnitt exemplarisch in der Abb. A.24 dargestellt. Wie bei call-Aktionen werden für den Aufruf benötigte Variablen ggf. zuerst deklariert und gebunden (hier die Objektvariable reference). Danach erfolgt der Methodenaufruf in einem eigenen Aktivitätenknoten.

Während in der DAL bei den beiden Operationen in Abb. A.24 keine Parameter und bei der redirect-Aktion keine Parameterübergaben spezifiziert wurden, besitzen die zugehörigen Methoden im Ecore-Modell je zwei Parameter c und d bzw. b und d. Der Methodenaufrufausdruck erhält in diesem Fall zu den Parametern passende Aufrufargumente. Diese sind analog zu den Argumenten aufgebaut, die aus dem Abbilden der bei call-Aktionen explizit modellierten Parameterübergaben resultieren (vgl. Abb. A.22, S. 273).

Die Typen der Parameter der aufrufenden und der aufgerufenen Methode sind entweder identisch (wie bei den Parametern d) oder zu einander passend (z.B. ist der Typ C des Parameters c ein Untertyp von B, dem Typ des Parameters b). Andernfalls wäre die Zuordnung der Ecore-Elemente zu den DAL-Elementen ungültig.

delegate-Aktionen

Eine delegate-Aktion ist eine spezielle Form einer redirect-Aktion. Allerdings beschreibt eine delegate-Aktion einen Aufruf einer Operation, bei dem nicht nur sämtliche Argumente unverändert an die aufgerufene Operation weitergereicht werden, sondern auch das Ergebnis des Aufrufs als eigenes Ergebnis zurückgegeben wird. Sämtliches Verhalten der aufrufenden Operation wird somit durch das Verhalten der aufgerufenen Operation bestimmt.

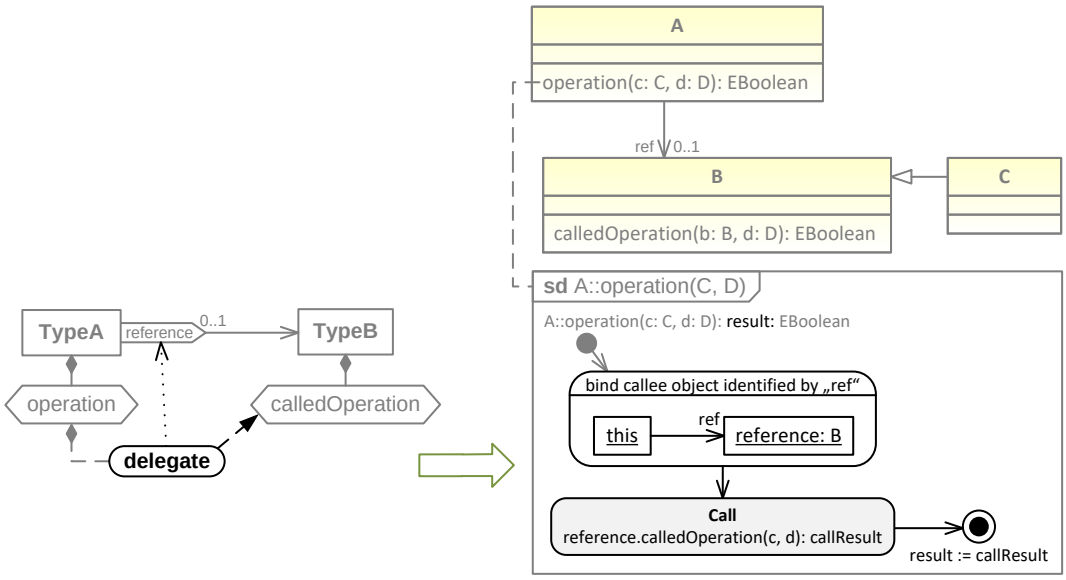


Abbildung A.25: Abbilden einer delegate-Aktion

Das Abbilden einer delegate-Aktion auf Story-Diagramm-Ausschnitte ist bis auf den Umgang mit dem Ergebnis des Aufrufs identisch zum Abbilden einer redirect-Aktion. Bei delegate-Aktionen wird grundsätzlich keine Ergebnisvariable spezifiziert, weil diese implizit durch die Semantik der delegate-Aktion gegeben ist und es keine folgende Aktion gibt, welche das Ergebnis weiterverwenden könnte (die delegate-Aktion beendet den Kontrollfluss der Methode durch Rückgabe des Aufrufergebnisses). Um das Ergebnis des Aufrufs als eigenes zurückzugeben, wird der Story-Diagramm-Ausschnitt, auf den eine delegate-Aktion abgebildet wird, um einen Endknoten ergänzt, welcher dem Methodenaufrufknoten folgt. Geben die aufgerufene und die aufrufende Methode wie im Beispiel aus der Abb. A.25 einen Wert zurück, so wird der von der aufgerufenen Methode gelieferte Wert in der Variable callResult zwischengespeichert und durch den Ausdruck result := callResult im Endknoten unverändert als Ergebnis der aufrufenden Methode zurückgegeben. Die Deklaration der Variable callResult erfolgt analog zur Deklaration einer Ergebnisvariable beim Abbilden von call-Aktionen (vgl. Abb. A.23, S. 274). Gibt die aufgerufene Methode keine Werte zurück, entfällt der Ausdruck im Endknoten.

Man beachte, dass die Methoden (die aufrufende und aufgerufene Methode) im Fall einer delegate-Aktion neben passenden Parametern auch zueinander passene Rückgabetypen haben müssen. Entweder geben beide Methoden keine Werte zurück oder der Rückgabetyp der aufgerufenen Methode ist identisch oder spezieller als der Typ der aufrufenden Methode. Außerdem darf das Zielojekt einer

delegate-Aktion nicht durch eine Referenz mit Kardinalität * beschrieben, weil das mehreren Delegationen entsprechen würde und nur eins der diversen Aufrufergebnisse als Ergebnis der aufrufenden Methode zurückgegeben werden kann.

create-Aktionen

Eine create-Aktion beschreibt das Instanziieren eines Typen. In einem Story-Diagramm wird das durch einen Aktivitätenknoten realisiert, in dem mit Hilfe eines Story Patterns eine neue Instanz der zum spezifizierten Typ gehörenden Klasse erzeugt und in einer neuen Objektvariable gespeichert wird.

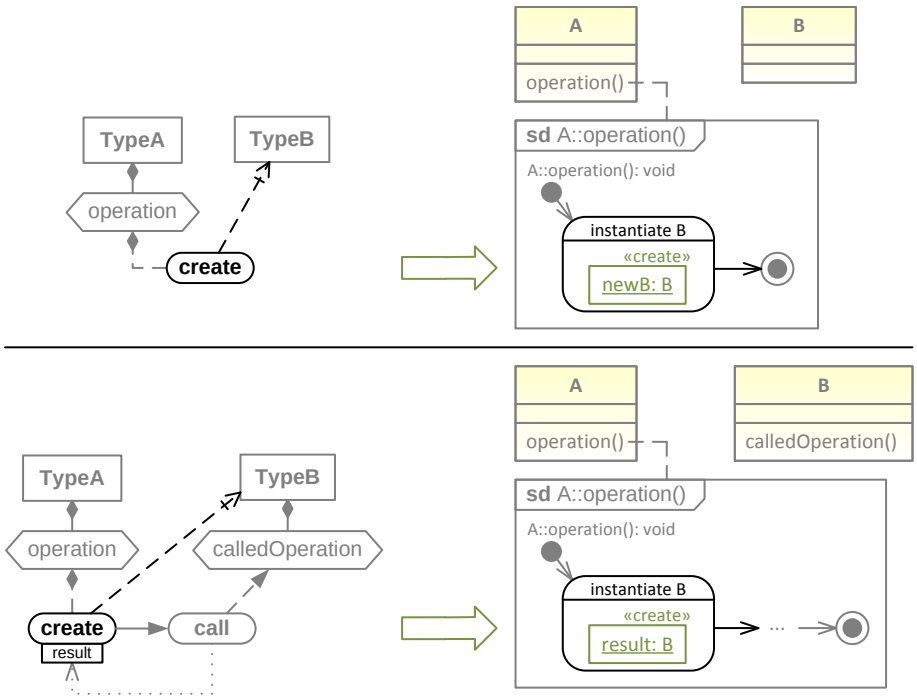


Abbildung A.26: Abbilden von create-Aktionen

Der aus einer create-Aktion resultierende Story-Diagramm-Ausschnitt ist in der Abb. A.26 anhand von zwei Beispielen dargestellt. In einem Aktivitätenknoten mit einem darin enthaltenen Story Pattern wird eine mit «create» markierte Objektvariable zur Beschreibung des zu erzeugenden Objekts verwendet. Der Typ der Variable entspricht der zu instanziierenden Klasse. Wird eine create-Aktion wie oben abgebildet ohne Ergebnisvariable verwendet, so erhält die Objektvariable den Namen newB, wobei B der Name der zu instanziierenden Klasse ist. Andernfalls wird der Name der Ergebnisvariable für die Objektvariable übernommen (siehe Abb. A.26 unten) und kann in folgenden Aktivitätenknoten verwendet werden.

produce-Aktionen

Eine produce-Aktion ist eine spezielle create-Aktion. Hierbei wird das erzeugte Objekt direkt als Ergebnis der Methode zurückgegeben. Das entspricht einer create-Aktion mit einer folgenden return-Aktion. Eine produce-Aktion vereinfacht jedoch

die Modellierung dieses in diversen Mustern wie Abstract Factory vorkommenden Verhaltens.

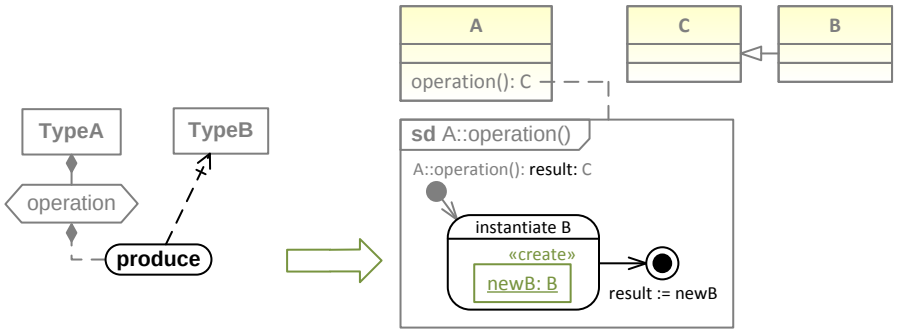


Abbildung A.27: Abbilden einer produce-Aktion

Die Abbildung einer produce-Aktion erfolgt analog zur Abbildung einer create-Aktion, wobei der resultierende Story-Diagramm-Ausschnitt um einen Endknoten ergänzt wird, in dem das erzeugte Objekt zurückgegeben wird (siehe Abb. A.27).

return-Aktionen

Eine return-Aktion beschreibt das Zurückgeben eines Variablenwertes als Ergebnis eines Methodenaufrufs. Dazu verweist die Aktion in einer Musterspezifikation auf die entsprechende Variable. Wie eine return-Aktion auf einen Story-Diagramm-Ausschnitt abgebildet wird, hängt von der Art der DAL-Variable ab.

In der Abb. A.28 wird für alle Variablenarten exemplarisch dargestellt, welchem Story-Diagramm-Ausschnitt eine return-Aktion entspricht.

Im Fall einer Referenz (a) wird zuerst der zurückzugebende Wert mit Hilfe eines Story Patterns an eine Variable gebunden und anschließend der Wert dieser Variable in einem Endknoten mit Hilfe eines entsprechenden Ausdrucks zurückgegeben (hier: result := reference). Der Teilausdruck reference (eine ObjectVariableExpression) verweist hierbei auf die zuvor deklarierte und gebundene Objektvariable gleichen Namens. Da nur ein Wert zurückgegeben werden kann, muss die Referenz die Kardinalität 0..1 haben.

In allen anderen Fällen ist die Variable, die den zurückzugebenden Wert enthält, bereits vorhanden, sodass außer dem Endknoten kein weiterer Aktivitätenknoten nötig ist. Soll der Wert eines Attributs (b), eines Parameters (c) oder einer Ergebnisvariable (d) zurückgegeben werden, enthält der Ausdruck im Endknoten einen entsprechenden Teilausdruck, der auf die jeweilige Variable verweist (AttributeExpression, ParameterExpression oder ObjectVariableExpression). Im Fall einer Selbstreferenz (e) und einer Verwendung des Wertes null (f) wird ein entsprechender Ausdruck verwendet (ObjectVariableExpression mit dem Schlüsselwort this bzw. null).

read-, write- und delete-Aktionen

Die Aktionen read, write und delete beschreiben Lese- und Schreibzugriffe. Allerdings beschreiben sie derartige Zugriffe nicht genau genug, um daraus ausführbare

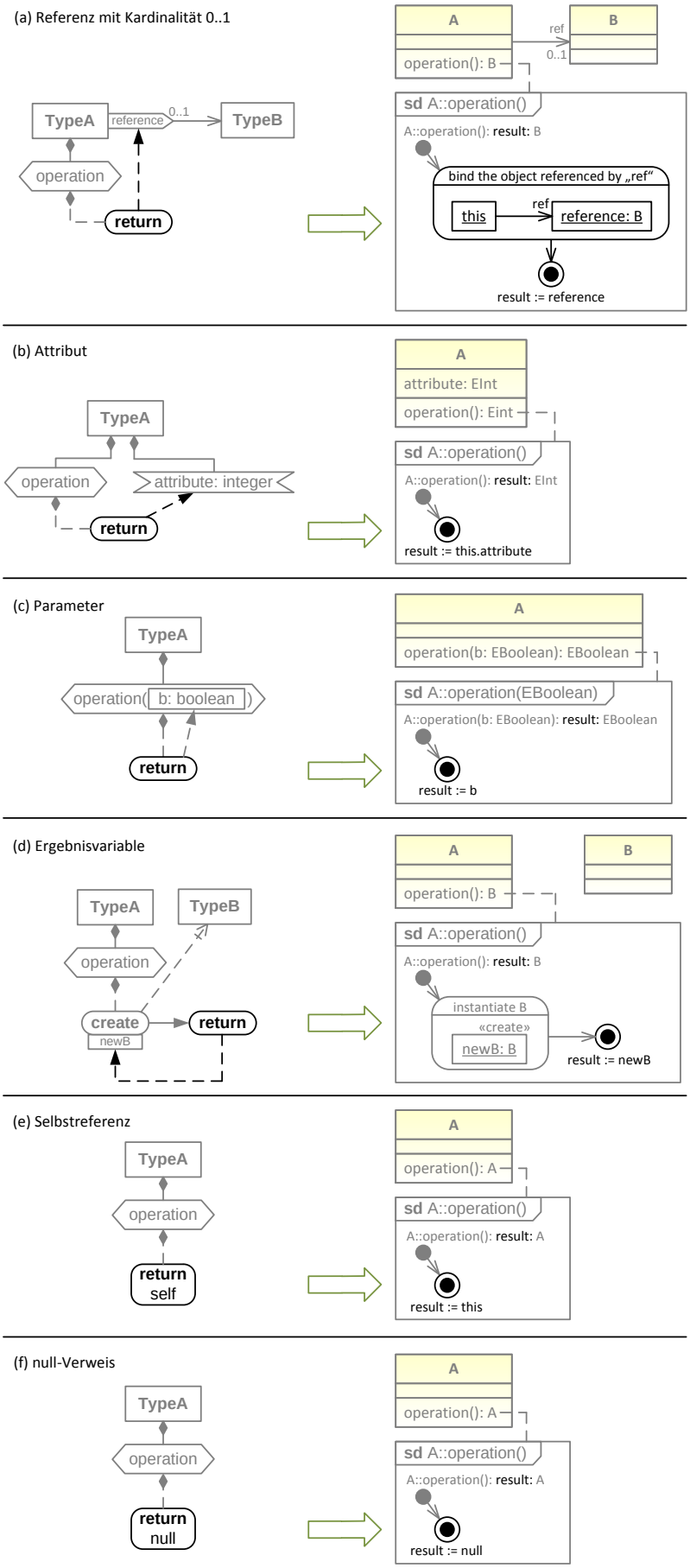


Abbildung A.28: Abbilden von return-Aktionen mit verschiedenen Variablen

Verhaltensmodelle wie Story-Diagramme (oder ausführbaren Code) abzuleiten. Es ist z.B. unklar wie eine Variable gelesen wird. Wird sie beim Aufruf einer Methode als Argument verwendet oder wird sie irgendeiner anderen Variable zugewiesen, wenn ja welcher? Wenn auf eine Variable schreibend zugegriffen wird, ist unklar, welcher Wert zugewiesen wird. Bei der delete-Aktion (einem speziellen Schreibzugriff) ist unklar, was getan werden muss, damit ein Objekt aus dem Speicher gelöscht wird (z.B. müssten alle Referenzen darauf entfernt werden, damit das Objekt in einer Java Virtual Machine per Garbage Collector eingesammelt wird). Aufgrund dieser fehlenden Detailinformationen zum spezifizierten Verhalten werden read-, write- und delete-Aktionen grundsätzlich nicht übersetzt. Diese werden bloß leeren Aktivitätenknoten zugeordnet. Entwickler können durch das Modellieren des für den konkreten Fall passenden Verhaltens die Aktivitätenknoten vervollständigen.

A.3.3 PSL: Formale Definition der Semantik von Set Fragments

In Entwurfsmustern werden immer wieder mehrfache Vorkommen gleicher Strukturen in den Entwurfslösungen beschrieben. So wird z.B. beim Observer-Muster eine Schnittstelle oder abstrakte Oberklasse Observer beschrieben, zu der es beliebig viele konkrete Unterklassen mit gleichen Eigenschaften geben kann ConcreteObserver (siehe Abb. 1.3 auf S. 5). Jede ConcreteObserver-Implementierung erhält laut Musterbeschreibung eine Referenz auf das beobachtete Objekt (ConcreteSubject) und eine ausimplementierte Operation update, die auf Änderungen des beobachteten Objektzustands reagiert.

Mit Hilfe von Set Fragments lassen sich solche Strukturen kompakt beschreiben. Der Teil des Entwurfs, der sich bei einer Musterimplementierung in gleicher Weise wiederholen kann, wird nur ein Mal eingerahmt durch ein Set Fragment spezifiziert (vgl. Abb. 3.7, S. 52) und beschreibt damit, dass die eingerahmte Struktur mit allen spezifizierten Eigenschaften als Ganzes mindestens ein Mal aber beliebig häufig in einer Musterimplementierung vorkommen darf.

Durch sich überschneidende Set Fragments wie in Tabelle A.5 (S. 263), Schachtelungen von Set Fragments und über Set Fragments hinausgehende Beziehungen lässt sich die genaue Bedeutung der Set Fragments nur schwer in Worte fassen. Die informelle Definition der Semantik von Set Fragments (siehe Abschnitt 3.5.2, S. 67) reicht für eine automatische Musteranwendung und eine automatische Validierung von Musterimplementierungen nicht aus. Darum definiere ich die Semantik dieses Sprachkonstruktes im Folgenden formal. Außerdem zeige ich die Wohlgeformtheit aller mit Hilfe von Set Fragments beschreibbaren und daraus herleitbaren Entwurfsmodellstrukturen, indem ich einige Eigenschaften dieser Semantik beweise wie z.B. den Erhalt aller Beziehungen zwischen den Elementen einer Musterspezifikation nach beliebiger Vervielfältigung der durch Set Fragments spezifizierten, wiederholbaren Teile eines Entwurfs.

Anders als bei den DAL-Sprachkonstrukten ist eine Abbildung von Set Fragments auf entsprechende Konstrukte einer Modellierungssprache nicht nötig. Es gibt keine direkte Entsprechung in Entwurfsmodellen. Um alle durch Set Fragments definierten Implementierungsvarianten einer Entwurfslösung eindeutig zu beschreiben, definiere ich stattdessen die Menge aller korrekten mathematischen

Abbildungen einer Musterspezifikation mit Set Fragments auf eine Musterspezifikation ohne Set Fragments (siehe Definition A.17, S. 290). Bei jeder dieser Abbildungen bleiben die in Set Fragments enthaltenen DAL-Elemente erhalten oder sie werden entsprechend der Semantik von Set Fragments vervielfältigt. Das Ergebnis einer solchen Abbildung – eine Musterspezifikation ohne Set Fragments – ist eine 1-zu-1-Repräsentation einer von vielen zu einer Musterspezifikation passenden Entwurfsmodellstrukturen bzw. einer Implementierungsvariante des spezifizierten Musters.

Jede dieser Abbildungen entspricht einer Graphtransformation eines typisierten, attributierten Graphen mit Vererbung und mit Set Fragments in einen typisierten, attributierten Graphen mit Vererbung ohne Set Fragments. Für meine Formalisierung stütze ich mich auf existierende Graphentheorien und Graphtransformationstheorien und orientiere mich dabei an den von Ehrig et al. [EEPT06, Kap. 2, 8, 13] formulierten Definitionen von typisierten, attributierten Graphen mit und ohne Vererbung. Diese Definitionen habe ich für meine Zwecke vereinfacht und um Set Fragments und Set-Fragment-Instanzen erweitert. Die Semantik von Set Fragments habe ich in der Definition A.17 festgelegt, zu welcher ich im Folgenden schrittweise führe.

Definition A.1 (Graph)**.** *Ein Graph G ist ein Tupel $G = (V, E, source, target)$ und besteht aus*

- *V, einer Menge von Knoten und E, einer Menge von Kanten,*
- *$source : E \to V$, einer Quellfunktion für Kanten und*
- *$target : E \to V$, einer Zielfunktion für Kanten.*

Definition A.2 (Graphmorphismus)**.** *Für zwei Graphen G_1 und G_2 mit $G_i = (V_i, E_i, source_i, target_i)$ für $i = 1, 2$ ist ein Graphmorphismus f definiert als $f : G_1 \to G_2, f = (f_V, f_E)$ bestehend aus zwei Funktionen*

- *$f_V : V_1 \to V_2$ und $f_E : E_1 \to E_2$ mit*
- *$f_V \circ source_1 = source_2 \circ f_E$ und $f_V \circ target_1 = target_2 \circ f_E$ (d.h. f_V und f_E erhalten die source- und target-Funktionen).*

Ein Graphmorphismus f ist injektiv (bzw. surjektiv), wenn beide Funktionen f_V und f_E injektiv (bzw. surjektiv) sind. f wird als isomorph bezeichnet, wenn f bijektiv ist, d.h. sowohl injektiv als auch surjektiv, dargestellt mit $f : G_1 \leftrightarrow G_2$.

Die Definitionen A.1 und A.2 entsprechen bis auf die Bezeichnungen im Wesentlichen den Definitionen 2.1 und 2.4 von Ehrig et al. [EEPT06, Kap. 2].

Definition A.3 (getypter Graph)**.** *Seien $G = (V, E, source, target)$ und $T = (V_T, E_T, source_T, target_T)$ Graphen, $t : G \to T, t = (type_V, type_E)$ ein Graphmorphismus bestehend aus den Funktionen $type_V : V \to V_T$ und $type_E : E \to E_T$, dann ist $G_T = (V, E, source, target, T, type)$ ein getypter Graph mit*

- *T, dem Typgraphen von G_T und*
- *$type : V \cup E \to V_T \cup E_T$, einer Typfunktion mit*
  $type(x) = \begin{cases} type_V(x) & \text{für } x \in V \\ type_E(x) & \text{für } x \in E \end{cases}$

Bei der Definition eines getypten Graphen habe ich mich an Ehrig et al. [EEPT06, Def. 2.6] orientiert, aber eine andere Tupelform gewählt, um die folgenden Definitionen zu vereinfachen. Auf die Definition eines Morphismus zwischen getypten Graphen [EEPT06, Def. 2.6] habe ich verzichtet, weil ich sie im Folgenden nicht benötige.

Definition A.4 (getypter Graph mit Set Fragments)**.** *Zu einem getypten Graphen* $G = (V, E, source, target, T, type)$ *ist ein getypter Graph mit Set Fragments* G_S *definiert als ein Tupel* $G_S = (V, E, source, target, T, type, S, contains_S)$ *mit*

(i) S, *der Menge von Set Fragments und*

(ii) $contains_S : S \times (V \cup S) \to \{wahr, falsch\}$, *der Enthaltensfunktion für Knoten und Set Fragments (*$contains_S(s, x)$ *bedeutet* s *enthält* x*), sodass*

(iii) $\forall s_1, s_2 \in S : contains_S(s_1, s_2) \Rightarrow s_1 \neq s_2$

(iv) $\forall s_1, s_2 \in S, x \in V : contains_S(s_1, s_2) \wedge contains_S(s_2, x) \Rightarrow \neg contains_S(s_1, x)$

(v) $\forall s_1, s_2, s_3 \in S, x \in V : contains_S(s_1, x) \wedge contains_S(s_2, x) \wedge contains_S(s_3, x) \Rightarrow s_1 = s_2 \vee s_1 = s_3 \vee s_2 = s_3$

(vi) $\forall s_1, s_2 \in S, x \in V : s_1 \neq s_2 \wedge contains_S(s_1, x) \wedge contains_S(s_2, x) \Rightarrow \exists y_1, y_2 \in V : contains_S(s_1, y_1) \wedge \neg contains_S(s_2, y_1) \wedge \neg contains_S(s_1, y_2) \wedge contains_S(s_2, y_2)$

Anmerkung A.5. *Zum besseren Verständnis erläutere ich im Folgenden einige der Punkte aus Definition A.4:*

(iii) *Dieser Punkt gibt an, dass kein Set Fragment sich selbst enthalten kann.*

(iv) *Enthält ein Set Fragment* s_1 *ein anderes Set Fragment* s_2*, so enthält es alle Knoten aus* s_2 *nur indirekt.*

(v) *Hier kommt die Einschränkung hinzu, dass ein Knoten in höchstens zwei Set Fragments enthalten sein kann, d.h. höchstens zwei Set Fragments dürfen sich überschneiden.*

(vi) *Diese Einschränkung gibt an, dass jedes Set Fragment, welches sich mit einem anderen überschneidet (ohne, komplett im anderen enthalten zu sein), mindestens einen, im anderen Set Fragment nicht enthaltenen Knoten enthalten muss (wie in Abb. 3.27, S. 68).*

Definition A.6 (getypter Graph mit Set-Fragment-Instanzen)**.** *Für einen getypten Graphen mit Set Fragments G_S mit einem Typgraphen T mit $G_S = (V_S, E_S, source_S, target_S, T, type_S, S, contains_S)$ und $T = (V_T, E_T, source_T, target_T)$ sowie einen Graphmorphismus $t : G \to T$, $t = (type_V, type_E)$ für den Graphen $G = (V, E, source, target)$ bestehend aus den Funktionen $type_V : V \to V_T$ und $type_E : E \to E_T$ ist ein getypter Graph mit Set-Fragment-Instanzen definiert als ein Tupel $G_I = (V, E, source, target, I, G_S, type, contains_I)$ mit*

(i) *V, einer Menge von Knoten und E, einer Menge von Kanten,*

(ii) *$source : E \to V$, einer Quellfunktion für Kanten,*

(iii) *$target : E \to V$, einer Zielfunktion für Kanten,*

(iv) *I, der Menge von Set-Fragment-Instanzen,*

(v) *$contains_I : I \times (V \cup I) \to \{wahr, falsch\}$, einer Enthaltensfunktion für Knoten und Set-Fragment-Instanzen ($contains_I(i, x)$ bedeutet i enthält x).*

(vi) *G_S, dem Set-Fragment-Graphen (ein getypter Graph mit Set Fragments),*

(vii) *einer Funktion $type_I : I \to S$ mit der Eigenschaft $\forall i_1, i_2 \in I : contains_I(i_1, i_2) \Rightarrow contains_S(type_I(i_1), type_I(i_2))$ und $\forall i \in I, x \in V : contains_I(i, x) \Rightarrow contains_S(type_I(i), type_V(x))$*

(viii) *$type : V \cup E \cup I \to V_T \cup E_T \cup I$, einer Typfunktion mit*

$$type(x) = \begin{cases} type_V(x) & \text{für } x \in V \\ type_E(x) & \text{für } x \in E \\ type_I(x) & \text{für } x \in I \end{cases}$$

Definition A.7 (Teilgraph)**.** *Für zwei getypte Graphen mit Set-Fragment-Instanzen $G_I = (V, E, source, target, I, G_S, type, contains_I)$ und $G'_I = (V', E', source', target', I', G'_S, type', contains'_I)$ ist G_I genau dann ein Teilgraph von G'_I (kurz $G_I \leq G'_I$), wenn gilt:*

- *$G_I = G'_I$ oder*
- *$G_I \neq G'_I$ und $G_S = G'_S$, $V \subseteq V'$, $E \subseteq E'$, $I \subseteq I'$, $type = type'|_{V \cup E \cup I}$, $source = source'|_E$, $target = target'|_E$, $contains = contains'|_{I \times (V \cup I)}$, wobei für eine Funktion $f : A \to C$ die Funktion $f|_B : B \to C$ für $B \subseteq A$ die Einschränkung der Funktion f auf den Wertebereich B ist.*

G_I ist genau dann ein echter Teilgraph von G'_I (kurz $G_I < G'_I$), wenn gilt:

- *$G_I \leq G'_I$ und $V \subset V' \vee E \subset E' \vee I \subset I'$*

Korollar A.8 (Die Teilgraphbeziehung ist transitiv)**.** *Für drei beliebige getypte Graphen mit Set-Fragment-Instanzen G_I, G'_I und G''_I gilt:*

$$G_I \leq G'_I \leq G''_I \Rightarrow G_I \leq G''_I,$$

d.h. die Teilgraphbeziehung $\leq$ ist transitiv. Ebenso gilt:

$$G_I < G'_I < G''_I \Rightarrow G_I < G''_I.$$

Definition A.9 (Indirekte Enthaltensbeziehung)**.** *Für einen getypten Graphen mit Set Fragments* $G_S = (V_S, E_S, source_S, target_S, T, type_S, S, contains_S)$ *sei eine Funktion* $in_{G_S} : S \times (V_S \cup S) \rightarrow \{wahr, falsch\}$ *definiert mit*

$$in_{G_S}(s, x) = contains_S(s, x) \vee (\exists s' \in S : contains_S(s, s') \wedge in_{G_S}(s', x))$$

Analog dazu sei auch für einen getypten Graphen mit Set-Fragment-Instanzen $G_I = (V_I, E_I, source_I, target_I, I, G_S, type_I, contains_I)$ *eine Funktion* $in_{G_I} : I \times (V_I \cup I) \rightarrow \{wahr, falsch\}$ *definiert mit*

$$in_{G_I}(i, x) = contains_I(i, x) \vee (\exists i' \in I : contains_I(i, i') \wedge in_{G_I}(i', x))$$

Definition A.10 (Instanziierung)**.** *Für einen getypten Graphen mit Set Fragments* $G_S = (V_S, E_S, source_S, target_S, T, type_S, S, contains_S)$ *ist die Instanziierung definiert als eine Abbildung auf einen getypten Graphen mit Set-Fragment-Instanzen* $G_I = (V_I, E_I, source_I, target_I, I, G_S, type_I, contains_I)$*. Die Abbildung ist definiert als Funktion* $instance : G_S \rightarrow G_I$ *(siehe Abb. A.29) mit*

(i) $V_I = V_S$, $E_I = E_S$, $source_I = source_S$, $target_I = target_S$,

(ii) $type_I(x) = \begin{cases} type_S(x) & \text{für } x \in V_I \cup E_I \\ type'(x) & \text{für } x \in I \end{cases}$
mit $type' : I \leftrightarrow S$ *eine totale, bijektive Abbildung von* I *nach* S*, sodass gilt:*

(iii) $\forall i \in I, j \in I : (contains_I(i, j) \Leftrightarrow contains_S(type_I(i), type_I(j)))$,

(iv) $\forall i \in I, v \in V_I = V_S : (contains_I(i, v) \Leftrightarrow contains_S(type_I(i), v))$.

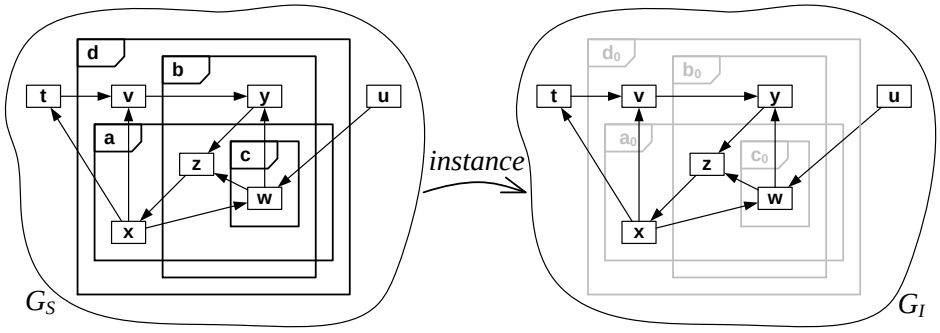


Abbildung A.29: Veranschaulichung der Instanziierung laut Definition A.10

Die Instanziierung laut Def. A.10 ist im Prinzip eine 1-zu-1-Abbildung eines getypten Graphen mit Set Fragments auf einen getypten Graphen mit Set-Fragment-Instanzen, wobei alle Set Fragments durch Set-Fragment-Instanzen ersetzt werden, welche das ersetzte Set Fragment als Typ erhalten. Die Instanziierung ist eine von drei Operationen bzw. Abbildungen bei einer Auffaltung (siehe Abb. 3.34, S. 71).

Definition A.11 (Projektion)**.** *Für einen getypten Graphen mit Set-Fragment-Instanzen* $G_I = (V_I, E_I, source_I, target_I, I, G_S, type_I, contains_I)$ *mit dem Set-Fragment-Graphen* $G_S = (V_S, E_S, source_S, target_S, T, type_S, S, contains_S)$ *und dem Typgraphen* $T = (V_T, E_T, source_T, target_T)$ *ist die Projektion* π *auf einen getypten Graphen* G_T *definiert als Funktion* $\pi : G_I \to G_T$ *(siehe Abb. A.30) mit*

$$\pi(G_I) = G_T = (V_I, E_I, source_I, target_I, T, type_I|_{V_I \cup E_I})$$

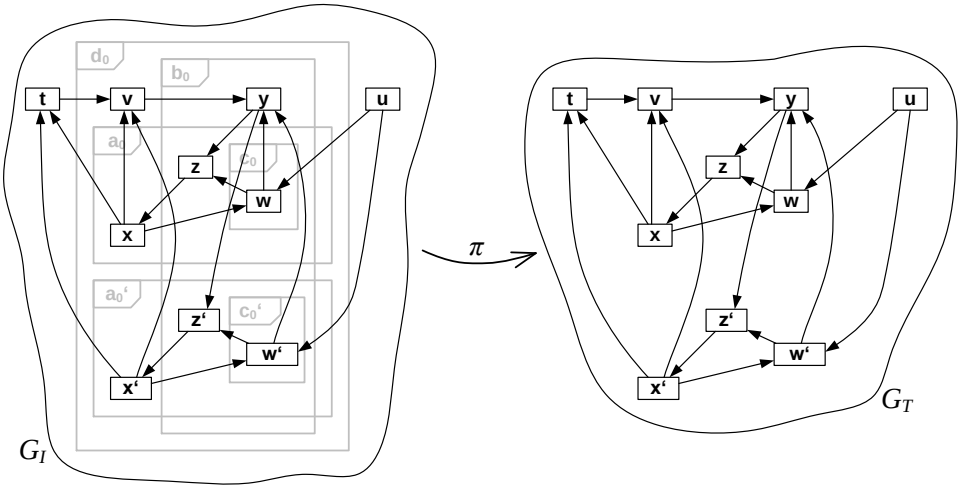


Abbildung A.30: Veranschaulichung der Projektion laut Definition A.11

Die Projektion eines getypten Graphen mit Set-Fragment-Instanzen auf einen getypten Graphen ohne Set-Fragment-Instanzen oder Set Fragments (Def. A.11) entspricht dem Weglassen von Set-Fragment-Instanzen, wobei alle Knoten und Kanten mit ihren Eigenschaften erhalten bleiben. Die Projektion entspricht der letzten Operation bei einer Auffaltung (siehe Abb. 3.34, S. 71).

Definition A.12 (Hinzufügen einer Set-Fragment-Instanz)**.** *Gegeben sei ein getypter Graph mit Set-Fragment-Instanzen* G_I *und eine Set-Fragment-Instanz* $i \in I$ *von* G_I*, dann ist für* $G_I = (V_I, E_I, source_I, target_I, I, G_S, type_I, contains_I)$ *mit dem Set-Fragment-Graphen* $G_S = (V_S, E_S, source_S, target_S, T, type_S, S, contains_S)$ *und dem Typgraphen* $T = (V_T, E_T, source_T, target_T)$ *das Hinzufügen einer Set-Fragment-Instanz definiert durch die Abbildung* $add : (G_I, i) \to G'_I$*, wobei* $G'_I = (V'_I, E'_I, source'_I, target'_I, I', G'_S, type'_I, contains'_I)$ *ein anderer getypter Graph mit Set-Fragment-Instanzen ist und für add Folgendes gilt (siehe Abb. A.31):*

(i) $G_I < G'_I$, $I \subset I'$ *und* $I' = I \cup I_{add}$, $V'_I = V_I \cup V_{add}$, $E'_I = E_I \cup E_{add}$

(ii) *Sei* $U_I = \{j \in I | j = i \vee in_{G_I}(i, j)\}$*. Dann ist* I_{add} *definiert als eine Menge von zusätzlichen Set-Fragment-Instanzen* $I_{add} = \{j' \notin I\}$*, sodass es eine totale, bijektive Abbildung* $img_I : U_I \leftrightarrow I_{add}$ *gibt mit*
$\forall j \in U_I : type'_I(img_I(j)) = type_I(j)$ *und den Eigenschaften* (v) *bis* (vii)

(iii) *Sei* $U_V = \{v \in V_I | in_{G_I}(i, v)\}$*. Dann ist* V_{add} *definiert als eine Menge von zusätzlichen Knoten* $V_{add} = \{v' \notin V_I\}$*, sodass es eine totale, bijektive Abbildung* $img_V : U_V \leftrightarrow V_{add}$ *gibt mit* $\forall v \in U_V : type'_I(img_V(v)) = type_I(v)$ *und den Eigenschaften* (vi) *bis* (viii)

(iv) *Sei* $U_E = \{e \in E_I | in_{G_I}(i, source_I(e)) \vee in_{G_I}(i, target_I(e))\}$. *Dann ist* E_{add} *definiert als eine Menge von zusätzlichen Kanten* $E_{add} = \{e' \notin E_I\}$, *sodass es eine totale, bijektive Abbildung* $img_E : U_E \leftrightarrow E_{add}$ *gibt mit* $\forall e \in U_E : type'_I(img_E(e)) = type_I(e)$ *und der Eigenschaft* (viii)

(v) $\forall j \in I : (contains_I(j, i) \Rightarrow contains'_I(j, img_I(i)))$

(vi) $\forall j, k \in U_I : (contains_I(j, k) \Rightarrow contains'_I(img_I(j), img_I(k)))$ *und* $\forall j \in U_I, v \in U_V : (contains_I(j, v) \Rightarrow contains'_I(img_I(j), img_V(v)))$

(vii) $\forall l, j \in U_I, k \in I \setminus U_I :$ $(contains_I(j, l) \wedge contains_I(k, l) \Rightarrow contains'_I(k, img_I(l)))$ *und* $\forall v \in U_V, j \in U_I, k \in I \setminus U_I :$ $(contains_I(j, v) \wedge contains_I(k, v) \Rightarrow contains'_I(k, img_V(v))$

(viii) $\forall e \in U_E :$

$$source'_I(img_E(e)) = \begin{cases} img_V(source_I(e)) & \text{für } source_I(e) \in U_V \\ source_I(e) & \text{für } source_I(e) \notin U_V \end{cases}$$

$\wedge$

$$target'_I(img_E(e)) = \begin{cases} img_V(target_I(e)) & \text{für } target_I(e) \in U_V \\ target_I(e) & \text{für } target_I(e) \notin U_V \end{cases}$$

Anmerkung A.13. *Nachfolgend ergänze ich die in der Definition A.12 beschriebenen Aussagen über die Abbildung add und erkläre deren Bedeutung:*

(i) G_I *ist echter Teilgraph von* G'_I *und es gibt mindestens eine Set-Fragment-Instanz in* G'_I, *die nicht in* G_I *ist.*

(ii) U_I *enthält* i *und alle darin direkt oder indirekt enthaltenen Set-Fragment-Instanzen.* I_{add} *enthält Kopien dieser Set-Fragment-Instanzen mit demselben Typ.*

(iii) U_V *enthält alle in* i *direkt oder indirekt enthaltenen Knoten.* V_{add} *enthält Kopien dieser Knoten mit demselben Typ.*

(iv) U_E *enthält alle Kanten, die mit einem Knoten verbunden sind, der direkt oder indirekt in* i *enthalten ist.* E_{add} *enthält zu jeder dieser Kanten eine andere Kante mit demselben Typ.*

(v) *Das Bild* $img_I(i)$ *von* i *ist in denselben Set-Fragment-Instanzen enthalten wie* i.

(vi) *Für alle kopierten Set-Fragment-Instanzen und Knoten gelten die gleichen Enthaltensbeziehungen wie für ihre Urbilder.*

(vii) *Ist eine Set-Fragment-Instanz oder ein Knoten* x *in zwei Set-Fragment-Instanzen* j *und* k, *wobei zu* k *keine Kopie existiert, dann enthält* k *auch die Kopie von* x.

(viii) *Eine Kante in* E_{add} *ist entweder eine Kopie einer Kante in* U_E *und verbindet zwei kopierte Knoten aus* V_{add} *miteinander oder verbindet einen kopierten Knoten aus* V_{add} *mit einem nicht kopierten Knoten aus* $V_I \setminus U_V$.

Das Hinzufügen einer Set-Fragment-Instanz laut Definition A.12 entspricht im Wesentlichen dem Kopieren einer Set-Fragment-Instanz samt ihrem Inhalt, also der enthaltenen Set-Fragment-Instanzen und Knoten. Diese Operation ist die letzte der in Abb. 3.34 (S. 71) dargestellten Operationen einer Auffaltung.

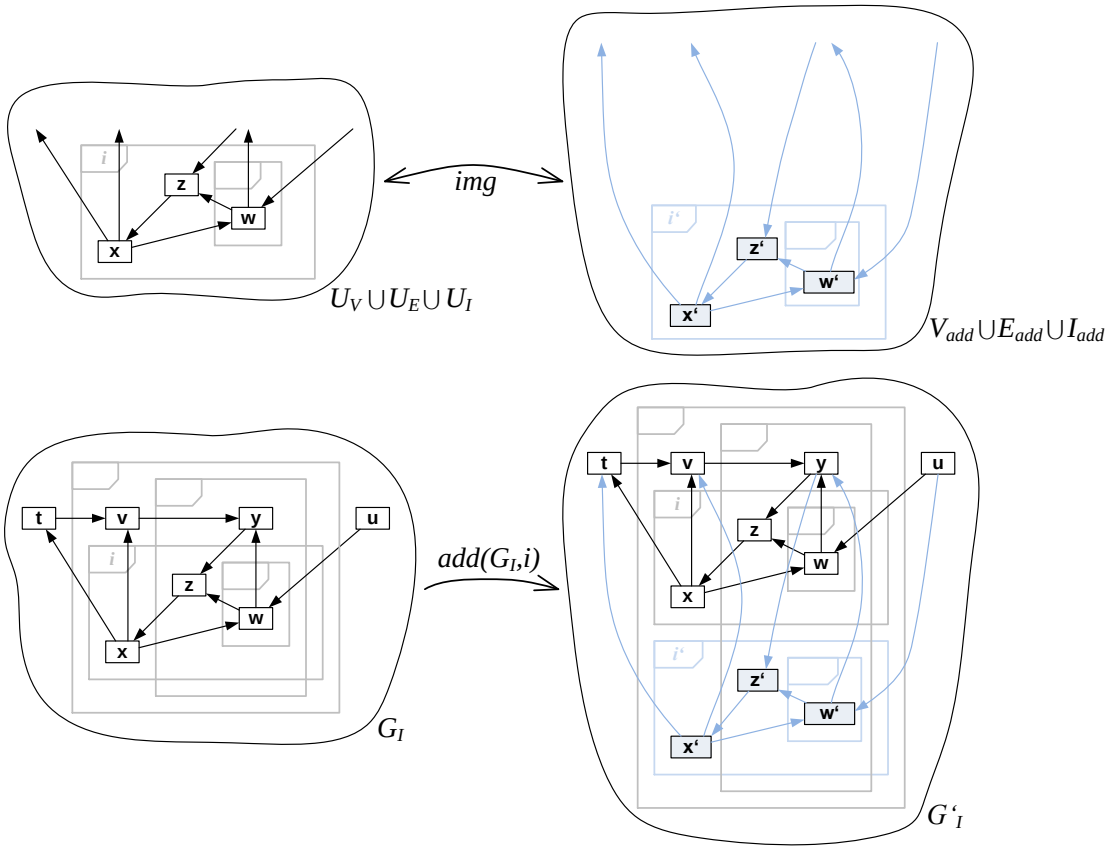


Abbildung A.31: Veranschaulichung der Abbildung $add(G_I, i)$ laut Definition A.12 mit einer bijektiven Abbildung $img : U_V \cup U_E \cup U_I \leftrightarrow V_{add} \cup E_{add} \cup I_{add}$,

$$img(x) = \begin{cases} img_V(x) & \text{für } x \in U_V \\ img_E(x) & \text{für } x \in U_E \\ img_I(x) & \text{für } x \in U_I \end{cases}$$

Anmerkung A.14. *Man beachte, dass die Abbildung $add(G_I, i) : G_I \to G'_I$ einige Konsistenz-erhaltende Bedingungen erfüllt, insb. folgende:*

(i) *Jede Kante hat (weiterhin) einen Quell- und einen Zielknoten. Das ist sichergestellt, da jede Kante in $E'_I = E_I \cup E_{add}$ (Def. A.12 (i)) entweder schon in E_I enthalten war und damit Quell- und Zielknoten besitzt oder in $E_{add} = img_E(U_E)$ liegt und damit durch Def. A.12 (viii) Quell- und Zielknoten definiert sind.*

(ii) *Die Knoten und Kanten in der hinzugefügten Set-Fragment-Instanz i' sind Typ-erhaltend und deckungsgleich (isomorph) zu ihren Vorbildern in der ursprünglichen Set-Fragment-Instanz i. Das stellen die bijektiven Abbildungen $img_V : U_V \leftrightarrow V_{add}$ und $img_E : U_E \leftrightarrow E_{add}$ sicher (Def. A.12 (iii), (iv)).*

Einige Beispiele für Abbildungen nach Def. A.12 sind zur Veranschaulichung in der Abb. A.32 dargestellt. Bei Überschneidungen von Set-Fragment-Instanzen bleibt die Enthaltensbeziehung der replizierten Set-Fragment-Instanzen zu nicht kopierten Set-Fragment-Instanzen erhalten. Komplett in der kopierten Set-Fragment-Instanz enthaltene Kanten werden mitkopiert und verbinden kopierte Kno-

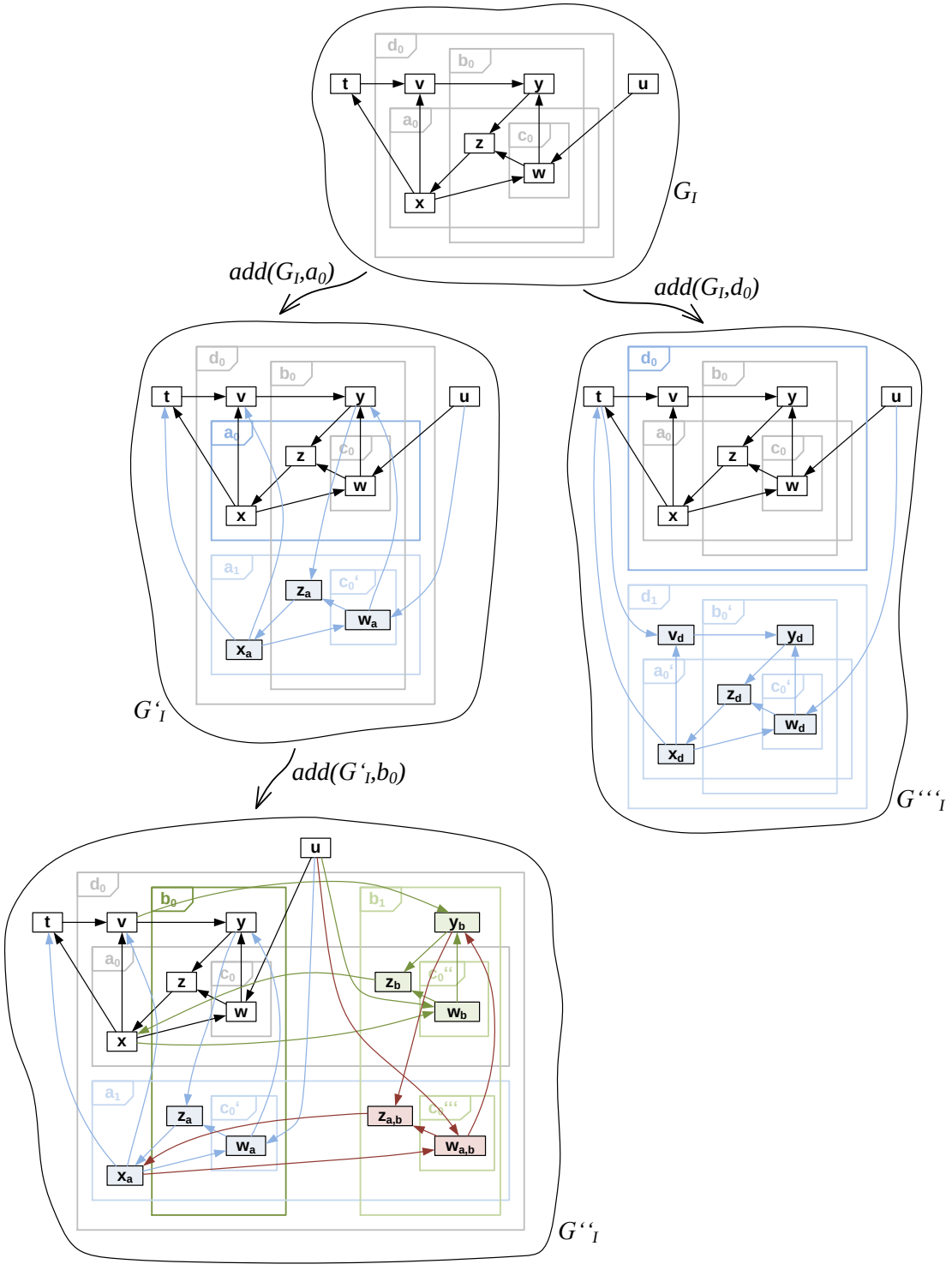


Abbildung A.32: Beispiele für Abbildungen $add(G_I, i)$ nach Definition A.12

ten auf gleiche Weise. Kanten, die einen Knoten innerhalb der kopierten Set-Fragment-Instanz mit einem Knoten außerhalb der kopierten Set-Fragment-Instanz verbinden, werden ebenfalls repliziert und verbinden den kopierten Knoten innerhalb der kopierten Set-Fragment-Instanz mit dem nicht kopierten Knoten außerhalb der kopierten Set-Fragment-Instanz.

Nachdem die einzelnen Schritte einer Auffaltung in Form von Abbildungen definiert sind, kann nun die Auffaltung wie folgt definiert werden.

Definition A.15 (Auffaltung eines getypten Graphen mit Set Fragments)**.** *Eine Auffaltung eines getypten Graphen mit Set Fragments G_S in einen getypten Graphen (ohne Set Fragments) G_T ist definiert als eine Abbildung $a_F : G_S \to G_T$, sodass es eine Folge von Set-Fragment-Instanzen F gibt mit:*

$$\begin{aligned} a_F(G_S) &= \pi(instance(G_S)) \text{ für die leere Folge } F = F_0 \text{ oder} \\ a_F(G_S) &= \pi(add(\ldots add(add(instance(G_S), i_1), i_2) \ldots, i_n)) \text{ für eine Folge} \\ &\quad F = (i_1, i_2, \ldots, i_n) \text{ für ein beliebiges } n \in \mathbb{N}^+ \end{aligned}$$

Anmerkung A.16. *Betrachtet man eine beliebige Auffaltung $a_F : G_S \to G_T$ laut Def. A.15, so erfüllt sie folgende Konsistenzeigenschaften.*

(i) *Jede Kante in G_T hat einen Quell- und einen Zielknoten. Die instance-Abbildung stellt das durch das Erhalten aller Knoten und Kanten sicher (Def. A.10* (i)*). Das gleiche gilt für die Projektion π (Def. A.11). Für jede add-Abbildung ist das durch den Punkt* (i) *der Anmerkung A.14 sichergestellt.*

(ii) *Jeder Knoten und jede Kante aus dem ursprünglichen getypten Graphen mit Set Fragments G_S (der Musterspezifikation) ist auch in G_T (einer Implementierungsvariante) enthalten. Das ist dadurch sichergestellt, dass die Abbildungen instance, add und π die Knoten und Kanten aus G_S erhalten (siehe Def. A.10* (i)*, Def. A.12* (i) *und Def. A.11).*

(iii) *Jeder Knoten und jede Kante in G_T (ein Teil einer Implementierungsvariante) kann seinem Urbild, d.h. genau einem Knoten bzw. genau einer Kante in G_S (einer Rolle der Musterspezifikation) zugeordnet werden, aus dem der Knoten bzw. die Kante in G_T entstanden ist. Für alle Knoten und Kanten, die bereits in G_S enthalten waren (diese Knoten und Kanten bleiben laut Punkt* (ii) *in G_T erhalten), ist diese Eigenschaft erfüllt, weil ihr Urbild sie selbst sind. Für alle anderen, also alle durch eine add-Abbildung hinzugefügten Knoten und Kanten ist ihr Urbild durch die Umkehrfunktionen img_V^{-1} und img_E^{-1} der bijektiven Abbildungen img_V und img_E definiert (Def. A.12* (iii), (iv)*). Ist G_T durch mehrere add-Abbildungen mit den Funktionen $img_{V_1} \cdots img_{V_n}$ und $img_{E_1} \cdots img_{E_n}$ entstanden, so ist das Urbild durch eine Hintereinanderanwendung der Umkehrfunktionen definiert, also durch $img_{V_n}^{-1} \circ \cdots \circ img_{V_1}^{-1}$ und $img_{E_n}^{-1} \circ \cdots \circ img_{E_1}^{-1}$.*

(iv) *Für alle Knoten und Kanten in G_S, die sich in Set Fragments befinden, enthält G_T mindestens ein, aber beliebig viele isomorphe Abbilder. Bei der Instanziierung und der Projektion bleiben alle Knoten erhalten (Def. A.10* (i) *und Def. A.11), was die Isomorphie für diese Abbildungen sicherstellt.*

Bei der add-Abbildung wird die Isomorphie zwischen den Teilgraphen zu einer Set-Fragment-Instanz i und seiner Kopie i′ (wobei die Knoten, welche Ziel oder Quelle einer Kante sind, welche in Bezug auf die Set-Fragment-Instanz i bzw. i′ eingehend oder ausgehend ist, zu den Teilgraphen gehören) durch Erhalten der bisherigen Knoten und Kanten (Def. A.12 (i)*), durch die bijektiven Abbildungen img_V und img_E (Def. A.12* (iii), (iv)*) und durch das Verbinden neuer und alter Knoten im Fall von Set-Fragment-Instanz-Grenzen überschreitenden Kanten (Def. A.12* (viii)*). Bei allen anderen Knoten und Kanten bleibt die zuvor vorliegende Isomorphie erhalten, weil sie unverändert in den neuen Graphen übernommen werden (Def. A.12* (i)*).*

Definition A.17 (Semantik eines getypten Graphen mit Set Fragments)**.** *Die Semantik eines getypten Graphen mit Set Fragments G_S bzw. seine Interpretation $\Im(G_S)$ wird definiert als die Menge aller möglichen Auffaltungen des Graphen:*

$$\Im(G_S) = \bigcup_F \{G_T | G_T = a_F(G_S)\}$$

Mit der Definition A.17 wird die Semantik von Set Fragments ausgedrückt als die Menge aller Möglichkeiten, einen getypten Graphen mit Set Fragments (eine Musterspezifikation) in einen getypten Graphen ohne Set Fragments (eine Implementierungsvariante) zu überführen. Anders ausgedrückt sind das alle erlaubten Möglichkeiten, die durch Set Fragments markierten Teilgraphen eines getypten Graphen mit Set Fragments zu vervielfältigen.

Anmerkung A.18 (Vererbung und attributierte Graphen)**.** *In meiner Formalisierung verwende ich nicht-attributierte getypte Graphen ohne Vererbung. Ehrig et al. haben für getypte attributierte Graphen und Transformationen darauf gezeigt, dass es zu jeder Graphtransformation und Grammatik basierend auf einem attributierten getypten Graphen mit Vererbung eine äquivalente Graphtransformation und Grammatik basierend auf einem attributierten getypten Graphen ohne Vererbung gibt [EEPT06, Kap. 13]. Folglich reicht es für meine Formalisierung aus, getypte Graphen ohne Verebung zu betrachten.*

Für die Definition einer Auffaltung eines getypten Graphen mit Set Fragments (Def. A.15) habe ich Knoten- und Kantenattribute nicht betrachtet. Meine Definitionen lassen sich jedoch auf attributierte Graphen analog zu den Definitionen von Ehrig et al. [EEPT06, Kap. 8] erweitern. In dem Fall würden beim Vervielfältigen von Knoten und Kanten laut Definition A.12 nicht nur der Typ, sondern auch die Attribute übernommen und die Attributwerte kopiert werden. Bei der Instanziierung (Def. A.10) und der Projektion (Def. A.11) würden Attribute entsprechend mitberücksichtigt werden.

Anhang B

Musterspezifikationen

Alle zu Evaluationszwecken erstellten Musterspezifikationen sind im Folgenden aufgeführt. Dazu gehören insbesondere Spezifikationen aller 23 Muster der Gang of Four [GHJV95] (Abschnitt B.1), aber auch einige exemplarische Spezifikationen anderer Entwurfsmuster sowie die einiger Architekturmuster und Idiome (Abschnitt B.2).

B.1 Spezifikation der Gang-of-Four-Muster

Die Entwurfsmuster der Gang of Four sind analog zu ihrer Auflistung im zugehörigen Buch [GHJV95], also in gleicher Reihenfolge und aufgeteilt in die Kategorien *Erzeugungsmuster*, *Strukturmuster* und *Verhaltensmuster* aufgeführt.

B.1.1 Erzeugungsmuster (Creational Patterns)

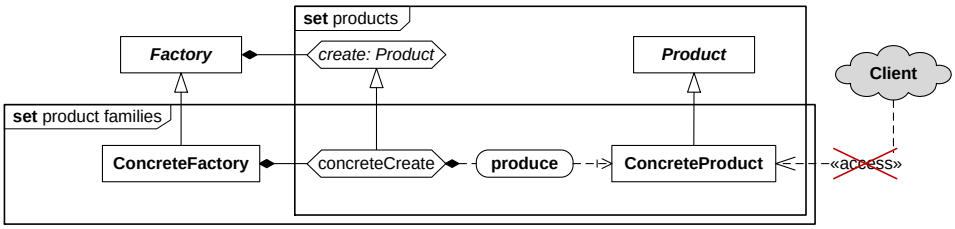


Abbildung B.1: Spezifikation des GoF-Entwurfsmusters „Abstract Factory“

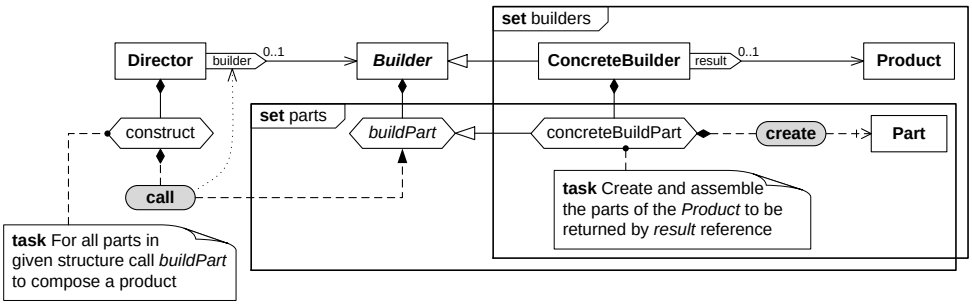


Abbildung B.2: Spezifikation des GoF-Entwurfsmusters „Builder“

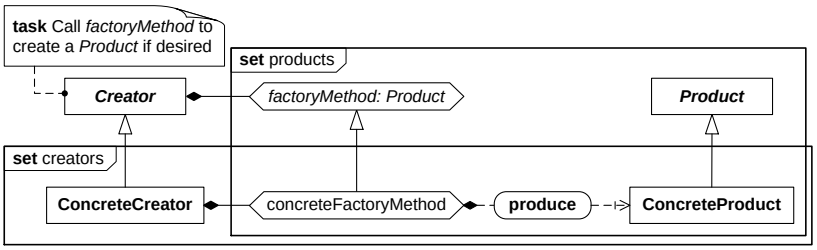


Abbildung B.3: Spezifikation des GoF-Entwurfsmusters „Factory Method“

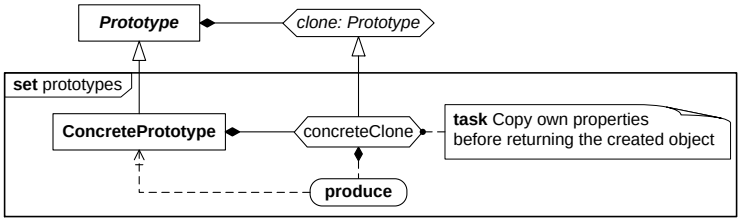


Abbildung B.4: Spezifikation des GoF-Entwurfsmusters „Prototype“

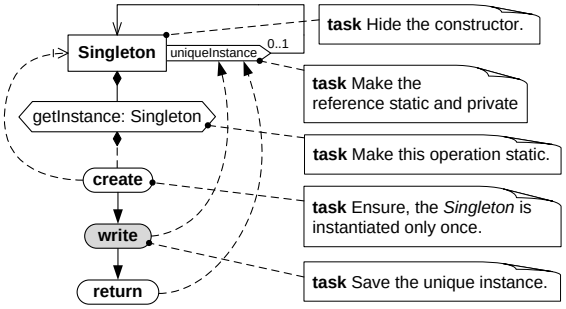


Abbildung B.5: Spezifikation des GoF-Entwurfsmusters „Singleton“

B.1.2 Strukturmuster (Structural Patterns)

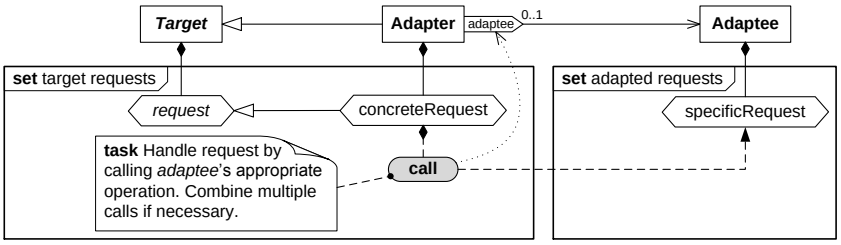


Abbildung B.6: Spezifikation des GoF-Entwurfsmusters „Object Adapter“

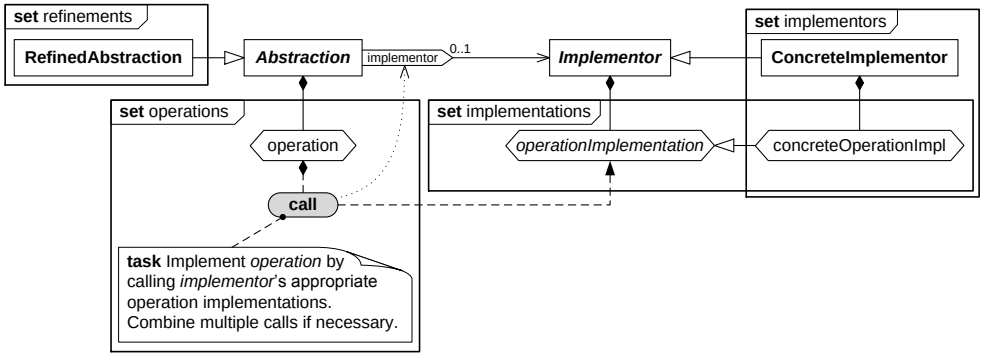


Abbildung B.7: Spezifikation des GoF-Entwurfsmusters „Bridge“

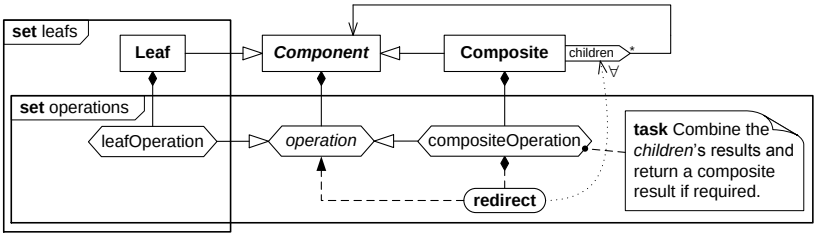


Abbildung B.8: Spezifikation des GoF-Entwurfsmusters „Composite“ mit separaten Schnittstellen für *Leaf* und *Composite*

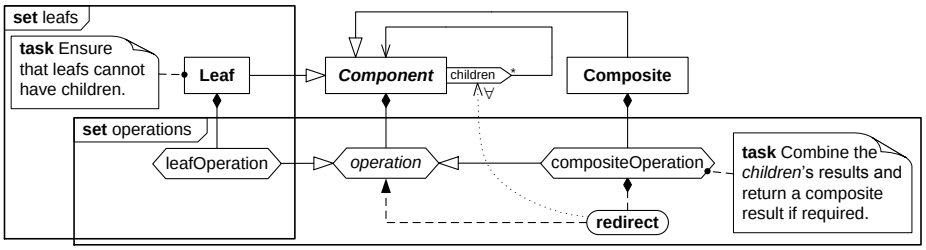


Abbildung B.9: Spezifikation des GoF-Entwurfsmusters „Composite“ mit einheitlicher Schnittstelle für *Leaf* und *Composite*

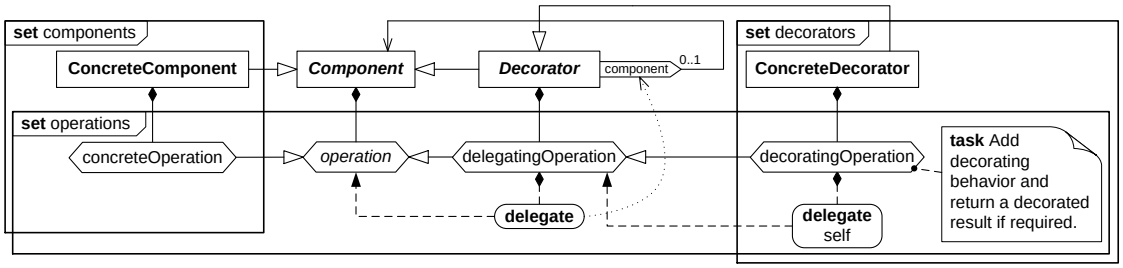


Abbildung B.10: Spezifikation des GoF-Entwurfsmusters „Decorator“

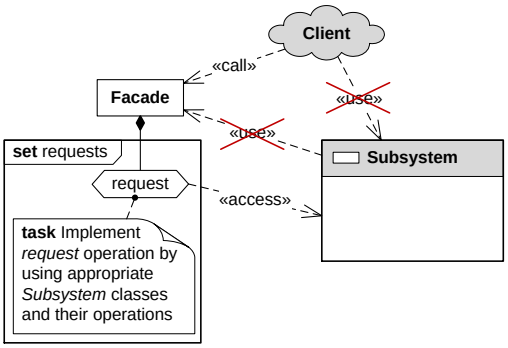


Abbildung B.11: Spezifikation des GoF-Entwurfsmusters „Facade“

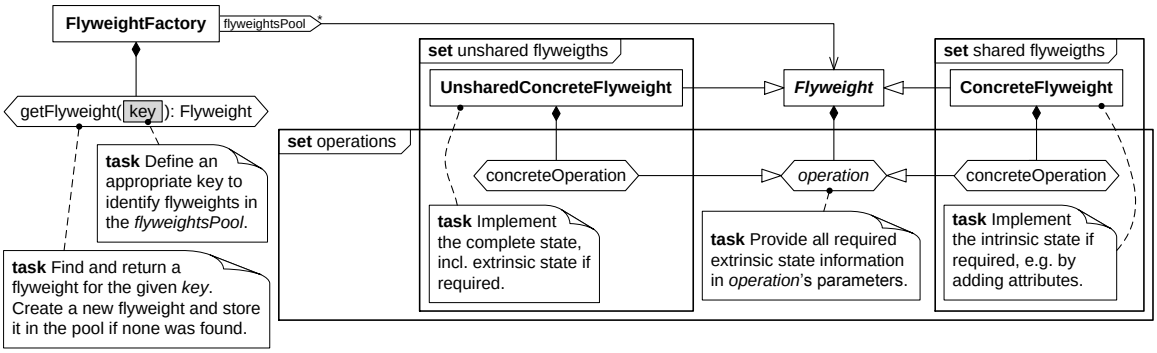


Abbildung B.12: Spezifikation des GoF-Entwurfsmusters „Flyweight“

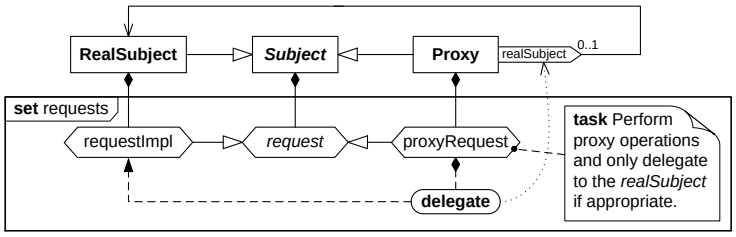


Abbildung B.13: Spezifikation des GoF-Entwurfsmusters „Proxy“

B.1.3 Verhaltensmuster (Behavioral Patterns)

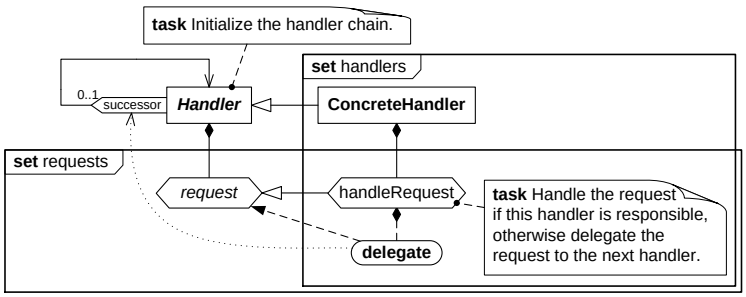


Abbildung B.14: Spezifikation des GoF-Entwurfsmusters „Chain of Responsibility“

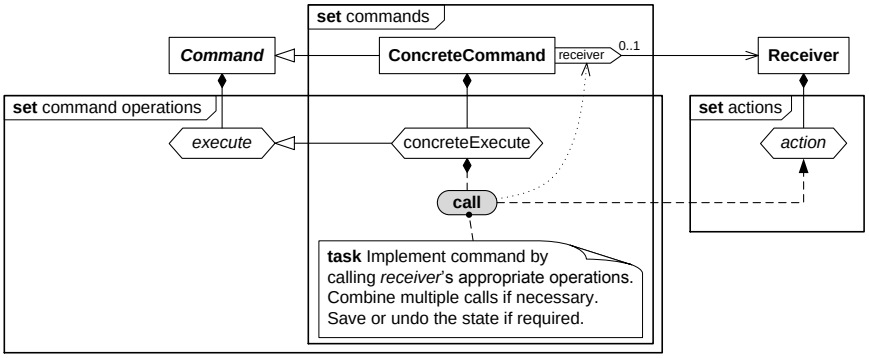


Abbildung B.15: Spezifikation des GoF-Entwurfsmusters „Command“

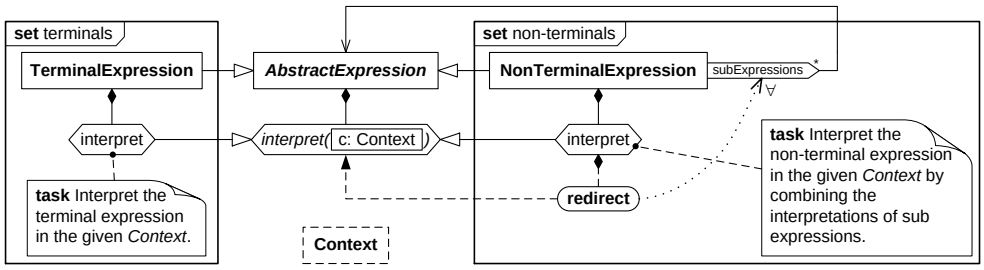


Abbildung B.16: Spezifikation des GoF-Entwurfsmusters „Interpreter“

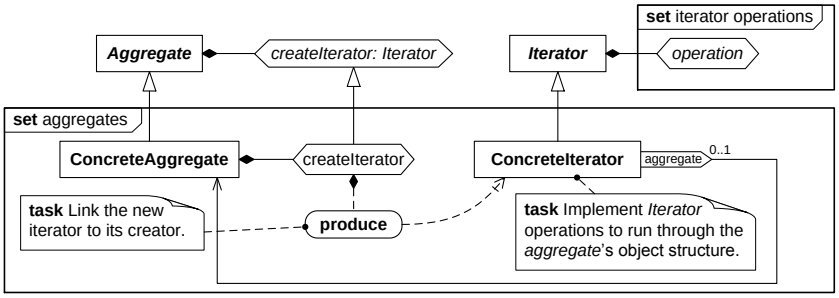


Abbildung B.17: Spezifikation des GoF-Entwurfsmusters „Iterator“

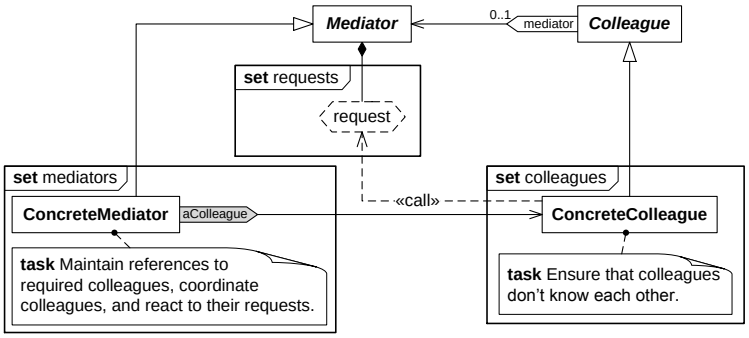


Abbildung B.18: Spezifikation des GoF-Entwurfsmusters „Mediator“

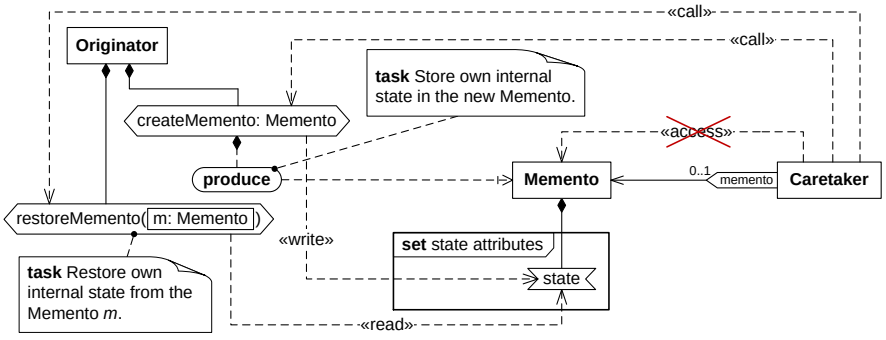


Abbildung B.19: Spezifikation des GoF-Entwurfsmusters „Memento“

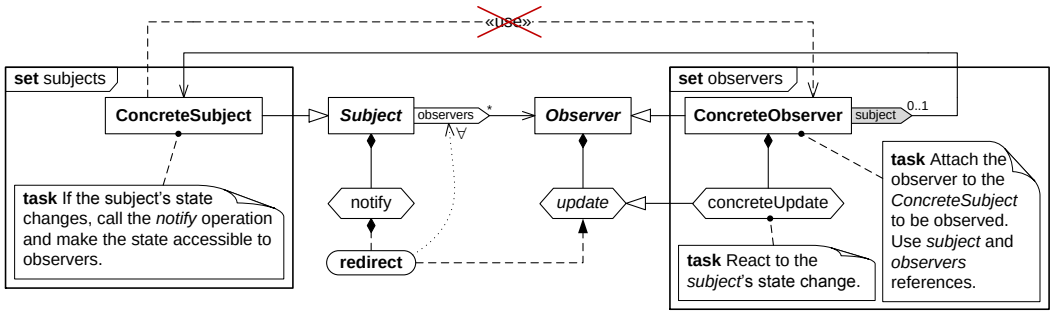


Abbildung B.20: Spezifikation des GoF-Entwurfsmusters „Observer“ – pull-Variante mit beliebig vielen Subjekten je Beobachter

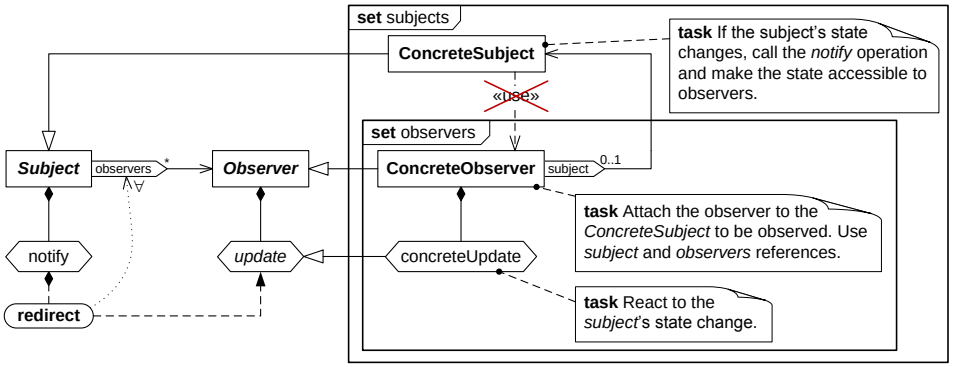


Abbildung B.21: Spezifikation des GoF-Entwurfsmusters „Observer“ – pull-Variante mit nur einem Subjekt je Beobachter

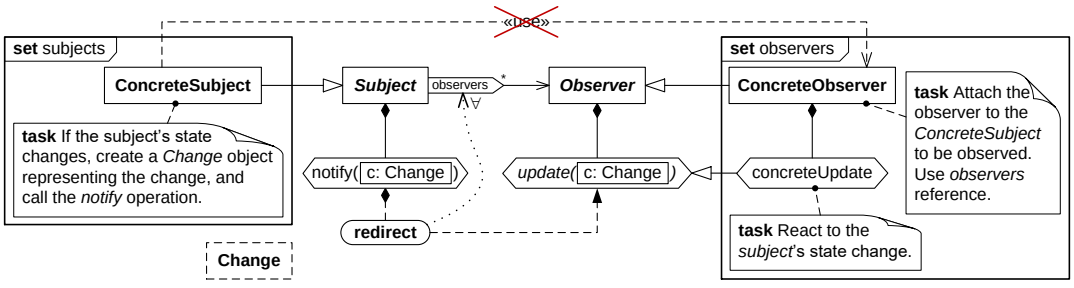


Abbildung B.22: Spezifikation des GoF-Entwurfsmusters „Observer“ – push-Variante mit beliebig vielen Subjekten je Beobachter

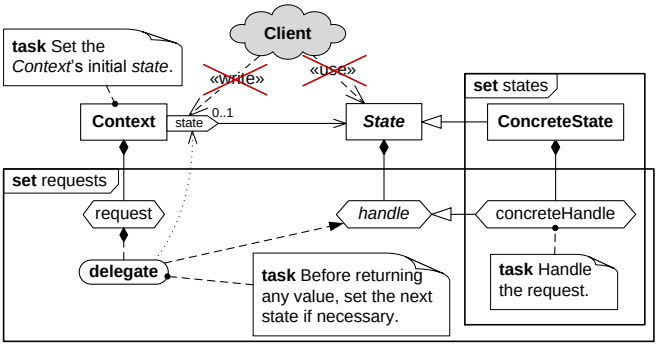


Abbildung B.23: Spezifikation des GoF-Entwurfsmusters „State“ – Variante 1: Kontext setzt nächsten Zustand

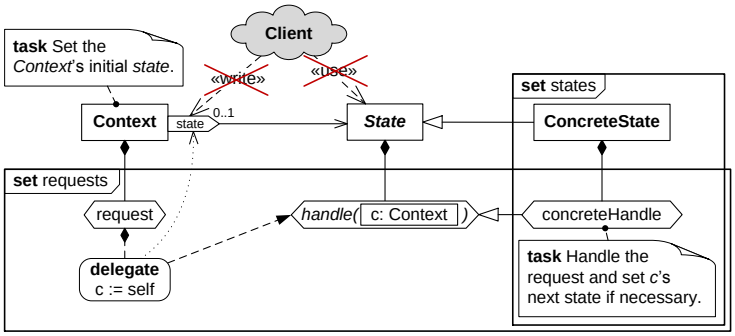


Abbildung B.24: Spezifikation des GoF-Entwurfsmusters „State“ – Variante 2: aktueller Zustand setzt nächsten Zustand

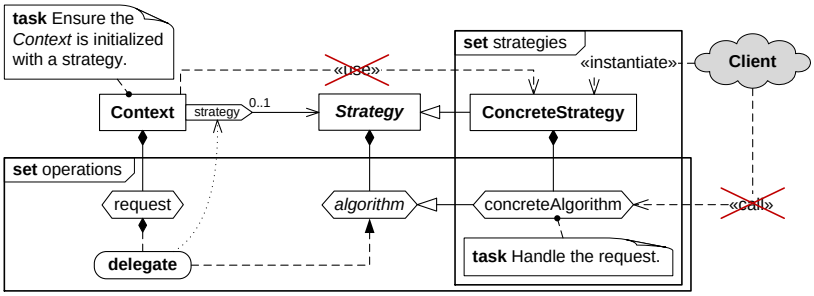


Abbildung B.25: Spezifikation des GoF-Entwurfsmusters „Strategy“

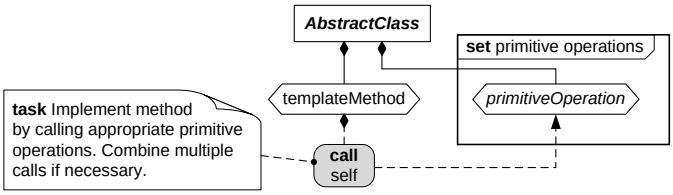


Abbildung B.26: Spezifikation des GoF-Entwurfsmusters „Template Method“

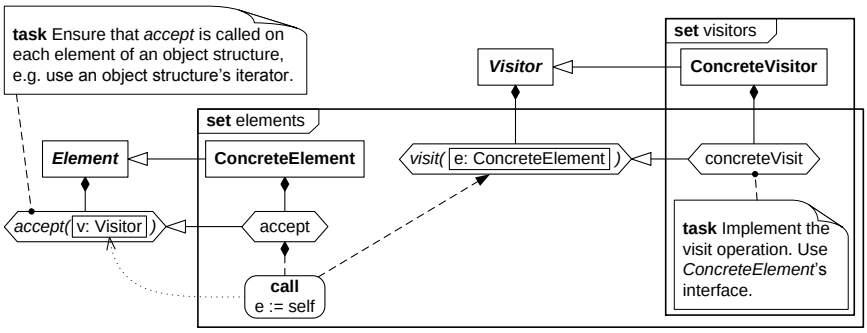


Abbildung B.27: Spezifikation des GoF-Entwurfsmusters „Visitor“

B.2 Spezifikation weiterer Entwurfsmuster, Architekturmuster und Idiome

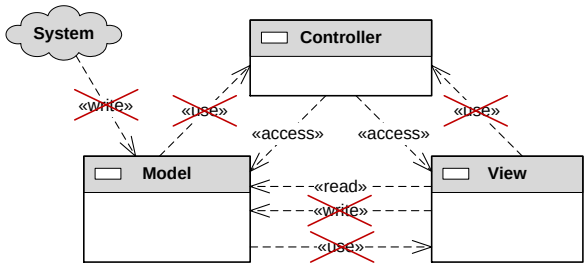


Abbildung B.28: Spezifikation des Architekturmusters „Model-View-Controller (MVC)“ [BMR+96, Fow02]

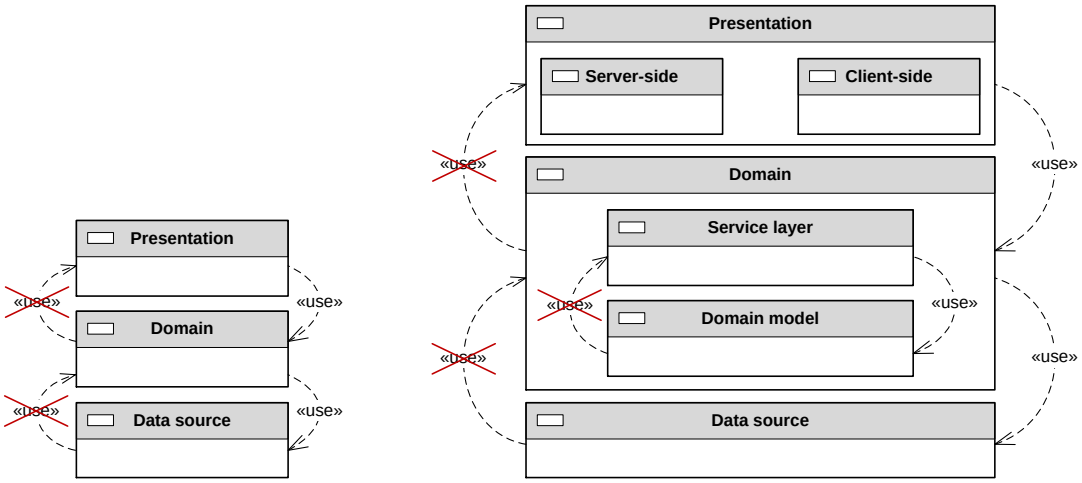


Abbildung B.29: Spezifikation zweier Ausprägungen des „Layers“-Architekturstils [BMR+96]. Links die von Fowler propagierte Form [Fow02, Kap. 8], rechts die erweiterte Form basierend auf Fowlers Aussagen [Fow02, Tab. 8.2, S. 104] und dem Muster „Service Layer“ [Fow02]

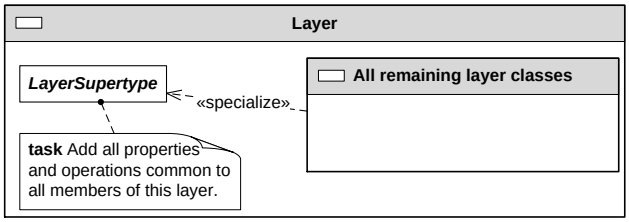


Abbildung B.30: Spezifikation des Enterprise Application Architecture Patterns „Layer Supertype“ [Fow02]

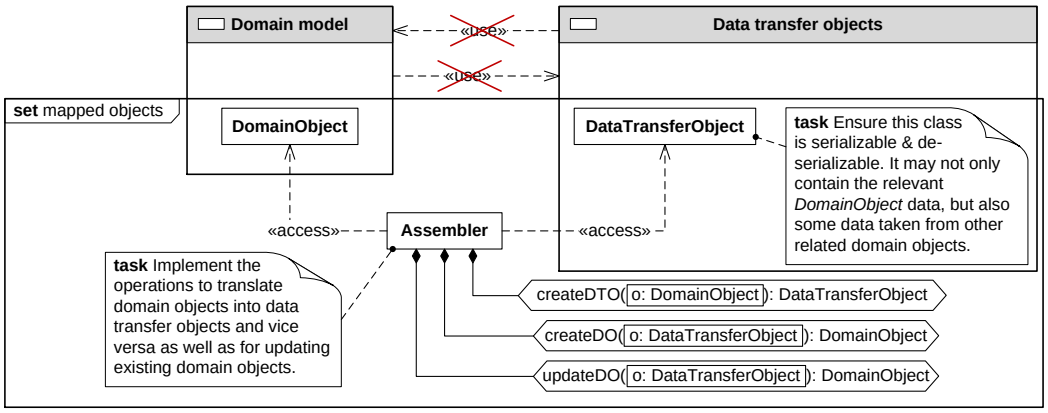


Abbildung B.31: Spezifikation des Enterprise Application Architecture Patterns „Data Transfer Object“ [Fow02]

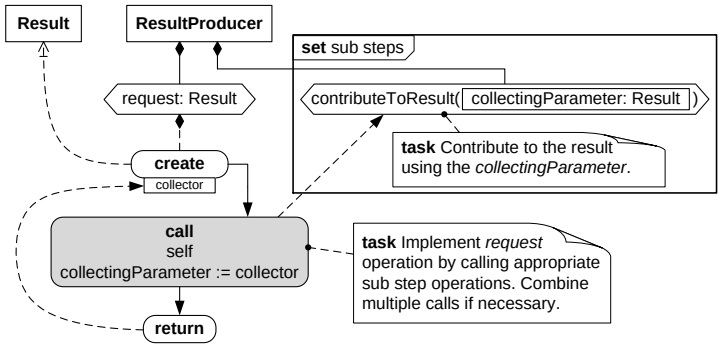


Abbildung B.32: Spezifikation des SmallTalk Best Practice Patterns „Collecting Parameter“ [Bec96]

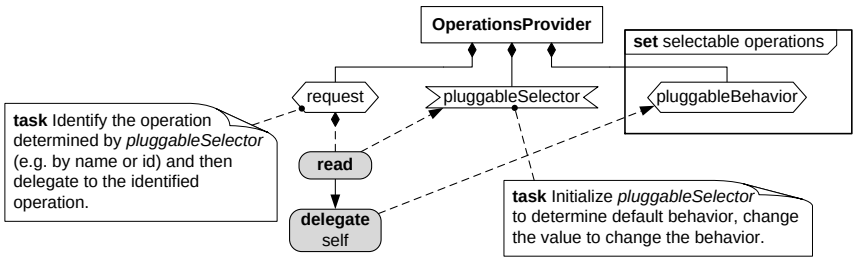


Abbildung B.33: Spezifikation des SmallTalk Best Practice Patterns „Pluggable Selector“ [Bec96]

Anhang C

Von der Musterspezifikation zur Musterimplementierung

Die Anwendung eines Entwurfsmusters wird in Kapitel 5 eingeführt. Ergänzend dazu gehe ich im Folgenden auf einige Details bei diesem Vorgehen ein. Insbesondere beschreibe ich detaillierter als bisher wie eine Anwendungsstelle im Entwurfsmodell markiert wird (Abschnitt C.1), wie eine Repräsentation der gewünschten Musterimplementierung durch die sogenannte Auffaltung entsteht (Abschnitt C.2) und schließlich wie diese Repräsentation der Musterimplementierung in das Entwurfsmodell übersetzt wird (Abschnitt C.3).

C.1 Korrespondenzmodell und Anwendungsstelle erstellen

Das Korrespondenzmodell wird schrittweise aufgebaut. Nachdem ein Entwickler eine Musterspezifikation für eine Musteranwendung ausgewählt hat, wird automatisch ein initiales Modell der zugehörigen Anwendungsstelle, also das Korrespondenzmodell, erstellt (Schritt 3 in Abb. 5.10, S. 115). Dieses enthält Korrespondenzknoten für jede Rolle der Musterspezifikation. Ein Korrespondenzknoten verknüpft eine Rolle der Musterspezifikation mit den Elementen im Softwareentwurf, welche die Rolle einnehmen sollen (oder bereits einnehmen). Spezielle Korrespondenzknoten ordnen einem Set Fragment der Musterspezifikation alle Ausprägungen der in einem Set Fragment beschrieben Struktur zu, also alle zugehörigen Set-Fragment-Instanzen. Details zur Struktur des Korrespondenzmodells und der Modellierung von Musterimplementierungen sind dem Kapitel 4 zu entnehmen.

In einem initialen Korrespondenzmodell werden zu allen Rollen und Set Fragments der Musterspezifikation entsprechende Korrespondenzknoten erstellt und mit den Rollen bzw. Set Fragments verknüpft. Erst danach werden die Knoten schrittweise mit Teilen des Entwurfs verknüpft. Ordnet der Entwickler Teile des Entwurfs einer Musterrolle zu (Schritt 5 in Abb. 5.10, S. 115), wird eine entsprechende Korrespondenz im Korrespondenzmodell aufgebaut und der entsprechende Korrespondenzknoten mit den Entwurfselementen verknüpft (Schritt 6 in Abb. 5.10, S. 115).

Wählt ein Entwickler zum Beispiel das Muster Strategy zur Anwendung aus, dann wird ein initiales Korrespondenzmodell erstellt, welches der Skizze in der Abb. C.1 entspricht. Hier ist das Korrespondenzmodell (welches Teil des Dekorationsmodells ist) grün dargestellt, die Musterspezifikation ist oberhalb davon schwarz abgebildet. Die Anwendungsstelle und die zugehörigen Korrespondenzen

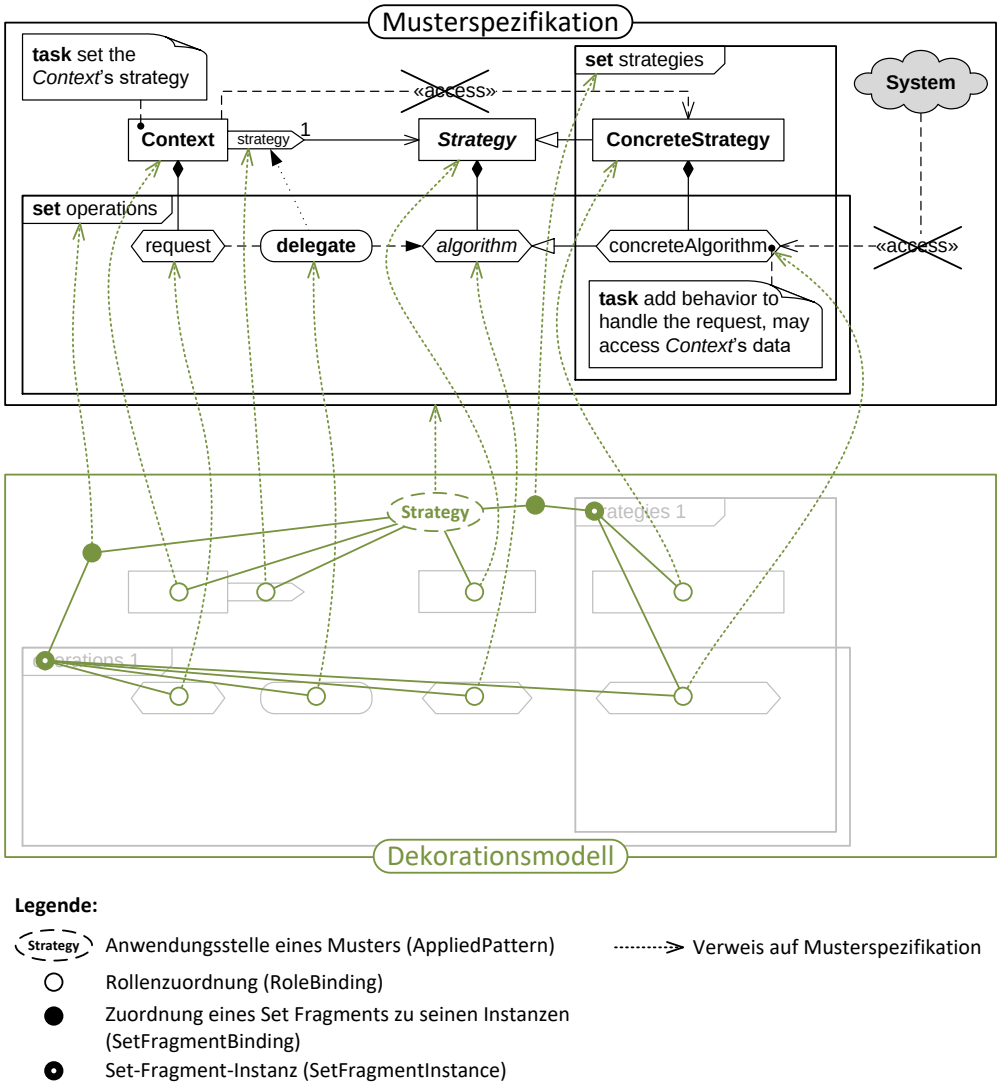


Abbildung C.1: Erstellen einer Anwendungsstelle am Beispiel Strategy

werden in einer baum-artigen Struktur festgehalten. Die Anwendungsstelle wird als eigener Knoten repräsentiert (AppliedPattern), der die Wurzel der baum-artigen Struktur bildet und auf die Musterspezifikation verweist. Die Korrespondenzknoten (RoleBinding) sind in der Abb. C.1 als Kreise dargestellt und sind jeweils mit der zugehörigen Rolle in der Musterspezifikation verknüpft. Zur Veranschaulichung der zu einem Korrespondenzknoten gehörenden Rolle wurde hier die Kontur des zur Rolle gehördenen visuellen Elements in der Musterspezifikation hell hinter dem Korrespondenzknoten eingezeichnet. Jedes Vorkommen der in einem Set Fragment definierten Struktur ist eine Set-Fragment-Instanz und wird durch einen entsprechenden Knoten (SetFragmentInstance) repräsentiert, der alle zugehörigen Korrespondenzknoten zusammenfasst. Auch hier sind zur Verdeutlichung das Set Fragment und der Index der Set-Fragment-Instanz hell hinter dem entsprechenden Knoten (SetFragmentInstance) eingezeichnet, sodass alle zur selben Set-Fragment-Instanz gehörenden Korrespondenzknoten von einem Rahmen umfasst sind. Alle zu einem Set Fragment gehörenden Set-Fragment-Instanzen werden ebenfalls in

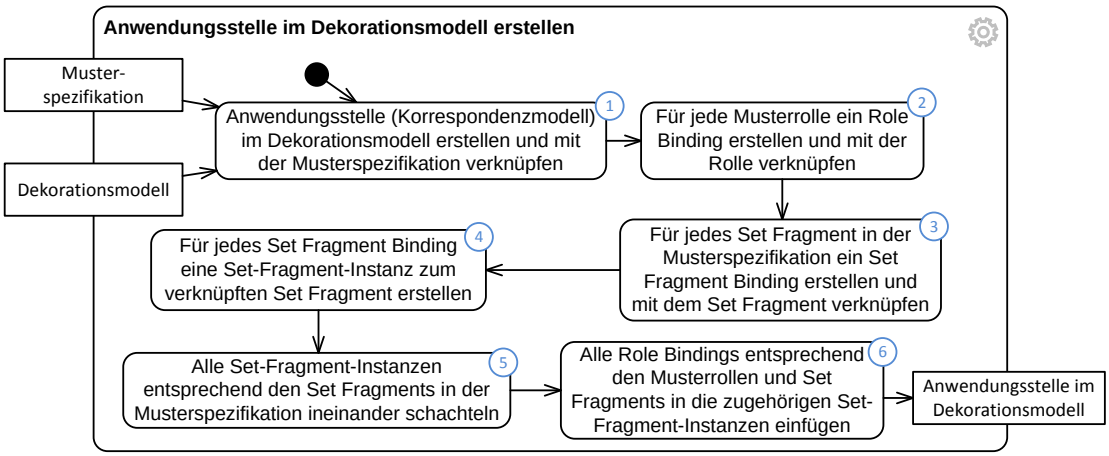


Abbildung C.2: Erstellen einer Anwendungsstelle im Dekorationsmodell

einem speziellen Knoten (SetFragmentBinding) zusammengefasst.

Das Erstellen eines initialen Korrespondenzmodells (Schritt 3 in Abb. 5.10, S. 115) wie z.B. des aus der Abb. C.1 erfolgt in mehreren Schritten. Diese Schritte sind in dem Aktivitätendiagramm in der Abb. C.2 aufgeführt. Zuerst wird die Anwendungsstelle (AppliedPattern) erstellt und mit der Musterspezifikation verknüpft (Schritt 1). Zu jeder Rolle der Musterspezifikation wird ein Korrespondenzknoten (RoleBinding) erstellt, welcher mit der Rolle verknüpft und zur Anwendungsstelle hinzugefügt wird (Schritt 2). Für jedes Set Fragment der Musterspezifikation wird ein spezieller Korrespondenzknoten, ein SetFragmentBinding, erstellt und mit dem Set Fragment verknüpft (Schritt 3). Dieser Korrespondenzknoten fasst alle Set-Fragment-Instanzen zu einem Set Fragment zusammen. Anschließend wird für jedes Set Fragment je eine Set-Fragment-Instanz (SetFragmentInstance) erstellt und dem SetFragmentBinding-Knoten hinzugefügt (Schritt 4). Die erstellten Set-Fragment-Instanzen werden genau so ineinander geschachtelt wie die zugehörigen Set Fragments (Schritt 5), sodass die Hierarchie von Set-Fragment-Instanzen der Hierarchie von Set Fragments entspricht. Schließlich werden die zuvor erstellten RoleBinding-Knoten so in die Set-Fragment-Instanzen eingefügt wie die zugehörigen Musterrollen der Musterspezifikation in den Set Fragments enthalten sind (Schritt 6). Insgesamt entsteht also eine Struktur von Korrespondenzknoten, die bis auf die Kanten der Musterspezifikation exakt der Struktur von Rollen und Set Fragments entspricht (siehe Bsp. in Abb. C.1).

Das Korrespondenzmodell kann anschließend dazu genutzt werden, durch den Entwickler gemachte Zuordnungen von Entwurfsteilen zu den von ihnen einzunehmenden Rollen der Musterspezifikation zu persistieren. Die in Abb. 5.7 dargestellten Zuordnungen, zum Beispiel, werden wie in der Abb. C.3 skizziert im Korrespondenzmodell festgehalten. Alle bei dieser Zuordnung erstellten Elemente sind in der Abbildung grün dargestellt. Auch vom Entwickler vorgesehene Namen für noch zu erzeugende Elemente im Entwurf werden im Korrespondenzmodell festgehalten. Zum Beispiel wird der Name „layout“ für die zu erzeugende Referenz bzw. Rolle strategy im entsprechenden Korrespondenzknoten festgehalten (siehe Abb. C.3).

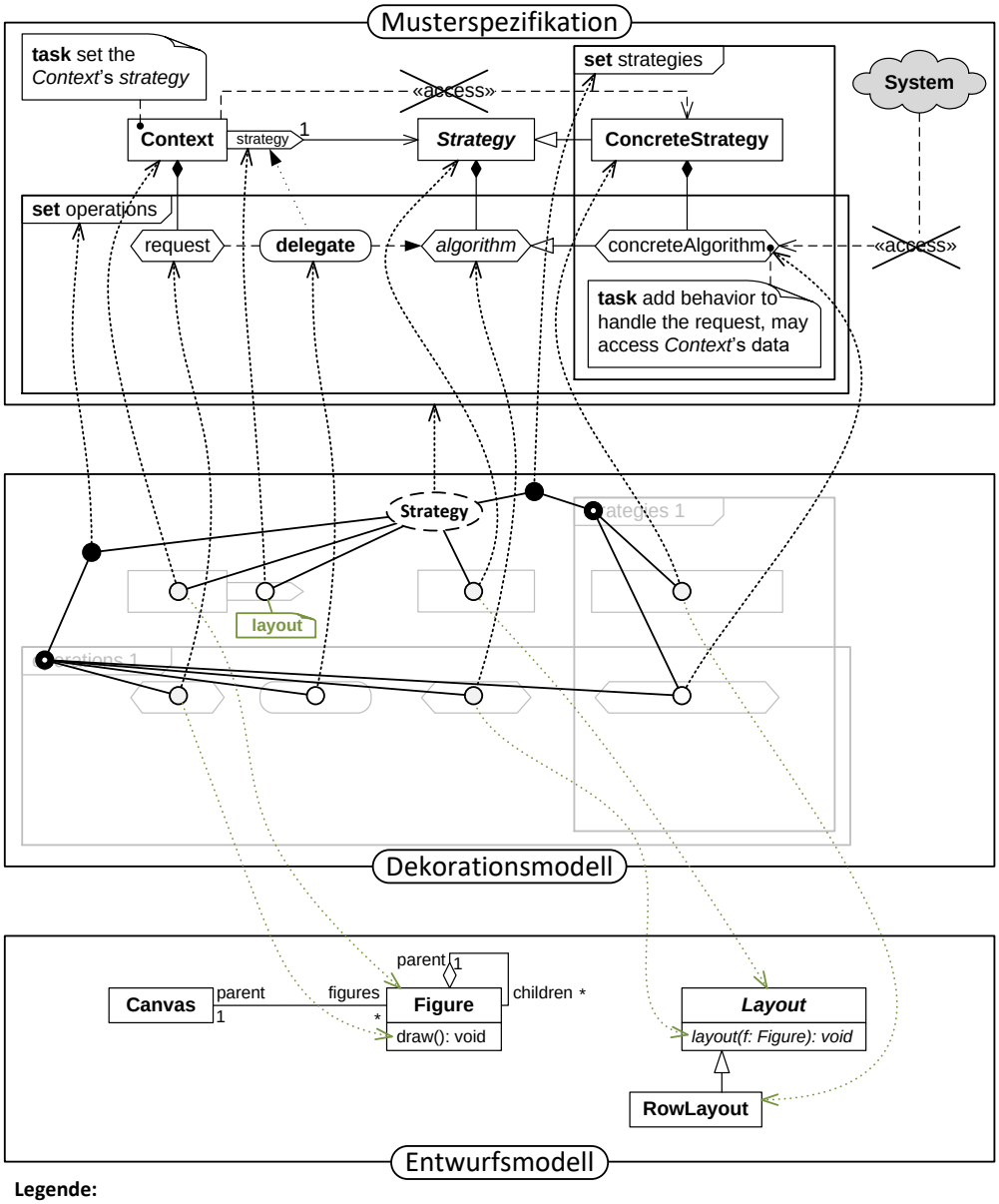


Abbildung C.3: Rollzuordnungen im Korrespondenzmodell am Beispiel Strategy

C.2 Auffaltung

Das Anwendungsmodell ist ein Zwischenergebnis bei der Überführung der allgemein beschriebenen Entwurfslösung aus einer Musterspezifikation in eine bestimme Implementierungsvariante des Musters in einem Softwareentwurfsmodell. Es ist das Ergebnis der Auffaltung einer Musterspezifikation (siehe Abb. 5.1, S. 101). Nach einer Erläuterung des Anwendungsmodells beschreibe ich im Folgenden, wie ich ein Anwendungsmodell auf Basis von Benutzereingaben und unter Berücksichtigung der formal definierten Set-Fragment-Semantik (siehe Abschnitt A.3.3) automatisch erzeugen lasse. Das Anwendungsmodell wird in mehreren Schritten erzeugt, die den Schritten 4, 6 und 8 der Abb. 5.10 (S. 115) entsprechen. Zunächst wird ein Anwendungsmodell durch „Kopieren“ der Musterspezifikation instanziiert (Abschnitt C.2.1), anschließend werden zusätzliche Vorkommen der in Set Fragments beschriebenen Entwurfsmodellstrukturen ergänzt (Abschnitt C.2.2). Wie diese Auffaltung mit der formalen Definition von Set Fragments zusammenhängt, beschreibe ich in Abschnitt C.2.3.

C.2.1 Erstellen eines initialen Anwendungsmodells (Instanziierung)

Ein Anwendungsmodell wird automatisch erstellt (Schritt 4 in Abb. 5.10, S. 115) bevor ein Entwickler vor einer Musteranwendung mit der Zuordnung von Entwurfsteilen zu Musterrollen beginnt (Schritte 5 und 7 derselben Abbildung).

Bezieht man das Korrespondenzmodell mit ein, welches unter anderem die Set-Fragment-Instanzen enthält, entspricht das initiale Erstellen des Anwendungsmodells mit zugehörigen Set-Fragment-Instanzen der Instanziierung laut Definition A.10 (S. 284).

In der Abb. C.4 wird die Struktur eines initial erzeugten Anwendungsmodells für das Muster Strategy skizziert. Neben dem Anwendungsmodell sind auch die Korrespondenzknoten des Korrespondenzmodells (also des Modells der Anwendungsstelle) und die Musterspezifikation eingezeichnet. Alle im Schritt 4 der Abb. 5.10 (S. 115) erzeugten Elemente sind grün dargestellt. Die Abb. C.4 verdeutlicht, dass die Struktur im Anwendungsmodell der Struktur der Musterspezifikation entspricht (gleiche Knoten und Kanten). Um die im Korrespondenzmodell erfassten Zugehörigkeiten der Entwurfselemente im Anwendungsmodell zu den Set-Fragment-Instnazen zu veranschaulichen, sind Set-Fragment-Instanzen bei der Darstellung des Anwendungsmodells durch hellgraue Rahmen mit Beschriftung angedeutet. Ein solcher Rahmen umschließt genau die Entwurfselemente im Anwendungsmodell, die zu der entsprechenden Set-Fragment-Instanz gehören.

Das initial erstellte Anwendungsmodell beschreibt eine Implementierungsvariante des spezifizierten Musters, in der es zu jeder Musterrolle genau ein entsprechendes Entwurfselement in der Musterimplementierung gibt. Durch Anlegen weiterer Vorkommen der in Set Fragments beschriebenen Strukturen – also weiterer Set-Fragment-Instanzen – (Schritt 7 in Abb. 5.10, S. 115) kann der Entwickler eine andere Implementierungsvariante des Musters wählen. Das Anwendungsmodell wird dann automatisch angepasst (Schritt 8 in Abb. 5.10, S. 115).

Das initiale Erzeugen des Anwendungsmodells ist als Aktivitätendiagramm in der Abb. C.5 beschrieben. Nach dem Anlegen eines leeren Anwendungsmodells

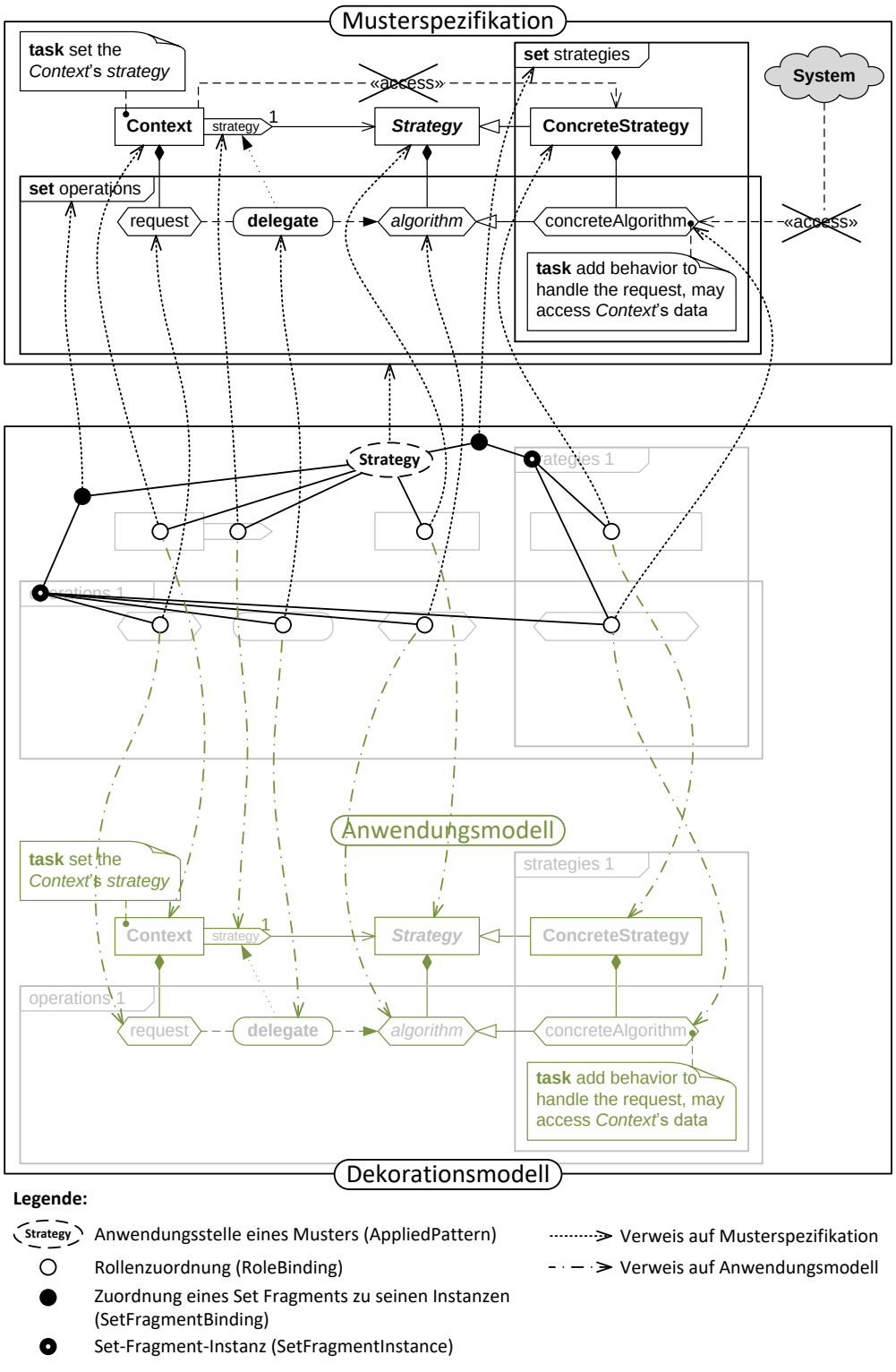


Abbildung C.4: Erstellen eines initialen Anwendungsmodells am Beispiel Strategy

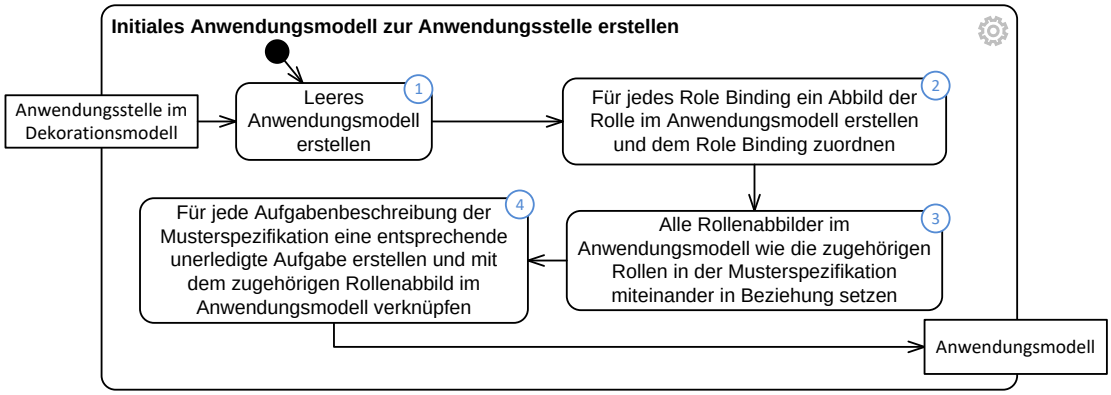


Abbildung C.5: Initiales Erstellen des Anwendungsmodells (Instanziierung)

(ApplicationModel, siehe Abb. 4.5, S. 86) zur zuvor erstellten Anwendungsstelle (AppliedPattern) (Schritt 1) wird zu jedem Korrespondenzknoten (RoleBinding) der Anwendungsstelle ein Abbild der damit verknüpften Rolle erstellt (Schritt 2). Jedes Abbild einer Rolle wird mit dem zugehörigen Korrespondenzknoten verknüpft (siehe auch Abb. C.4) und erhält dadurch die Zuordnung zu seinen bis zu 2 Set-Fragment-Instanzen.

Das Abbild einer Rolle ist im Wesentlichen eine Kopie der ursprünglichen Rolle. Im Gegensatz zu einer Rolle in der Musterspezifikation, repräsentiert ein solches Abbild der Rolle im Anwendungsmodell genau ein die Rolle einnehmendes Element, das im Entwurfsmodell erzeugt werden oder schon vorhanden sein soll. Zum Beispiel gibt es in der Abb. 5.9 (S. 114) für jede Ausprägung des Typs ConcreteStrategy aus der Musterspezifikation je einen Typ im Anwendungsmodell, der die zu erzeugende bzw. vorhandene Klasse repräsentiert, hier RowLayout und XYLayout. Direkt nach dem Erzeugen eines Anwendungsmodells gibt es nur je ein Abbild zu einer Rolle. Bei dem Beispiel aus Abb. 5.9 gibt es anfangs also nur einen Typ im Anwendungsmodell für die Rolle ConcreteStrategy (siehe Abb. C.4).

Das initial erstellte Anwendungsmodell soll bis auf Set Fragments exakt der Struktur in der Musterspezifikation entsprechen (vgl. Def. A.10, S. 284). Darum werden alle Rollenabbilder im Anwendungsmodell im nächsten Schritt genauso miteinander verknüpft, wie ihre Rollen in der Musterspezifikation (Schritt 3). Das heißt, alle Beziehungen wie Referenzen, Vererbungsbeziehungen, Aufrufe von Operationen, etc. (Kanten) werden zwischen den Rollenabbildern (Knoten) so aufgebaut wie sie zwischen den Rollen der Musterspezifikation angegeben wurden. Somit entsteht ein bis auf Set Fragments und Kopplungsregeln zur Musterspezifikation isomorphes Abbild aller Rollen und ihrer Beziehungen (vgl. Abb. A.29, S. 284).

Abschließend werden jedem Rollenabbild im Anwendungsmodell die laut Musterspezifikation manuell zu erledigenden Aufgaben hinzugefügt (Schritt 4). Dazu wird zu jeder Aufgabenbeschreibung der Musterspezifikation eine entsprechende Aufgabe erstellt und dem Rollenabbild hinzugefügt, an dessen Rolle die Aufgabenbeschreibung geheftet ist. In der Abb. C.4 ist die Notation für Aufgabenbeschreibungen und Aufagen identisch.

Das Erstellen einer Anwendungsstelle im Korrespondenzmodell und des initialen Anwendungsmodells entspricht, wie zuvor erwähnt, der Instanziierung aus

Definition A.10 (S. 284). Weil es beliebig viele, unterschiedliche Anwendungsstellen geben kann und eine Musterspezifikation nicht bei jeder Musteranwendung verändert werden soll, enthält das Anwendungsmodell (der Graph G_I) abweichend von der Definition A.10 (i) nicht die Knoten und Kanten aus der Musterspezifikation (dem Graphen G_S), sondern neue Knoten und Kanten, welche denen aus der Musterspezifikation entsprechen, dieselben Typen haben und auf die gleiche Weise miteinander verbunden sind. Das Anwendungsmodell enthält also ein zur Musterspezifikation isomorphes Abbild der Knoten und Kanten, wobei alle Knoten und Kanten denselben Typ haben wie ihre Vorbilder in der Musterspezifikation. Das Korrespondenzmodell enthält Set-Fragment-Instanzen, welche isomorph zu den Set Fragments der Musterspezifikation sind, wobei jede Set-Fragment-Instanz das zugehörige Set Fragment als Typ erhält. Die Bedingungen (ii) bis (iv) der Definition A.10 bleiben somit von dieser Abweichung unberührt.

C.2.2 Ergänzen einer Set-Fragment-Instanz (add-Operation)

Ohne weiteres Zutun durch den Entwickler wird bei Anwendung eines Musters genau die in der Musterspezifikation definierte Struktur im Entwurfsmodell erzeugt. Dabei wird zu jeder Musterrolle genau ein die Rolle einnehmendes Entwurfselement im Entwurfsmodell erstellt.

Durch Set Fragments in Musterspezifikationen können Strukturen definiert werden, die als Ganzes beliebig häufig in einer Implementierung des Musters vorkommen können. Jedes solche Vorkommen der Struktur in einer Musterimplementierung wird als Instanz des Set Fragments angesehen. Vor einer Musteranwendung legt der Entwickler für seinen Anwendungsfall fest, wie häufig Set-Fragment-Instanzen in seiner Musterimplementierung vorkommen sollen. Dazu fügt der Entwickler jede Set-Fragment-Instanz einzeln zu seiner Musterimplementierung hinzu (Schritte 7 und 8 in Abb. 5.10, S. 115).

Formal ist diese Operation in der Definition A.12 (S. 285) festgelegt und in den Abb. A.31 (S. 287) und A.32 (S. 288) skizziert. Ein Beispiel für das Hinzufügen einer Set-Fragment-Instanz im Dekorationsmodell ist in der Abb. C.6 für das Muster Strategy skizziert (vgl. Abb. C.4, S. 308).

Technisch erfolgt das Hinzufügen einer Set-Fragment-Instanz mit Hilfe einer schon existierenden Set-Fragment-Instanz, die als Vorbild für die neue Set-Fragment-Instanz fungiert. Bei dem Beispiel aus Abb. C.6 dient die Set-Fragment-Instanz strategies 1 als Vorbild für die neue Set-Fragment-Instanz strategies 2. Der Entwickler gibt an, dass er eine weitere Set-Fragment-Instanz in seiner Musterimplementierung haben möchte, welche mit der schon existierenden Set-Fragment-Instanz strukturell übereinstimmt.

Wie in der Abb. C.6 dargestellt, kann es in einer Set-Fragment-Instanz mehr als nur eine Ausprägung einer Musterrolle geben. Zum Beispiel gibt es in der Set-Fragment-Instanz operations 1 aus Abb. C.6 zwei concreteAlgorithm-Operationen. Das entspricht der Definition A.12 (S. 285) und hängt mit der Überschneidung der Set Fragments operations und strategies zusammen. Die Definition A.12 sorgt dafür, dass bei Überschneidung zweier Set Fragments jede Instanz eines Set Fragments mit jeder Instanz des anderen Set Fragments kombiniert ist und somit nur zur Musterspezifikation konforme Strukturen in einer Musterimplementierung vor-

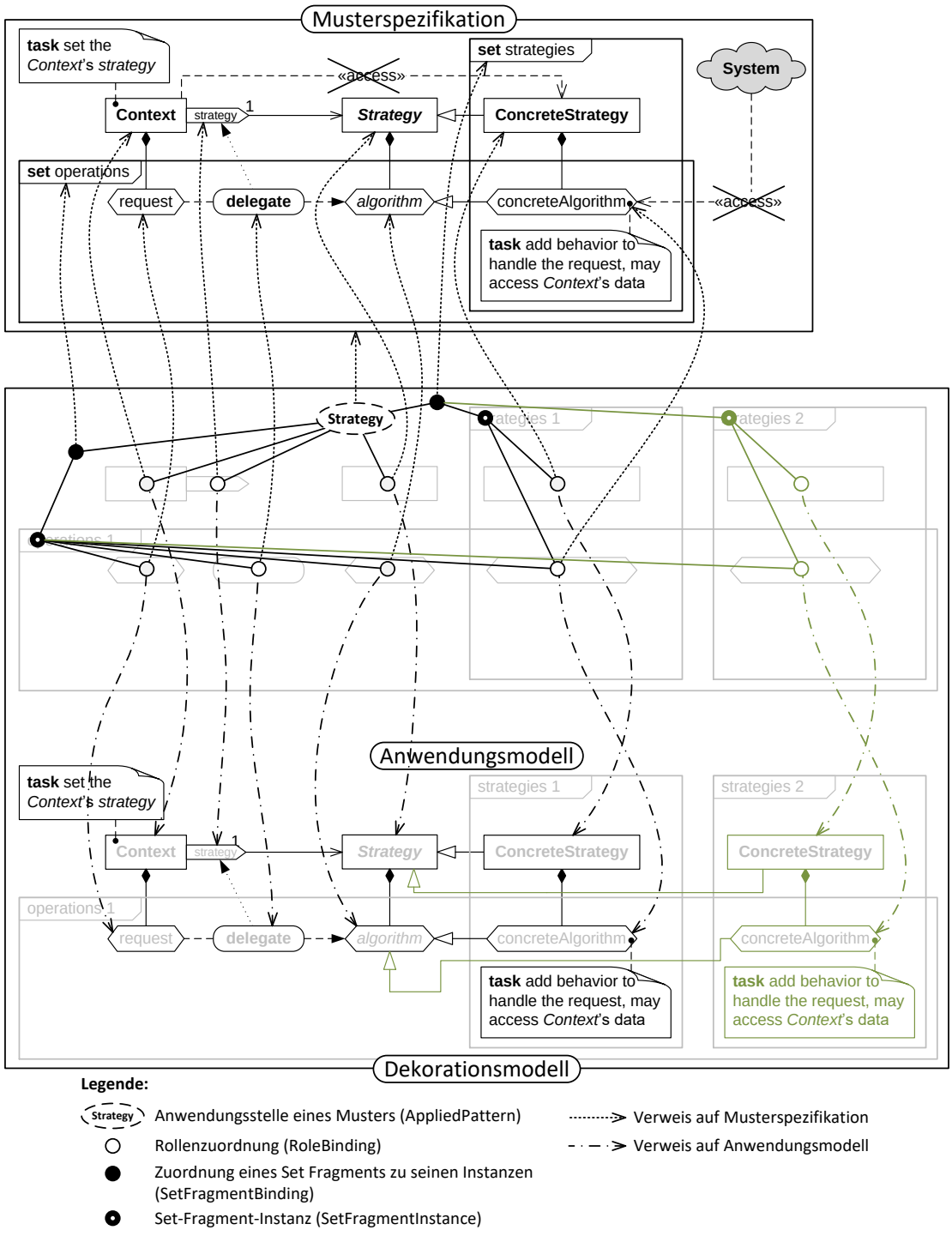


Abbildung C.6: Ergänzen einer Set-Fragment-Instanz am Beispiel Strategy

Codefragment C.1: Operation addSetFragmentInstance

```
1  addSetFragmentInstance(i: SetFragmentInstance): SetFragmentInstance
2  {                                  // addSetFragmentInstance entspricht der add-Opeation laut Def. A.12
3    // temporäre Zuordnung von kopierten RoleBindings bzw. Entwurfselementen auf ihre Replikate erstellen
4    img_V := new Map<RoleBinding, RoleBinding>;                                    // Def. A.12 (iii)
5
6    // i rekursiv replizieren, i dient als Vorbild für die neue Set-Fragment-Instanz i'
7    i' := recursivelyCopySetFragmentInstance(i, i, img_V);      // Codefragment C.2, S. 314
8
9    // i rekursiv die Kanten aus i replizieren und in i' ergänzen
10   recursivelyCopyEdgesInSetFragmentInstance(i, i, img_V);   // Codefragment C.4, S. 316
11
12   // Neue Set-Fragment-Instanz i' in die gleichen Set-Fragment-Instanzen einfügen wie i
13   forEach j: SetFragmentInstance                                                 // Def. A.12 (v)
14     in i.containingSetFragmentInstances do
15   {
16     j.containedSetFragmentInstances→add(i');
17   }
18
19   return i';
20 }
```

kommen. Dem Entwickler wird in der Musteranwendungsansicht nur je eine der existierenden Set-Fragment-Instanzen präsentiert und damit nur Ausschnitte seiner gewählten Musterimplementierung (siehe Abb. 5.7 und 5.8, S. 110 ff.). Diese Ausschnitte sind immer kongruent[1] zur Musterspezifikation.

Der Algorithmus zur Erzeugung einer Set-Fragment-Instanz entspricht dem Schritt 8 in der Abb. 5.10 (S. 115) und erfüllt die Eigenschaften der *add*-Operation aus Definition A.12 (S. 285). Im Folgenden beschreibe ich den Algorithmus und stelle ihn in Beziehung zur Definition der *add*-Operation.

Den Algorithmus zur Erzeugung einer zusätzlichen Set-Fragment-Instanz habe ich in der Operation addSetFragmentInstance definiert und durch den Pseudocode im Codefragment C.1 beschrieben. Im Pseudocode werden Bezeichner aus der abstrakten Syntax des Dekorationsmodells verwendet (siehe Abb. 4.5, S. 86). Die Operation addSetFragmentInstance erhält eine existierende Set-Fragment-Instanz i als Argument und erzeugt eine zu i strukturell identische neue Set-Fragment-Instanz i'. Strukturell identisch bedeutet im Prinzip, dass die neue Set-Fragment-Instanz i' die gleichen Elemente enthält wie i. Dazu werden alle in i enthaltenen Entwurfselemente (Knoten, Kanten) und Set-Fragment-Instanzen rekursiv repliziert und ihre Replikate in i' eingefügt.

Die Operation addSetFragmentInstance ist eine zweigeteilte, rekursive Operation. Sie kopiert zunächst i und alle direkt oder indirekt in i enthaltenen Set-Fragment-Instanzen und Entwurfselemente (Knoten ohne Kanten) (Codefragment C.1, Zeile 7). Nachdem alle Knoten repliziert wurden, kopiert die Operation alle Kanten, die mindestens einen in i direkt oder indirekt enthaltenen Knoten als Quelle oder Ziel haben (Zeile 10) (siehe Def. A.12 (iv) und Abb. A.31, S. 287). Abschließend wird die Kopie von i, nämlich die neue Set-Fragment-Instanz i', in dieselben Set-Fragment-Instanzen eingefügt wie i (Zeilen 13–17) (vgl. Def. A.12 (v)).

Zur Replikation von Kanten wird die Zuordnung von replizierten Knoten zu ihren Replikaten benötigt. Dazu wird vor dem Replizieren der Set-Fragment-Instanz

[1] deckungsgleich

i eine temporäre, vorerst leere Tabelle (Map) für solche Zuordnungen angelegt (Zeile 4). Die Zuordnungen entsprechen der Abbildung $img_V : U_V \to V_{add}$ aus Def. A.12 (iii) (S. 285). Die in Zeile 7 aufgerufene Operation recursivelyCopySetFragmentInstance füllt beim Replizieren der Knoten aus i die Tabelle img_V und ordnet jedem replizierten Knoten $v \in U_V$ des bisherigen Anwendungsmodells den dazu entsprechenden neuen Knoten $v' = img_V(v) \in V_{add}$ in der neuen Set-Fragment-Instanz i' zu. Die in Zeile 10 aufgerufene Operation recursivelyCopyEdgesInSetFragmentInstance nutzt die Tabelle bzw. Abbildung img_V dazu, die neuen Kanten entsprechend der Definition A.12 (viii) zu erzeugen.

Der Pseudocode zur Operation recursivelyCopySetFragmentInstance ist dem Codefragment C.2 zu entnehmen. Diese rekursive Operation erhält eine zu replizierende Set-Fragment-Instanz j, die in der Hierarchie oberste zu replizierende Set-Fragment-Instanz i und eine Tabelle (Map) img_V als Argumente (Zeilen 1–3). Dabei ist i die als Argument an die Operation addSetFragmentInstance überreichte Set-Fragment-Instanz (siehe Codefragment C.1, Zeilen 1 und 7). Nachdem j repliziert wurde, wird das Replikat j' als Ergebnis des Aufrufs der Operation recursivelyCopySetFragmentInstance zurückgegeben (Zeilen 4 und 64).

Als Erstes wird eine neue Set-Fragment-Instanz j' erzeugt (Zeile 7) und dem SetFragmentBinding-Knoten der schon existierenden Set-Fragment-Instanz j hinzugefügt (Zeile 8). Weil ein SetFragmentBinding-Knoten alle Instanzen zu einem Set Fragment zusammenfasst, wird dadurch für j' der gleiche Typ[2] festgelegt wie für j (vgl. Def. A.12 (ii)).

Als Nächstes werden alle in j direkt oder indirekt enthaltenen Set-Fragment-Instanzen repliziert sowie in das Replikat j' eingefügt (Zeilen 10–28). Dazu werden alle direkt in j enthaltenen Set-Fragment-Instanzen l durchlaufen (Zeilen 10, 11) und durch einen rekursiven Aufruf der Operation recursivelyCopySetFragmentInstance repliziert (Zeile 14). Die Replikate l' werden in j' eingefügt (Zeile 17), um die gleiche Enthaltensbeziehung wie zwischen j und den darin enthalten Instanzen l zu erreichen (Def. A.12 (vi)). Für den Fall von sich überschneidenden Set Fragments und zur Erfüllung der Bedingung (vii) aus Definition A.12 (S. 285) werden die Replikate l' anschließend in alle schon vorhanden Set-Fragment-Instanzen $k \neq i$ eingefügt, welche nicht direkt oder indirekt in i enthalten sind (Zeilen 20–27). Die Enthaltensbeziehung wird durch die überladene Operation in aus Codefragment C.3 (S. 315) geprüft (vgl. Def. A.9, S. 284).

Nach dem Replizieren aller in j enthaltenen Set-Fragment-Instanzen werden die in j enthaltenen Knoten (Entwurfselemente, Aufgaben und Korrespondenzknoten) repliziert (Zeilen 30–62). Zunächst wird jeder in j enthaltene Korrespondenzknoten (RoleBinding) repliziert und der Anwendungsstelle (AppliedPattern) hinzugefügt (Zeilen 34–36). Anschließend wird das zum Korrespondenzknoten r und damit auch zur Set-Fragment-Instanz j gehörende Entwurfselement, der Knoten v, repliziert, indem ein neuer Knoten v' gleichen Typs erzeugt und dem Anwendungsmodell (ApplicationModel) hinzugefügt wird (Zeilen 39–42). Um das spätere Ergänzen der Kanten in j' zu ermöglichen, wird der neu gebildete Knoten v' mit Hilfe der Tabelle img_V dem Knoten v zugeordnet, aus dem v' entstanden ist. Das

[2]Set-Fragment-Instanzen haben laut Meta-Modell keinen explizit modellierten Typ (siehe Abschnitt 4.2). Dieser wird implizit über den SetfragmentBinding-Knoten bestimmt, dem eine Set-Fragment-Instanz immer zugeordnet wird.

Codefragment C.2: Operation recursivelyCopySetFragmentInstance

```
1  recursivelyCopySetFragmentInstance(j: SetFragmentInstance,
2                                     i: SetFragmentInstance,
3                                     img_V: Map<RoleBinding, RoleBinding>):
4                                                          SetFragmentInstance
5  {
6    // Set-Fragment-Instanz j replizieren
7    j' := new SetFragmentInstance;
8    j.setFragmentBinding.instances→add(j');          // type'_I(j') = type_I(j), Def. A.12 (ii)
9
10   forEach l: SetFragmentInstance
11     in j.containedSetFragmentInstances do
12   {
13     // in j enthaltene Set-Fragment-Instanz l rekursiv replizieren
14     l' := recursivelyCopySetFragmentInstance(l, i, img_V);
15
16     // Neue Set-Fragment-Instanz l' in j' einfügen
17     j'.containedSetFragmentInstances→add(l');                          // Def. A.12 (vi)
18
19     // Neue Set-Fragment-Instanz l' in nicht kopierte Set-Fragment-Instanz k einfügen
20     forEach k: SetFragmentInstance                                     // Def. A.12 (vii)
21       in l.containingSetFragmentInstances
22     {
23       if (k ≠ j and k ≠ i and not in(i, k)) then          // k ∈ I \ U_I, Def. A.12 (vii)
24       {                                                  // in(i, k) laut Codefragment C.3, S. 315
25         k.containedSetFragmentInstances→add(l');
26       }
27     }
28   }
29
30   forEach r: RoleBinding
31     in j.roleBindings do
32   {
33     // in j enthaltenen Korrespondenzknoten r replizieren
34     r' := new RoleBinding;
35     r.appliedPattern.roleBindings→add(r');
36     r'.role := r.role;
37
38     // in j enthaltenes Entwurfselement v replizieren
39     v := r.applicationModelElement;
40     v' := v→type().new;                          // type'_I(v') = type_I(v), Def. A.12 (iii)
41     r'.applicationModelElement := v';
42     r.appliedPattern.applicationModel.designElements→add(v');
43
44     // temporär die Zuordnung r ↦ r' (und damit auch v ↦ v') merken
45     img_V →put(r, r');                          // img_V : U_V → V_add, Def. A.12 (iii)
46
47     // Neues Entwurfselement v' (über r') in j' einfügen
48     j'.roleBindings→add(r');                                           // Def. A.12 (vi)
49
50     // Neues Entwurfselement v' (über r') in nicht kopierte Set-Fragment-Instanz k einfügen
51     forEach k: SetFragmentInstance                                     // Def. A.12 (vii)
52       in r.containingSetFragmentInstances
53     {
54       if (k ≠ j and k ≠ i and not in(i, k)) then          // k ∈ I \ U_I, Def. A.12 (vii)
55       {                                                  // in(i, k) laut Codefragment C.3, S. 315
56         k.roleBindings→add(r');
57       }
58     }
59
60     // Zu erledigende Aufgaben für r' entsprechend der Musterspezifikation erstellen
61     ...
62   }
63
64   return j';
65 }
```

Codefragment C.3: Operation in

```
1  in(i: SetFragmentInstance, j: SetFragmentInstance): boolean    // vgl. Def. A.9, S. 284
2  {
3    forEach k: SetFragmentInstance
4        in i.containedSetFragmentInstances
5    {
6      if (k = j or in(k, j)) then
7        return true;
8    }
9    return false;
10 }
11
12 in(i: SetFragmentInstance, v: RoleBinding): boolean          // vgl. Def. A.9, S. 284
13 {
14   if (i.roleBindings→contains(v) then
15       return true;
16
17   forEach k: SetFragmentInstance
18       in i.containedSetFragmentInstances
19   {
20     if (in(k, v)) then
21       return true;
22   }
23   return false;
24 }
```

geschieht durch die Zuordnung der zu v und v' gehörenden Korrespondenzknoten r und r' (Zeile 45). Die Zuordnung des neuen Knotens v' zur neuen Set-Fragment-Instanz j' wird durch Einfügen des zu v' gehörenden Korrespondenzknotens r' in die Set-Fragment-Instanz j' festgehalten (Zeile 48). Abschließend werden zur korrekten Behandlung von sich überschneidenden Set Fragments die in j' enthaltenen Knoten v' entsprechend der Bedingung (vii) aus Definition A.12 (S. 285) in alle schon vorhanden Set-Fragment-Instanzen $k \neq i$ eingefügt, welche nicht direkt oder indirekt in i enthalten sind (Zeilen 51–58). Zu jedem erzeugten Korrespondenzknoten r' werden außerdem zu erledigende Aufgaben (Task) nach Vorgabe durch die Aufgabenbeschreibungen (TaskDescription) in der Musterspezifikation erzeugt bzw. nach Vorlage des Korrespondenzknotens r repliziert. Dieser Schritt ist simpel und wurde der Übersicht halber weggelassen (Zeilen 60, 61).

Das Erstellen der Kanten zu den zuvor replizierten Knoten (Entwurfselementen im Anwendungsmodell) wird durch den Pseudocode zur Operation recursivelyCopyEdgesInSetFragmentInstance im Codefragment C.2 beschrieben. Diese Operation erhält die gleichen Argumente wie die Operation recursivelyCopySetFragmentInstance (vgl. Setfragment C.2, S. 314). j ist die replizierte Set-Fragment-Instanz, in dessen Replikat nun die Kanten ergänzt werden sollen. i ist die in der Hierarchie oberste replizierte Set-Fragment-Instanz, also die als Argument an die Operation addSetFragmentInstance überreichte Set-Fragment-Instanz (siehe Codefragment C.1, Zeilen 1 und 7). Die Tabelle (Map) img_V enthält die Zuordnungen der replizierten Knoten (der Entwurfselemente) zu ihren Replikaten und entspricht der Abbildung $img_V : U_V \rightarrow V_{add}$ aus Def. A.12 (iii) (S. 285).

Vor dem Erstellen der Kanten zu den Knoten in der Set-Fragment-Instanz j werden rekursiv die Kanten zu den darin enthaltenen Set-Fragment-Instanzen l erstellt (Zeilen 5–10). Anschließend werden alle zur Set-Fragment-Instanz j gehörenden, zu replizierenden Kanten durchlaufen und repliziert. Diese haben einen Quell-

Codefragment C.4: Operation recursivelyCopyEdgesInSetFragmentInstance

```
1  recursivelyCopyEdgesInSetFragmentInstance(j: SetFragmentInstance,
2                                            i: SetFragmentInstance,
3                                            img_V: Map<RoleBinding, RoleBinding>): void
4  {
5    forEach l: SetFragmentInstance
6      in j.containedSetFragmentInstances do
7    {
8      // Kanten in enthaltener Set-Fragment-Instanz l rekursiv replizieren
9      recursivelyCopyEdgesInSetFragmentInstance(l, i, img_V);
10   }
11
12   // Durchlaufe alle in j enthaltenen Knoten und ihre ein- und ausgehenden Kanten
13   forEach r_1: RoleBinding
14     in j.roleBindings do
15   {
16     v_1 := r_1.applicationModelElement;      // in j enthaltener Knoten v_1 ∈ U_V, siehe Def. A.12 (iii)
17     r'_1 := img_V →get(r_1);
18     v'_1 := r'_1.applicationModelElement;                  // v'_1 = img_V(v_1), siehe Def. A.12 (iii)
19
20     // Durchlaufe alle ausgehenden Kanten e_out ∈ U_E von v_1                          Def. A.12 (iv)
21     forEach e_out: DesignElementRelation
22       in v_1.outgoingRelations do
23     {
24       v_2 := e_out.target;                                  // v_1 = source_I(e_out), v_2 = target_I(e_out)
25       r_2 := getRoleBinding(v_2, r_1.appliedPattern);          // Codefragment C.5, S. 317
26
27       // Erstelle neue Kante e'_out = img_E(e_out)
28       e'_out := e_out →type().new;                  // Def. A.12 (iv), type'_I(e'_out) = type_I(e_out)
29
30       e'_out.source := v'_1;      // source'_I(img_E(e_out)) = img_V(source_I(e_out)) = v'_1, Def. A.12 (viii)
31
32       // Falls v_1 ∈ U_V ∧ v_2 ∈ U_V nur von v_1 ausgehende Kanten replizieren ⇒ keine doppelten Replikate
33       if (in(i, r_2)) then          // d.h.: v_2 ∈ U_V, Def. A.12 (iii), (viii), Codefragment C.3, S. 315
34       {
35         r'_2 := img_V →get(r_2);
36         v'_2 := r'_2.applicationModelElement;                // v'_2 = img_V(v_2), siehe Def. A.12 (iii)
37
38         e'_out.target := v'_2;    // target'_I(img_E(e_out)) = img_V(target_I(e_out)) = v'_2, Def. A.12 (viii)
39       }
40       else                                                  // d.h.: v_2 ∉ U_V, Def. A.12 (iii), (viii)
41       {
42         e'_out.target := v_2;          // target'_I(img_E(e_out)) = target_I(e_out) = v_2, Def. A.12 (viii)
43       }
44     }
45
46     // Durchlaufe alle eingehenden Kanten e_in ∈ U_E von v_1                           Def. A.12 (iv)
47     forEach e_in: DesignElementRelation
48       in v_1.incomingRelations do
49     {
50       v_2 := e_in.source;                                   // v_2 = source_I(e_in), v_1 = target_I(e_in)
51       r_2 := getRoleBinding(v_2, r_1.appliedPattern);          // Codefragment C.5, S. 317
52
53       if (not in(i, r_2)) then    // d.h.: v_2 ∉ U_V, Def. A.12 (iii), (viii), Codefragment C.3, S. 315
54       {
55         // Erstelle neue Kante e'_in = img_E(e_in)
56         e'_in := e_in →type().new;                  // Def. A.12 (iv), type'_I(e'_in) = type_I(e_in)
57
58         e'_in.source := v_2;          // source'_I(img_E(e_in)) = source_I(e_in) = v_2, Def. A.12 (viii)
59         e'_in.target := v'_1;      // target'_I(img_E(e_in)) = img_V(target_I(e_in)) = v'_1, Def. A.12 (viii)
60       }
61     }
62   }
63 }
```

Codefragment C.5: Operation getRoleBinding

```
1  getRoleBinding(v: DesignElement, application: AppliedPattern): RoleBinding
2  {
3    forEach r: RoleBinding
4      in application.roleBindings do
5    {
6      if (r.applicationModelElement = v) then
7      {
8        return r;
9      }
10   }
11   return null;
12 }
```

oder Zielknoten (oder beide) in der Set-Fragment-Instanz j (vgl. Def. A.12 (iv)). Darum werden die Kanten durchlaufen, indem zunächst alle Knoten $v \in U_V$ (und zugehörige Korrespondenzknoten r mit r.applicationModelElement$= v$) in j durchlaufen werden (Zeilen 13 ff.) und dann alle ein- und ausgehenden Kanten dieser Knoten betrachtet werden (Zeilen 21 ff. und 47 ff.).

Zu jedem Korrespondenzknoten r_1 in j wird der zugehörige Knoten im Anwendungsmodell $v_1 \in U_V$ herausgesucht (Zeile 16), wodurch der Quell- oder Zielknoten der zu replizierenden Kanten $e \in U_E$ (vgl. Def. A.12 (iv)) bekannt ist. Es werden außerdem das aus v_1 erzeugte Replikat $v'_1 = img_V(v_1)$ sowie der zugehörige Korrespondenzknoten r'_1 mit Hilfe der Tabelle img_V bestimmt (Zeilen 17, 18).

Bei den zu replizierenden Kanten $e \in U_E$ ergeben sich aus der Definition A.12 (iv), (viii) und dem rekursiven Aufbau der Operation recursivelyCopyEdgesInSetFragmentInstance (es gilt: $source_I(e) = v_1 \vee target_I(e) = v_1$) drei zu unterscheidende Fälle, die sich daraus ergeben, dass der Quell- bzw. Zielknoten einer neuen Kante e' dem ursprünglichen Knoten $v \in U_V$ oder dessen Replikat $v' = img_V(v)$ entsprechen kann. Die drei Fälle sind: (a) sowohl Quell- als auch Zielknoten von e sind direkt oder indirekt in i enthalten (d.h.: beide liegen in U_V), (b) nur der Quellknoten von e ist direkt oder indirekt in i enthalten oder (c) nur der Zielknoten von e ist direkt oder indirekt in i enthalten. Durch den Aufbau der rekursiven Operation recursivelyCopyEdgesInSetFragmentInstance entsprechen der Quellknoten in Fall (b) und der Zielknoten in Fall (c) dem Knoten v_1, der direkt in j enthalten ist.

Durch das Durchlaufen aller in j direkt enthaltener Knoten $v \in U_V$ und der zugehörigen Kanten werden die Kanten in den Fällen (b) und (c) ein Mal, im Fall (a) jedoch zwei Mal betrachtet[3]. Um zu vermeiden, dass eine Kante mehr als ein Mal repliziert wird, erzeuge ich im Fall (a) nur beim Durchlaufen der von v_1 ausgehenden Kanten e (d.h.: $v_1 = source_I(e)$) eine neue Kante e'.

Beim Durchlaufen der zu replizierenden Kanten werden zunächst der zweite zur Kante gehörende Knoten v_2 und der zugehörige Korrespondenzknoten r_2 bestimmt (Zeilen 24, 25 und 50, 51). Dabei wird die Hilfsoperation getRoleBinding verwendet

[3]Entweder sind Quell- und Zielknoten direkt in j enthalten und dieselbe Kante wird ein Mal als ausgehende und ein Mal als eingehende Kante betrachtet oder nur einer der Knoten ist direkt in j enthalten und der andere ist in einer anderen Set-Fragment-Instanz innerhalb von i enthalten, sodass v_1 bei verschiedenen Aufrufen der Operation recursivelyCopyEdgesInSetFragmentInstance ein Mal den Quell- und ein Mal den Zielknoten repräsentiert.

(Codefragment C.5, S. 317).

Beim Betrachten der von v_1 ausgehenden Kanten e_{out} (Zeilen 21–44) werden nun die Fälle (a) und (b) behandelt. Es wird in beiden Fällen eine neue Kante e'_{out} erzeugt (Zeile 28), die denselben Typ wie e_{out} hat. Der Quellknoten für die neue Kante ergibt sich aus der Definition A.12 (viii) und entspricht dem zuvor erzeugten Knoten $v'_1 = img_V(v_1) \in V_{add}$ (Zeile 30), der für die beiden Fälle (a) und (b) identisch ist. Der Zielknoten hängt jedoch davon ab, ob Fall (a) oder (b) vorliegt und somit Quell- und Zielknoten oder nur der Quellknoten von e_{out} sich direkt oder indirekt in i befindet. Ob der Fall (a) vorliegt, wird in der Zeile 33 geprüft. Liegt er vor, werden mit Hilfe der Tabelle img_V der aus dem Knoten $v_2 \in U_V$ erzeugte Knoten $v'_2 \in V_{add}$ und der zugehörige Korrespondenzknoten r'_2 bestimmt (Zeilen 35, 36). Entsprechend der Definition A.12 (viii) wird dann $v'_2 = img_V(v_2) \in V_{add}$ als Zielknoten von e'_{out} festgelegt (Zeile 38). Im Fall (b) dagegen wird nach der Definition A.12 (viii) der Knoten $v_2 \notin U_V$ als Zielknoten bestimmt (Zeile 42).

Wenn in v_1 eingehende Kanten e_{in} betrachtet werden (Zeilen 47–61) wird der Fall (c) behandelt. Es wird also nur dann eine neue Kante e'_{in} erzeugt, wenn nur der Zielknoten von e_{in} direkt oder indirekt in i enthalten ist, also nur dann, wenn der Quellknoten v_2 von e_{in} (siehe Zeile 50) nicht innerhalb von i liegt (d.h.: $v_2 \notin U_V$). Diese Bedingung wird in der Zeile 53 geprüft. Ist sie erfüllt, wird eine neue Kante e'_{in} desselben Typs wie e_{in} erzeugt (Zeile 56). Wie in der Definition A.12 (viii) bestimmt, werden als Zielknoten der neuen Kante $v'_1 = img_V(v_1) \in V_{add}$ und als Quellknoten der Knoten $v_2 \in U_V$ gewählt (Zeilen 58, 59).

Nach dem Erweitern des Anwendungs- und des Korrespondenzmodells mit der hier beschriebenen add-Operation bzw. der Operation addSetFragmentInstance (vgl. Def. A.12, S. 285 und Codefragment C.1, S. 312) um neue Ausprägungen der in Set Fragments spezifizierten Entwurfsstrukturen kann ein Entwickler, wenn nötig, mit der Zuordnung der Teile seines bisherigen Entwurfs zu den Musterrollen fortfahren (Schritte 5 und 6, Abb. 5.10, S. 115) und schließlich eine Implementierung des Musters in seinem Entwurfsmodell automatisch synthetisieren lassen. Ein komplettes Modell der Anwendungsstelle, der Rollenbelegungen und ein komplettes Anwendungsmodell können für das Beispiel aus der Abb. 5.9 (S. 114) aussehen wie in der Abb. C.7 skizziert.

C.2.3 Konformität zur formal definierten Semantik von Set Fragments

Im Abschnitt A.3.3 (S. 280 ff.) habe ich die Semantik von Set Fragments formal definiert, indem ich eine Auffaltung einer Musterspezifikation in eine Musterimplementierung, also die Herleitung einer bestimmten aus mehreren spezifizierten Implementierungsvarianten eines Musters (vgl. Abb. 5.1, S. 101), mathematisch beschrieben habe. Dazu habe ich eine Auffaltung als eine Abbildung a_F eines getypten Graphen mit Set Fragments auf einen getypten Graphen ohne Set Fragments beschrieben (siehe Def. A.15, S. 289). Im Folgenden erläutere ich den Zusammenhang der im Abschnitt 5.4 (sowie in C.2.1 und C.2.2) beschriebenen Umsetzung einer Auffaltung und der im Abschnitt A.3.3 formal definierten Semantik einer Auffaltung.

Die Abbildung a_F ist aus mehreren anderen Abbildungen zusammengesetzt.

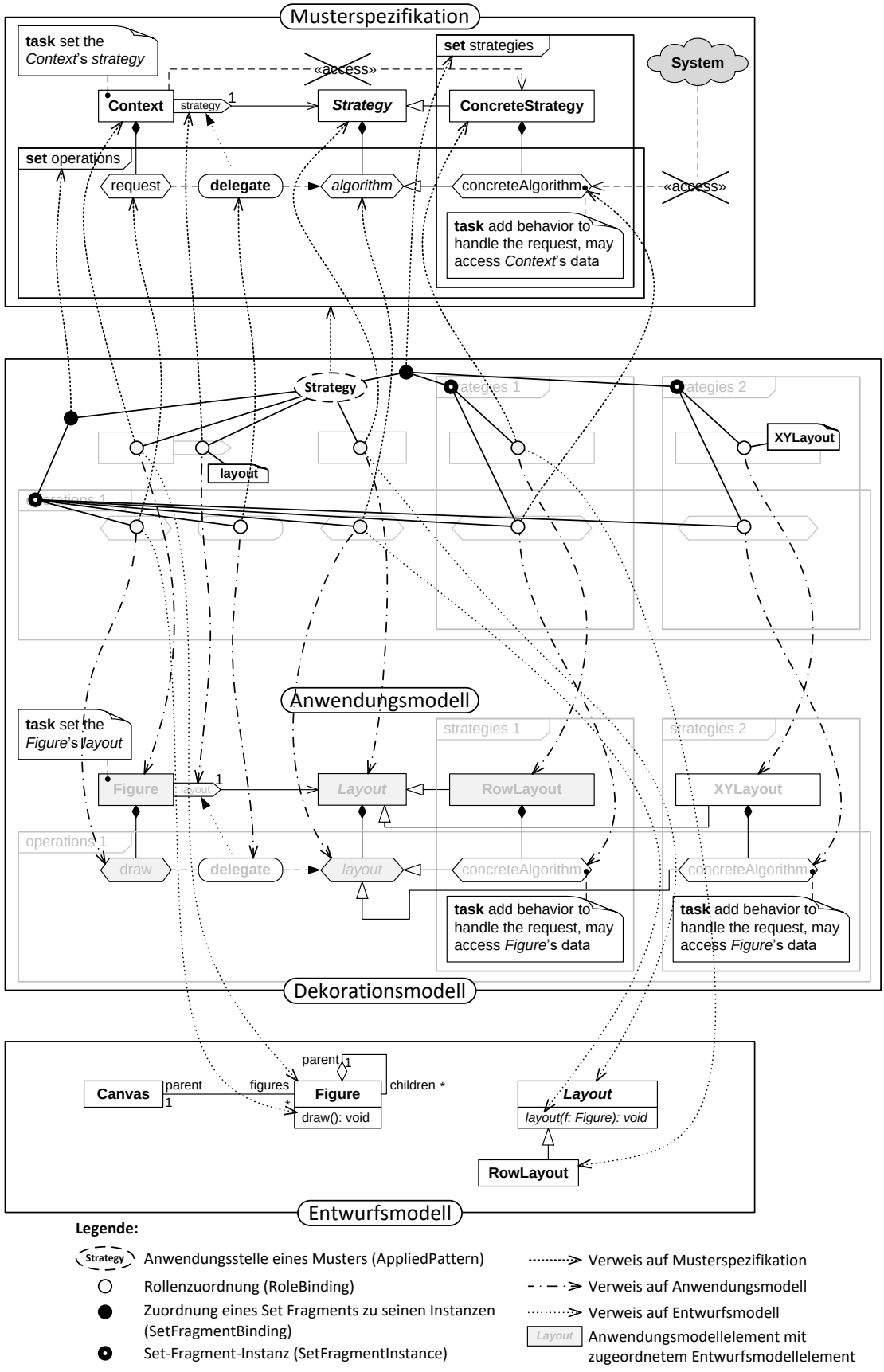


Abbildung C.7: Korrespondenz- und Anwendungsmodell vor einer Musteranwendung am Beispiel Strategy

Zunächst wird ein getypter Graph mit Set Fragments – dieser entspricht einer Musterspezifikation – durch die Abbildung *instance* (Def. A.10, S. 284) auf einen getypten Graphen mit Set-Fragment-Instanzen abgebildet. Die Abbildung *instance* entspricht in meiner Umsetzung dem initialen Erstellen eines Anwendungsmodells (siehe Abschnitt C.2.1) und dem vorhergehenden Erzeugen von Set-Fragment-Instanzen im Korrespondenzmodell (siehe Abschnitt 5.4.3). Der Bezug zur Definition der Instanziierung (*instance*-Abbildung, Def. A.10) wird zusammen mit dem Algorithmus zur initialen Erzeugung eines Anwendungsmodells in Abschnitt C.2.1 hergestellt.

Der durch die Abbildung *instance* erzeugte getypte Graph mit Set-Fragment-Instanzen wird anschließend in mehreren Schritten auf einen anderen getypten Graphen mit Set-Fragment-Instanzen abgebildet. Das erfolgt durch eine beliebige, auch leere Folge von *add*-Abbildungen (siehe Def. A.15, S. 289 und Def. A.12, S. 285). In meiner Umsetzung entspricht die in Abschnitt C.2.2 beschriebene Operation addSetFragmentInstance (Codefragment C.1, S. 312) der *add*-Operation aus Definition A.12. In dem die Operation definierenden Pseudocode wird auf die durch den Code eingehaltenen Bedingungen aus der Definition A.12 eingegangen und in der Beschreibung des Algorithmus der Bezug zur Definition hergestellt. Wie man der Abb. 5.10 (S. 115) entnehmen kann, wird die Operation addSetFragmentInstance beliebig häufig nach der Instanziierung – also nach dem initialen Erzeugen des Anwendugnsmodells – aufgerufen, was mit der Definition A.15 (S. 289) übereinstimmt.

Als letzter Schritt der Auffaltung wird eine Projektion π des getypten Graphen mit Set-Fragment-Instanzen auf einen getypten Graphen ohne Set-Fragment-Instanzen definiert (siehe Def. A.15 und Def. A.11, S. 284). Durch die Trennung des Anwendungsmodells von dem Korrespondezmodell und somit von den Set-Fragment-Instanzen entfällt bei meiner Umsetzung die Projektion. Betrachtet man das Anwendungsmodell nach der Auffaltung ohne das zugehörige Korrespondenzmodell, so entspricht das Anwendungsmodell dem getypten Graphen ohne Set-Fragment-Instanzen, der durch Instanziierung, eine beliebige Folge von *add*-Operationen und eine abschließende Projektion entsteht.

Somit entspricht meine Umsetzung der Auffaltung der zuvor formal definierten Abbildung a_F (Auffaltung) nach Definition A.15. Folglich gelten alle für a_F beschriebenen Eigenschaften (siehe Anmerkung A.16, S. 289) auch für meine Implementierung der Auffaltung.

Eine Auffaltung kann auch als eine Graphtransformation aufgefasst werden. Dabei wird die Musterspezifikation – ein getypter, attributierter Graph – in eine Musterimplementierung beschrieben in der DAL – also einen anderen getypten, attributierten Graphen – überführt. Ehrig et al. definieren getypte, attributierte Graphen und auf solchen Graphen basierende Graphtransformationen und Transformationssysteme [EEPT06]. Meine Formalisierung aus Abschnitt A.3.3 ist an die Graphendefinitionen von Ehrig et al. angelehnt.

Um die Formalisierung nicht unnötig kompliziert zu machen, basieren meine Definitionen jedoch im Gegensatz zu denen von Ehrig et al. nur auf getypten, nicht-attributierten Graphen (siehe Def. A.3, A.4, A.6, S. 281 ff.). Allerdings lassen sich meine Definitionen der Graphen und deren Abbildungen aufeinander analog zu den Definitionen von Ehrig et al. wie in Anmerkung A.18 (S. 290) beschrieben

um Attribute für Knoten und Kanten erweitern.

In meiner Formalisierung der Auffaltung verwende ich keine Vererbung im Typgraphen. Ehrig et al. haben gezeigt, dass es zu jeder Graphtransformation und Grammatik, welche auf getypten, attributierten Graphen *mit* Vererbung basiert, auch eine äquivalente Graphtransformation und Grammatik gibt, welche auf getypten, attributierten Graphen *ohne* Vererbung basiert [EEPT06, Theorem 13.24, Kap. 13]. Damit ist meine Formalisierung aus Abschnitt A.3.3 auch auf getypte, attributierte Graphen mit Vererbung im Typgraphen übertragbar (siehe Anmerkung A.18, S. 290). Weil die Typgraphen bei meiner Umsetzung den Meta-Modellen entsprechen (wo natürlich auch Vererbung vorkommt), während die getypten, attributierten Graphen den Modellen von Musterspezifikationen, Musterimplementierungen und deren Korrespondenzen entsprechen, passt die formale Definition der Auffaltung auch zu meiner in diesem Kapitel beschriebenen Umsetzung.

C.3 Übersetzung

In Abschnitt 5.5 (S. 121 ff.) habe ich beschrieben wie ein Anwendungsmodell in ein Entwurfsmodell überführt wird, welche Herausforderungen es dabei zu bewältigen gibt und wie mein Lösungsansatz aussieht. Dort habe ich nur einige Besipiele für konkrete Transformationsregeln gezeigt. Im Folgenden gebe ich dazu einen ausführlicheren Einblick. Zunächst gebe ich in Abschnitt C.3.1 einen Überblick über alle Transformationsregeln und ihre Zusammenhänge. Anschließend stelle ich in den Abschnitten C.3.2 und C.3.3 wie Klassenstrukturen und Verhalten (Aktionen) übersetzt werden.

C.3.1 Transformationsregeln bzw. Elementübersetzer

Die Transformation ist aus insgesamt 26 Elementübersetzern[4] aufgebaut. Diese können auch als Transformationsregeln angesehen werden. Die Übersetzer sind in der Abb. C.8 (S. 322) zusammen mit ihren Abhängigkeiten untereinander aufgeführt. Für alle übersetzbaren Konstrukte der Design Abstraction Language (DAL) gibt es entsprechende Übersetzer. Zum Beispiel gibt es Übersetzer für Typen (Type2EClass), Operationen (Operation2EOperation) und für verschiedene Aktionen (z.B. CallAction2ControlFlow). Subsysteme der DAL dagegen sind so allgemein gefasst[5], dass sie nur zur Überprüfung von Kopplungseigenschaften und nicht zur Generierung entsprechender Konstrukte im Entwurfsmodell verwendet werden. Darum gibt es keine Übersetzer dafür.

Einige Übersetzer triggern die Übersetzung von Teilen oder einzelnen Eigenschaften eines Elements. Zum Beispiel triggert der Übersetzer für Typen (Type2EClass) die Übersetzung der Vererbungsbeziehung zu einem Obertyp mit Hilfe eines anderen Übersetzers (TypeInheritance2EClassInheritance).

[4] Abstrakte Übersetzer ausgenommen.

[5] Sie könnten z.B. Paketstrukturen, Komponenten-, Bibliotheks- und Frameworkgrenzen beschreiben, ohne auf eines der Konzepte beschränkt zu sein.

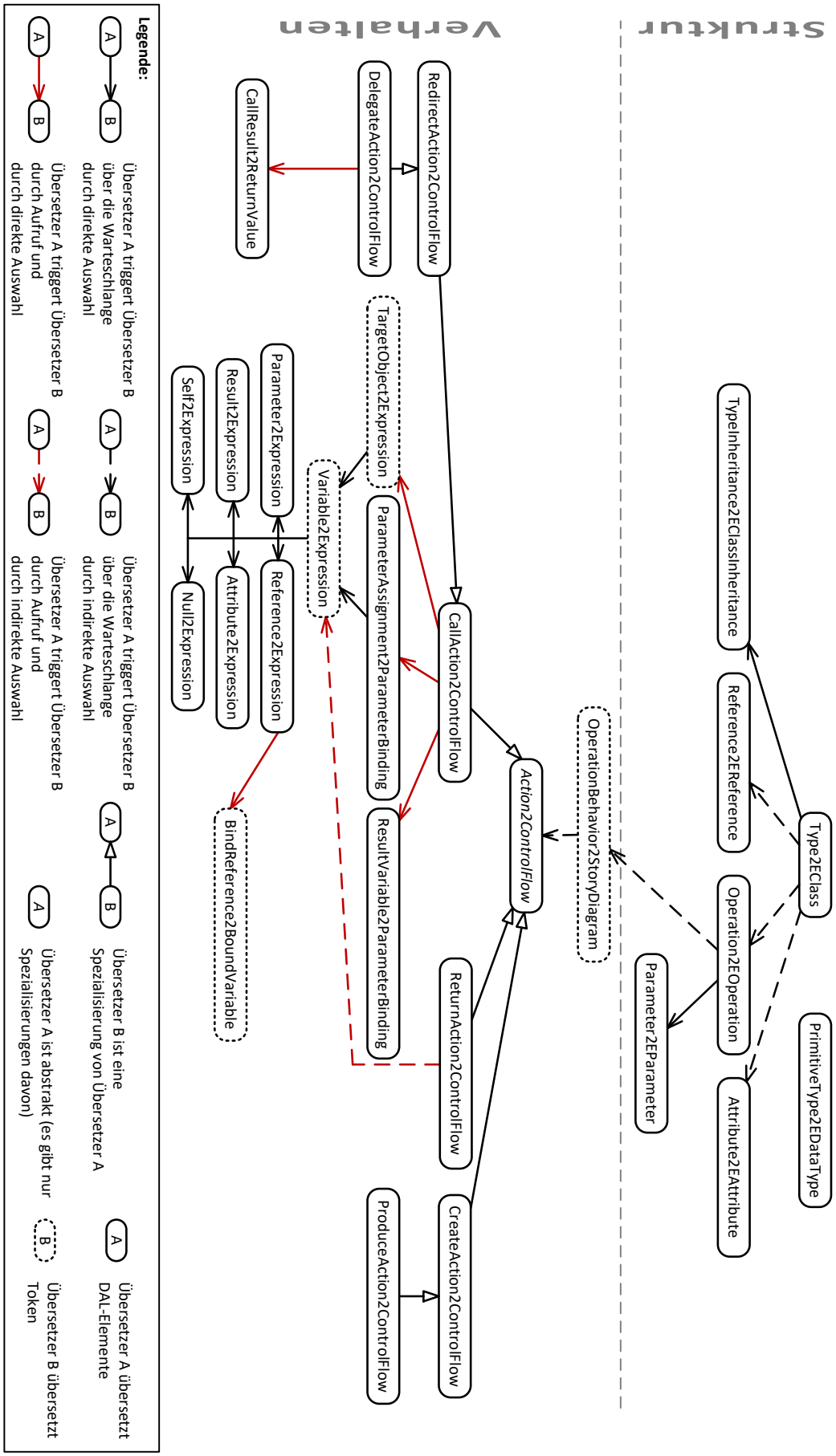


Abbildung C.8: Übersicht aller Elementübersetzer bzw. Transformationsregeln

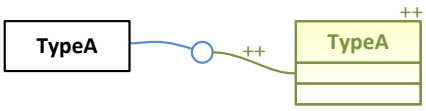


Abbildung C.9: Type2EClass

Bei komplexeren, mehrschrittigen Transformationen eines Elements, insbesondere einer Aktion, werden Tokens erzeugt, welche logische Teilstrukturen der zu erzeugenden Konstrukte repräsentieren und einzeln durch dafür verantwortliche Übersetzer transformiert werden. Zum Beispiel werden bei der Übersetzung einer call-Aktion Tokens verwendet (Details in Abschnitt C.3.3, S. 328 ff.), um das Objekt, auf dem der Aufruf erfolgt, einzeln zu übersetzen (TargetObject-Token) oder Teilergebnisse wie den generierten Kontrollflussabschnitt zu markieren (ControlFlow-Token).

Zwecks Übersichtlichkeit und Gleichbehandlung ähnlicher Fälle habe ich Vererbungsbeziehungen zwischen den Übersetzern verwendet und in der Abb. C.8 dargestellt. So wird zum Beispiel eine Aktion immer in einen Kontrollflussausschnitt eines Story-Diagramms übersetzt, weshalb es eine gemeinsame Schnittstelle für solche Übersetzer gibt (Action2ControlFlow). Abhängig vom Typ der Aktion sind jedoch verschiedene Übersetzer zuständig (z.B. CallAction2ControlFlow oder CreateAction2ControlFlow). Abstrakte Übersetzer, also (analog zu abstrakten Klassen) Übersetzer, von denen es keine konkrete Ausprägung gibt, werden in der Abbildung kursiv hervorgehoben.

Die Übersetzer sind in Java implementiert. Darum kann ich keine Spezifikation der Übersetzer als Transformationsregeln in einer der existierenden Transformationssprachen präsentieren. Um dennoch einen Eindruck von der Realisierung der einzelnen Übersetzer zu geben, stelle ich im Folgenden einige Übersetzer und die wesentlichen Konzepte dahinter exemplarisch in einer an relationale Transformationssprachen angelehnten, aus Platzgründen jedoch hauptsächlich auf konkreter Syntax basierenden Notation dar. Ergänzend dazu verwende ich Aktivitätendiagramme zur Beschreibung des Kontrollflusses einzelner Übersetzer.

C.3.2 Übersetzung von Klassenstrukturen in ein Ecore-Modell

Typen

Wie zuvor beschrieben beginnt meine Übersetzung mit den Typen im Anwendungsmodell (siehe Abb. 5.18 auf S. 129 und C.8 auf S. 322). Für die Übersetzung von nicht primitiven, als generierbar markierten Typen ist der Übersetzer Type2EClass verantwortlich, dessen wesentliches Verhalten ich in der Abb. C.9 skizziert habe. Er erzeugt eine zu dem Typ TypeA passende Klasse gleichen Namens im Ecore-Modell (ein EClass-Objekt) und verknüpft diese mit dem zugehörigen Typ im Anwendungsmodell. Diese Korrespondenz – blau dargestellt – wird als Kreis mit zwei ausgehenden Kanten abgebildet. Die erzeugten Elemente werden grün hervorgehoben und mit einem ++ markiert.

Das Verhalten des Übersetzers wird in der Abb. C.10 durch ein Aktivitätendiagramm beschrieben (vgl. Abb. 5.19 auf S. 130). Wie bei allen Übersetzern wird zunächst geprüft, ob der zu übersetzende Typ s bereits übersetzt oder einer existierenden Klasse zugeordnet wurde. Wenn nicht, wird eine neue Klasse (EClass)

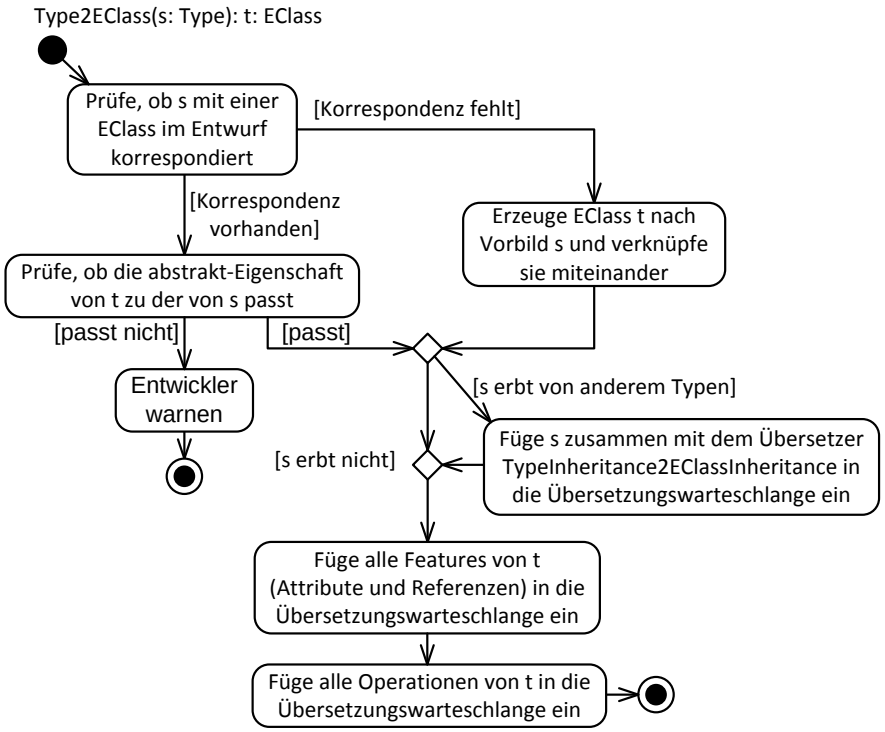


Abbildung C.10: Kontrollfluss des Übersetzers Type2EClass

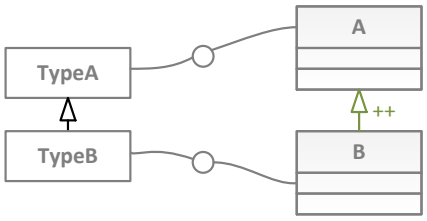


Abbildung C.11: TypeInheritance2EClassInheritance

mit den zum abstract-Attribut des Typen passenden Eigenschaften erzeugt und mit dem Typen in Korrespondenz gesetzt. Existiert eine mit dem Typ korrespondierende Klasse t schon, so wird nur die Konformität des abstract-Attributwerts des Typen zu den Eigenschaften der Klasse geprüft und der Entwickler im Fall einer Diskrepanz gewarnt. Ist der Typ s ein Untertyp eines anderen, wird anschließend die Übersetzung der Vererbungsbeziehung vorbereitet, indem der dafür zuständige Übersetzer TypeInheritance2EClassInheritance zusammen mit dem Typen s in die Übersetzungswarteschlange eingefügt wird. Abschließend werden alle Kindelemente des Typen, nämlich Attribute, Referenzen und Operationen, jeweils mit einem durch den Transformationsmechanismus bestimmten Übersetzer in die Warteschlange eingefügt und somit für eine spätere Übersetzung vorbereitet.

Die Vererbungsbeziehung ist nur eine Eigenschaft eines Typ-Objekts, wird also im Anwendungsmodell (bzw. der Musterspezifikation) nicht durch ein eigenes Objekt repräsentiert. Darum wird stellvertretend das Typ-Objekt, zu dem die Eigenschaft gehört, in die Übersetzungswarteschlange eingefügt sowie der Übersetzer direkt bestimmt und nicht vom Transformationsmechanismus gewählt.

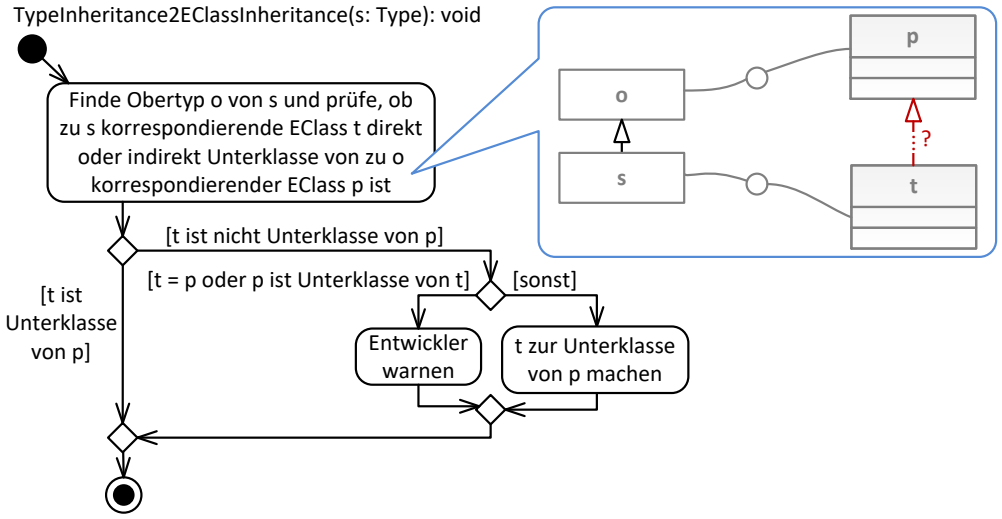


Abbildung C.12: Kontrollfluss des Übersetzers TypeInheritance2EClassInheritance

Vererbung

Die Übersetzung einer Vererbungsbeziehung eines Typen zu einem Obertyp wird vom Übersetzer TypeInheritance2EClassInheritance übernommen. Sein Verhalten ist in der Abb. C.11 skizziert. Dieser Übersetzer hat eine Vorbedingung, die erfüllt sein muss, bevor die Übersetzung erfolgen kann. Zu den beiden Typen, zwischen denen die Vererbungsbeziehung modelliert ist, muss jeweils eine damit korrespondierende Klasse im Entwurfsmodell existieren. Ist die Bedingung erfüllt, so wird zwischen den entsprechenden Klassen im Entwurfsmodell eine Vererbungsbeziehung erstellt. Andernfalls wird die Übersetzung durch den Transformationsmechanismus verzögert (siehe Schritte 3–5 in Abb. 5.18 auf S. 129).

Die Vorbedingung besteht bei diesem und anderen Übersetzern aus Elementen im Anwendungs- und Entwurfsmodell, die vor der Übersetzung vorhanden sein und auf bestimmte Weise in Korrespondenz zueinander stehen müssen (analog zu TGG und QVT). Solche Vorbedingungen stelle ich im Folgenden in Grau dar, die zu übersetzenden Elemente werden schwarz oder farbig dargestellt, die bei der Übersetzung erzeugten Elemente werden durch grüne Kanten und Schrift sowie das Label ++ hervorgehoben.

In der Abb. C.12 wird das Verhalten des Übersetzers TypeInheritance2EClassInheritance als Aktivitätendiagramm dargestellt. Weil das zu übersetzende Element – die Vererbungsbeziehung – in diesem Fall nicht durch ein Objekt im Anwendungsmodell repräsentiert wird, weicht das Verhalten etwas von dem eines Elementübersetzers ab. Es wird keine Korrespondenz eines Objekts im Anwendungsmodell zu einem im Entwurfsmodell geprüft, sondern die zu übersetzende Eigenschaft – die Vererbungsbeziehung der mit dem übergebenen Typen s korrespondierenden Klasse t.

Eine Vererbungsbeziehung zwischen zwei Typen im Anwendungsmodell repräsentiert eine direkte oder indirekte Vererbungsbeziehung in einem Entwurfsmodell. Darum wird geprüft, ob zwischen den Klassen t und p im Entwurfsmodell, welche mit den in Vererbungsbeziehung stehenden Typen s und o im Anwendungsmodell korrespondieren, eine direkte oder indirekte Vererbungsbeziehung vorliegt (siehe Skizze oben rechts in Abb. C.12). Liegt sie nicht vor, wird eine direkte

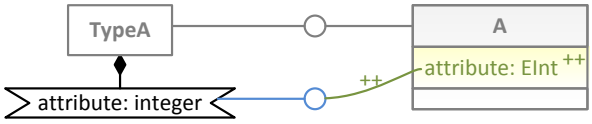


Abbildung C.13: Attribute2EAttribute

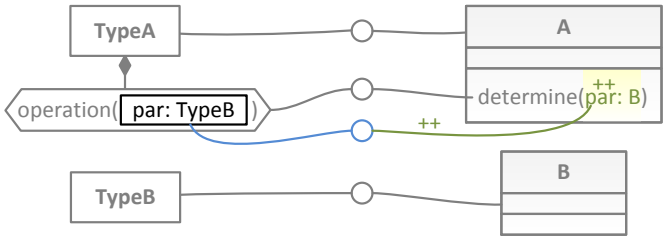


Abbildung C.14: Parameter2EParameter

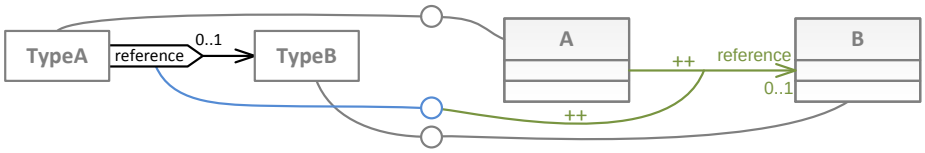


Abbildung C.15: Reference2EReference

Vererbungsbeziehung ergänzt, falls nicht schon eine Vererbungsbeziehung in umgekehrter Richtung vorliegt.

Referenzen, Attribute, Parameter & Co.

Die meisten anderen Übersetzer sind sehr ähnlich nach dem in der Abbildung 5.19 (S. 130) beschriebenen Prinzip aufgebaut, so auch zum Beispiel die in den Abb. C.13, C.14 und C.15 skizzierten Elementübersetzer. Sie haben eine Vorbedingung (grau dargestellt), welche die Verfügbarkeit der für die Übersetzung notwendigen Elemente im Entwurfsmodell fordert, erzeugen ein zu dem zu übersetzenden Element des Anwendungsmodell entsprechendes Element im Entwurfsmodell und stellen diese in Korrespondenz.

Operationen

Eine Besonderheit tritt bei der Übersetzung von Operationen auf. Zum einen ist hier die Übersetzung nicht nur von der zu übersetzenden Operation im Anwendungsmodell abhängig, sondern auch von schon existierenden Methoden im Entwurfsmodell. Zum anderen werden hier Tokens eingesetzt, um die Übersetzung des Operationsverhaltens in mehreren Schritten und dennoch nachvollziehbar durchzuführen.

Das Verhalten des Übersetzers ist in den Abb. C.16 und C.17 skizziert. Hier werden zwei Fälle unterschieden.

Im einfachsten Fall soll eine Operation übersetzt werden, die nicht in einer Spezialisierungsbeziehung zu einer anderen Operation steht (Abb. C.16). In diesem Fall müssen ausschließlich die Eigenschaften der zu übersetzenden Operation im Anwendungsmodell betrachtet werden. Gibt es keine mit der Operation korrespondierende Methode im Entwurfsmodell, so wird eine neue Methode gleichen Namens erzeugt. Ist ein Rückgabetyp spezifiziert, wird dieser entsprechend übersetzt, andernfalls wird void angenommen. Sind Parameter spezifiziert, werden auch diese

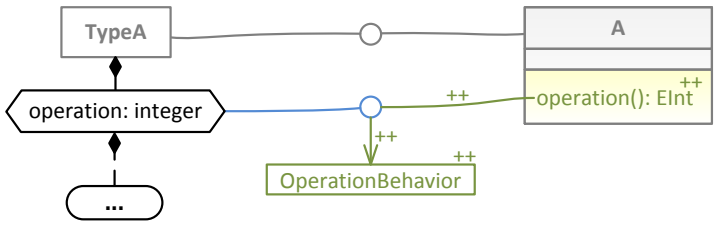


Abbildung C.16: Operation2EOperation ohne Spezialisierung

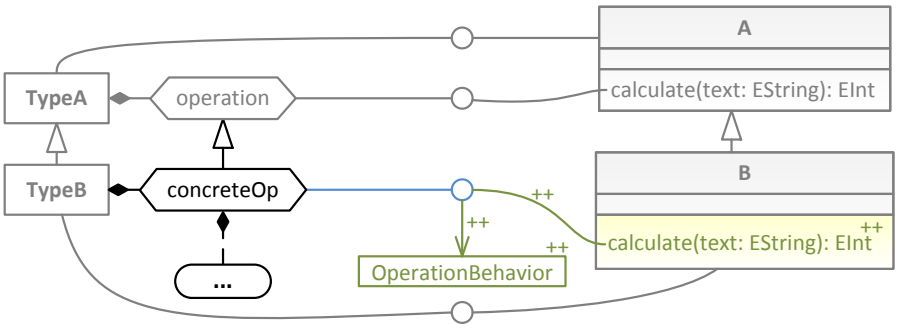


Abbildung C.17: Operation2EOperation mit Spezialisierung

mit Hilfe des zuständigen Übersetzers übersetzt.

Ist für die Operation Verhalten in Form von Aktionen spezifiziert, so wird ein OperationBehavior-Token (dargestellt als Kasten mit dem Tokentyp als Label) erstellt und an den zur Operation gehörenden Korrespondenzknoten gehängt. Das Token repräsentiert das gesamte Verhalten einer Operation[6] und damit eine logische, nicht explizit im Anwendungsmodell vertretene Einheit, welche einzeln durch einen dafür zuständigen Übersetzer übersetzt werden soll. In diesem Fall repräsentiert das Token ein noch zu erstellendes Story-Diagramm-Modell.

In dem Fall, dass die zu übersetzende Operation als Spezialisierung einer anderen Operation spezifiziert wurde (Abb. C.17), müssen bei der Übersetzung nicht nur die Eigenschaften der zu übersetzenden Operation im Anwendungsmodell betrachtet werden, sondern auch der Kontext der bei der Übersetzung zu erzeugenden Methode im Entwurfsmodell. Wird die Operation concreteOp aus Abb. C.17 übersetzt, ergibt sich die Signatur der erzeugten Methode aus der Signatur derjenigen Methode, die mit der spezialisierten – d.h. implementierten oder überschriebenen – Operation operation korrespondiert. Obwohl für die Operation concreteOp im Anwendungsmodell keine Parameter und kein Rückgabetyp spezifiziert wurden, erhält die Übersetzung der Operation beides.

Die Unterscheidung der beiden Fälle und die Behandlung des letzteren führen zu einem relativ komplexen Kontrollfluss des Übersetzers für Operationen. Dieser ist in vereinfachter Form in der Abb. C.18 dargestellt. Im Fall einer Spezialisierung wird die Signatur der spezialisierten Methode aus dem Entwurfsmodell anhand einer Hilfsoperation für die neue Methode übernommen. Andernfalls werden Name, Rückgabetyp und Parameter aus dem Anwendungsmodell als Vorgabe genommen. Existiert eine mit der Operation korrespondierende Methode bereits und fehlen

[6] Das Verhalten einer Operation kann durch mehrere Aktionen beschrieben werden.

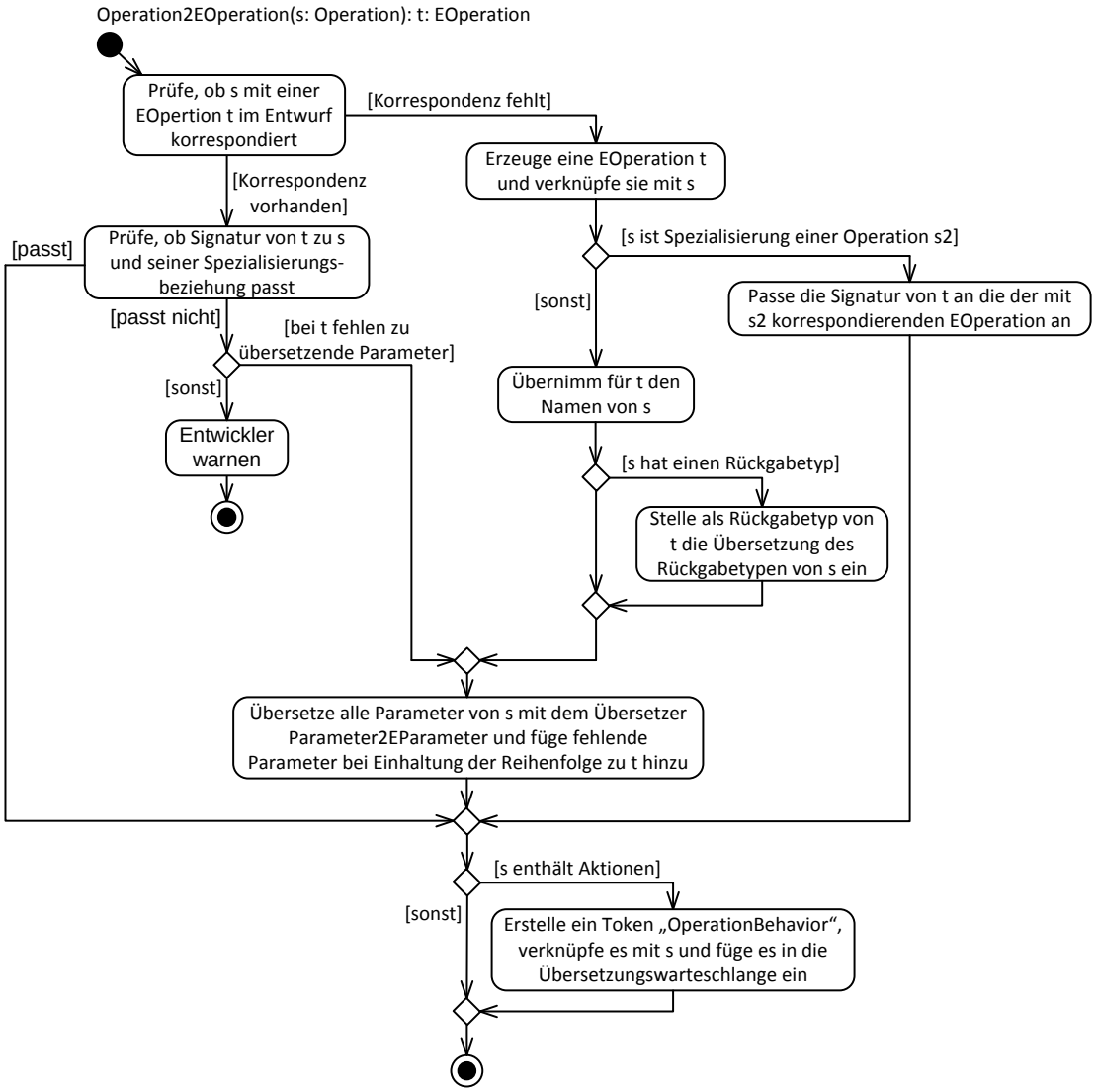


Abbildung C.18: Kontrollfluss des Übersetzers Operation2EOperation

ihr Parameter, welche denen der Operation entsprechen, so werden die fehlenden Parameter ergänzt[7]. Die Übersetzung von Parametern wird an den zuständigen Übersetzer delegiert. Im letzten Schritt wird durch das Erzeugen eines Operation-Behavior-Tokens die Übersetzung des Operationsverhaltens durch einen anderen Übersetzer vorbereitet, womit wir zur Generierung von Story-Diagrammen aus Aktionen kommen (siehe auch Abb. C.8 auf S. 322).

C.3.3 Übersetzung von Aktionen in Story-Diagramme

Das Verhalten einer Operation wird in Musterspezifikationen und somit auch in Anwendungsmodellen durch Aktionen beschrieben. Bei der Übersetzung des auf diese Wiese beschriebenen Verhaltens wird je ein Story-Diagramm generiert und

[7]Man beachte, dass Musterspezifikationen nur eine Mindestanforderung an die Implementierung stellen. Die Übersetzung einer Operation kann somit neben den für eine Operation spezifizierten auch zusätzliche Parameter enthalten.

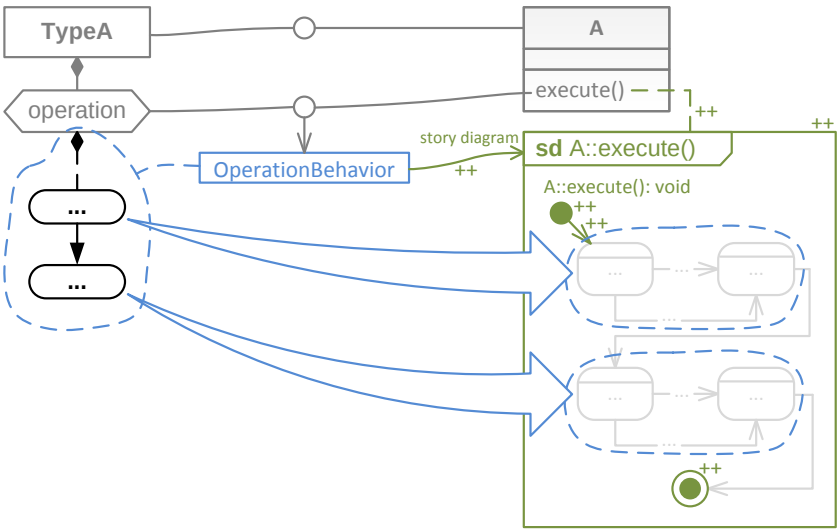


Abbildung C.19: OperationBehavior2StoryDiagram

mit der zur Operation gehörenden Methode im Ecore-Modell verknüpft (siehe Abb. C.19). Die zur Operation gehörenden Aktionen werden jeweils in Kontrollflussausschnitte im Story-Diagramm übersetzt. Diese können aus mehreren Aktivitätenknoten bestehen und werden analog zu den Aktionen zu einer Sequenz verknüpft.

Anders als bei der Übersetzung der Struktur in ein Ecore-Modell unterscheiden sich die Abstraktions- und Detaillevel der zu übersetzenden Aktionen und der daraus abgeleiteten Story-Diagramm-Strukturen deutlich. Aus diesem Grund wird die Übersetzung einer Aktion mit Hilfe von Tokens in mehrere kleinere Übersetzungsschritte zerlegt.

Warum Tokens?

Tokens ermöglichen zum einen eine logische Zerlegung der Elemente im Anwendungsmodell, insb. Aktionen, in einzeln übersetzbare Einheiten, wobei ein Token je eine übersetzbare Einheit repräsentiert. Zum anderen dokumentieren Tokens im Detail woraus die Story-Diagramm-Elemente schrittweise entstanden sind.

Die Übersetzung einer Operation wird mit Hilfe eines OperationBehavior-Tokens (siehe Abb. C.16 und C.17 auf S. 327 und Abb. C.19) in zwei Schritte zerlegt. Das Token repräsentiert das gesamte Verhalten einer Operation und wird einzeln übersetzt. Dadurch wird die Übersetzung des Operationsverhaltens in ein Story-Diagramm von der Übersetzung einer Operation in eine Methode entkoppelt.

Verhalten einer Operation bzw. OperationBehavior-Token

Für die Übersetzung des Operationsverhaltens bzw. des OperationBehavior-Tokens ist der Übersetzer OperationBehavior2StoryDiagram zuständig. Sein Verhalten ist in der Abb. C.19 skizziert. Er übersetzt alle Aktionen einer Operation in ein entsprechendes Verhaltensmodell – ein Story-Diagramm-Modell – und verknüpft dieses mit Hilfe des Tokens mit der Operation. Die Übersetzung der einzelnen Aktionen wird an die jeweils zuständigen Übersetzer delegiert und ihre Ergebnisse zu einem vollständigen Story-Diagramm kombiniert.

Im Detail ist diese Übersetzung im Aktivitätendiagramm in der Abb. C.20 dargestellt. Wenn es noch kein mit dem OperationBehavior-Token s verknüpftes Story-Diagramm t gibt, wird ein neues Story-Diagramm erstellt und mit dem Token unter dem Schlüssel story diagram verknüpft (siehe auch Abb. C.19). Die

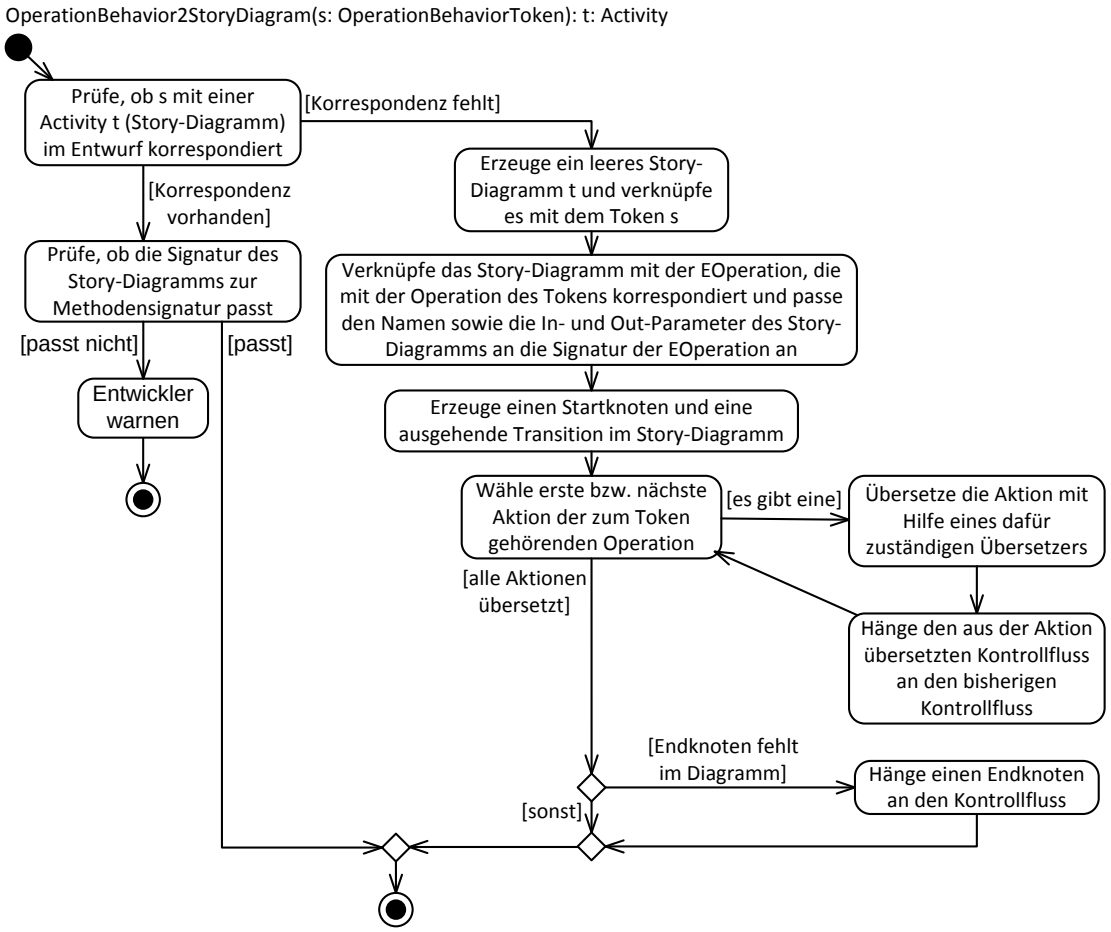


Abbildung C.20: Kontrollfluss des Übersetzers OperationBehavior2StoryDiagram

mit der Operation des Tokens korrespondierende Methode erhält einen Verweis auf das erzeugte Story-Diagramm, welches nun als Verhaltensmodell der Methode dient. Die Signatur des Story-Diagramms wird anschließend an die Signatur der damit verknüpften Methode angepasst. Dazu werden Name sowie Ein- und Ausgabeparameter analog zu denen der Methode eingerichtet.

Der Kontrollfluss im Story-Diagramm wird erzeugt, indem zunächst ein Startknoten und eine ausgehende Transition erstellt werden. Anschließend werden die Aktivitäten nacheinander mit Hilfe des für sie zuständigen Übersetzers übersetzt. Dabei wird jeweils ein Kontrollflussausschnitt erzeugt, welcher mit dem Startknoten bzw. dem zuletzt erzeugten Kontrollflussausschnitt verknüpft wird, sodass eine Sequenz solcher Kontrollflussausschnitte entsteht. Sollte der letzte Knoten dieser Sequenz kein Endknoten sein, wird abschließend ein Endknoten ergänzt und das Story-Diagramm somit vervollständigt.

Ist das OperationBehavior-Token bereits mit einem Story-Diagramm verknüpft, wird die Signatur der zum Story-Diagramm gehörenden Methode und die des Story-Diagramms auf Konformität zueinander geprüft und der Entwickler ggf. auf Abweichungen hingewiesen. Auf die Überprüfung des gesamten im Story-Diagramm modellierten Verhaltens auf Konformität mit den Aktionen im Anwendungsmodell wird aus Gründen der Komplexität verzichtet. Diese Überprüfung

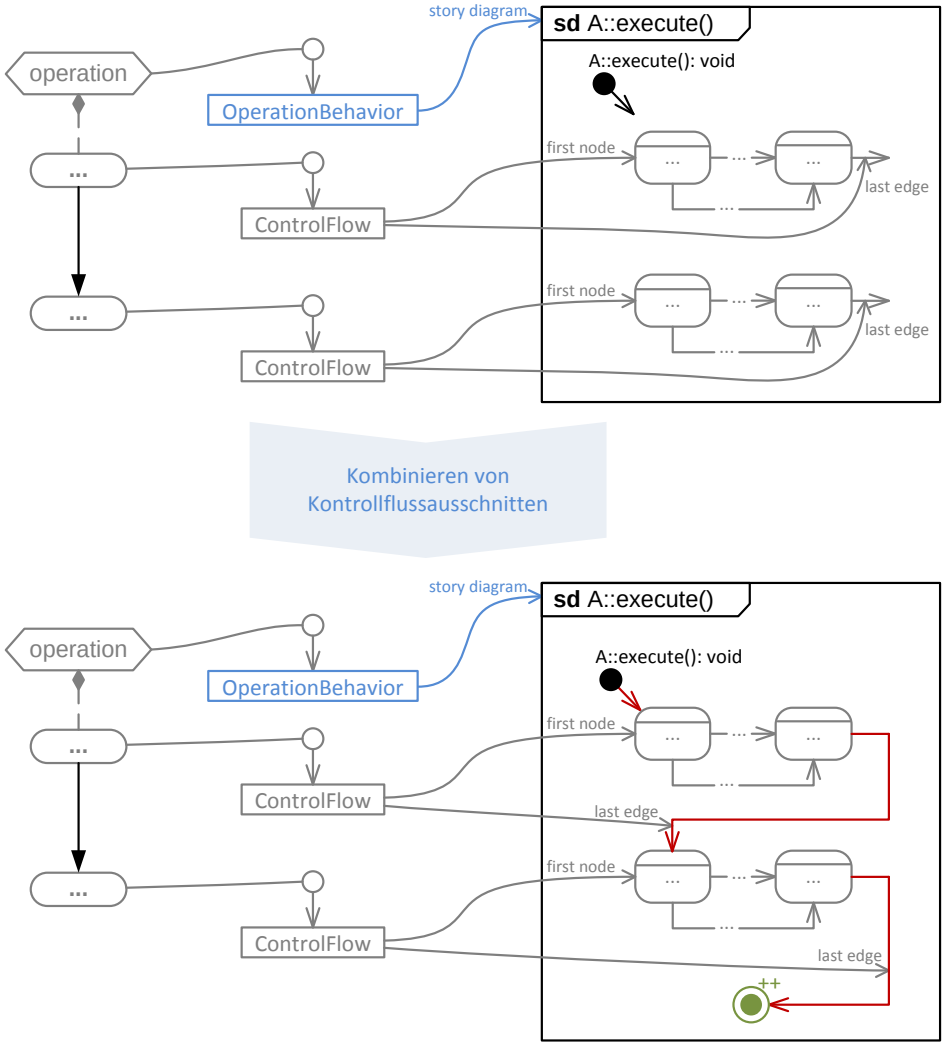


Abbildung C.21: Kombinieren der Übersetzungen von Aktionen

ist nur zum Teil automatisierbar (siehe Kapitel 6) und muss daher manuell vorgenommen werden.

Damit die einzelnen aus Aktionenübersetzungen entstandenen Kontrollflussausschnitte korrekt miteinander verbunden werden können, markieren die Übersetzer von Aktionen jeweils den ersten Aktivitätenknoten und die letzte Aktivitätenkante des erzeugten Kontrollflussausschnitts mit Hilfe eines ControlFlow-Tokens (siehe Abb. C.21 oben). Beim Kombinieren dieser Ausschnitte zu einem zusammenhängenden Kontrollfluss wird immer die letzte Kante des letzten Ausschnitts mit dem ersten Knoten des nächsten Ausschnitts verbunden (siehe Abb. C.21 unten).

Da der Übersetzer OperationBehavior2StoryDiagram auf die Ergebnisse anderer Übersetzer angewiesen ist, werden die Übersetzer direkt aufgerufen und nicht zeitversetzt vom Transformationsmechanismus über die Übersetzungswarteschlange. Das gleiche trifft auch auf die anderen für das Verhalten zuständigen Übersetzer zu.

Das Verhalten von Operationen wird in Musterspezifikationen ausschließlich

Aktionen

durch Aktionen beschrieben. Die Übersetzung der verschiedenen Aktionen erfolgt immer nach demselben Prinzip. Wie in der Abb. C.21 dargestellt, entstehen dabei immer Kontrollflussausschnitte eines Story-Diagramms, die zu einer Sequenz verbunden werden. Jeder Ausschnitt besteht aus mindestens einem Aktivitätenknoten. Wenn der letzte Aktivitätenknoten kein Endknoten ist, gibt es außerdem eine letzte Aktivitätenkante, die zum nächsten Story-Diagramm-Abschnitt führt. Außerdem setzen die Übersetzer von Aktionen Tokens ein, um die Übersetzung in logische Teilschritte zu zerlegen und die Teilergebnisse im Korrespondenzmodell zu persistieren.

Aufgrund ihrer Ähnlichkeit und zur besseren Wartbarkeit sind die Übersetzer von Aktionen in einer Vererbungshierarchie mit der abstrakten Oberklasse Action2ControlFlow organisiert. Die Abb. C.8 (S. 322) gibt einen Überblick über diese und andere Übersetzer.

Weil die Ergebnisse der Aktionenübersetzungen in Anhang A.3.2 beschrieben werden und alle Aktionenübersetzungen ähnlich aufgebaut sind, gehe ich im Folgenden nur exemplarisch auf die Übersetzung einer call-Aktion ein.

call-Aktion

Eine call-Aktion repräsentiert den Aufruf einer Operation (siehe Abschnitt 3.4.2, S. 61 ff.). Neben der aufrufenden und aufgerufenen Operation werden das Objekt, auf dem der Aufruf erfolgen soll, und ggf. die zu übergebenden Argumente spezifiziert. Wenn das Ergebnis des Aufrufs weiterverwendet werden soll, kann außerdem eine Variable deklariert werden, der das Ergebnis des Aufrufs zugewiesen wird und in folgenden Aktionen verwendet werden kann.

In einem Story-Diagramm wird aus einer call-Aktion ein Methodenaufruf, der als zusammengesetzter Ausdruck in einem Aktivitätenknoten modelliert wird. Zur Modellierung dieses Methodenaufrufs werden die folgenden vier Informationen benötigt:

(i) die aufzurufende Methode,
(ii) das Objekt, auf dem der Aufruf erfolgen soll (Zielobjekt),
(iii) die Argumente für den Aufruf, falls die Methode Parameter hat,
(iv) ggf. eine Variable zur Speicherung des Aufrufergebnisses.

Verweist die call-Aktion auf eine Referenz, muss die Referenz vor dem Methodenaufruf im Story-Diagramm aufgelöst (dereferenziert) werden, was weitere Aktivitätenknoten bei der Übersetzung erfordert.

Für die Übersetzung einer call-Aktion ist der Übersetzer CallAction2ControlFlow zuständig. Weil die Übersetzung abhängig von den Punkten (ii) bis (iv) stark variiert, wird sie in mehrere Teilschritte mit jeweils dafür zuständigen Übersetzern zerlegt. Die beteiligten Übersetzer sind in der Übersicht in der Abb. C.8 (S. 322) aufgeführt.

Der Übersetzer CallAction2ControlFlow erzeugt zunächst einen Aktivitätenknoten mit einer ausgehenden Kante und einem vorerst unvollständigen Methodenaufruf-Ausdruck. Der Ausdruck – eine MethodCallExpression – beschreibt (i) die aufzurufende Methode. Die Übersetzung der Punkte (ii) bis (iv) wird an die jeweils zuständigen Übersetzer delegiert und anschließend zu einem zusammenhängenden Story-Diagramm-Abschnitt bestehend aus mindestens einem Aktivitätenknoten und einem vollständigen Methodenaufruf-Ausdruck kombiniert. Der resultierende Story-Diagramm-Abschnitt ist in der Abb. C.22 dargestellt. Die von anderen

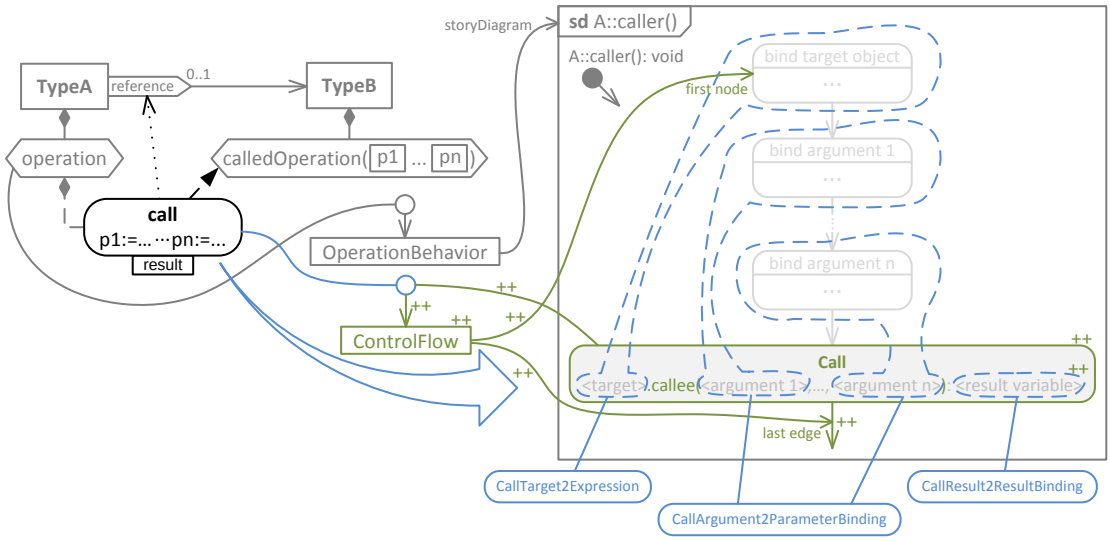


Abbildung C.22: Dekomposition der Übersetzung einer call-Aktion

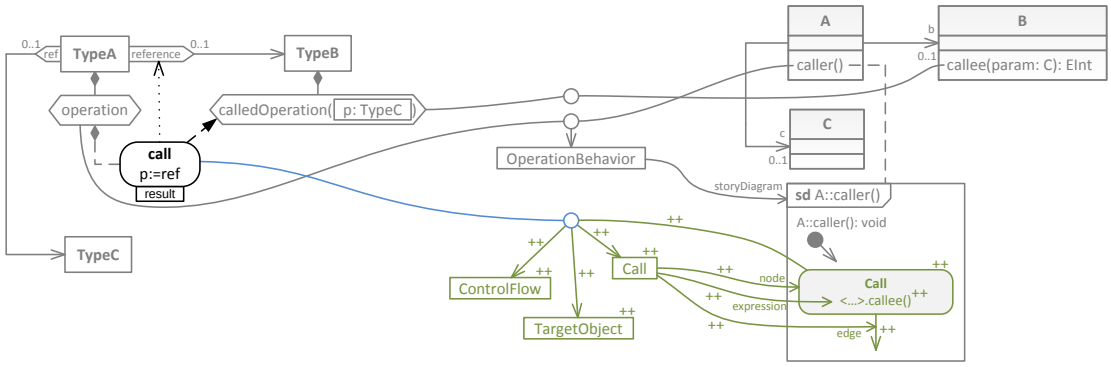


Abbildung C.23: CallAction2ControlFlow

Übersetzern erzeugten Zwischenergebnisse sind in Hellgrau angedeutet und getrichelt eingerahmt. Die jeweils zuständigen Übersetzer sind ebenfalls angegeben.

Zur Repräsentation der Teilaufgaben bzw. der einzeln übersetzten Teile einer call-Aktion werden Tokens verwendet, sofern kein entsprechendes Element im Anwendungsmodell vorhanden ist. So wird z.B. für das Zielobjekt ein eigenes Token TargetObject erzeugt und einzeln übersetzt. Außerdem werden Tokens zur Speicherung von Informationen verwendet. Zum Beispiel wird das ControlFlow-Token dazu verwendet, den ersten Knoten und die letzte ausgehende Kante eines Kontrollflussabschnitts in einem Story-Diagramm zu markieren (siehe Abb. C.22). Alle von dem Übersetzer CallAction2ControlFlow erzeugten Tokens sind in der Abbildung C.23 dargestellt. So wird auch ein Call-Token verwendet, um den bei der Übersetzung erzeugten Aktivitätenknoten, die zugehörige ausgehende Kante und den Ausdruck mit dem darin enthalenen Methodenaufruf zu markieren.

Das Verhalten des Übersetzers ist in der Abb. C.24 in Form eines Aktivitätendiagramms im Detail beschrieben. Wenn eine call-Aktion noch nicht übersetzt wurde, wird sie wie in der Abb. C.23 dargestellt zunächst in einen Aktivitätenknoten (StatementNode) mit vorerst unvollständigem Ausdruck für den Methodenaufruf

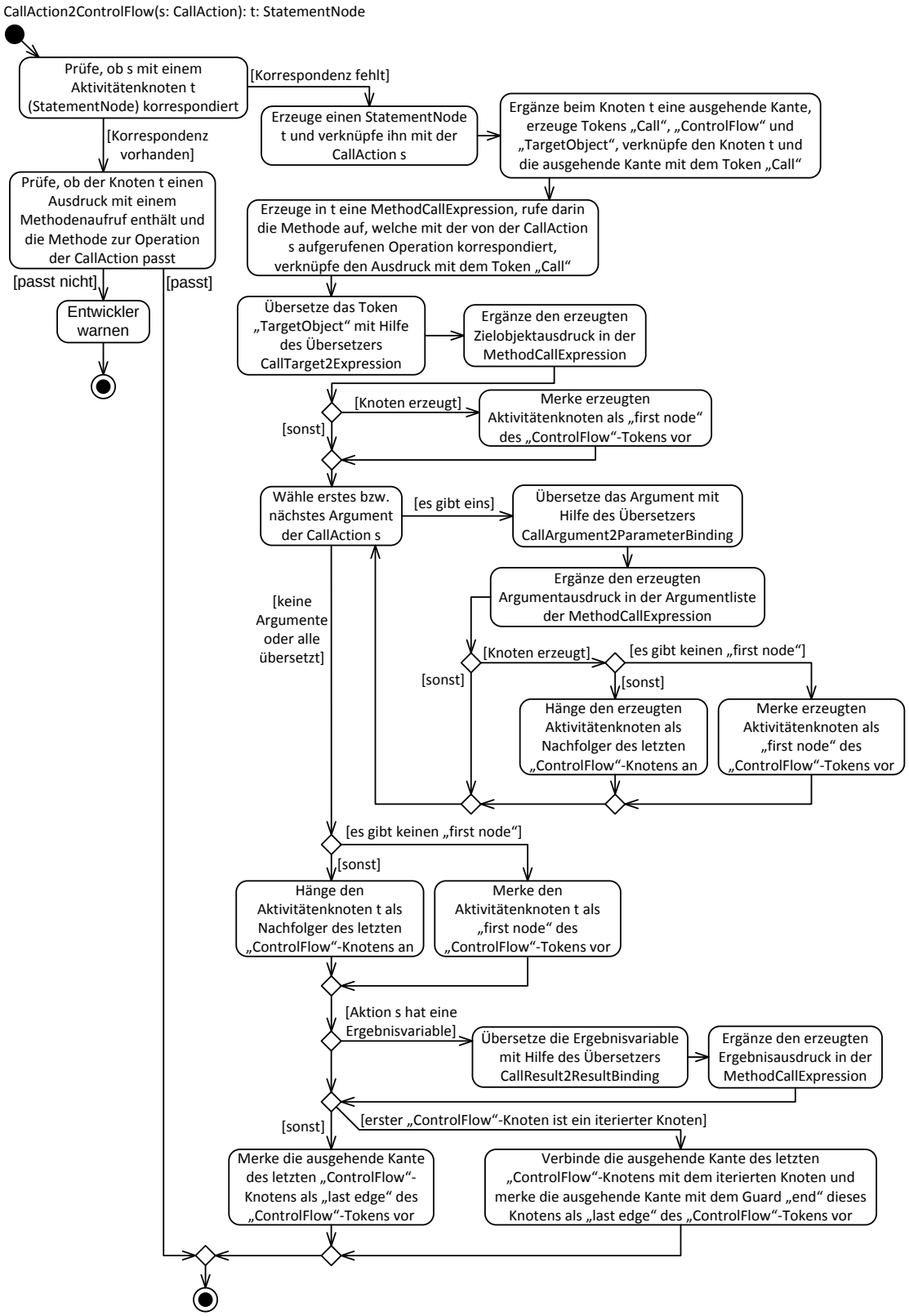


Abbildung C.24: Kontrollfluss des Übersetzers CallAction2ControlFlow

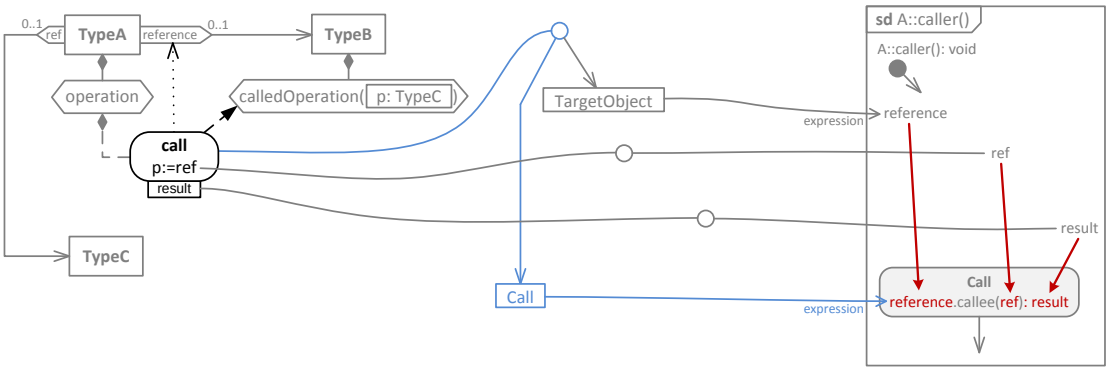


Abbildung C.25: Zusammenführen der Teilausdrücke zu einem

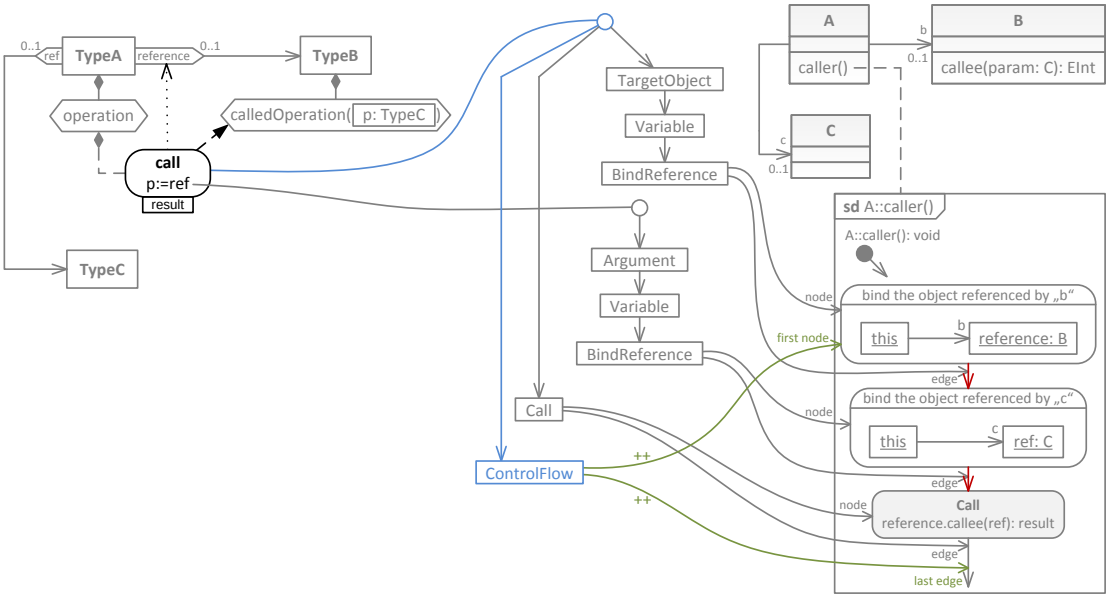


Abbildung C.26: Bilden einer Sequenz von Aktivitätenknoten

(MethodCallExpression) übersetzt. Die dabei erzeugten Elemente werden mit Hilfe des Call-Tokens markiert.

Das erzeugte TargetObject-Token wird zur Übersetzung an einen anderen Übersetzer gereicht. Schritt für Schritt werden auch die mit der call-Aktion spezifizierten Argumente und ggf. die Ergebnisvariable übersetzt. Die Teilergebnisse werden anschließend wie in der Abb. C.25 skizziert peu à peu zu einem vollständigen Methodenaufruf-Ausdruck zusammengeführt.

Gleichzeitig werden die ggf. bei der Übersetzung von Zielobjekt und Argumenten erzeugten Aktivitätenknoten zusammen mit dem zuvor erzeugten Aktivitätenknoten – darin ist der Methodenaufruf enthalten – zu einem zusammenhängenden Kontrollflussabschnitt in einem Story-Diagramm verbunden (Details zum Ablauf in Abb. C.24). Typischerweise ist der entstehende Kontrollflussabschnitt eine Sequenz von Aktivitätenknoten wie in den Abb. C.22 (S. 333) und C.26 dargestellt. Die ausgehenden Kanten der zuvor erzeugten Aktivitätenknoten werden mit den jeweils folgenden Aktivitätenknoten verbunden. Als letzter Aktivitätenknoten wird der Knoten mit dem Methodenaufruf angehängt. Der so ent-

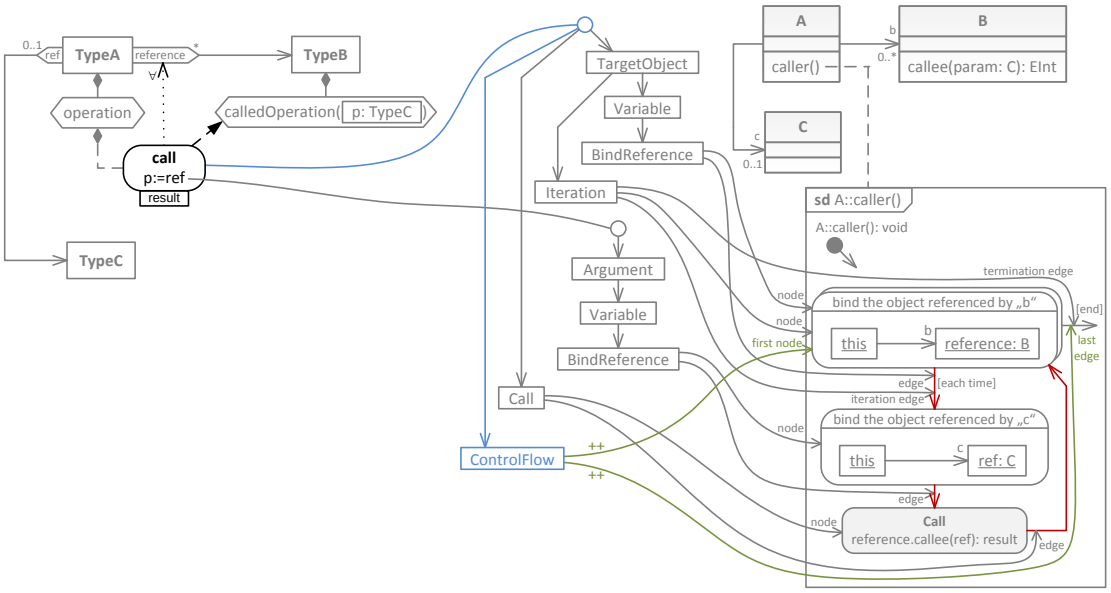


Abbildung C.27: Bilden einer iterierten Sequenz von Aktivitätenknoten

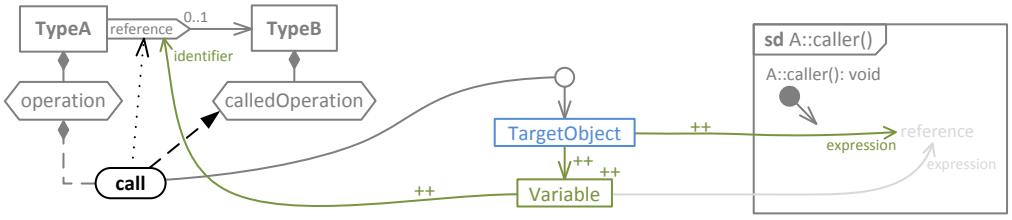


Abbildung C.28: CallTarget2Expression

standene Kontrollflussabschnitt wird mit Hilfe des ControlFlow-Tokens und seiner Verknüpfungen first node und last edge markiert.

Wird das Zielobjekt einer call-Aktion durch eine Referenz mit Kardinalität * ausgedrückt, so bedeutet das, dass nicht nur ein Methodenaufruf erfolgen soll, sondern mehrere; je ein Methodenaufruf auf jedem über die Referenz verknüpften Objekt. Darum entsteht bei der Übersetzung der call-Aktion in diesem Fall keine Sequenz von Aktivitätenknoten, sondern eine Schleife wie sie in der Abbildung C.27 dargestellt ist. In diesem Fall wird die Kante mit dem Guard [end] als last edge des ControlFlow-Tokens markiert.

Zielobjekt bzw. TargetObject-Token

Das Objekt, auf dem der Aufruf erfolgen soll (kurz: das Zielobjekt), wird durch den Übersetzer CallTarget2Expression übersetzt. Das Zielobjekt kann in Musterspezifikationen durch verschiedene Variablen, nämlich durch eine Referenz, einen Parameter, eine Ergebnisvariable einer anderen Aktion oder durch einen Selbstverweis beschrieben werden. Alle vier Fälle führen zu unterschiedlichen Ausdrücken bei der Übersetzung und werden deswegen von verschiedenen Übersetzern einzeln behandelt. Die zu übersetzende Variable wird durch ein Variable-Token mit einer identifier-Verknüpfung markiert (siehe Abb. C.28) und an einen der zuständigen Variable2Expression-Übersetzer (siehe Abb. C.8, S. 322) weitergereicht. Anschließend wird der bei der Übersetzung entstandene Ausdruck unter dem Schlüssel expression persistiert (siehe Abb. C.28).

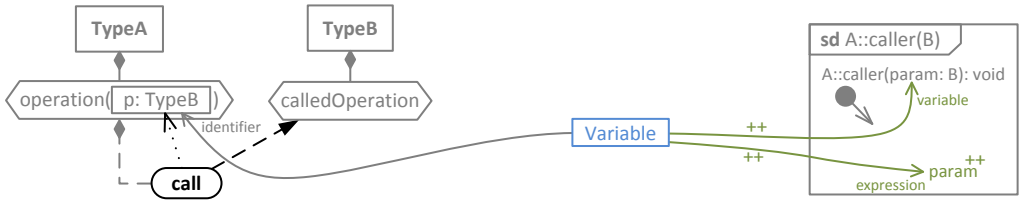


Abbildung C.29: Parameter2Expression

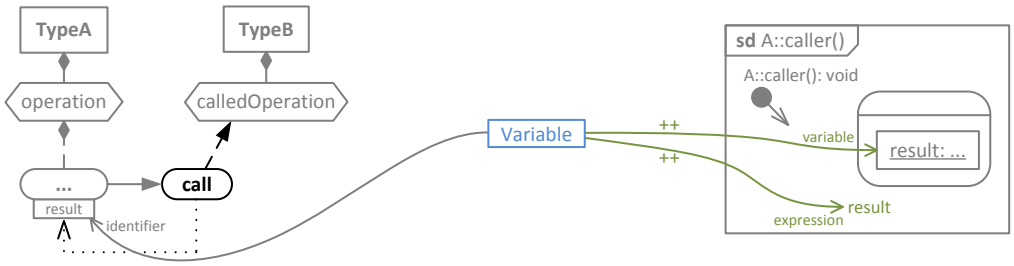


Abbildung C.30: Result2Expression

Variable-Token

Die Variable2Expression-Übersetzer identifizieren eine der angegebenen Variable im Anwendungsmodell entsprechende Variable im Story-Diagramm und erzeugen einen Ausdruck, der auf die Variable im Story-Diagramm verweist. Zum Beispiel wird aus einem Verweis auf einen Parameter im Anwendungsmodell ein Ausdruck, der auf einen entsprechenden Methodenparameter verweist (siehe Abb. C.29). Aus einem Verweis auf das Ergebnis einer vorhergehenden Aktion wird ein Ausdruck, der auf die entsprechende Objektvariable im Story-Diagramm verweist (siehe Abb. C.30). Bei allen Variable2Expression-Übersetzern ist der erzeugte Ausdruck eine ObjectVariableExpression oder bei Verwendung eines Parameters eine ParameterExpression. Mit Hilfe des Variable-Tokens und der Verknüpfungen expression und variable werden der erzeugte Ausdruck und die Objektvariable bzw. der Parameter, welcher im Ausdruck verwendet wird, markiert (siehe Abb. C.29 und C.30).

Ist die Variable im Anwendungsmodell eine Referenz, gibt es im Story-Diagramm keine entsprechende Variable. Die im Ausdruck verwendete Objektvariable muss zuerst deklariert und gebunden werden, was in einem zusätzlichen Aktivitätenknoten geschieht. Der für Referenzen zuständige Übersetzer Reference2Expression delegiert das Erzeugen dieses zusätzlichen Aktivitätenknotens an den Übersetzer

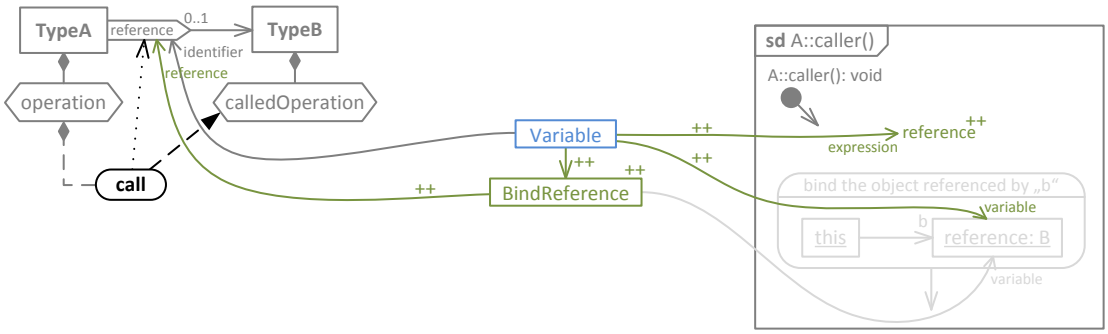


Abbildung C.31: Reference2Expression

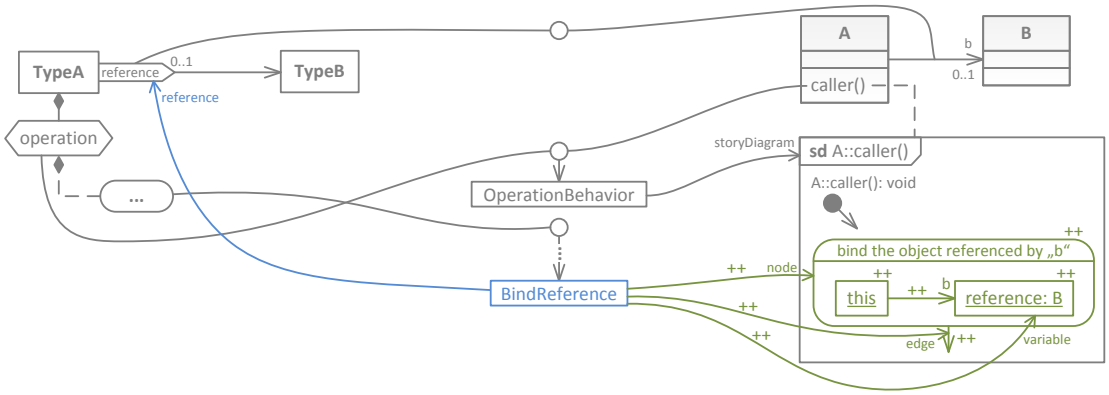


Abbildung C.32: Reference2BoundVariable

Reference2BoundVariable (siehe Abb. C.8, S. 322). Dazu wird mit dem BindReference-Token die Referenz markiert, deren Entsprechung im Entwursmodell dereferenziert und an eine neue Objektvariable gebunden werden soll (siehe Abb. C.31). In einem neuen Aktivitätenknoten mit einem darin enthaltenen Story-Diagramm wird die benötigte Objektvariable deklariert und an den aktuellen Wert der Referenz gebunden. Nach der Übersetzung verweist das BindReference-Token über die variable-Verknüpfung auf diese Objektvariable, sodass der Übersetzer Reference2Expression die Variable für das Erzeugen des auf die Variable verweisenden Ausdrucks nutzen kann.

BindReference-Token

Der Übersetzer Reference2BoundVariable übersetzt das Token BindReference in einen Aktivitätenknoten, in welchem die vom Token angegebene Referenz aufgelöst und das Ergebnis in einer neuen Objektvariable gebunden wird. Dazu wird ein Knoten (MatchingStoryNode) mit einem Story Pattern erzeugt (siehe Abb. C.32). In dem Story Pattern[8] wird von einer immer verfügbaren this-Variable ausgehend (eine Selbstreferenz) über die angegebene Referenz zu einem darüber referenzierten Objekt navigiert und dieses Objekt einer neuen Objektvariable zugewiesen (die Variable wird an das Objekt gebunden). Die neue Objektvariable erhält den Namen der Referenz im Anwendungsmodell. Ist der Name schon verwendet worden, wird eine Zahl an den Namen gehängt, die den Variablennamen im Story-Diagramm einmalig macht.

Iteration über mehrere Zielobjekte

Wird das Zielobjekt einer call-Aktion durch eine Referenz mit Kardinalität * in Kombination mit einem Allquantor beschrieben, so drückt die call-Aktion je einen Aufruf auf jedem referenzierten Objekt aus. Die dazu notwendige Schleife wird durch einen iterierten Aktivitätenknoten realisiert. In diesem Fall wird der bei der Übersetzung des BindReference-Tokens entstandene Aktivitätenknoten anschließend zu einem iterierten Knoten geändert (in der Abb. C.33 rot dargestellt, vgl. Abb. C.32). Für diese Änderung ist der Übersetzer MultipleCallTargets2ExpressionAndLoop verantwortlich, der eine Spezialisierung des Übersetzers CallTarget2Expression ist (siehe Abb. C.8, S. 322). Um die Iteration komplett zu machen, wird außerdem die ausgehende Kante des ursprünglichen Aktivitätenknotens mit dem Guard [each time] versehen und es wird eine zusätzliche ausgehende

[8] In diesem Fall ist es sogar ein Matching Pattern (MatchingPattern), also ein spezielles Story Pattern, das keine Änderungen am Wirtsgraphen vornimmt.

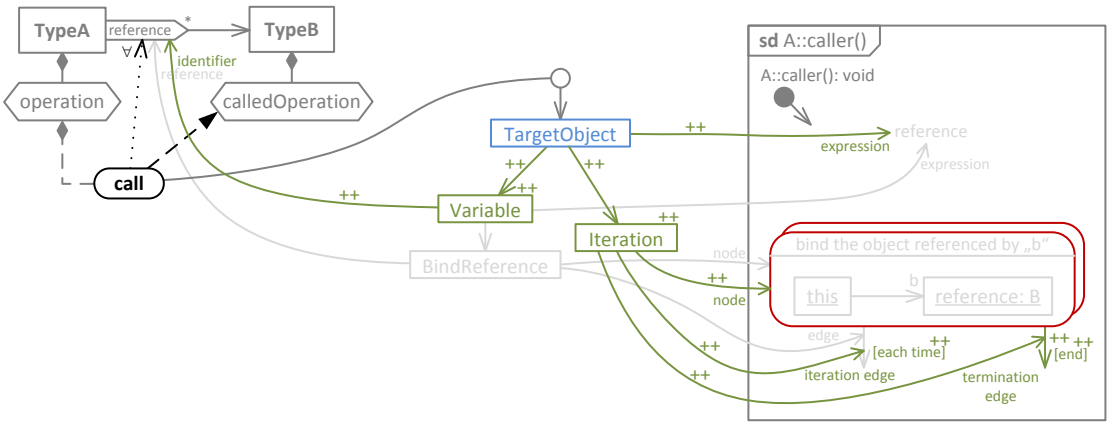


Abbildung C.33: MultipleCallTargets2ExpressionAndLoop

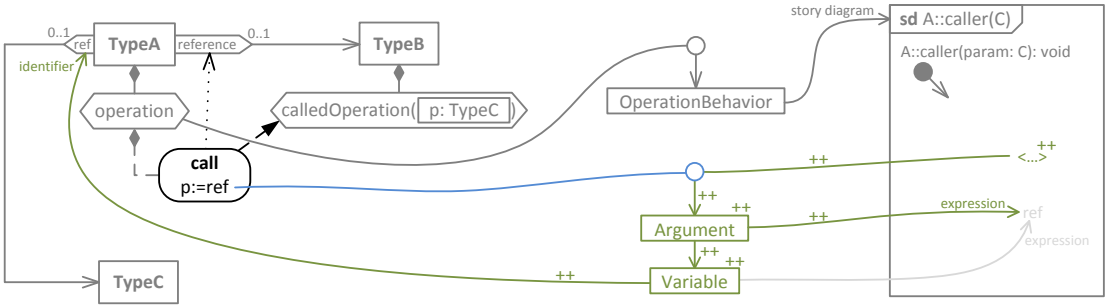


Abbildung C.34: CallArgument2ParameterBinding

Kante mit dem Guard [end] erstellt. Mit Hilfe eines Iteration-Tokens werden alle die Iteration definierenden Teile markiert: der iterierte Knoten, die Kante zu ggf. weiteren Aktivitätenknoten innerhalb der Iteration ([each time]) und die Kante zum ersten Knoten nach der Iteration ([end]).

Ist das Zielobjekt einer call-Aktion übersetzt, werden als Nächstes die Argumente des Aufrufs übersetzt. Dafür ist der Übersetzer CallArgument2ParameterBinding zuständig (siehe Abb. C.8, S. 322).

Die Argumente eines Operationsaufrufs werden in Musterspezifikationen und Anwendungsmodellen durch Zuweisungen von Variablenwerten an Parameter der aufgerufenen Operation (ParameterAssignment) beschrieben. Zum Beispiel wird in der Abb. C.34 dem Parameter p der Operation calledOperation der Wert der Variable (Referenz) ref zugewiesen. In Story-Diagrammen entsprechen die Parameterzuweisungen dem Binden einzelner durch Ausdrücke beschriebener Werte an Parameter der aufgerufenen Methode (InParameterBinding). Bei der Übersetzung erzeugt der CallArgument2ParameterBinding-Übersetzer zuerst ein InParameterBinding, welches der Parameterzuweisung im Anwendungsmodell entspricht, auf den Parameter der aufgerufenen Methode verweist (param in Abb. C.34), aber noch keinen zugewiesenen Ausdruck enthält (in Abb. C.34 dargestellt als <...>). Der zu erzeugende Ausdruck variiert abhängig von der Art der verwendeten Variable in der Parameterzuweisung im Anwendungsmodell und wird wie bei der Übersetzung von ein Zielobjekt beschreibenden Ausdrücken mit Hilfe eines von 6 Variable2Expression-Übersetzern erstellt (siehe Abb. C.8, S. 322). Erst danach wird der erzeug-

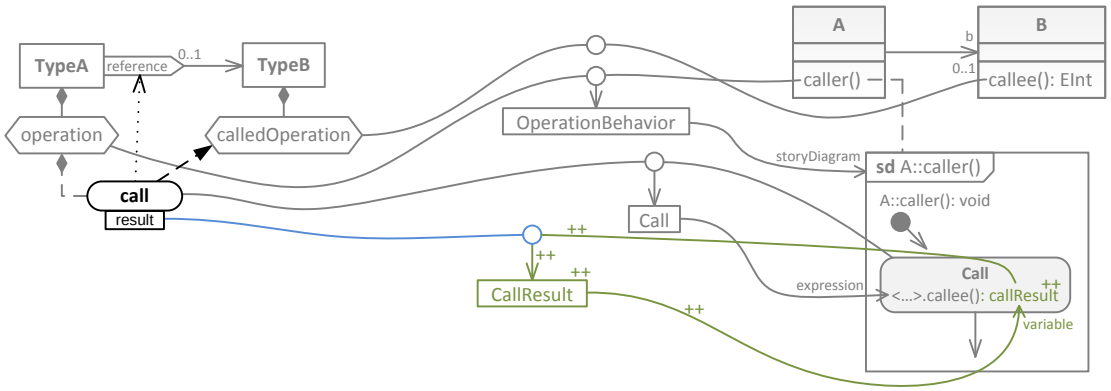


Abbildung C.35: CallResult2ResultBinding

te Ausdruck mit Hilfe des Argument-Tokens und seiner expression-Verknüpfung markiert. Zur Übersetzung der in der Parameterzuweisung verwendeten Variable im Anwendungsmodell wird die Variable mit Hilfe des Variable-Tokens über die identifier-Verknüpfung markiert und das Token zur Übersetzung an einen Variable2Expression-Übersetzer weitergereicht (vgl. Abb. C.34 und C.28, S. 336).

Wenn vorhanden, wird nach den Argumenten die Ergebnisvariable einer call-Aktion übersetzt. Diese Variable beschreibt das Ergebnis eines Operationenaufrufs zum Zweck der Weiterverwendung in einer folgenden Aktion. Auch in Story-Diagrammen kann das Rückgabeergebnis eines Methodenaurufs einer Variable zugewiesen werden. In dem den Methodenaufruf beschreibenden Ausdruck wird dazu eine neue Variable deklariert und ihr der Wert des Ausgabeparameters der aufgerufenen Methode zugewiesen (OutParameterBinding).

Bei der Übersetzung der Ergebnisvariable aus dem Anwendungsmodell durch den dafür zuständigen Übersetzer CallResult2ResultBinding wird eine neue Variable zur Speicherung des Ergebnisses eines Methodenaufrufs deklariert. Der Typ dieser Variable entspricht dem Rückgabetyp der aufgerufenen Methode (in Abb. C.35 die Variable callResult vom Typ EInt). Ein CallResult-Token markiert diese Variable über eine variable-Verknüpfung. Die Zuweisung des Aufrufergebnisses an diese Variable wird durch ein erzeugtes OutParameterBinding-Objekt beschrieben (in Abb. C.35 nicht dargestellt), welches auf die deklarierte Ergebnisvariable und auf den Ausgabeparameter der aufgerufenen Methode verweist. Existiert der Ausgabeparameter bei der aufgerufenen Methode trotz eines vorhandenen Rückgabetypen noch nicht, wird er ergänzt[9]. Man beachte, dass jede Methode mit Rückgabetyp $\neq$ void implizit einen unbenannten Ausgabeparameter hat. Dieser wird für die Verwendung in Story-Diagrammen explizit gemacht und benannt.

Sind das Zielobjekt, die Argumente und die Ergebnisvariable einer call-Aktion übersetzt, werden die Ergebnisse von dem CallAction2ControlFlow-Überstzer in dem Methodenaufruf-Ausdruck kombiniert (siehe Abb. C.25, S. 335). Die call-Aktion ist somit vollständig übersetzt. Andere Aktionen werden analog dazu schrittweise übersetzt und verwenden dabei zum Teil die gleichen Übersetzer zur Erzeugung von Zwischenergebnissen (siehe Abb. C.8, S. 322).

[9]Bei der aufgerufenen Methode wird ein OperationExtension-Objekt ergänzt, welches auf den neuen Ausgabeparameter (EParameter) verweist.

Anhang D

Schrittweise Musteranwendungen im JUnit-Framework

Zwecks Evaluation des im Rahmen dieser Arbeit vorgestellten Verfahrens zur modellgetriebenen Musteranwendung (siehe Abschnitt 8.2.3, S. 200 ff.) und zwecks Einschätzung seiner Praxistauglichkeit an einem realistischen Anwendungsbeispiel habe ich mit meinem Prototypen mehrere aufeinander folgende Musteranwendungen durchgeführt. Als Anwendungsbeispiel habe ich das Test-Framework JUnit[1] verwendet.

Zu diesem Framework in der Version 3.8.x existiert eine Entwicklerdokumentation, in der erläutert wird, welche Entwurfsmuter und in welcher Reihenfolge bei der Entwicklung des Frameworks angewandt wurden [Gam01]. Angelehnt an diese Dokumentation wiederhole ich die Entwicklungsschritte und Musteranwendungen mit dem von mir propagierten Verfahren. Im Gegensatz zum ursprünglichen Vorgehen, den Entwurf direkt in Java auszuprogrammieren, wende ich eine modellgetriebene Entwicklung an und modelliere sämtliche Klassen, ihr Verhalten und Musteranwendungen. Zur Modellierung von Klassenstrukturen dient mir dabei Ecore[2], zur Modellierung von Verhalten verwende ich Story-Diagramme [FNTZ00, vDHH+12].

D.1 Anwendung des Command-Musters vorbereiten

In einem ersten Schritt wird laut Entwicklerdokumentation [Gam01, Kap. 3.1] das Entwurfsmuster Command [GHJV95, S. 233 ff.] (siehe Spezifikation in Abb. B.15, S. 295) auf einer neuen Klasse TestCase in einem bisher leeren Entwurf (bzw. in einer bisher nicht vorhandenen Implementierung) angewandt. Zu diesem Zeitpunkt ist nur bekannt, dass die Klasse TestCase die Rolle Command in dem Muster einnehmen soll.

Unabhängig von dem Command-Muster werden anschließend weitere Eigenschaften der Klasse ergänzt. Das Ergebnis daraus ist das Codefragment D.1. Die Methode run soll die Rolle der execute-Operation einnehmen. Analog dazu habe ich die Klasse mit ihren Eigenschaften im Klassendiagramm links in Abb. D.1 modelliert und die Rollen des Musters den Entwurfselementen zugeordnet (rechts im Bild). Die eigentliche Musteranwendung (Ergänzung aller Entwurfselemente, welche die verbliebenen Rollen einnehmen) erfolgt in diesem Fall erst später.

[1] http://junit.org/

[2] Ecore ist ein Teil des Eclipse Modeling Frameworks (EMF) [SBPM08] und ist eine Implementierung der Essential MOF (EMOF), einer Teilmenge der MOF [OMG11c]

Codefragment D.1: Initiale Implementierung des JUnit-Frameworks

```
public abstract class TestCase implements Test {
   private final String fName;

   public TestCase(String name) {
      fName = name;
   }

   public abstract void run();

   ...
}
```

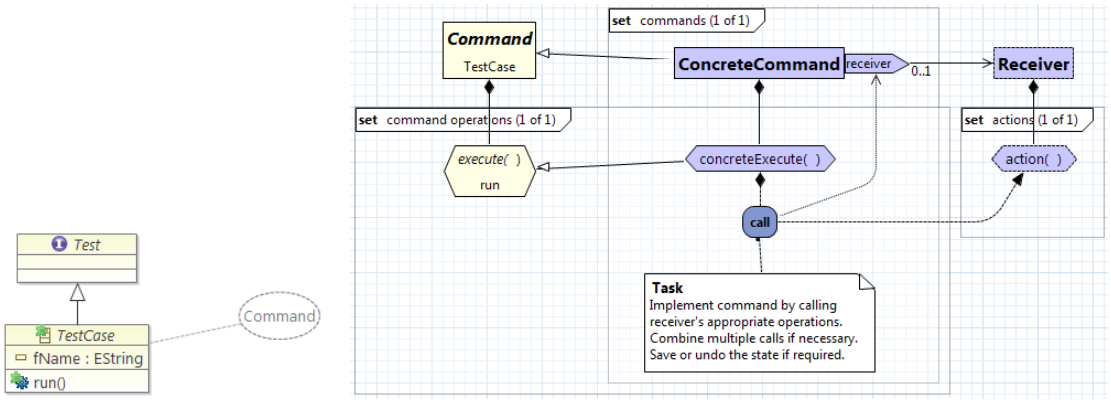


Abbildung D.1: Partielle Rollenzuordnung vor Anwendung des Command-Musters

D.2 Anwenden des Musters Template Method

Nach Anlegen der Klasse TestCase wird das Verhalten ihrer bisher abstrakten run-Methode festgelegt [Gam01, Kap. 3.2]. Sie soll drei Methoden nacheinander aufrufen, deren Verhalten durch Unterklassen bestimmt werden soll. Dazu wird das Muster Template Method [GHJV95, S. 325 ff.] (siehe Spezifikation in Abb. B.26, S. 298) angewandt und die run-Methode konkret gemacht. Das Ergebnis ist im Codefragment D.2 zu sehen.

Codefragment D.2: Implementierung der Methode *TestCase::run()*

```
public void run() {
   setUp();
   runTest();
   tearDown();
}
```

Die bereits am Command-Muster beteiligte run-Methode soll nun zusätzlich eine Rolle im Muster Template Method einnehmen. Die drei aufzurufenden Methoden fehlen bisher im Entwurf und sollen jeweils die Rolle einer primitiven Operation einnehmen, welche von der Template-Methode aufgerufen wird. Zwecks Anwendung des Musters definiere ich die Namen der zu erzeugenden primitiven Operationen in der Musteranwendungssicht und ordne der bereits existierenden run-Methode ihre Rolle zu (siehe Rollenzuordnungen in Abb. D.2 rechts).

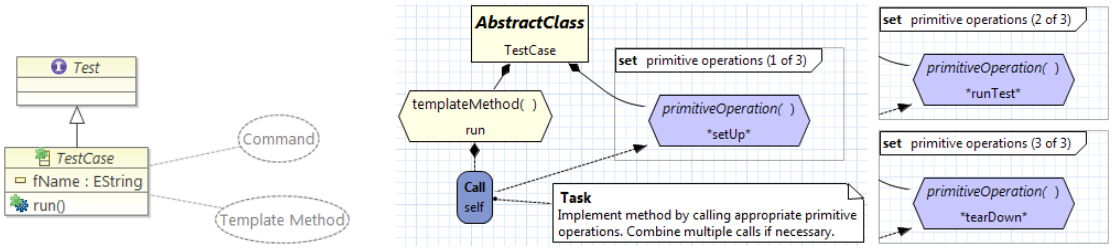


Abbildung D.2: Rollenzuordnung vor Anwendung des Musters Template Method

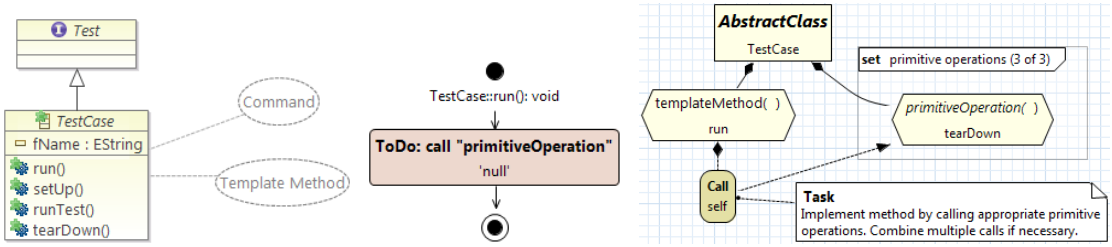


Abbildung D.3: Ergebnis nach Anwendung des Musters Template Method

Man beachte, dass sich die beiden Rollen der run-Methode widersprechen. Während die execute-Rolle des Command-Musters von der run-Methode verlangt, dass sie abstrakt ist, wird laut dem Muster Template Method eine konkrete run-Methode erwartet. Mit einer automatischen Validierung dieser beiden Anwendungsstellen ließe sich der Konflikt aufdecken.

Nach automatischer Musteranwendung sind die drei primitiven Operationen setUp, runTest und tearDown (links in Abb. D.3) sowie ein Story-Diagramm zur Modellierung des Verhaltens der run-Methode generiert worden (Abb. D.3, Mitte). Das Story-Diagramm hat neben Start- und Endknoten auch einen Aktivitätenknoten für das manuell zu modellierende Verhalten, hier die Aufrufe der drei primitiven Operationen. Dieser Aktivitätenknoten wird zusammen mit dem Story-Diagramm der call-Aktion der Musterspezifikation zugeordnet. Damit ist die Musteranwendung bis auf die offene Entwurfsaufgabe komplett und alle Rollen sind Elementen im Entwurf zugeordnet (rechts in Abb. D.3).

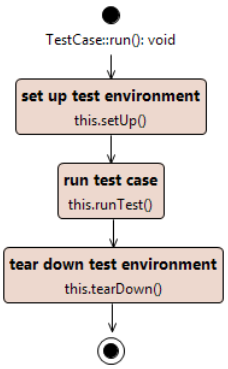


Abbildung D.4: Manuelle Vervollständigung des Verhaltens bei der Anwendung des Musters Template Method

Die noch offene Entwurfsaufgabe wird erledigt, indem der generierte Aktivitätenknoten um einen Methodenaufruf ergänzt wird und weitere Methodenaufrufe wie in der Abb. D.4 modelliert werden. Nach Abschließen der Aufgabe würde man die Entwurfsaufgabe in der Musteranwendungsansicht als erledigt markieren (vgl. Abb. 4.8, S. 89), was im vorliegenden Prototypen nicht möglich ist.

D.3 Anwenden der Musters Collecting Parameter

Als Nächstes soll die Klasse TestCase um eine Möglichkeit erweitert werden, die Ergebnisse der Testausführungen in einem Objekt zu sammeln. Dazu wird eine neue Klasse TestResult eingeführt [Gam01, Kap. 3.3]. Auch diese Klasse ist Teil einer Musteranwendung. An dieser Stelle wird das SmallTalk Best Practice Pattern [Bec96] Collecting Parameter angewandt (siehe Spezifikation in Abb. B.32, S. 301), in welchem die Klasse TestResult als Collecting Parameter dient. Die Implementierung dieser Klasse wird in der Entwicklerdokumentation von JUnit wie im Codefragment D.3 beschrieben.

Codefragment D.3: Implementierung der neuen Klasse *TestResult*

```
public class TestResult {
   protected int fRunTests;

   public TestResult() {
      fRunTests = 0;
   }
}
```

Diesen Entwicklungsschritt führe ich ebenfalls aus. Dazu modellieren ich zuerst die neue Klasse TestResult mit ihrem Attribut fRunTests[3] (siehe Abb. D.5 links) und ordne dieser Klasse anschließend die Rolle Result des Musters Collecting Parameter zu während die Klasse TestCase die Rolle ResultProducer erhält (siehe Abb. D.5 rechts).

[3]Den Konstruktor aus Codefragment D.3 habe ich der Einfachheit halber im Modell weggelassen. Er ließe sich jedoch in einer .genmodel-Datei samt Verhalten mitmodellieren, sodass hier keine Einschränkung im Vorgehen besteht.

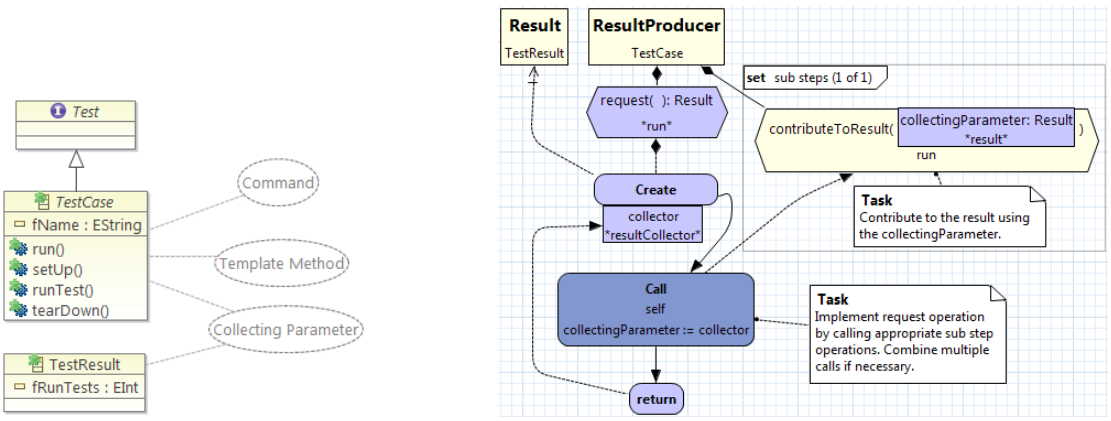


Abbildung D.5: Rollenzuordnung vor Anwendung des Musters Collecting Parameter

Codefragment D.4: Implementierung der Methode *TestCase::run(TestResult)*

```
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    runTest();
    tearDown();
}
```

In einem weiteren Schritt wird die bisher parameterlose Methode run der Klasse TestCase um einen Parameter vom Typ TestResult erweitert und das Verhalten der Methode erweitert, um die Ergebnisse der Testausführungen zu sammeln (siehe Codefragment D.4, vgl. Codefragement D.2, S. 342).

Die so erweiterte run-Methode befüllt also den Collecting Parameter und nimmt damit die Rolle contributeToResult der zugehörigen Musterspezifikation ein. Darum ordne ich diese Rolle entsprechend der run-Methode zu (siehe Abb. D.5 rechts). Der neue Parameter ist im Modell bisher nicht verfügbar und soll erst durch die automatische Musteranwendung ergänzt werden. Der zu verwendende Parametername (Rolle collectingParameter) ist in der Musteranwendungssicht mit result vordefiniert (siehe Abb. D.5 rechts).

Man beachte, dass die run-Methode bereits eine Rolle im Muster Temlate Method einnimmt (siehe Abb. D.2, S. 343). Obwohl die dort spezifizierte Rolle templateMethod keinen Parameter enthält, die Rolle contributeToResult des Musters Collecting Parameter jedoch einen Parameter spezifiziert, stellt das keinen Widerspruch dar. Bei meinem Ansatz sind Musterimplementierungen auch dann korrekt, wenn Methoden im Entwurf mehr Parameter enthalten als bei ihren Rollen in der zugehörigen Musterspezifikation definiert ist.

Neben der Erweiterung der bisher existierenden run-Methode um einen Parameter soll laut Entwicklerdokumentation auch eine neue parameterlose run-Methode in der Klasse TestCase ergänzt werden. Diese nimmt die Rolle request der Spezifikation des Musters Collecting Parameter ein. Sie erzeugt zunächst ein leeres TestResult-Objekt, führt einen Testfall aus und lässt dabei das Objekt mit Ausführungsergebnissen befüllen, abschließend gibt sie das befüllte Objekt als Ergebnis zurück. Der laut Entwicklerdokumentation [Gam01] zugehörige Java-Code ist der Abb. D.5 zu entnehmen.

Diese neue, parameterlose run-Methode fehlt bisher im Entwurf und soll ebenfalls durch die automatische Musteranwendung erzeugt werden. Darum definiere ich in der Musteranwendungssicht des Musters Collecting Parameter nur den

Codefragment D.5: Implementierung weiterer Methoden in der Klasse *TestCase*

```
public TestResult run() {
    TestResult result = createResult();
    run(result);
    return result;
}

protected TestResult createResult() {
    return new TestResult();
}
```

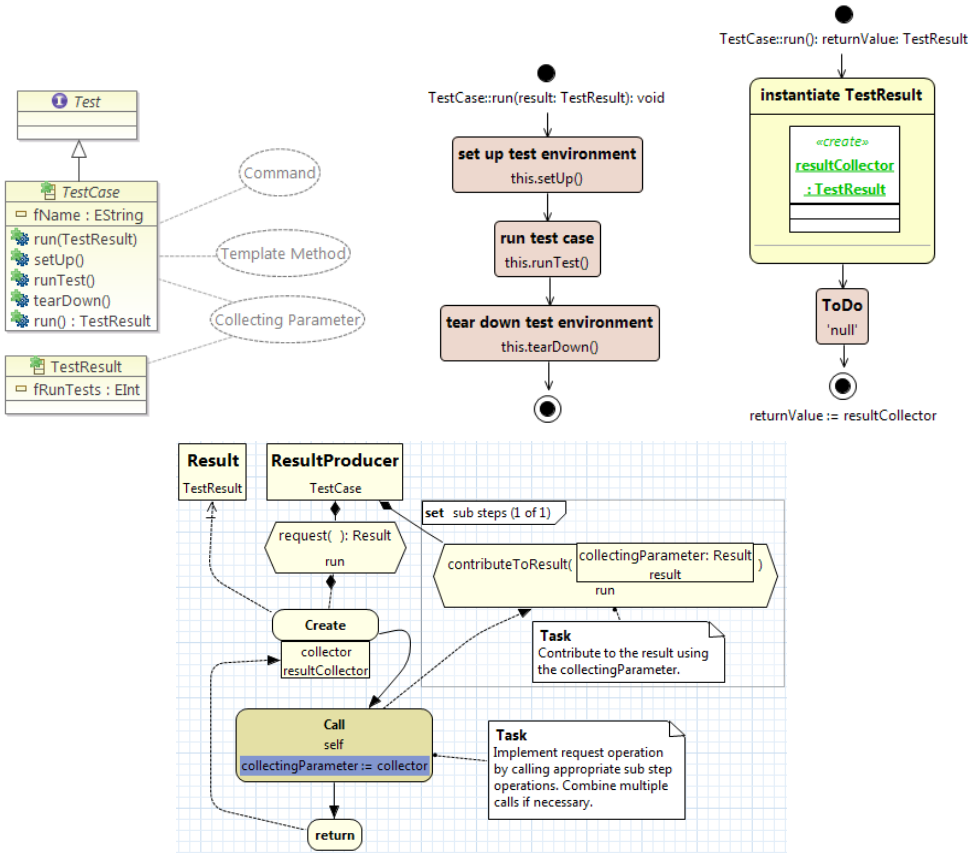


Abbildung D.6: Ergebnis nach Anwendung des Musters Collecting Parameter

Namen der zu erzeugenden Methode: run (siehe Abb. D.5 rechts). Wie in der Musterspezifikation definiert (ebenfalls Abb. D.5 rechts), soll diese Methode das Collecting-Parameter-Objekt – hier ein TestResult-Objekt (Rolle Result) – erzeugen (create-Aktion), durch Aufruf der Operation (Rolle) contributeToResult – hier run(TestResult) – befüllen lassen (call-Aktion) und schließlich das Ergebnis zurückgeben (return-Aktion und Rückgabetyp der Operation request).

Das Ergebnis nach automatischer Anwendung des Musters ist in der Abb. D.6 dargestellt. Im Klassendiagramm (oben links im Bild) wurde die existierende run-Methode um einen Parameter erweitert und eine neue run-Methode mit dem Rückgabetyp TestResult wurde generiert. Letztere ist mit einem ebenfalls generierten Verhaltensmodell in Form eines Story-Diagramms verknüpft (oben rechts im Bild). Das Verhaltensmodell der bisher parameterlosen run-Methode wurde um einen TestResult-Parameter ergänzt (Mitte oben im Bild). Die Rollenzuordnungen mit Ausnahme der Parameterübergabe sind erfasst worden (unten im Bild). Unter anderem wurden die Aktionen der Musterspezifikation den zugehörigen Aktivitätenknoten im Story-Diagramm zugeordnet. Für die call-Aktion wurde ein leerer Statement-Aktivitätenknoten mit dem Namen „ToDo“ generiert (oben rechts im Bild). Hier muss noch manuell der spezifizierte, nicht generierbare[4] Metho-

[4]Laut Spezifikation können beliebig viele contributeToResult-Operationen aufgerufen werden. Welche Operationen aufgerufen werden, ist vom Anwendungsfall abhängig, weswegen die Entscheidung vom Musteranwender gefällt wird und der Aufruf nicht generiert werden kann.

denaufruf ergänzt werden. Ebenso muss analog zum Codefragment D.4 (S. 345) der Aufruf der noch zu ergänzenden Methode startTest im Verhalten der Methode run(TestResult) der Klasse TestCase eingefügt werden.

Das Verhaltensmodell der Methode run der Klasse TestCase (siehe Abb. D.6 oben rechts) weicht hier etwas von dem Code in Codefragment D.5 (S. 345) ab, denn es gibt keine separate Methode createResult. Bis auf diesen Unterschied ist das Verhalten jedoch dasselbe.

Dateinamenskonflikt

Vor automatischer Anwendung des Musters war aufgrund einer Einschränkung im Prototypen noch ein manueller Schritt nötig. Die Werkzeuge, mit denen Story-Diagramme modelliert und mit Ecore-Modellen (Klassenmodellen) verknüpft werden, legen jedes Story-Diagramm in je zwei Dateien ab (einer Modelldatei und einer Diagrammdatei). Dabei wird der Dateiname auf Basis der Methodensignatur bestimmt. Zum Beispiel heißt die Modelldatei des Story-Diagramms für die parameterlose run-Methode aus Abb. D.4 (S. 343) vor Anwendung des Musters Collecting Parameter *framework.TestCase~run.sdm*. Nach Anwendung des Musters ändert sich die Methodensignatur von run() zu run(TestResult). Damit müsste sich eigentlich auch der Dateiname zu *framework.TestCase~run-TestResult.sdm* ändern. Der vorliegende Prototyp führt die Umbenennung der Dateien nicht selbstständig durch. Darum musste dieser Schritt vor Anwendung des Musters manuell erfolgen. So wurde u.a. ein Konflikt mit dem Dateinamen der bei Anwendung des Musters generierten Story-Diagramm-Modelldatei für die neue parameterlose run-Methode (Abb. D.6 oben rechts, S. 346) vermieden. Diese Modelldatei erhält den gleichen Dateinamen wie die Modelldatei der schon vorher existierenden, bisher parameterlosen und nun um einen Parameter erweiterten run-Methode (Abb. D.6 Mitte oben, S. 346).

Um den bisher fehlenden Methodenaufruf in der Implementierung des Musters Collecting Parameter im Entwurfsmodell zu ergänzen und damit die erste der beiden Entwurfsaufgaben zu erledigen, ergänze ich im Story-Diagramm der Methode run() manuell den Aufruf der Methode run(TestResult) (siehe Abb. D.7 rechts, vgl. Abb. D.6 oben rechts). Dabei übergebe ich analog zur Implementierung im Codefragment D.5 (S. 345) und konform zur Musterspezifikation unten in Abb. D.6

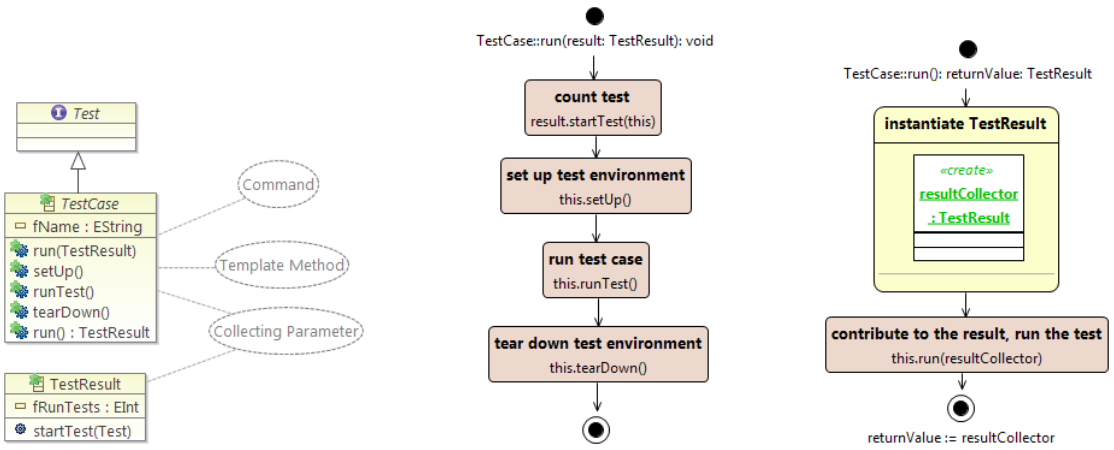


Abbildung D.7: Manuelle Entwurfsanpassungen nach Anwendung des Musters Collecting Parameter

(S. 346) das zuvor erzeugte TestResult-Objekt als Argument, welches im vorhergehenden Aktivitätenknoten in der Objektvariable resultCollector zwischengespeichert wurde (siehe Abb. D.7 rechts). Der Aufruf ist also als this.run(resultCollector) modelliert.

Als Nächstes ergänze ich im Klassendiagramm in der Klasse TestResult die Methode startTest(Test), damit ich anschließend den Aufruf dieser Methode im Story-Diagramm der Methode run(TestResult) ergänzen kann. Dieser Methodenaufruf ist im Aktivitätenknoten namens count test in der Mitte der Abb. D.7 modelliert. Als Argument des Aufrufs übergebe ich hier analog zum Codefragment D.4 (S. 345) das aufrufende TestCase-Objekt.

Codefragment D.6: Implementierung der Methode *TestResult::startTest(Test)*

```
public synchronized void startTest(Test test) {
   fRunTests++;
}
```

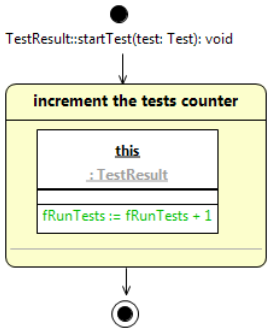


Abbildung D.8: Verhaltensmodell der Methode *TestResult::startTest(Test)*

Das Verhalten der Methode startTest(Test) wird in der Entwicklerdokumentation wie in Codefragment D.6 beschrieben. Es lässt sich wie im Story-Diagramm in Abb. D.8 modellieren. Damit ist auch die zweite Entwurfsaufgabe der Spezifikation des Musters Collecting Parameter erledigt (siehe Abb. D.6 unten, S. 346): der Collecting Parameter wird nun gefüllt (zählen der ausgeführten Tests). Leider kann die letzte fehlende Rollenzuordnung (Parameterübergabe) unten in Abb. D.6 (S. 346) nicht manuell vervollständigt werden, weil der Prototyp dafür keine Möglichkeit bietet.

D.4 Ergänzen einer Fehlerbehandlung

In einem weiteren Schritt wird die Methode run(TestResult) laut Entwicklerdokumentation [Gam01, Kap. 3.3] erneut angepasst. Es wird eine Fehlerbehandlung (Exception Handling) in Form eines `try`- und zugehöriger `catch`- und `finally`-Blöcke im Code ergänzt (vgl. Codefragement D.7 und D.4, S. 345). Exceptions werden bei Ausführung von Tests durch assert...-Methoden erzeugt. Die Implementierung einer dieser Methoden ist in Codefragment D.8 aufgeführt. Außerdem werden Methoden addFailure und addError in der Klasse TestResult ergänzt. Ihre

Codefragment D.7: Implementierung der Methode *TestCase::run(TestResult)*

```
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
        runTest();
    }
    catch (AssertionFailedError e) { // 1
        result.addFailure(this, e);
    }
    catch (Throwable e) { // 2
        result.addError(this, e);
    }
    finally {
        tearDown();
    }
}
```

Codefragment D.8: Implementierung der Methode *TestCase::assertTrue(boolean)*

```
protected void assertTrue(boolean condition) {
    if (!condition)
        throw new AssertionFailedError();
}
```

Codefragment D.9: Implementierung weiterer Methoden der Klasse *TestResult*

```
public synchronized void addError(Test test, Throwable t) {
    fErrors.addElement(new TestFailure(test, t));
}

public synchronized void addFailure(Test test, Throwable t) {
    fFailures.addElement(new TestFailure(test, t));
}
```

Codefragment D.10: Implementierung der Klasse *TestFailure*

```
public class TestFailure {
    protected Test fFailedTest;
    protected Throwable fThrownException;
}
```

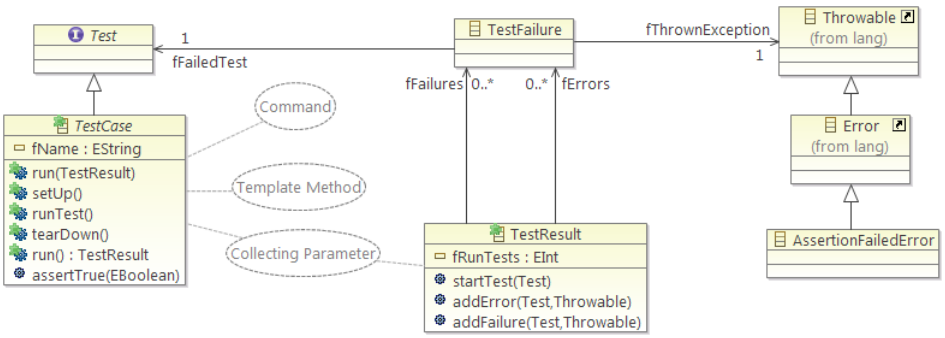


Abbildung D.9: Klassendiagramm nach manueller Ergänzung einiger Klassen

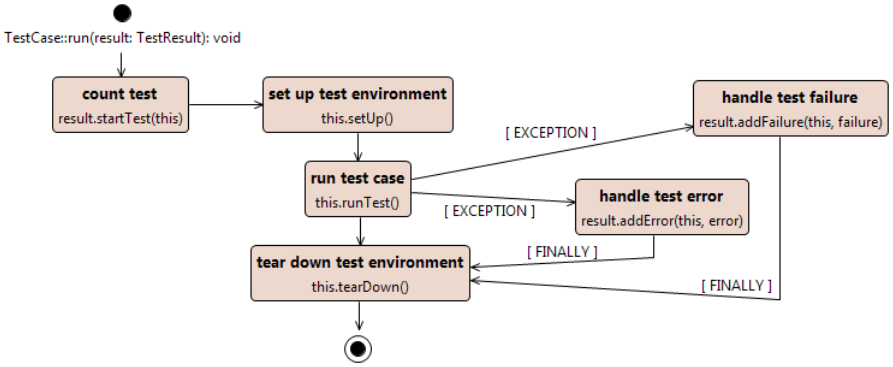


Abbildung D.10: Story-Diagramm der Methode *TestCase::run(TestResult)* nach manueller Erweiterung

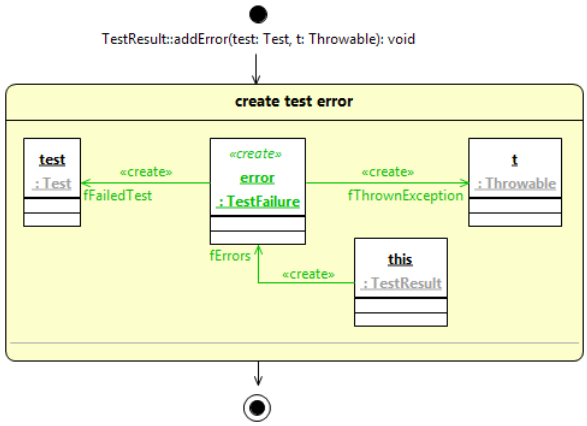


Abbildung D.11: Verhalten der Methode *TestResult::addError(Test, Throwable)*

Implementierung ist in Codefragment D.9 dargestellt. Zusätzlich wird eine neue Klasse TestFailure wie in Codefragment D.10 implementiert.

Um diese Schritte auch in meinem Entwurfsmodell nachzumachen, ergänze ich im Klassendiagramm die fehlenden Klassen. Das Ergebnis ist in der Abb. D.9 dargestellt (vgl. Abb. D.7 links, S. 347). Die Klasse TestResult erhält die bisher fehlenden Methoden addError und addFailure sowie zwei Assoziationen fFailures und fErrors zu der neuen Klasse TestFailure, um die gleichnamigen Objektvariablen aus Codefragment D.9 befüllen zu können. Die Objektvariablen der Klasse TestFailure aus Codefragment D.10 werden ebenfalls durch gleichnamige Assoziationen im Klassendiagramm in Abb. D.9 modelliert. Die Klasse TestCase erhält die Methode assertTrue aus Codefragment D.8.

Das Verhaltensmodell der Methode run(TestResult) der Klasse TestCase habe ich analog zum Codefragment D.7 um eine Fehlerbehandlung erweitert. Das resultierende Story-Diagramm ist in der Abb. D.10 dargestellt. Die beiden Kanten mit dem Guard „EXCEPTION“ entsprechen den beiden `catch`-Blöcken in Codefragment D.7. Sie werden nur dann durchlaufen, wenn innerhalb des Aktivitätenknotens mit der Beschriftung „run test case“ eine Exception vom Typ AssertionFailedError bzw. Throwable auftritt. Die Typen der Exceptions werden im vorliegenden Story-Diagramm-Editor nicht zusammen mit dem Guard an der

Kante angezeigt, sind aber mitmodelliert. Die Kanten mit dem Guard „FINALLY“ entsprechen dem `finally`-Block in Codefragment D.7.

Das Verhalten der Methode addError der Klasse TestResult ist im Story-Diagramm aus Abb. D.11 modelliert und entspricht dem Verhalten aus Codefragment D.9. Das Verhalten der Methode addFailure ist analog dazu modelliert. Das Verhalten der Methode assertTrue (siehe Codefragment D.8, S. 349) lässt sich mit dem vorliegenden Story-Diagramm-Editor nicht modellieren, das er keine Möglichkeit bietet, das Werfen einer Exception zu modellieren.

D.5 Anwenden des Adapter-Musters

Im nächsten Schritt wird in der Entwicklerdokumentation [Gam01, Kap. 3.4] die Anwendung des Adapter-Musters [GHJV95, S. 139 ff.] beschrieben. Dabei wird die Mustervariante Class Adapter verwendet (es gibt außerdem die verbreitetere Variante Object Adapter).

Codefragment D.11: Implementierung eines Tests mit anonymer, innerer Klasse

```
TestCase test = new MoneyTest("testMoneyEquals") {
      protected void runTest() {
         testMoneyEquals();
      }
   };
```

Laut Dokumentation wird das Muster im Java-Code angewandt, indem eine anonyme, innere Klasse instanziiert wird wie es in Codefragment D.11 exemplarisch dargestellt ist [Gam01, Kap. 3.4]. Hier wird eine anonyme Unterklasse der Klasse MoneyTest implementiert und instanziiert. Die exemplarisch implementierte Klasse MoneyTest ist eine Unterklasse von TestCase und bietet diverse Testfall-Implementierungen. Einen dieser Testfälle implementiert die Klasse MoneyTest in der Methode testMoneyEquals. Durch Überschreiben der Methode runTest aus der Klasse TestCase in der anonymen Klasse und durch Aufrufen der Methode testMoneyEquals wird in Codefragment D.11 festgelegt, dass hier nur der Testfall „testMoneyEquals“ ausgeführt werden soll.

Die Klasse MoneyTest ist nur ein Beispiel für eine mögliche Implementierung von Testfällen. Diese Klasse gehört eigentlich nicht direkt zum JUnit-Framework, sondern stellt einen Anwendungsfall des Frameworks dar. In einem modellgetriebenem Entwurf des JUnit-Frameworks würde man so eine Klasse meiner Meinung nach normalerweise nicht mitmodellieren.

Außerdem wird im Codefragment D.11 ein Java-spezifisches Programmierkonstrukt verwendet (anonyme, innere Klassen), welches sich in Ecore- und Story-Diagramm-Modellen nicht nachmodellieren lässt.

Aus diesen Gründen kann das Adapter-Muster in meinem Softwareentwurfsmodell nicht angewandt werden. Das ist jedoch auch nicht nötig, denn laut Entwicklerdokumentation wird vom JUnit-Framework noch eine zweite, häufiger genutzte Methode zum Aufrufen eines Testfalls angeboten. Bei dieser Methode kommen Java Reflection und das nächste angewandte Muster – Pluggable Selector – zum Einsatz.

D.6 Anwenden des Musters Pluggable Selector

Das nächste eingesetzte Muster ist Pluggable Selector [Gam01, Kap. 3.4], welches zu den SmallTalk Best Practice Patterns gehört [Bec96] (siehe Spezifikation in Abb. B.33, S. 301). Die Grundidee hinter dem Muster ist, das Verhalten einer Methode, sagen wir request(), flexibel zu gestalten, indem ihr Verhalten durch den Aufruf einer von mehreren anderen Methoden bestimmt wird. Welche Methode aufgerufen wird, bestimmt eine Variable, die auf die aufzurufende Methode verweist. Der Wert dieser Variable kann zur Laufzeit geändert und damit auch das Verhalten der request-Methode ausgetauscht werden.

Codefragment D.12: Implementierung der Methode *TestCase::runTest()* – Ausführen eines Testfalls per Reflection

```
protected void runTest() throws Throwable {
   Method runMethod= null;
   try {
      runMethod = getClass().getMethod(fName, new Class[0]);
   }
   catch (NoSuchMethodException e) {
      assertTrue("Method \"" + fName + "\" not found", false);
   }
   try {
      runMethod.invoke(this, new Class[0]);
   }
   // catch InvocationTargetException and IllegalAccessException
}
```

Im JUnit-Framework wurde das Muster mit Hilfe von Java Reflection angewandt. Dabei soll das Verhalten der Methode runTest der Klasse TestCase zur Laufzeit geändert werden können und darin eine der in Unterklassen zu implementierenden Testmethoden (ein Testfall) aufgerufen werden. Dazu wird der Name der aufzurufenden Methode in der Objektvariable fName der Klasse TestCase gespeichert (siehe Klassendiagramm in Abb. D.9, S. 349). Die Methode runTest wird wie in Codefragment D.12 implementiert. Sie sucht abhängig vom aktuellen Wert der Variable fName per Reflection die aufzurufende Methode heraus und ruft diese anschließend auf. (Da im JUnit-Framework erwartet wird, dass alle Testmethoden parameterlos sind, reicht der Name der aufzurufenden Testmethode, um sie eindeutig zu bestimmen.)

Zwecks Anwendung dieses Musters habe ich die Rollen der Musterspezifikation wie in der Abb. D.12 dargestellt den zugehörigen Elementen im Softwareentwurf zugeordnet. Die Klasse TestCase nimmt die Rolle OperationsProvider ein, ihr Attribut fName nimmt die Rolle pluggableSelector ein, während ihre Methode runTest die Rolle der request-Operation einnimmt (rechts in der Abbildung).

Laut Musterspezifikation soll die request-Operation den aktuellen Wert des Attributs pluggableSelector auslesen und abhängig von diesem Wert die zugehörige, aufzurufende pluggableBehavior-Operation aufrufen, wovon es beliebig viele geben kann.

Nach automatischer Anwendung des Musters erhält man das in der Abb. D.13 dargestellte Ergebnis. Die fehlende Operation pluggableBehavior wurde im Klassendiagramm in der Klasse TestCase ergänzt (oben links im Bild). Das Verhal-

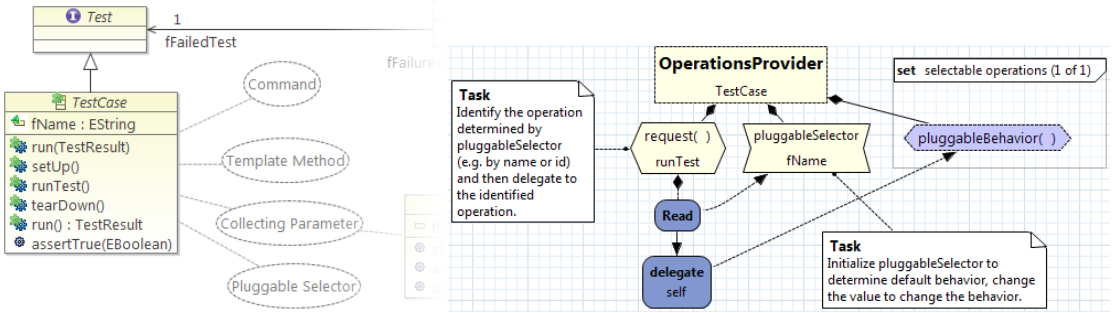


Abbildung D.12: Rollenzuordnung vor Anwendung des Musters Pluggable Selector

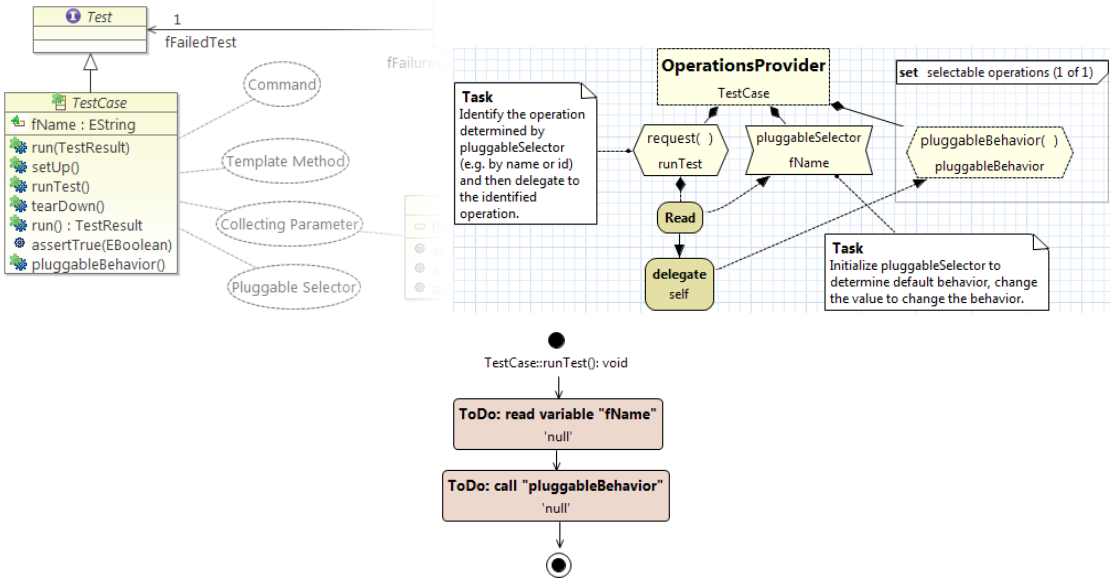


Abbildung D.13: Ergebnis nach Anwendung des Musters Pluggable Selector

tensmodell der Methode runTest wurde angelegt und sieht aus wie das Story-Diagramm in Abb. D.13 unten. Für die Aktionen read und delegate wurden leere Aktivitätenknoten generiert, weil aus der Spezifikation nicht ersichtlich ist wie genau der Wert der Variable fName ausgelesen und verwendet werden soll und an welche der beliebig vielen pluggableBehavior-Operationen delegiert werden soll.

generierte Methode muss entfernt werden

In diesem speziellen Anwendungsfall ist die generierte Methode pluggableBehavior sogar fehl am Platz. Die Rolle pluggableBehavior der Musterspezifikation soll von den später in Unterklassen von TestCase zu implementierenden Testmethoden eingenommen werden (analog zu testMoneyEquals in dem Beispielcode aus Codefragment D.11, S. 351). Welche das sein werden, ist zu diesem Zeitpunkt noch völlig unbekannt. Es sollte also ausnahmsweise noch gar keine Methode zu der Rolle pluggableBehavior der Musterspezifikation generiert werden.

Diese kleine Abweichung von der Intention in diesem Anwendungsfall des Musters lässt sich leicht korrigieren. Die generierte Methode pluggableBehavior wird aus der Klasse TestCase und aus dem Klassenmodell wieder entfernt. Ebenso wird auch die zugehörige Rollenzuordnung wieder aufgehoben (siehe Abb. D.14).

Um das Verhalten der Methode runTest analog zum Java-Code in Codefrag-

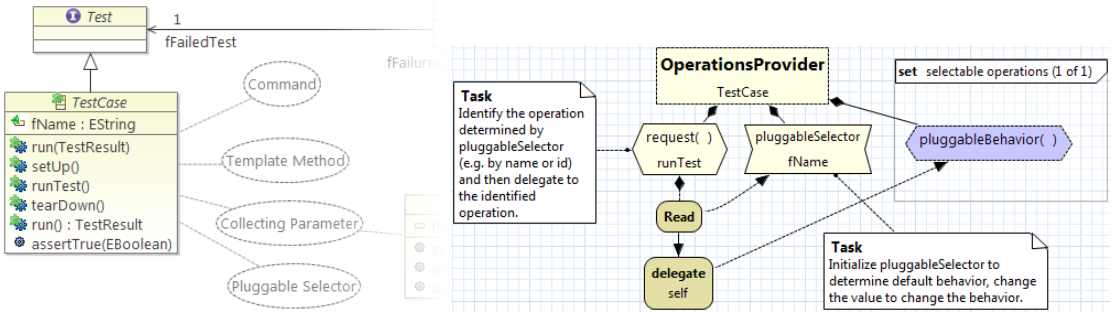


Abbildung D.14: Zustand nach Entfernen der unnötigerweise generierten Methode *TestCase::pluggableSelector()*

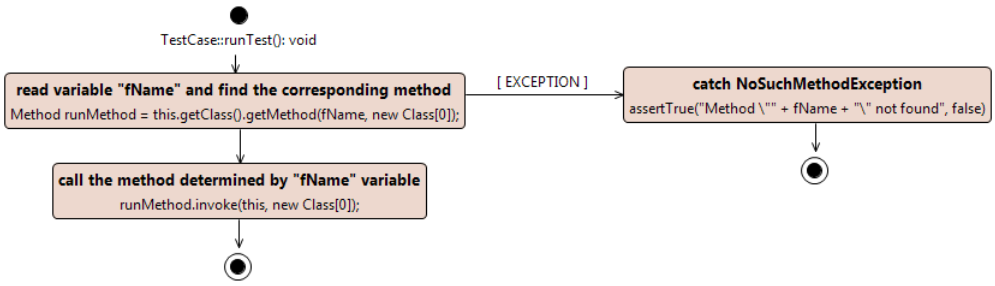


Abbildung D.15: Angepasstes Verhalten der Methode *TestCase::runTest()*

ment D.12 (S. 352) zu modellieren und die erste von zwei spezifizierten Entwurfsaufgaben zu erledigen, wird das zuvor generierte Story-Diagramm der runTest-Methode wie in der Abb. D.15 dargestellt erweitert (vgl. Abb. D.13 unten). Die Statement-Aktivitätenknoten des Story-Diagramms enthalten hier einige Java-Anweisungen aus Codefragment D.12, weil sich Reflection in Story-Diagrammen bisher nicht anders modellieren lässt.

Bis auf die offen gebliebene Entwurfsaufgabe (Initialisieren des pluggableSelector-Attributs) und die fehlenden pluggableBehavior-Operationen ist die Musteranwendung nun vollständig. Es geht nun erstmal mit der nächsten Musteranwendung weiter.

D.7 Anwenden des Composite-Musters

In dem nun folgenden Schritt [Gam01, Kap. 3.5] wird das Entwurfsmuster Composite [GHJV95, S. 163 ff.] angewandt (siehe Spezifikation in Abb. B.8, S. 293). Dieses Muster soll dabei helfen, eine beliebige Teilmenge von implementierten Testfällen in sogenannten Test Suiten zusammenzufassen, Test Suiten hierarchisch zu kombinieren sowie einzelne Testfälle und ganze Test Suiten auf gleiche Weise auszuführen.

Die Java-Schnittstelle Test soll dabei eine einheitliche Schnittstelle für einzelne Testfälle – bereits durch die Klasse TestCase implementiert – und Test Suiten darstellen und wird dazu um eine Methode run(TestResult) erweitert (siehe Codefragment D.13), analog zu der gleichnamigen Methode der Klasse TestCase (vgl. Abb. D.14 links). Diese Schnittstelle nimmt die Rolle Component des Musters ein.

Codefragment D.13: Erweiterung der Schnittstelle *Test*

```
public interface Test {
   public abstract void run(TestResult result);
}
```

Codefragment D.14: Implementierung der Klasse *TestSuite*

```
public class TestSuite implements Test {
   private Vector fTests = new Vector();

   public void addTest(Test test) {
      fTests.addElement(test);
   }

   public void run(TestResult result) {
      for (Enumeration e = fTests.elements(); e.hasMoreElements(); ) {
         Test test = (Test) e.nextElement();
         test.run(result);
      }
   }
}
```

Für die Rolle Composite wird eine neue Klasse TestSuite implementiert. Diese repräsentiert hierarchische Kompositionen von TestCase-Objekten oder Test Suiten. Die Klasse wird laut Entwicklerdokumentation wie in Codefragment D.14 implementiert.

Die bereits vorhandene Klasse TestCase nimmt die Leaf-Rolle ein und kann somit nicht aus mehreren TestCase-Objekten oder Test Suiten zusammengesetzt werden.

Das Besondere an der Composite-Klasse TestSuite ist, dass sie beliebig viele Test-Objekte enthalten kann und damit aus TestCase- und TestSuite-Objekten zusammengesetzt werden kann und dass sie ihr Verhalten in der Methode run(TestResult) an alle Kindobjekte delegiert (siehe Codefragment D.14).

manuelle Erweiterung der Schnittstelle nötig

Nach manuellem Erweitern der Test-Schnittstelle[5] im Klassendiagramm um die Methode run(TestResult) analog zum Codefragment D.13 kann die automatische Musteranwendung vorbereitet werden, indem die Rollen der Musterspezifikation wie in Abb. D.16 den Entwurfselementen zugeordnet werden. Wie beschrieben nehmen die Klassen Test und TestCase die Rollen Component bzw. Leaf ein. Die run-Methode der Schnittstelle Test nimmt die Rolle operation ein, die der Klasse TestCase die Rolle leafOperation. Für die Rolle Composite gibt es noch keine Entsprechung im Entwurf, hier wird nur der zu verwendende Name TestSuite festgelegt. Bei der Rolle compositeOperation braucht vor Musteranwendung nichts näher definiert zu werden, da die Methodensignatur anhand der Spezialisierungsbeziehung zur Rolle operation automatisch hergeleitet wird. Für die Rolle children – eine Referenz von Composite zu Component – wird ebenfalls ein zu verwendender Name festgelegt, hier fTests (analog zum Namen der Objektvariable in Codefragment D.14).

Nach automatischer Musteranwendung erhält man das in der Abb. D.17 darge-

[5] Die Methodensignatur kann bei einer automatischen Musteranwendung im vorliegenden Prototypen nicht von einer Unterklasse zu einer ihrer Oberklassen propagiert werden. Die Propagierung wird nur abwärts in der Klassenhierarchie unterstützt.

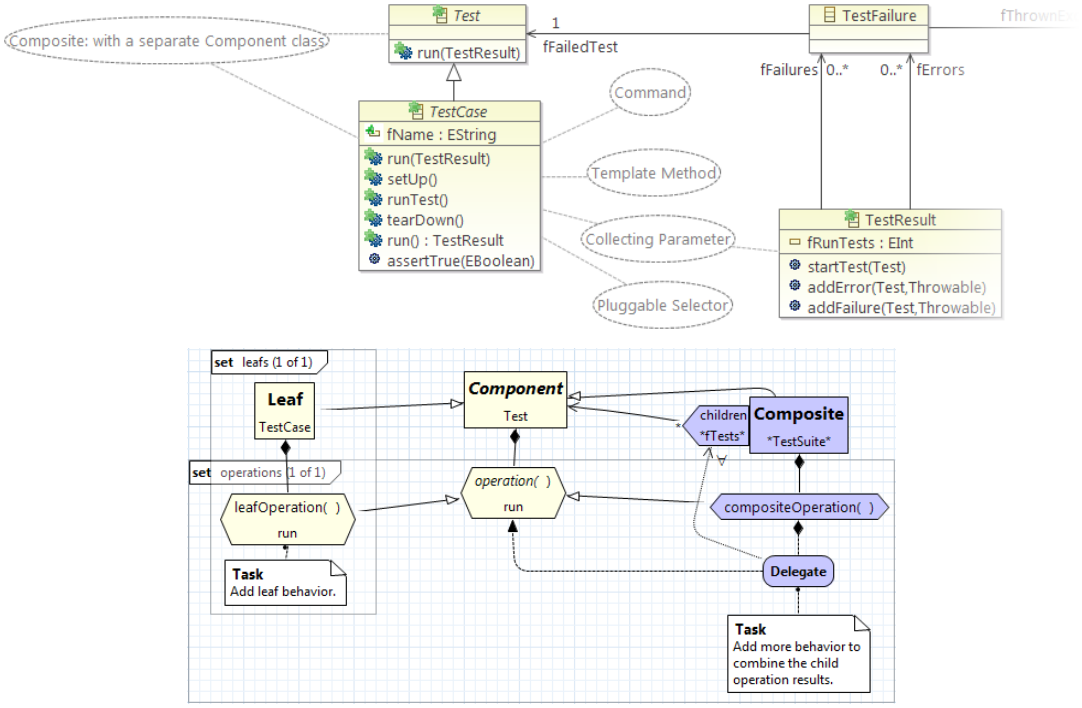


Abbildung D.16: Rollzuordnung vor Anwendung des Composite-Musters

stellte Ergebnis. Die Klasse TestSuite wurde im Klassendiagramm ergänzt. Sie hat eine unidirektionale Assoziation fTests mit Kardinalität 0..* zur abstrakten Klasse Test[6] erhalten, um darüber die komponierten Tests zu sammeln. Außerdem hat sie, wie in der Musterspezifikation angegeben, die in der Oberklasse Test definierte Methode run(TestResult) samt ihrer kompletten Methodensignatur übernommen, obwohl in der Musteranwendungssicht keine Details zu dieser Methode an der Rolle compositeOperation angegeben waren (vgl. Abb. D.16 unten). Hier hat der Algorithmus zur automatischen Musteranwendung die Information aus der Musterspezifikation genutzt, dass die Operation compositeOperation eine Spezialisierung[7] der Operation operation ist und damit die gleiche Signatur benötigt.

Neben der Erweiterung des Klassenmodells wird auch ein Verhaltensmodell für die Methode run(TestResult) der Klasse TestSuite generiert. Das zugehörige Story-Diagramm wird ebenfalls in der Abb. D.17 dargestellt (Mitte des Bildes). Hier ist die in der Musterspezifikation durch eine delegate-Aktion definierte Delegation des Verhaltens an die Kindobjekte modelliert. Auf allen referenzierten Test-Objekten soll die Methode run(TestResult) aufgerufen werden und die erhaltenen Argumente (hier ein TestResult-Objekt) unverändert weitergereicht werden.

Das Story-Diagramm enthält einen Aktivitätenknoten mit einem Story Pattern. Dieses durchläuft ausgehend von einem TestSuite-Objekt (this) alle über die

[6] In Ecore-Klassendiagrammen kann nicht direkt angegeben werden, dass eine Klasse als Java-Schnittstelle (`interface`) dienen soll. Diese Information lässt sich aber in einer ergänzenden .genmodel-Datei modellieren, sodass bei der Generierung von Java-Code ein `interface` verwendet wird.

[7] Unter Spezialisierung verstehe ich hier das Überschreiben einer konkreten oder Implementieren einer abstrakten Methode.

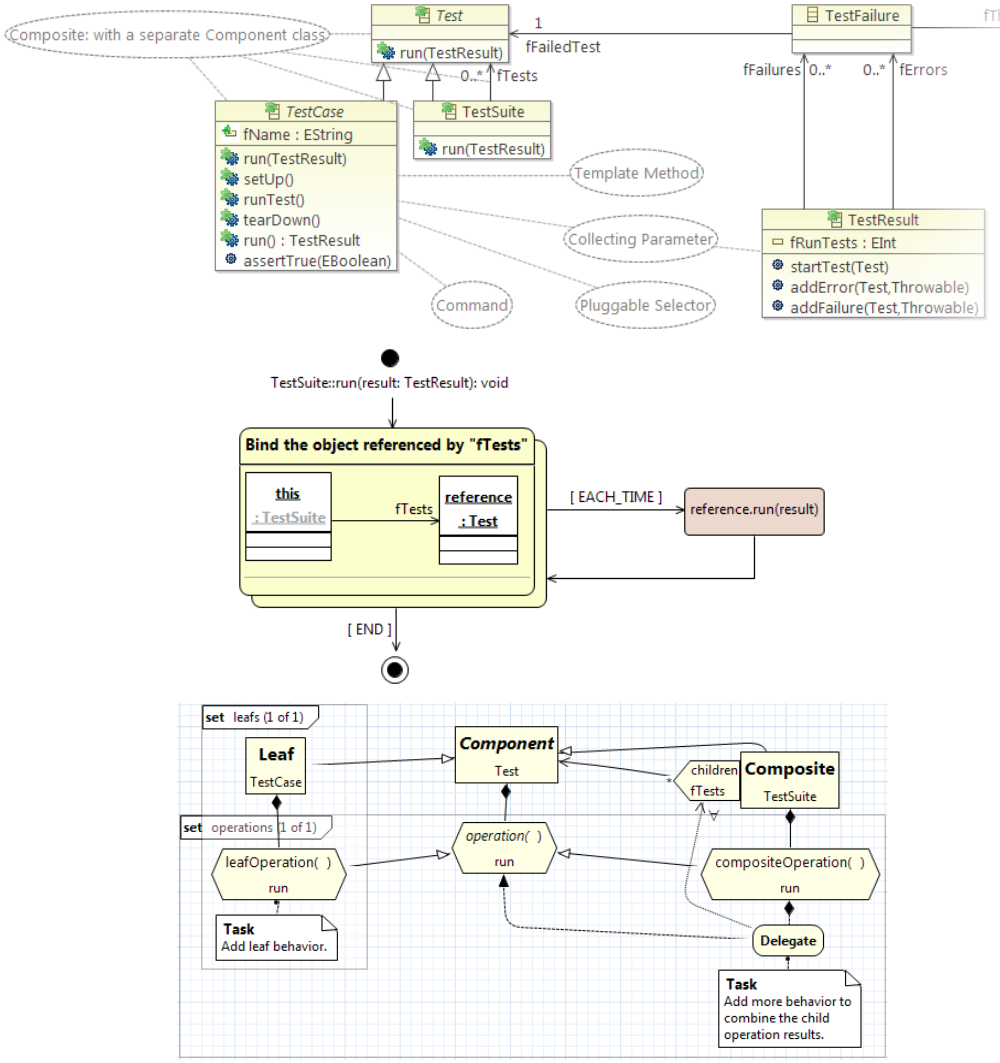


Abbildung D.17: Ergebnis nach Anwendung des Composite-Musters

Assoziation fTests erreichbaren (referenzierten) Test-Objekte und führt auf jedem dieser Test-Objekte den über die Aktivitätenkante mit dem Guard „EACH_TIME“ erreichbaren Aktivitätenknoten aus. Dieser wiederum beschreibt einen Aufruf der run-Methode auf einem referenzierten Test-Objekt (reference). Dabei wird das erhaltene Argument result (Parameter des Story-Diagramms) auch als Argument im modellierten Methodenaufruf verwendet. Nachdem alle referenzierten Test-Objekte durchlaufen wurden, wird der Kontrollfluss über die Aktivitätenkante mit dem Guard „END“ verlassen.

Alle Rollen der Musterspezifikation sind den Elementen im Entwurf zugeordnet. Die erste der beiden Entwurfsaufgaben, das Verhalten der Blattklassen (Rolle Leaf) zu ergänzen, ist bereits erledigt, denn das geforderte Verhalten der Methode run(TestResult) der Klasse TestCase wurde bereits zuvor modelliert (siehe Abb. D.11, S. 350). Aber auch die zweite spezifizierte Entwurfsaufgabe, die bei der Delegation an die Kindobjekte ggf. entstandenen Ergebnisse der Methodenaufrufe zu einem eigenen Ergebnis zu kombinieren, ist bereits erledigt. In diesem Fall liefert die auf den Kindobjekten aufgerufene Methode run(TestResult) keine

Rückgabewerte, was auch für die gleichnamige Methode der Klasse TestSuite gilt. Darum müssen in der TestSuite-Klasse Ergebnisse der Kindtests nicht explizit zu einem Gesamtergebnis kombiniert werden. Sie werden dank des Musters Collecting Parameter bereits im weitergereichten TestResult-Objekt kombiniert.

Damit scheint das Composite-Muster vollständig angewendet zu sein. Allerdings fällt auf, dass TestCase im Klassendiagramm als abstrakte Klasse modelliert ist, während sie in der Musterspezifikation mit der zugehörigen Rolle Leaf als konkrete Klasse spezifiziert ist. Eine automatische Validierung der Anwendungsstelle könnte so eine Abweichung von der Spezifikation aufdecken, ist jedoch im Prototypen nicht implementiert.

Streng genommen müsste die Rolle Leaf der Musterspezifikation in diesem Anwendungsbeispiel nicht der abstrakten Klasse TestCase zugeordnet werden, sondern den davon erbenden, konkreten Testfallimplementierungen. Ein Beispiel für eine solche Implementierung liefern die Entwickler des JUnit-Frameworks selbst mit: die Klasse MoneyTest, welche als konkrete Unterklasse von TestCase implementiert ist. Die Zuordnung der Rolle Leaf zu der Klasse TestCase (siehe Abb. D.17 unten) müsste also eigentlich wieder zurückgenommen werden.

D.8 Vervollständigen der Musteranwendungen durch Hinzunahme einer konkreten Testfallimplementierung

Die Klasse MoneyTest gehört nach meinem Verständnis eigentlich nicht zum JUnit-Framework, sondern stellt nur ein Beispiel für eine Verwendung des Frameworks dar. Nimmt man diese Klasse jedoch im Entwurfsmodell auf, so lassen sich die Anwendung des Composite-Musters sowie die anderen, unvollständigen Musteranwendungen vervollständigen.

D.8.1 Composite

Ergänzt man die eben erwähnte MoneyTest-Klasse im Klassendiagramm und korrigiert die Zuordnung der der Composite-Musterspezifikation, erhält man das in der Abb. D.18 dargestellte Ergebnis.

Die Klasse MoneyTest enthält hier eine Testfallimplementierung in Form der Methode testMoneyEquals aus den Beispielen in der Enwicklerdokumentation [Gam01]. Tatsächlich stellt diese Klasse noch 21 weitere Testmethoden bereit, auf deren Modellierung ich hier der Einfachheit und Übersicht halber verzichte.

Die Zuordnung der Leaf-Rolle in der Musteranwendungssicht in Abb. D.18 (unten) wurde von der Klasse TestCase zu der Klasse MoneyTest geändert. Man beachte, dass es beliebig viele Testklassen zu der Rolle Leaf geben kann, was die Musterspezifikation durch die Verwendung des Set Fragments leafs vorsieht (siehe Abb. D.18 unten). Die Rolle leafOperation bleibt der Methode run(TestResult) der Klasse TestCase zugeordnet. Obwohl die Methode zur Rolle leafOperation nun nicht mehr in der Klasse zur Rolle Leaf liegt, lässt die Semantik der Musterspezifkationen diese Zuordnung zu. Die Klasse MoneyTest (Rolle Leaf) stellt die geforderte leafOperation (Methode run(TestResult)) durch Erben von einer Oberklasse bereit. Damit ist die Anwendung des Composite-Musters korrigiert und komplett.

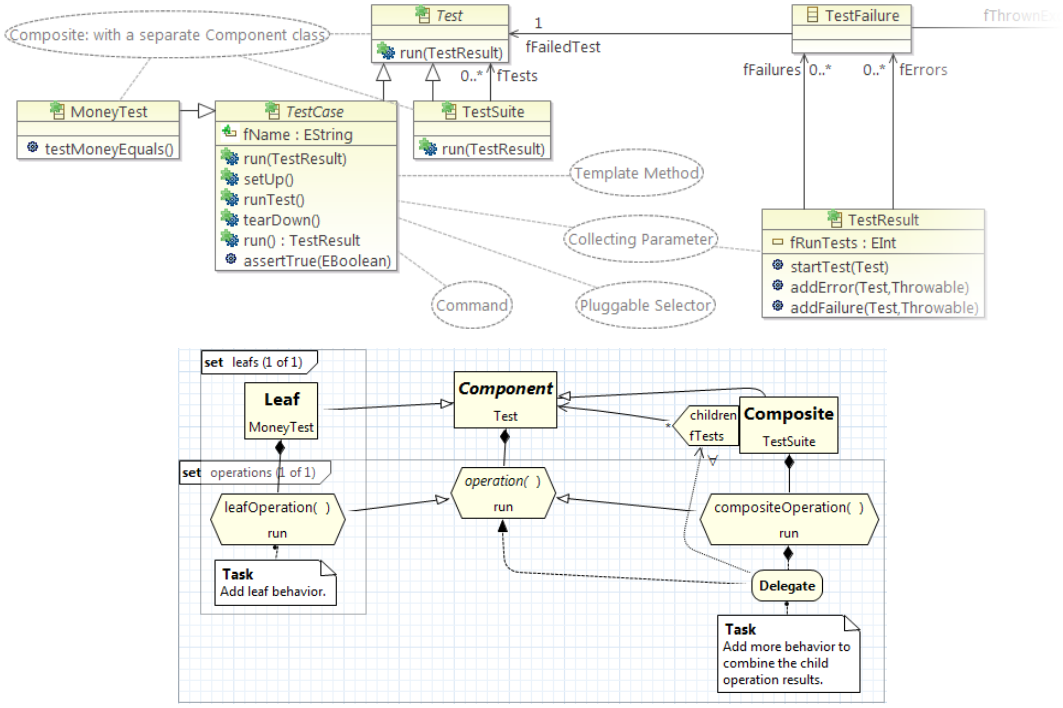


Abbildung D.18: Korrigierte Rollzuordnung in der Anwendung des Composite-Musters

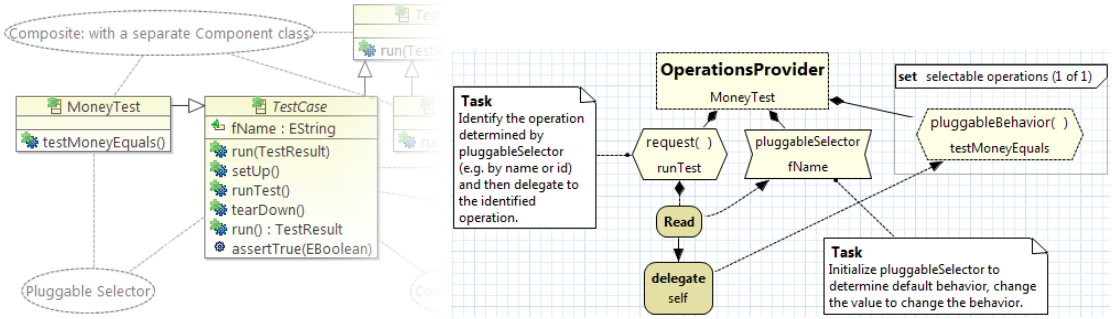


Abbildung D.19: Korrigierte Rollzuordnung in der Anwendung des Musters Pluggable Selector

D.8.2 Pluggable Selector

Die Anwendung des Musters Pluggable Selector ist bisher ebenfalls unvollständig (siehe Abb. D.14, S. 354). Es fehlt zur der Rolle pluggableBehavior der Musterspezifikation mindestens eine entsprechende Methode im Entwurf. Bei dieser Musteranwendung soll diese Rolle von den Testmethoden konkreter Testimplementierungen eingenommen werden. Bisher fehlten diese, sind aber mit der Klasse MoneyTest und seiner Testmethode testMoneyEquals ergänzt worden. Somit kann die bisher unbelegte Rolle pluggableBehavior wie in der Abb. D.19 dargestellt mit der Methode testMoneyEquals belegt werden. Gleichzeitig wird die Zuordnung der Rolle OperationsProvider, welche bisher der Klasse TestCase zugeordnet war (vgl. Abb. D.14, S. 354), korrigiert. Dazu wird diese Rolle der Klasse MoneyTest zugeordnet, denn das ist die Klasse, welche die Operationen (Testmethoden) bereitstellt, wovon eine mit Hilfe des Musters zur Laufzeit ausgewählt und ausgeführt werden soll.

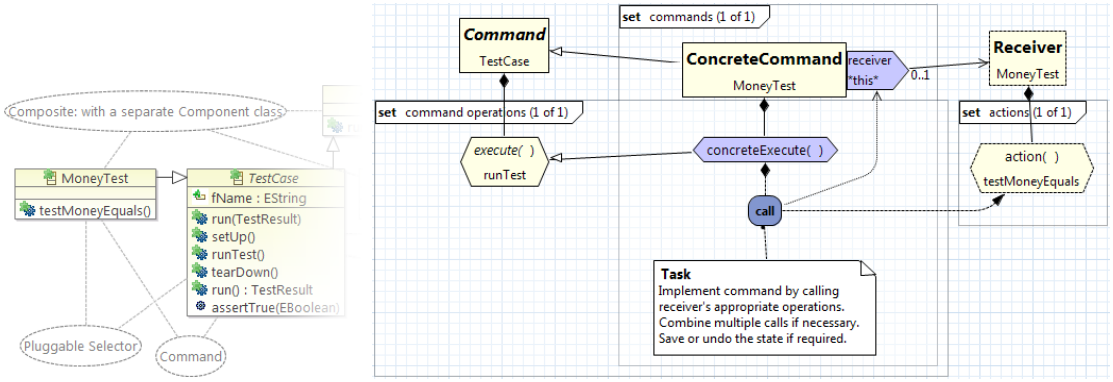


Abbildung D.20: Rollenzuordnung vor Anwendung des Command-Musters

Analog zur Anwendung des Composite-Musters stellt man auch hier fest, dass die Rolle OperationsProvider einer Klasse MoneyTest zugeordnet wird, die Operation request und das Attribut pluggableSelector jedoch nicht in der Klasse MoneyTest modelliert sind wie die Spezifikation möglicherweise vermuten lassen würde. Hier werden die Operation und das Attribut von der Oberklasse TestCase der Klasse MoneyTest geerbt. Damit sind diese in der Klasse MoneyTest verfügbar und die Rollenzuordnung ist damit konsistent mit der Musterspezifikation (erfüllt alle geforderten Bedingungen).

Zum Abschließen der Musteranwendung muss noch die offen gebliebene Entwurfsaufgabe (Initialisieren des pluggableSelector-Attributs) abgeschlossen werden. Dazu wird ein Konstruktor der Klasse TestCase analog zum Codefragment D.1 (S. 342) ergänzt. Die Initialisierung des Attributswertes erfolgt in diesem Fall bei Aufruf des Konstruktors oder per Reflection. Beides modelliere ich hier nicht explizit und betrachte das Muster Pluggable Selector nun als vollständig angewandt.

D.8.3 Command

Bei der Anwendung des Command-Musters wurden bisher nur die Rollen Command und execute belegt (siehe Abb. D.1, S. 342). Die automatische Musteranwendung wurde noch nicht durchgeführt. Es fehlt somit ein signifikanter Teil der Musteranwendung. Nun kann dieser Teil mit Hilfe der Klasse MoneyTest vervollständigt werden. Diese nimmt nun wie in der Abb. D.20 dargestellt die Rollen ConcreteCommand und Receiver ein. Die Testmethode testMoneyEquals nimmt die Rolle action ein.

implizit immer vorhandene Selbstreferenz ergänzt

Bei der vorliegenden Anwendung des Command-Musters soll ein MoneyTest-Objekt in der Implementierung der Rolle concreteExecute seine eigenen Testmethoden (Rolle action) aufrufen. Das Objekt, auf welchem die call-Aktion ausgeführt werden soll (das Zielobjekt), würde in diesem Fall in Java mit dem Schlüsselwort `this` beschrieben werden. In Musterspezifikationen würde man das entsprechend mit dem Schlüsswort self analog zu Fall (e) in Abb. A.21 (S. 271) ausdrücken. In der vorliegenden Spezifikation wird jedoch davon abweichend auf eine Referenz receiver verwiesen. Anstatt eine weitere Variante der Spezifikation des Musters Command anzulegen und darin eine Selbstreferenz wie in Abb. A.21 zu spezifizieren (das wäre der korrektere Weg), habe ich mich der Einfachheit halber entschieden,

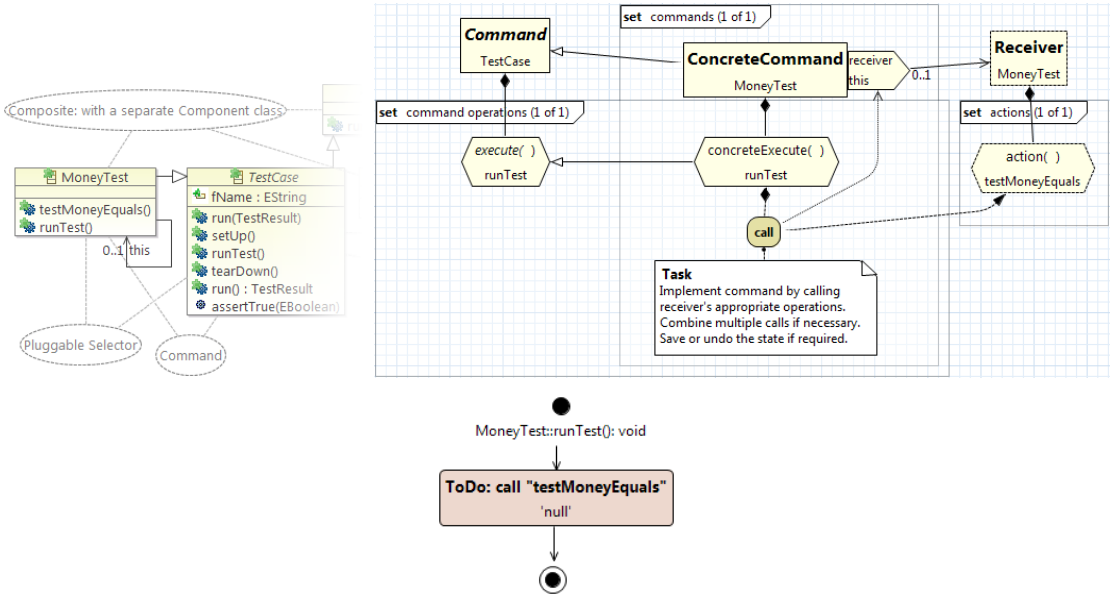


Abbildung D.21: Ergebnis nach Anwendung des Command-Musters

in diesem Fall die vorliegende Musterspezifikation zu nutzen und als Work-around die Referenz receiver mit „this“ zu bennenen (siehe Abb. D.20).

Dadurch erhält man nach automatischer Anwendung des Musters ein Ergebnis wie in der Abb. D.21 dargestellt. Neben der neuen Methode runTest in der Klasse MoneyTest wurde der Klasse auch eine eigentlich unnötige Selbstassoziation namens this hinzugefügt (oben links im Bild). Außerdem wurde für die call-Aktion ein Story-Diagramm mit dem Verhalten der Methode runTest generiert (unten im Bild). Alle Rollen der Musterspezifikation sind nun Entwurfsteilen zugeordnet (rechts im Bild).

Nun muss nur noch die verbliebene Entwurfsaufgabe erledigt werden. Dazu muss das Verhalten im generierten Story-Diagramm zu Ende modelliert und der Aufruf der Testmethode testMoneyEquals ergänzt werden. Nach manueller Anpassung des Modells erhält man ein Story-Diagramm wie in der Abb. D.22. Hier wird die generierte Assoziation this zur Modellierung des Methodenaufrufs mit dem Ausdruck this.testMoneyEquals() verwendet. Anders lässt sich der Ausdruck in Story-Diagrammen derzeit nicht modellieren, weswegen ich mich wie zuvor beschrieben dazu entschieden habe, bei der vorliegenden Spezifikation des Command-Musters zu bleiben.

Die Rolle action in der Spezifikation des Command-Musters (siehe Abb. D.21

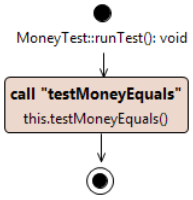


Abbildung D.22: Angepasstes Verhalten der Methode *MoneyTest::runTest()*

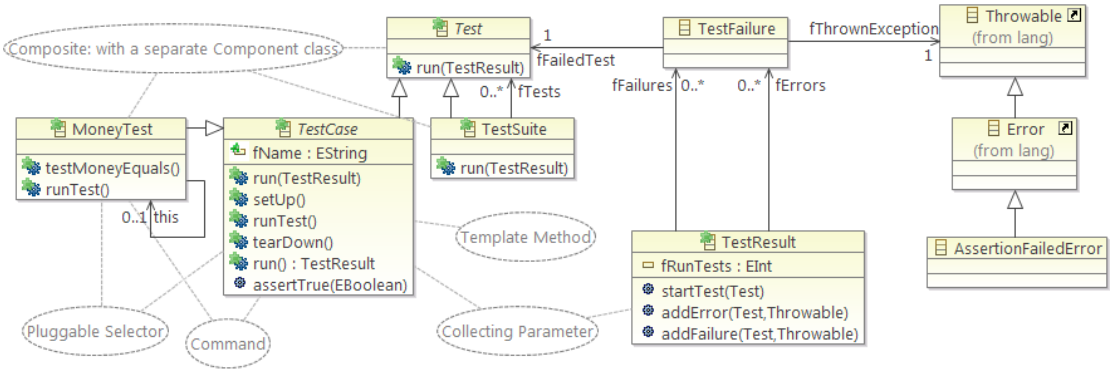


Abbildung D.23: Entwurf nach Anwendung aller Muster

oben rechts) wird im Fall der Klasse MoneyTest eigentlich von 22 implementierten Testmethoden eingenommen, modelliert ist hier der Einfachheit halber nur eine davon, nämlich testMoneyEquals. Die anderen, nicht modellierten Testmethoden könnten aber mit Hilfe des Set Fragments actions ebenfalls der Rolle action zugeordnet werden.

Das hier modellierte Überschreiben der Methode runTest der Klasse TestCase ist optional und entspricht im Wesentlichen der Anwendung des Adapter-Musters analog zum Codefragment D.11 (S. 351). Stattdessen wird das JUnit-Framework meist so eingesetzt, dass alle verfügbaren Testmethoden der Klasse MoneyTest vom Framework per Reflection nacheinander aufgerufen werden. Dazu wird u.a. der Java-Code aus Codefragment D.12 (S. 352) bzw. das modellierte Verhalten aus dem Story-Diagramm in Abb. D.15 (S. 354) verwendet.

D.9 Fazit

Bis auf das Adapter-Muster sind nun alle Muster vollständig angewendet und die zugehörigen Anwendungsstellen explizit modelliert. Der fertige Entwurf ist im Klassendiagramm in Abb. D.23 dargestellt. Die Rollenzuordnungen sind mit Ausnahme der Parameterübergabe in der Anwendung des Musters Collecting Parameter[8] (siehe Abb. D.6, S. 346) vollständig und alle Entwurfsaufgaben sind erledigt.

Das Anwendungsbeispiel JUnit mit all seinen Musteranwendungen zeigt damit, dass das im Rahmen dieser Arbeit vorgestellte Verfahren zur modellgetriebenen Anwendung von Entwurfsmustern bis auf wenige Einschränkungen auch in realistischen Anwendungsszenarien eingesetzt werden kann. Ein Großteil der Einschränkungen bei den Musteranwendungen im Fall von JUnit kommen durch Einschränkungen in der Ausdrucksmächtigkeit der eingesetzten Ecore- und Story-Diagramm-Modelle.

[8] Die Benutzerschnittstelle des vorliegenden Prototypen stellt keine Möglichkeit bereit, das Zuordnen der hier spezifizierten Parameterübergabe zu Elementen eines Story-Diagramms durchzuführen.

Anhang E

Implementierungsdetails

Die in Kapitel 7 beschriebene prototypische Implementierung des vorgestellten Ansatzes basiert auf Eclipse Juno Service Release 2 (eclipse modeling distribution) und Java Standard Edition 6 (JDK 6 Update 45). Das ist insb. die Kombination von Eclipse- und Java-Versionen, zu denen die Story-Diagramm-Eclipse-Plug-ins kompatibel sind (ich habe sie wegen diverser Plug-in-Inkompatibilitäten nicht auf aktuellere Eclipse- und Java-Versionen migriert).

Der gesamte Quellcode ist frei verfügbar und kann über die GitLab-Plattform[1] inkl. der von SVN[2] zu Git[3] migrierten Historie der Code-Repositories heruntergeladen werden. Der Quellcode ist in zwei GitLab-Projekten abgelegt (siehe Tab. E.1). Der POSE-Quellcode liegt im Git-Branch *master*, während der in dieser Arbeit verwendete Story-Diagramm-Plug-ins-Quellcode im Git-Branch *POSE-travkin-branch-for-PhD* liegt (der nicht für diese Arbeit modifizierte Quellcode liegt ebenfalls im Git-Branch *master*).

Tabelle E.1: Repositories mit dem Quellcode des Prototypen

POSE	GitLab-Projekt: https://gitlab.com/travkin/pose-tools Git-Repository: https://gitlab.com/travkin/pose-tools.git
Story Diagrams	GitLab-Projekt: https://gitlab.com/story-diagrams/storydiagrams Git-Repository: https://gitlab.com/story-diagrams/storydiagrams.git

Einige Informationen zum Umfang der entwickelten oder angepassten Software können der Tabelle E.2 entnommen werden. Die Entwicklung der POSE Tool Suite erstreckt sich laut Git-Historie über einen Zeitraum vom 17.05.2011 bis heute (18.12.2017) und 684 Commits[4], wovon 583 von mir, 70 von Aljoscha Hark, einer studentischen Hilfskraft, und 31 von Andre Backofen sind, einem Studenten, der seine Master-Arbeit in diesem Forschungsgebiet geschrieben hat [Bac11]. Aus der Historie für meine Anpassungen an den Story-Diagramm-Werkzeugen geht hervor, dass die Entwicklung aller Story-Diagramm-Werkzeuge sich über eine Zeitspanne vom 03.09.2010 bis heute (18.12.2017) erstreckt, wobei (in meinem Git-Branch) 212 der 854 Commits auf mich fallen[5].

Zur Nachvollziehbarkeit und einfacheren Installation liegt der Abgabe-Version dieser Dissertation eine DVD-ROM mit einer vorbereiteten Eclipse-Installation, den Git-Repositories, einem JDK 6 und einer Kurzanleitung bei.

[1] GitLab: https://gitlab.com
[2] Subversion (SVN), ein Versionskontrollsystem, https://subversion.apache.org/
[3] Git, ein derzeit weit verbreitetes Versionskontrollsystem, https://git-scm.com/
[4] Details unter https://gitlab.com/travkin/pose-tools/graphs/master
[5] https://gitlab.com/story-diagrams/storydiagrams/graphs/POSE-travkin-branch-for-PhD

Tabelle E.2: Implementierungsumfang in Lines of Code (LOC) und Eclipse-Plug-ins / Eclipse-Projekten

Beschreibung	Plug-ins	Test-Projekte	Summe Projekte	Plug-ins LOC manuell	Plug-ins LOC generiert	Tests LOC manuell	Tests LOC generiert	Plug-ins LOC	Tests LOC	Summe LOC
Musterspezifikations-Meta-Modell	2	0	2	5.206	34.735	0	0	39.941	0	39.941
Musteranwendungsstellen-Meta-Modell	1	0	1	916	8.564	0	0	9.480	0	9.480
Musterspezifikations-Editor	4	0	4	19.095	9.139	0	0	28.234	0	28.234
Musteranwendungsoperationen	1	0	1	5.375	0	0	0	5.375	0	5.375
Rollenzuordnungs-Editor, Musteranwendung, Validierun	2	2	4	6.088	2.473	2.299	1.434	8.561	3.733	12.294
Erweiterter Klassendiagramm-Editor	1	0	2	3.543	0	0	0	3.543	0	3.543
Erweiterter Stroy-Diagramm-Editor	1	0	1	88	0	0	0	88	0	88
Andere (z.B. Feature-Definitionen & Update Sites)	0	0	2	0	0	0	0	0	0	0
Summe POSE:	**12**	**2**	**17**	**40.311**	**54.911**	**2.299**	**1.434**	**95.222**	**3.733**	**98.955**
Story-Diagramm-Meta-Modell	4	0	4	4.269	94.352	0	0	98.621	0	98.621
Story-Diagramm-Editor	10	1	15	15.022	74.847	290	0	89.869	290	90.159
Story-Diagramm-Interpreter	8	2	13	21.651	0	1.058	2.447	21.651	3.505	25.156
Story-Diagramm-Traces	3	0	4	976	29.705	0	0	30.681	0	30.681
Story-Diagramm-Debugger	3	0	4	6.691	0	0	0	6.691	0	6.691
Basis für graphische Editoren	3	6	10	2.324	0	412	24.515	2.324	24.927	27.251
Andere (z.B. Feature-Definitionen & Update Sites)	0	0	8	0	0	0	0	0	0	0
Summe Story-Diagramme:	**31**	**9**	**58**	**50.933**	**198.904**	**1.760**	**26.962**	**249.837**	**28.722**	**278.559**
Summe gesamt:	**43**	**11**	**75**	**91.244**	**253.815**	**4.059**	**28.396**			
				345.059		**32.455**				
				377.514						

Glossar

In meiner Arbeit verwendete ich zahlreiche Begriffe. Einige davon unterscheiden sich nur geringfügig in ihrer Bedeutung voneinander. Einige habe ich selbst eingeführt. In beiden Fällen bedürfen die Begriffe einer Erklärung. Die wichtigsten der verwendeten Begriffe führe ich im Folgenden auf und erläutere sie. Begriffe rund um Softwareentwurfsmuster werden in Abschnitt 2.1.3 erklärt und mit Hilfe einer Ontologie (siehe Abb. 2.1, S. 26) in Beziehung gesetzt.

Aktion ein in der Musterspezifikationssprache eingeführtes Sprachkonstrukt zur Beschreibung von Verhalten (siehe Tab. A.3, S. 259)

Anwendungsfall eine Situation / Konstellation, in welcher ein Entwurfsmuster angewandt werden könnte

Anwendungsmodell eine kompakte, abstrakte Repräsentation einer zu erzeugenden Musterimplementierung, beschrieben in der DAL (siehe Abschnitt 5.4.1 und Abb. 4.3, S. 83)

Anwendungsstelle siehe Musteranwendungsstelle

Dekorationsmodell ein Modell, welches ein anderes Modell markiert und mit zusätzlichen Informationen anreichert. Im Rahmen dieser Arbeit wird ein Entwurfsmodell mit Informationen zu angewandten oder anzuwendenden Entwurfsmustern angereichert (dekoriert). Das Dekorationsmodell umfasst hier das Korrespondenz- und das Anwendungsmodell (siehe Abb. 4.3, S. 83).

Design siehe Entwurf

Design Abstraction Language (DAL) der Teil der Musterspezifikationssprache zur abstrakten Repräsentation bestimmter Konzepte im Softwareentwurf (siehe Abb. 3.4, S. 45) und Abb. 4.3, S. 83

Entwurf siehe Softwareentwurf

Entwurfselement Teil des Softwareentwurfs wie z.B. eine Klasse, Methode, Attribut, Assoziation, Parameter,...

Entwurfslösung das in einer Musterbeschreibung erläuterte Konzept zum Aufbau eines Softwareentwurfs (siehe Abb. 2.1, S. 26)

Entwurfsmodell das Modell oder die Modelle des Softwareentwurfs (hier Klassen- und Story-Diagramm-Modelle)

Entwurfsmuster siehe Softwareentwurfsmuster

Entwurfsskizze Diagramm in einer Musterbeschreibung, meist exemplarische, skizzenhafte Darstellung eines partiellen Softwareentwurfs

Entwurfsstruktur die Struktur des Softwareentwurfs, z.B. Klassen,

ihre Eigenschaften und Beziehungen oder eine Softwarekomponentenstruktur

Entwurfsverhalten das (modellierte / implementierte) Verhalten eines Softwareentwurfs, z.B. in Story-Diagrammen, State Machines, Quellcode

Implementierungsvariante siehe Musterimplementierungsvariante

Intention Zweck einer Musteranwendung, z.B. bestimmte Teile des Softwareentwurfs zu entkoppeln

Interaktion siehe Aktion

Konsequenz in dieser Arbeit: Folge einer Musteranwendung, insb. bestimmte Qualitätsmerkmale wie Austauschbarkeit, Erweiterbarkeit, Effizienz,...

Korrespondenzmodell ein spezielles Traceability-Modell, eine Repräsentation der Zuordnungen von Musterrollen und Set Fragments zu den sie implementierenden Entwurfsteilen, dient u.a. als Repräsentation einer Musteranwendungsstelle (siehe Abb. 4.3, S. 83).

Lösung siehe Entwurfslösung

Lösungsidee siehe Lösungsskizze

Lösungsskizze Skizze einer Entwurfslösung

Lösungsvariante eine von mehreren Entwurfslösungen

Lösungsvorschlag siehe Entwurfslösung

Muster siehe Entwurfsmuster

Musteranwendung die Tätigkeit, eine Entwurfslösung in einem Softwareentwurf zu implementieren und eine bestimmte Implementierungsvariante zu erzeugen

Musteranwendungsstelle die Stelle einer erfolgten oder geplanten Musteranwendung, charakterisiert durch die Zuordnung aller oder einiger der Musterrollen der zugehörigen Entwurfslösung zu bereits vorhandenen Elementen im Softwareentwurf (siehe Kapitel 4)

Musterausprägung siehe Musterinstanz

Musterbeschreibung informelle Beschreibung eines Entwurfsmusters, meist natürlichsprachlich, oft ergänzt durch Diagramme und Beispielcode

Musterimplementierung eine Implementierung einer allgemein beschriebenen Entwurfslösung, eine Anwendungsstelle eines Entwurfsmusters

Musterimplementierungsvariante eine von mehreren Implementierungsmöglichkeiten zu einem Entwurfsmuster

Musterinstanz eine Musterimplementierung

Musterkatalog eine Sammlung mehrerer Entwurfsmuster bzw. ihrer Spezifikationen

Musterrealisierung siehe Musterimplementierung

Musterrealisierungsvariante siehe Musterimplementierungsvariante

Musterrolle ein benannter Teil einer Entwurfslösung, wird von einem

Teil einer Musterimplementierung eingenommen

Musterspezifikation die formale Repräsentation einer Entwurfslösung, siehe Kapitel 3

Musterspezifikationssprache eine Sprache zur Beschreibung der Entwurfslösungen zu Entwurfsmustern, z.B. die Pattern Specification Language

Mustervariante ein Entwurfsmuster, welches eine Alternative zu einem anderen Entwurfsmuster darstellt, z.B. Class Adapter vs. Object Adapter [GHJV95, S. 139 ff.]

Musterverwendung jede Form des Einsatzes von Entwurfsmustern, insb. die Implementierung von Entwurfsmustern sowie die Dokumentation oder die Validierung von Musterimplementierungen

Mustervorkommen siehe Musterimplementierung

Pattern Application Language (PAL) die im Rahmen dieser Arbeit entwickelte Sprache zur Erfassung von Anwendungsstellen (Korrespondenzen zwischen Musterspezifikationen und Musterimplementierungen), siehe Abschnitt 4.2 und Abb. 4.3, S. 83

Pattern Specification Language (PSL) die im Rahmen dieser Arbeit entwickelte Musterspezifikationssprache, siehe Kapitel 3 und Abb. 4.3, S. 83

Realisierungsvariante siehe Musterrealisierungsvariante

Rolle siehe Musterrolle

Rollenzuordnung die Zuordnung von Teilen eines Softwareentwurfs zu den Rollen eines Entwurfsmusters bzw. einer Musterspezifikation (siehe Abb. 4.3, S. 83)

Set Fragment Sprachkonstrukt in der Musterspezifikationssprache zur Kennzeichnung von sich als Ganzes wiederholenden Entwurfsteilen in einer Musterimplementierung, z.B. Produkte und zugehörige create-Operationen beim Muster Abstract Factory (vgl. Abb. 3.3, S. 43 und Abb. B.1, S. 291)

Set-Fragment-Instanz ein spezieller Korrespondenzknoten der PAL. Repräsentiert ein Vorkommen / eine Implementierung der in einem Set Fragment spezifizierten Entwurfsteile und verknüpft sie mit dem zugehörigen Set Fragment (siehe Abb. 4.3, S. 83).

Softwareentwurf der vorliegende, ggf. partielle Entwurf einer Software, umfasst Struktur und Verhalten, kann als Diagrammskizze, Modell oder als Quellcode vorliegen

Softwareentwurfsmuster wiederverwendbare Lösung für ein Entwurfsproblem, siehe Abschnitt 2.1

Eigene Veröffentlichungen

[ACE+08] Alhawash, Kahtan, Toni Ceylan, Tobias Eckardt, Masud Fazal-Baqaie, Joel Greenyer, Christian Heinzemann, Stefan Henkler, Renate Ristov, Dietrich Travkin und Coni Yalcin: *The Fujaba Automotive Tool Suite*. In: Aßmann, Uwe, Jendrik Johannes und Albert Zündorf (Herausgeber): *Proceedings of the 6th International Fujaba Days 2008*, Nummer TUD-FI08-09 in *Technical Report*, Seiten 36–39. Technische Universität Dresden, September 2008.

[FTvD11] Fockel, Markus, Dietrich Travkin und Markus von Detten: *Interpreting Story Diagrams for the Static Detection of Software Patterns*. In: Norbisrath, Ulrich und Ruben Jubeh (Herausgeber): *Proceedings of the 8th International Fujaba Days*, Nummer 2012, 1 in *Kasseler Informatikschriften (KIS)*, Seiten 6–10. Universität Kassel, Mai 2011.

[HRvD+11] Heinzemann, Christian, Jan Rieke, Markus von Detten, Dietrich Travkin und Marius Lauder: *A new Meta-Model for Story Diagrams*. In: Norbisrath, Ulrich und Ruben Jubeh (Herausgeber): *Proceedings of the 8th International Fujaba Days*, Nummer 2012, 1 in *Kasseler Informatikschriften (KIS)*, Seiten 1–5. Universität Kassel, Mai 2011.

[PvDT11] Platenius, Marie Christin, Markus von Detten und Dietrich Travkin: *Visualization of Pattern Detection Results in Reclipse*. In: Norbisrath, Ulrich und Ruben Jubeh (Herausgeber): *Proceedings of the 8th International Fujaba Days*, Nummer 2012, 1 in *Kasseler Informatikschriften (KIS)*, Seiten 33–37. Universität Kassel, Mai 2011.

[ST07] Stürmer, Ingo und Dietrich Travkin: *Automated Transformation of MATLAB Simulink and Stateflow Models*. In: Gehrke, Matthias, Holger Giese und Joachim Stroop (Herausgeber): *Preliminary Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, Band tr-ri-07-286, Seiten 57–62. Universität Paderborn, 2007.

[ST08] Stürmer, Ingo und Dietrich Travkin: *Tool Supported Quality Assessment and Improvement in MATLAB Simulink and Stateflow Models*. In: Gehrke, Matthias, Holger Giese und Joachim Stroop (Herausgeber): *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, Band 236 der Reihe *HNI Verlagsschriftenreihe*, Seiten 61–68. Universität Paderborn, 2008, ISBN 978-3-939350-55-2.

[TM05] Travkin, Dietrich und Matthias Meyer: *Generation of Type Safe Association Implementations*. In: Giese, Holger und Albert Zündorf (Herausgeber): *Proceedings of the 3rd International Fujaba Days 2005*, Band tr-ri-05-259, Seiten 63–66. Universität Paderborn, September 2005.

[Tra05] Travkin, Dietrich: *Generierung typsicherer Implementierungen für Assoziationen in UML-Modellen*. Studienarbeit, Universität Paderborn, Februar 2005. http://dx.doi.org/10.17619/UNIPB/1-333, URN: urn:nbn:de:hbz:466:2-30867, DOI: 10.17619/UNIPB/1-333.

[Tra06] Travkin, Dietrich: *Bewertung automatisch erkannter Instanzen von Software-Mustern*. Diplomarbeit, Universität Paderborn, August 2006. http://dx.doi.org/10.17619/UNIPB/1-334, URN: urn:nbn:de:hbz:466:2-30872, DOI: 10.17619/UNIPB/1-334.

[Tra07] Travkin, Dietrich: *Bewertung automatisch erkannter Ausprägungen von Software-Mustern*. In: Bleek, Wolf Gideon, Henning Schwentner und Heinz Züllighoven (Herausgeber): *Proceedings of the Software Engineering 2007 Conference - Workshop Contributions*, Band P-106 der Reihe *LNI*, Seiten 369–372. Gesellschaft für Informatik, März 2007.

[Tra10] Travkin, Dietrich: *Towards Better Support for Pattern-Oriented Software Development*. In: *Proceedings of the 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE-DS)*, Seiten 55–59, Oktober 2010.

[vDHH+12] Detten, Markus von, Christian Heinzemann, Stephan Hildebrandt, Marie Christin Platenius, Jan Rieke und Dietrich Travkin: *Story Diagrams – Syntax and Semantics*. Technischer Bericht tr-ri-12-324, Universität Paderborn, Juli 2012. Ver. 0.2.

[vDHP+12] Detten, Markus von, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Julian Suck und Dietrich Travkin: *Story Diagrams – Syntax and Semantics*. Technischer Bericht tr-ri-12-320, Universität Paderborn, April 2012. Ver. 0.1.

[vDMT10a] Detten, Markus von, Matthias Meyer und Dietrich Travkin: *Reclipse – A Reverse Engineering Tool Suite*. Technischer Bericht tr-ri-10-312, Universität Paderborn, März 2010.

[vDMT10b] Detten, Markus von, Matthias Meyer und Dietrich Travkin: *Reverse Engineering with the Reclipse Tool Suite*. In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, Seiten 299–300. ACM Press, Mai 2010.

[vDT10] Detten, Markus von und Dietrich Travkin: *An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw*. Technischer Bericht tr-ri-10-322, Universität Paderborn, Oktober 2010.

Literatur

[AACGJ01] Albin-Amiot, Hervé, Pierre Cointe, Yann-Gaël Guéhéneuc und Narendra Jussien: *Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together*. In: Richardson, Debra J., Martin Feather und Michael Goedicke (Herausgeber): *Proceedings of the* 16^{th} *IEEE Conference on Automated Software Engineering (ASE)*, Seiten 166–173. IEEE Computer Society Press, November 2001. 217, 229

[AAG01] Albin-Amiot, Hervé und Yann-Gaël Guéhéneuc: *Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis*. In: Broek, Pim van den, Pavel Hruby, Motoshi Saeki, Gerson Sunyé und Bedir Tekinerdogan (Herausgeber): *Proceedings of the* 1^{st} *ECOOP Workshop on Automating Object-Oriented Software Development Methods (AOOSDM)*, Seiten 20–27. University of Twente, Oktober 2001. TR-CTIT-01-35. 217, 229

[ACM13] Alur, Deepak, John Crupi und Dan Malks: *Core J2EE Patterns – Best Practices and Design Strategies*. Prentice Hall / Sun Microsystems Press, 2. Auflage, Dezember 2013, ISBN 978-0133807462. 24, 247

[AIS77] Alexander, Christopher, Sara Ishikawa und Murray Silverstein: *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977, ISBN 978-0195019193. 19

[AN04] Arlow, Jim und Ila Neustadt: *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Object Technology Series. Addison-Wesley Professional, 1. Auflage, Januar 2004, ISBN 978-0321112309. 24

[Anj14] Anjorin, Anthony: *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*. Dissertation, Technische Universität Darmstadt, Oktober 2014. 126, 154, 239, 246

[Bac11] Backofen, Andre: *Werkzeug-gestützte Anwendung von Entwurfsmustern unter Berücksichtigung von Verhaltensmodellen*. Masterarbeit, Universität Paderborn, Oktober 2011. 208, 363

[BBB+12] Becker, Steffen, Christian Brenner, Christopher Brink, Stefan Dziwok, Christian Heinzemann, Renate Löffler, Uwe Pohlmann, Wilhelm Schäfer, Julian Suck und Oliver Sudmann: *The MechatronicUML Design Method – Process, Syntax, and Semantics*. Technischer Bericht tr-ri-12-326, Universität Paderborn, August 2012. Vers. 0.3. 24, 76

[BBF+11] Backofen, Andre, Adnan Biser, Adrian Fahle, Stefan Loehers, Joachim Meyer, Maik Niggemann, Dennis Nobel, Marie Christin Platenius und Jan Schmalor: *Projektgruppe Pattern-Oriented Software Engineering (PG POSE)*. Abschlussbericht, Universität Paderborn, März 2011.

https://www.hni.uni-paderborn.de/swt/lehre/projektgruppenarchiv/pg-pose, Web-Seite: http://www-old.cs.uni-paderborn.de/no_cache/fachgebiete/fachgebiet-softwaretechnik/lehre/lehrveranstaltungen/projektgruppen/pg-pose.html, besucht: 18.12.2017. 42, 155, 207, 217, 218

[Bec96] Beck, Kent: *SmallTalk Best Practice Patterns*. Prentice Hall, Oktober 1996, ISBN 978-0134769042. 24, 180, 181, 200, 301, 344, 352

[Bec07] Beck, Kent: *Implementation Patterns*. Addison-Wesley, Upper Saddle River, NJ, 2007, ISBN 978-0-321-41309-3. 24

[BFYV96] Budinsky, Frank J., Marylin A. Finnie, Patsy S. Yu und John M. Vlissides: *Automatic Code Generation from Design Patterns*. IBM Systems Journal, 35(2):151–171, 1996, ISSN 0018-8670. 8, 228, 230

[BHS07a] Buschmann, Frank, Kevlin Henney und Douglas C. Schmidt: *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley Series in Software Design Patterns. Wiley, März 2007, ISBN 978-0470059029. 24, 62

[BHS07b] Buschmann, Frank, Kevlin Henney und Douglas C. Schmidt: *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. Wiley Series in Software Design Patterns. John Wiley and Sons, April 2007, ISBN 978-0471486480. 20, 21, 22, 23, 24, 42, 208

[Ble07] Blewitt, Alex: *Design Patterns Formalization Techniques*, Kapitel VI: SPINE: Language for Pattern Verification, Seiten 109–122. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209

[BLS06] Briand, Lionel C., Yvan Labiche und Alexandre Sauvé: *Guiding the Application of Design Patterns Based on UML Models*. In: *22nd IEEE International Conference on Software Maintenance (ICSM)*, Seiten 234–243, September 2006. 234

[BMMM98] Brown, W. J., R. C. Malveau, H. W. McCormick und T. J. Mombray: *Anti Patterns – Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., März 1998. 238

[BMR+96] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley Series in Software Design Patterns. Wiley, Juli 1996, ISBN 978-0-471-95869-7. 24, 65, 73, 156, 178, 179, 198, 300

[BMS03] Baniassad, Elisa L. A., Gail C. Murphy und Christa Schwanninger: *Design Pattern Rationale Graphs: Linking Design to Source*. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Seiten 352–362, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1877-X. 221

[Bor] *Together*. Borland. http://www.borland.com/de-DE/Products/Requirements-Management/Together , besucht: 18.12.2017. 8

[Bor11] *Borland Together 2008 – Borland Together Modeling Guide*. Micro Focus, Juni 2011. https://supportline.microfocus.com/Documentation/books/Together/2008R3/Together.pdf , besucht: 18.12.2017, Together-Vers. 12.6. 216, 220, 233, 237

[BP99] Beuze, Gerrit und Rene Post: *Design Patterns Reference – ModelMaker 5 (Anleitung)*. ModelMaker Tools, 1999. http://www.modelmakertools.com/i/o/mm-design-patterns-manual.html , besucht: 18.12.2017, Kurzbeschreibung: http://www.modelmakertools.com/modelmaker/design-patterns.html. 220, 230

[Bro04] Brown, Alan W.: *Model driven architecture: Principles and practice*. Software and Systems Modeling (SoSyM), 3(4):314–327, August 2004, ISSN 1619-1366. 27

[BTGH06] Babar, Muhammad Ali, Antony Tang, Ian Gorton und Jun Han: *Industrial Perspective on the Usefulness of Design Rationale for Software Maintenance: A Survey*. In: *Proceedings of the Sixth International Conference on Quality Software (QSIC)*, Seiten 201–208, Oktober 2006. 79, 145

[BZ07] Bayley, Ian und Hong Zhu: *Formalising Design Patterns in Predicate Logic*. In: *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, Seiten 25–36. IEEE Computer Society, 2007. 209

[CC90] Chikofsky, Elliot J. und James H. Cross II: *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, 7(1):13–17, Januar 1990. 28

[CH06] Czarnecki, Krzysztof und Simon Helsen: *Feature-Based Survey of Model Transformation Approaches*. IBM Systems Journal – Model-Driven Software Development, 45(3):621–645, Juli 2006, ISSN 0018-8670. 30, 34, 121, 127, 128

[CHE05] Czarnecki, Krzysztof, Simon Helsen und Ulrich Eisenecker: *Formalizing Cardinality-based Feature Models and their Specialization*. Software Process: Improvement and Practice, 10(1):7–29, 2005, ISSN 1099-1670. 76

[CHV00] Chambers, Craig, Bill Harrison und John Vlissides: *A Debate on Language and Tool Support for Design Patterns*. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Seiten 277–289, Januar 2000. 6, 10

[Cin00] Cinnéide, Mel Ó: *Automated Application of Design Patterns: A Refactoring Approach*. Dissertation, University of Dublin, Trinity College, Oktober 2000. 228

[CN99] Cinnéide, Mel Ó und Paddy Nixon: *A Methodology for the Automated Introduction of Design Patterns*. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Seiten 463–472, 1999. 228

[CR06] Clayberg, Eric und Dan Rubel: *Eclipse – Building Commercial-Quality Plug-ins*. The Eclipse Series. Addison-Wesley, 2. Auflage, März 2006, ISBN 978-0-321-42672-7. 156

[CS95] Coplien, James O. und Douglas C. Schmidt (Herausgeber): *Pattern Languages of Program Design*, Band 1 der Reihe *Software Patterns Series*. Addison-Wesley, Mai 1995, ISBN 978-0201607345. 24

[DAC07] Dong, Jing, Paulo Alencar und Donald Cowan: *Design Patterns Formalization Techniques*, Kapitel V: Formal Specification and Verification of Design Patterns, Seiten 94–108. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209, 238

[DACY07] Dong, Jing, Paulo S. C. Alencar, Donald D. Cowan und Sheng Yang: *Composing Pattern-based Components and Verifying Correctness*. Journal of Systems and Software, 80(11):1755–1769, November 2007. 209, 238

[Dai11] Daigneau, Robert: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 1. Auflage, November 2011, ISBN 978-0321544209. 24, 247

[DBHT12] Dziwok, Stefan, Kathrin Bröker, Christian Heinzemann und Matthias Tichy: *A Catalog of Real-Time Coordination Patterns for Advanced Mechatronic Systems*. Technischer Bericht tr-ri-12-319, Universität Paderborn, Februar 2012. 24, 76

[Dur14] Durdik, Zoya: *Architectural Design Decision Documentation through Reuse of Design Patterns*. Dissertation, Karlsruher Institut für Technologie, Juni 2014. 240

[DYZ07] Dong, Jing, Sheng Yang und Kang Zhang: *Visualizing Design Patterns in Their Applications and Compositions*. IEEE Transactions on Software Engineering, 33(7):433–453, Juli 2007. 222, 223

[EBM07] El Boussaidi, Ghizlane und Hafedh Mili: *A Model-Driven Framework for Representing and Applying Design Patterns*. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC), Beijing, China, 24–27 Juli 2007*, Band 1, Seiten 97–100, Juli 2007. 215, 231

[Ede00] Eden, Amnon H.: *Precise Specification of Design Patterns and Tool Support in their Application*. Dissertation, Department of Computer Science, University of Tel Aviv, 2000. 211, 212, 224, 235

[EEPT06] Ehrig, Hartmut, Karsten Ehrig, Ulrike Prange und Gabriele Taentzer: *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science – An EATCS Series. Springer, Februar 2006, ISBN 978-3540311874. 31, 32, 35, 70, 281, 282, 290, 320, 321

[Ehm13] Ehms, Dirk: *PatternBox: Design Pattern Editor Plugin for Eclipse*, 2013. http://www.patternbox.com/download/patternbox.pdf , besucht: 18.12.2017, Eclipse-Plug-in: http://www.patternbox.com/projekte/eclipse-plugin.html. 229

[ELN+92] Engels, Gregor, Claus Lewerentz, Manfred Nagl, Wilhelm Schäfer und Andy Schürr: *Building Integrated Software Development Environments. Part I: Tool Specification*. Transactions on Software Engineering and Methodology (TOSEM), 1(2):135–167, April 1992, ISSN 1049-331X. 34

[EN11] Eden, Amnon H. und Jonathan Nicholson: *Codecharts – Roadmaps and Blueprints for Object-Oriented Programs*. John Wiley and Sons, April 2011, ISBN 978-0470626948. 39, 209, 210, 211, 212, 220, 224, 235, 236

[Eng86] Engels, Gregor: *Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung*. Dissertation, Universität Osnabrück, 1986. 34

[ENG07] Eden, Amnon H., Jonathan Nicholson und Epameinondas Gasparis: *The 'Gang of Four' Companion*. Technischer Bericht CSM-472, University of Essex, Dezember 2007. http://lepus.org.uk/ref/companion/. 211, 212, 213

[Erl00] Erlikh, Len: *Leveraging Legacy System Dollars for E-Business*. IT Professional, 2(3):17–23, Mai 2000, ISSN 1520-9202. 1

[EYG97] Eden, Amnon H., Amiram Yehudai und Joseph Gil: *Precise Specification and Automatic Application of Design Patterns*. In: *Proceedings of the 12th International Conference on Automated Software Engineering (ASE)*, Seiten 143–152. IEEE Computer Society, 1997. 228

[FCA07] Flores, Andrés, Alejandra Cechich und Gabriela Aranda: *Design Patterns Formalization Techniques*, Kapitel III: A Generic Model of Object-Oriented Patterns Specified in RSL, Seiten 44–72. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208

[Fer13] Fernandez, Eduardo B.: *Security Patterns in Practise – Designing Secure Architectures Using Software Patterns*. Wiley Series in Software Design Patterns. John Wiley and Sons, April 2013, ISBN 978-1119998945. 24, 247

[FKGS04] France, Robert B., Dae-Kyoo Kim, Sudipto Ghosh und Eunjee Song: *A UML-Based Pattern Specification Technique*. IEEE Transactions on Software Engineering, 30(3):193–206, März 2004. 214, 215

[FMvW97] Florijn, Gert, Marco Meijers und Pieter van Winsen: *Tool Support for Object-Oriented Patterns*. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Band 1241 der Reihe *Lecture Notes in Computer Science*, Seiten 472–495. Springer-Verlag Berlin Heidelberg, 1997, ISBN 978-3540630890. 8, 210, 211, 220, 228, 235

[FNT98] Fischer, Thorsten, Jörg Niere und Lars Torunski: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*. Diplomarbeit, Universität Paderborn, Juli 1998. 33, 34

[FNTZ00] Fischer, Thorsten, Jörg Niere, Lars Torunski und Albert Zündorf: *Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java*. In: *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, Band 1764 der Reihe *Lecture Notes in Computer Science*, Seiten 296 – 309. Springer Berlin / Heidelberg, 2000. 16, 33, 35, 102, 341

[Fow96] Fowler, Martin: *Analysis Patterns – Reusable Object Models*. The Addison-Wesley Object Technology Series. Addison-Wesley, November 1996, ISBN 978-0201895421. 24

[Fow99] Fowler, Martin: *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, Juni 1999, ISBN 978-0201485677. 238

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 1. Auflage, November 2002, ISBN 978-0321127426. 23, 24, 65, 73, 156, 178, 179, 180, 198, 247, 300, 301

[Fra92] Frazer, A.: *Reverse Engineering - hype, hope or here?* UNICOM Applied Information Technology, 12:209–243, 1992. 1

[Gam96] Gamma, Erich: *Applying Design Patterns in Java.* Java Report, 1(6):47–53, 1996. Später veröffentlicht in [Gam98]. 222, 223

[Gam98] Gamma, Erich: *Applying Design Patterns in Java.* Java Gems – Jewels from Java Report, Seiten 105–114, Juli 1998. 376, 374

[Gam01] Gamma, Erich: *JUnit: A Cook's Tour*, Juli 2001. https://github.com/junit-team/junit/tree/master/doc/cookstour, siehe auch http://junit.org. 16, 180, 200, 201, 223, 244, 341, 342, 344, 345, 348, 351, 352, 354, 358

[Gas07] Gasparis, Epameinondas: *Design Patterns Formalization Techniques*, Kapitel XVI: LePUS: A Formal Language for Modeling Design Patterns, Seiten 357–372. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208, 209

[GB03] Gamma, Erich und Kent Beck: *Contributing to Eclipse – Principles, Patterns, and Plug-Ins.* The Eclipse Series. Addison-Wesley, Oktober 2003, ISBN 978-0-321-20575-9. 156

[GENK08] Gasparis, Epameinondas, Amnon H. Eden, Jonathan Nicholson und Rick Kazman: *The Design Navigator: Charting Java Programs.* In: *Companion of the 30th International Conference on Software Engineering (ICSE)*, Seiten 945–946. ACM, 2008. 209, 210, 212, 235

[GHJV95] Gamma, Erich, Richard Helm, Ralph E. Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, USA, 1995, ISBN 0201633612. 2, 3, 4, 5, 10, 19, 20, 21, 23, 40, 42, 43, 44, 47, 48, 58, 59, 61, 65, 67, 72, 73, 76, 129, 158, 169, 170, 172, 173, 175, 176, 182, 184, 193, 194, 200, 209, 211, 212, 225, 228, 229, 244, 252, 253, 256, 291, 341, 342, 351, 354, 367

[GHS09] Giese, Holger, Stephan Hildebrandt und Andreas Seibel: *Improved Flexibility and Scalability by Interpreting Story Diagrams.* In: Margaria, Tiziana, Julia Padberg und Gabriele Taentzer (Herausgeber): *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, Band 18. Electronic Communications of the EASST, 2009. 34, 244

[GM05] Gil, Joseph (Yossi) und Itay Maman: *Micro patterns in Java code.* In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seiten 97–116. ACM, 2005, ISBN 1-59593-031-0. 24

[GR10] Greenyer, Joel und Jan Rieke: *Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars.* Software & Systems Modeling (SoSyM), 9(1):21–46, 2010, ISSN 1619-1366. Published online July 15, 2009. 126

[Gro09] Gronback, Richard C.: *Eclipse Modeling Project – A Domain-Specific Language (DSL) Toolkit.* The Eclipse Series. Addison-Wesley, März 2009, ISBN 978-0-321-53407-1. 156

[GSR05] Geiger, Leif, Christian Schneider und Carsten Reckord: *Template- and modelbased code generation for MDA-Tools.* In: Giese, Holger und Albert Zündorf (Herausgeber): *Proceedings of the 3rd International Fujaba Days 2005*, Seiten 1–6. Universität Paderborn, September 2005. tr-ri-06-275. 34, 244

[Han07] Hanmer, Robert S.: *Patterns for Fault Tolerant Software.* Wiley Series in Software Design Patterns. John Wiley and Sons, Oktober 2007, ISBN 978-0470319796. 24

[HC07] Henninger, Scott und Victor Corrêa: *Software Pattern Communities: Current Practices and Challenges.* In: *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP)*, Seiten 14:1–14:19. ACM, September 2007, ISBN 978-1605584119. 24

[HFR99] Harrison, Neil, Brian Foote und Hans Rohnert (Herausgeber): *Pattern Languages of Program Design*, Band 4 der Reihe *Software Patterns Series.* Addison-Wesley, November 1999, ISBN 978-0201433043. 24

[HKM07] Helin, Joni, Pertti Kellomäki und Tommi Mikkonen: *Design Patterns Formalization Techniques*, Kapitel IV: Patterns of Collective Behavior in Ocsid, Seiten 73–93. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209

[HLG+13] Hildebrandt, Stephan, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin und Andy Schürr: *A Survey of Triple Graph Grammar Tools.* In: Margaria, Tiziana, Julia Padberg und Gabriele Taentzer (Herausgeber): *Proceedings of the 2nd International Workshop on Bidirectional Transformations (BX 2013)*, Band 57. Electronic Communications of the EASST, 2013. 126, 239

[HM08] Hesse, Wolfgang und Heinrich C. Mayr: *Modellierung in der Softwaretechnik: eine Bestandsaufnahme.* Informatik-Spektrum, 31:377–393, 2008, ISSN 0170-6012. 10.1007/s00287-008-0276-7. 29, 41

[HMN07] Herranz, Angel und Juan José Moreno-Navarro: *Design Patterns Formalization Techniques*, Kapitel X: Modeling and Reasoning about Design Patterns in SLAM-SL, Seiten 206–235. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209

[HRvD+11] Heinzemann, Christian, Jan Rieke, Markus von Detten, Dietrich Travkin und Marius Lauder: *A new Meta-Model for Story Diagrams.* In: *Proceedings of the 8th International Fujaba Days*, Mai 2011. 33

[HW03] Hohpe, Gregor und Bobby Woolf: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 1. Auflage, Oktober 2003, ISBN 978-0321200686. 24, 247

[IBM] *IBM Rational Software Architect*. IBM. http://www-01.ibm.com/software/rational/products/swarchitect/ , besucht: 18.12.2017. 8

[KC05] Konrad, Sascha und Betty H. C. Cheng: *Real-time Specification Patterns*. In: Roman, Gruia Catalin, William G. Griswold und Bashar Nuseibeh (Herausgeber): *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, Seiten 372–381. ACM, Mai 2005. 24

[KCC04] Konrad, Sascha, Betty H. C. Cheng und Laura A. Campbell: *Object analysis patterns for embedded systems*. IEEE Transactions on Software Engineering, 30(12):970–992, Dezember 2004, ISSN 0098-5589. 24

[Ker04] Kerievsky, Joshua: *Refactoring to Patterns*. Addison-Wesley, August 2004, ISBN 978-0321213358. 232

[Kim07] Kim, Dae Kyoo: *Design Patterns Formalization Techniques*, Kapitel IX: The Role-Based Metamodeling Language for Specifying Design Patterns, Seiten 183–205. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208, 214

[KJ04] Kircher, Michael und Prashant Jain: *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley Series in Software Design Patterns. Wiley, April 2004, ISBN 978-0-470-84525-7. 24

[KKZB08] Koch, Nora, Alexander Knapp, Gefei Zhang und Hubert Baumeister: *UML-Based Web Engineering*. In: Rossi, Gustavo, Oscar Pastor, Daniel Schwabe und Luis Olsina (Herausgeber): *Web Engineering: Modelling and Implementing Web Applications*, Human Computer Interaction Series, Seiten 157–191. Springer London, 2008, ISBN 978-1-84628-923-1. 44, 105

[KM10] Kajsa, Peter und L'ubomír Majtás: *Design Patterns Instantiation Based on Semantics and Model Transformations*. In: Leeuwen, Jan van, Anca Muscholl, David Peleg, Jaroslav Pokorný und Bernhard Rumpe (Herausgeber): *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Band 5901 der Reihe *Lecture Notes in Computer Science*, Seiten 540–551. Springer Berlin Heidelberg, 2010, ISBN 978-3642112669. 227, 231

[KN12] Kajsa, Peter und Pavol Návrat: *Design Pattern Support Based on the Source Code Annotations and Feature Models*. In: Bieliková, Mária, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser und György Turán (Herausgeber): *Proceedings of the 38th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Band 7147 der Reihe *Lecture Notes in Computer Science*, Seiten 467–478. Springer Berlin Heidelberg, 2012, ISBN 978-3642276606. 227, 229, 231

[Kno12] Knoernschild, Kirk: *Java Application Architecture – Modularity Patterns with Examples Using OSGi*. Robert C. Martin Series. Prentice Hall, März 2012, ISBN 978-0321247131. 24

[Kön11] Könemann, Patrick: *Integrating Design Decision Management with Model-based Software Development*. Dissertation, Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, 2011. IMM-PHD-2011-249. 232, 233, 240

[KS08] Kim, Dae-Kyoo und Wuwei Shen: *Evaluating Pattern Conformance of UML Models: A Divide-and-Conquer Approach and Case Studies*. Software Quality Journal, 16(3):329–359, 2008, ISSN 0963-9314. 8, 214, 219, 236

[Küh99] Kühne, Thomas: *A Functional Pattern System for Object-Oriented Design*. Verlag Dr. Kovac, Hamburg, Germany, 1999, ISBN 978-3860647707. http://homepages.mcs.vuw.ac.nz/~tk/fps/. 24

[Küh06a] Kühne, Thomas: *Clarifying matters of (meta-) modeling: an author's reply*. International Journal on Software and Systems Modeling (SoSyM), 5(4):395–401, Dezember 2006, ISSN 1619-1366. 30

[Küh06b] Kühne, Thomas: *Matters of (Meta-) Modeling*. International Journal on Software and Systems Modeling (SoSyM), 5(4):369–385, Dezember 2006, ISSN 1619-1366. 30

[KW05] Kim, Dae-Kyoo und Jon Whittle: *Generating UML Models from Domain Patterns*. In: *Proceedings of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, Seiten 166–173, August 2005. 214, 230

[KW07] Kindler, Ekkart und Robert Wagner: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Technischer Bericht tr-ri-07-284, Universität Paderborn, Juni 2007. 126, 239

[Lan07] Lano, Kevin: *Design Patterns Formalization Techniques*, Kapitel VIII: Formalising Design Patterns as Model Transformations, Seiten 156–182. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209

[LAS17] Leblebici, Erhan, Anthony Anjorin und Andy Schürr: *Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques*. In: Huisman, Marieke und Julia Rubin (Herausgeber): *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Seiten 191–207. Springer-Verlag Berlin Heidelberg, 2017. 126, 154, 239, 246

[LAST16] Leblebici, Erhan, Anthony Anjorin, Andy Schürr und Gabriele Taentzer: *Multi-amalgamated triple graph grammars: Formal foundation and application to visual language translation*. Journal of Visual Languages & Computing, 42(Supplement C):99–121, 2016, ISSN 1045-926X. 143, 239, 246

[LB05] Lau, Christina und Jim Bonanno: *Content authoring demystified: Authoring Rational Software Architect patterns and transforms*. IBM developerWorks, November 2005. http://www.ibm.com/developerworks/rational/library/05/1101_lao-bonanno/ , besucht: 18.12.2017. 220, 234

[Lea99] Lea, Doug: *Concurrent Programming in Java – Design Principles and Patterns*. The Java Series. Addison-Wesley, 2. Auflage, Juni 1999, ISBN 978-0201310092. 24

[Leh96] Lehman, Meir M.: *Laws of Software Evolution Revisited.* In: *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT)*, Seiten 108–124, London, UK, 1996. Springer-Verlag, ISBN 3-540-61771-X. 1

[LGSJ00] Le Guennec, Alain, Gerson Sunyé und Jean Marc Jézéquel: *Precise Modeling of Design Patterns.* In: Evans, Andy, Stuart Kent und Bran Selic (Herausgeber): *Proceedings of the 3rd International Conference on «UML» 2000 – The Unified Modeling Language, Advancing the Standard*, Band 1939 der Reihe *Lecture Notes in Computer Science*, Seiten 482–496. Springer-Verlag Berlin Heidelberg, 2000, ISBN 978-3540400110. 212, 220

[LK08] Lee, Larix und Philippe Kruchten: *A Tool to Visualize Architectural Design Decisions.* In: *Proceedings of the 4th International Conference on Quality of Software-Architectures (QoSA): Models and Architectures*, Seiten 43–54, Berlin, Heidelberg, Oktober 2008. Springer-Verlag, ISBN 978-3-540-87878-0. 240

[LSV07] Lovatt, Howard, Anthony M. Sloane und Dominic R. Verity: *Design Patterns Formalization Techniques*, Kapitel XV: A Pattern Enforcing Compiler (PEC) for Java: A Practical Way to Formally Specify Patterns, Seiten 324–356. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208

[Mak04] Mak, Ka Hing: *Precise Specification of Deisgn Patterns and Compound Patterns.* Dissertation, The Hong Kong Polytechnic University, 2004. 210, 214

[MAW07] Mussbacher, Gunter, Daniel Amyot und Michael Weiss: *Design Patterns Formalization Techniques*, Kapitel XIV: Formalizing Patterns with the User Requirements Notation, Seiten 302–323. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208

[MCL03] Mak, Jeffrey K. H., Clifford S. T. Choy und Daniel P. K. Lun: *Precise Specification to Compound Patterns with ExLePUS.* In: *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, Seiten 440–445, November 2003. 210, 214

[MCL04] Mak, Jeffrey K. H., Clifford S. T. Choy und Daniel P. K. Lun: *Precise Modeling of Design Patterns in UML.* In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Seiten 252–261, Mai 2004. 214, 220

[MDE97] Meijler, Theo Dirk, Serge Demeyer und Robert Engel: *Making Design Patterns Explicit in FACE: A Framework Adaptive Composition Environment.* SIGSOFT Software Engineering Notes, 22(6):94–110, November 1997, ISSN 0163-5948. 224

[MEB05] Mili, Hafedh und Ghizlane El Boussaidi: *Representing and Applying Design Patterns: What Is the Problem?* In: Briand, Lionel und Clay Williams (Herausgeber): *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Band 3713 der Reihe *Lecture Notes in Computer Science*, Seiten 186–200. Springer-Verlag Berlin Heidelberg, Oktober 2005, ISBN 978-3540320579. 215, 231

[Mey09] Meyer, Matthias: *Musterbasiertes Re-Engineering von Softwaresystemen.* Dissertation, Universität Paderborn, Dezember 2009. 232, 238

[MGC13] Macedo, Nuno, Tiago Guimarães und Alcino Cunha: *Model Repair and Transformation with Echo*. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Seiten 694–697, November 2013. 239

[MHG02] Maplesden, David, John Hosking und John Grundy: *Design Pattern Modelling and Instantiation using DPML*. In: *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, Seiten 3–11. Australian Computer Society, 2002. 9, 213, 219, 230, 236

[MHG07] Maplesden, David, John Hosking und John Grundy: *Design Patterns Formalization Techniques*, Kapitel II: A Visual Language for Design Pattern Modeling and Instantiation, Seiten 20–43. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 9, 208, 213, 219, 230, 236

[Mik98] Mikkonen, Tommi: *Formalizing Design Patterns*. In: *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, Seiten 115–124. IEEE Computer Society Press, 1998. 209

[Moo09] Moody, Daniel L.: *The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering*. IEEE Transactions on Software Engineering, 35(6):756–779, November 2009, ISSN 0098-5589. 45, 50, 186

[MRB97] Martin, Robert C., Dirk Riehle und Frank Buschmann (Herausgeber): *Pattern Languages of Program Design*, Band 3 der Reihe *Software Patterns Series*. Addison-Wesley, November 1997, ISBN 978-0201310115. 24

[MSM04] Mattson, Timothy G., Beverly A. Sanders und Berna L. Massingill: *Patterns for Parallel Programming*. Software Patterns Series. Addison-Wesley, 2004, ISBN 978-0321228116. 24, 247

[MvH09] Moody, Daniel L. und Jos van Hillegersberg: *Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams*. In: Gašević, Dragan, Ralf Lämmel und Eric Van Wyk (Herausgeber): *Proceedings of the 1st International Conference on Software Language Engineering (SLE). Revised Selected Papers*, Band 5452 der Reihe *Lecture Notes in Computer Science*, Seiten 16–34. Springer Berlin Heidelberg, 2009, ISBN 978-3642004346. 50, 51

[MVN06] Manolescu, Dragos, Markus Völter und James Noble (Herausgeber): *Pattern Languages of Program Design*, Band 5 der Reihe *Software Patterns Series*. Addison-Wesley, April 2006, ISBN 978-0321321947. 24

[NGEK09] Nicholson, Jonathan, Epameinondas Gasparis, Amnon H. Eden und Rick Kazman: *Automated Verification of Design Patterns with LePUS3*. In: *1st NASA Formal Methods Symposium*, 2009. 8, 39, 209, 210, 211, 212, 220, 235, 236

[Nie04] Niere, Jörg: *Inkrementelle Entwurfsmustererkennung*. Dissertation, Universität Paderborn, Juni 2004. 218

[NNZ00] Nickel, Ulrich A., Jörg Niere und Albert Zündorf: *Tool Demonstration: The FUJABA Environment*. In: *Proceedings of the* 22^{nd} *International Conference on Software Engineering (ICSE)*, 2000. 33, 34

[NSN] Neelakant, Vineeth, Priyananda Shenoy und Keerti L. Narayan: *DpaToolkit Manual*. http://dpatoolkit.sourceforge.net/DPA_MANUAL_PAGE.html , besucht: 18.12.2017. 233

[NSW+02] Niere, Jörg, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals und Jim Welsh: *Towards Pattern-Based Design Recovery*. In: *Proceedings of the 24th International Conference on Software Engineering*, Seiten 338–348. ACM Press, Mai 2002. 34, 42, 218, 238

[NZJ13] Norbisrath, Ulrich, Albert Zündorf und Ruben Jubeh: *Story Driven Modeling*. CreateSpace Independent Publishing Platform, April 2013, ISBN 978-1483949253. 16, 33, 34, 35, 38, 102

[OA10] Ortega-Arjona, Jorge Luis: *Patterns for Parallel Software Design*. Wiley Series in Software Design Patterns. John Wiley and Sons, Januar 2010, ISBN 978-0470697344. 24, 247

[OMG03] *MDA guide version 1.0.1*. Object Management Group, 2003. Document omg/2003-06-01. 10, 28

[OMG11a] *Unified Modeling Language (UML) Infrastructure Specification Ver. 2.4.1*. Object Management Group, 2011. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF, Document formal/2011-08-05. 13, 28, 44, 185

[OMG11b] *Unified Modeling Language (UML) Superstructure Specification Ver. 2.4.1*. Object Management Group, 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, Document formal/2011-08-06. 13, 28, 44, 66, 93, 185, 220, 221, 222

[OMG11c] *Meta Object Facility (MOF) Core Specification Ver. 2.4.1*. Object Management Group, 2011. http://www.omg.org/spec/MOF/2.4.1/PDF, Document formal/2011-08-07. 13, 44, 156, 341

[OMG11d] *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Ver. 1.1*. Object Management Group, 2011. http://www.omg.org/spec/QVT/1.1/PDF/, Document formal/2011-01-01. 28, 126, 239

[OMG14] *Object Constraint Language (OCL) Ver. 2.4*. Object Management Group, 2014. http://www.omg.org/spec/OCL/2.4/PDF, Document formal/14-02-03. 154

[Par94] Parnas, David Lorge: *Software Aging*. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, Seiten 279–287. IEEE Computer Society Press, 1994. 1

[Par07] Park, Jaeyong: *Perfective and Corrective UML Pattern-based Design Maintenance with Design Constraints for Information Systems*. Dissertation, George Mason University, Fairfax, Virginia, USA, 2007. 215, 237

[Par10] Parr, Terence: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1. Auflage, Januar 2010, ISBN 978-1934356456. 24

[PG10] Porras, Gerardo Cepeda und Yann Gaël Guéhéneuc: *An Empirical Study on the Efficiency of Different Design Pattern Representations in UML Class Diagrams*. Empirical Software Engineering, 15(5):493–522, Februar 2010. 223

[Pre01] Prechelt, Lutz: *Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik*. Springer Verlag, Berlin, Heidelberg, New York, Januar 2001, ISBN 3-540-41257-3. 98, 169

[PRW08] Park, Jaeyong, David C. Rine und Elizabeth White: *Assessing Conformance of Pattern-based Design in UML*. In: *Proceedings of the 46th Annual Southeast Conference (ACMSE)*, ACM-SE 46, Seiten 298–303. ACM, 2008, ISBN 978-1-60558-105-7. 215, 237

[PULPT02] Prechelt, Lutz, Barbara Unger-Lamprecht, Michael Philippsen und Walter Tichy: *Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance*. IEEE Transactions on Software Engineering, 28(6):595–606, Juni 2002, ISSN 0098-5589. 2, 20, 23, 79, 145

[PvDB12] Platenius, Marie Christin, Markus von Detten und Steffen Becker: *Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies*. In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, Seiten 255 – 264. IEEE, März 2012, ISBN 978-0769546667. 238

[PvDT11] Platenius, Marie Christin, Markus von Detten und Dietrich Travkin: *Visualization of Pattern Detection Results in Reclipse*. In: Norbisrath, Ulrich und Ruben Jubeh (Herausgeber): *Proceedings of the 8th International Fujaba Days*, Nummer 2012, 1 in *Kasseler Informatikschriften (KIS)*, Seiten 33–37. Universität Kassel, Mai 2011. 226, 227

[RCOH07] Raje, Rajeev R., Sivakumar Chinnasamy, Andrew M. Olson und William M. Higdon: *Design Patterns Formalization Techniques*, Kapitel XI: The Applications and Enhancement of LePUS for Specifying Design Patterns, Seiten 236–257. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208

[Rie11] Riehle, Dirk: *Lessons Learned from Using Design Patterns in Industry Projects*. In: Noble, James, Ralph Johnson, Paris Avgeriou, Neil B. Harrison und Uwe Zdun (Herausgeber): *Transactions on Pattern Languages of Programming II: Special Issue on Applying Patterns*, Seiten 1–15. Springer-Verlag, 2011, ISBN 978-3-642-19431-3. 23

[Rie14] Rieke, Jan: *Model Consistency Management for Systems Engineering*. Dissertation, Universität Paderborn, Juli 2014. 126, 154, 239, 246

[Ris00] Rising, Linda: *The Pattern Almanac 2000*. Software Patterns Series. Addison-Wesley, Boston, MA, USA, Januar 2000. 24

[Roz97] Rozenberg, Grzegorz: *Handbook of Graph Grammars and Computing by Graph Transformation*, Band 1 (Foundations). World Scientific Publishing Co. Pte. Ltd., Februar 1997, ISBN 978-9810228842. 31, 35

[SBPM08] Steinberg, David, Frank Budinsky, Marcelo Paternostro und Ed Merks: *EMF – Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2. Auflage, Dezember 2008, ISBN 978-0-321-33188-5. 13, 16, 44, 81, 87, 102, 156, 232, 244, 341

[SCG+05] Swithinbank, Peter, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie und Larry Yusuf: *Patterns: Model-Driven Development Using IBM Rational Software Architect*. IBM Redbooks, Dezember 2005, ISBN 978-0738492889. http://www.redbooks.ibm.com/redbooks/pdfs/sg247105.pdf. 234

[Sch86] Schäfer, Wilhelm: *Eine integrierte Software-Entwicklungsumgebung Konzepte, Entwurf und Implementierung*. Dissertation, Universität Osnabrück, 1986. 34

[Sch91] Schürr, Andy: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991, ISBN 978-3-8244-2021-6. 34

[Sch94] Schürr, Andy: *Specification of Graph Translators with Triple Graph Grammars*. In: Mayr, Ernst W., Gunther Schmidt und Gottfried Tinhofer (Herausgeber): *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, Band 903 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 151–163. Springer Berlin Heidelberg, 1994, ISBN 978-3-540-59071-2. 126, 143, 239

[SH04] Soundarajan, Neelam und Jason O. Hallstrom: *Responsibilities and Rewards: Specifying Design Patterns*. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Seiten 666–675, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7695-2163-0. 209

[SH07] Soundarajan, Neelam und Jason O. Hallstrom: *Design Patterns Formalization Techniques*, Kapitel XIII: Precision, Flexibility, and Tool Support: Essential Elements of Pattern Formalization, Seiten 280–301. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 209

[Sip97] Sipser, Michael: *Introduction to the Theory of Computation*. International Thomson Publishing, 1. Auflage, Januar 1997, ISBN 978-0534947286. 31

[SK98] Schauer, Reinhard und Rudolf K. Keller: *Pattern Visualization for Software Comprehension*. In: *Proceedings of the* 6^{th} *International Workshop on Program Comprehension*, Seiten 1–9. IEEE Computer Society Press, Juni 1998. 222, 223

[SK08] Schürr, Andy und Felix Klar: *15 Years of Triple Graph Grammars*. In: Ehrig, Hartmut, Reiko Heckel, Grzegorz Rozenberg und Gabriele Taentzer (Herausgeber): *Proceedings of the 4th International Conference on Graph Transformations (ICGT)*, Seiten 411–425, Berlin, Heidelberg, 2008. Springer-Verlag, ISBN 978-3-540-87405-8. 126, 143

[SLGJ00] Sunyé, Gerson, Alain Le Guennec und Jean Marc Jézéquel: *Design Patterns Application in UML*. In: Bertino, Elisa (Herausgeber): *Proceedings of the 14th European*

Conference on Object-Oriented Programming (ECOOP), Band 1850 der Reihe *Lecture Notes in Computer Science*, Seiten 44–62. Springer-Verlag Berlin Heidelberg, 2000, ISBN 978-3540451020. 212, 233

[Smi11] Smith, Jason McColm: *The Pattern Instance Notation: A Simple Hierarchical Visual Notation for the Dynamic Visualization and Comprehension of Software Patterns*. Journal of Visual Languages & Computing, 22(5):355–374, 2011, ISSN 1045-926X. 76, 224, 225, 226

[Smi12] Smith, Jason McColm: *Elemental Design Patterns*. Addison-Wesley Professional, 1. Auflage, April 2012, ISBN 978-0321711922. 1, 25, 64, 209, 224, 225

[Som07] Sommerville, Ian: *Software Engineering*. Pearson Studium, 8. Auflage, April 2007, ISBN 978-3827372574. 1

[Spaa] *Enterprise Architect*. Sparx Systems. http://www.sparxsystems.de , besucht: 18.12.2017. 8

[Spab] *UML Patterns*. Sparx Systems. http://www.sparxsystems.com.au/resources/developers/uml_patterns.html , besucht: 18.12.2017, Details: http://www.sparxsystems.com.au/resources/developers/use_uml_patterns.html, http://www.sparxsystems.com.au/resources/developers/create_uml_patterns.html. 216, 233

[SS03] Smith, Jason McColm und David Stotts: *SPQR: Flexible Automated Design Pattern Extraction From Source Code*. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, Seiten 215–224. IEEE Computer Society Press, Oktober 2003. 225

[SS07] Smith, Jason McColm und David Stotts: *Design Patterns Formalization Techniques*, Kapitel VII: Intent-Oriented Design Pattern Formalization Using SPQR, Seiten 123–155. In: Taibi, Toufik [Tai07a], 2007, ISBN 978-1599042190. 208, 209

[SSRB00] Schmidt, Douglas C., Michael Stal, Hans Rohnert und Frank Buschmann: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley Series in Software Design Patterns. Wiley, August 2000, ISBN 978-0-471-60695-6. 24, 62, 247

[ST08] Stürmer, Ingo und Dietrich Travkin: *Tool Supported Quality Assessment and Improvement in MATLAB Simulink and Stateflow Models*. In: Gehrke, Matthias, Holger Giese und Joachim Stroop (Herausgeber): *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, Band 236 der Reihe *HNI Verlagsschriftenreihe*, Seiten 61–68. Universität Paderborn, 2008, ISBN 978-3-939350-55-2. 153

[Sta73] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Springer, Wien, Dezember 1973, ISBN 978-3211811061. 29, 41

[SV06] Stahl, Thomas und Markus Völter: *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley and Sons, Ltd., 1. Auflage, Mai 2006, ISBN 978-0470025703. 10, 27, 28, 30

[SWZ95] Schürr, Andy, Andreas J. Winter und Albert Zündorf: *Graph Grammar Engineering with PROGRES*. In: Schäfer, Wilhelm und Pere Botella (Herausgeber): *Proceedings of the 5th European Software Engineering Conference (ESEC)*, Seiten 219–234, Berlin, Heidelberg, 1995. Springer-Verlag, ISBN 978-3-540-45552-3. 34

[Szy98] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998, ISBN 0-201-17888-5. 66

[Tai07a] Taibi, Toufik (Herausgeber): *Design Patterns Formalization Techniques*. IGI Publishing, Hershey, Pennsylvania, USA, 2007, ISBN 978-1599042190. 9, 208, 372, 374, 375, 376, 377, 378, 379, 380, 381, 383, 384, 385, 386, 370, 373, 382

[Tai07b] Taibi, Toufik: *Design Patterns Formalization Techniques*, Kapitel I: An Integrated Approach to Design Pattern Formalization, Seiten 1–19. In: *Tai07a* [Tai07a], 2007, ISBN 978-1599042190. 209

[Tra06] Travkin, Dietrich: *Bewertung automatisch erkannter Instanzen von Software-Mustern*. Diplomarbeit, Universität Paderborn, August 2006. http://dx.doi.org/10.17619/UNIPB/1-334, DOI: 10.17619/UNIPB/1-334. 217, 238

[Tra10] Travkin, Dietrich: *Towards Better Support for Pattern-Oriented Software Development*. In: *Proceedings of the 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)*, Seiten 55–59, 2010. 207

[Vai15] Vaisnoriene, Daiva: *MagicDraw 18.2 Documentation – Creating a custom design pattern*, Juli 2015. http://docs.nomagic.com/display/MD182/Creating+a+custom+design+pattern , besucht: 18.12.2017, Details: http://docs.nomagic.com/display/MD182/Creating+a+pattern und http://docs.nomagic.com/display/MD182/Target+concept. 234

[VB04] Völter, Markus und Jorn Bettin: *Patterns for Model-Driven Software-Development*. In: *Proceedings of the 9th European Conference on Pattern Languages of Programs (EuroPLoP)*, Seiten D3-1–D3-54, Juli 2004. 24

[vD13] Detten, Markus von: *Reengineering of Component-Based Software Systems in the Presence of Design Deficiencies*. Dissertation, Universität Paderborn, März 2013. 238

[vDHH+12] Detten, Markus von, Christian Heinzemann, Stephan Hildebrandt, Marie Christin Platenius, Jan Rieke und Dietrich Travkin: *Story Diagrams – Syntax and Semantics*. Technischer Bericht tr-ri-12-324, Universität Paderborn, Juli 2012. Ver. 0.2. 16, 33, 34, 35, 36, 102, 166, 244, 341

[vDMT10a] Detten, Markus von, Matthias Meyer und Dietrich Travkin: *Reclipse – A Reverse Engineering Tool Suite*. Technischer Bericht tr-ri-10-312, Universität Paderborn, März 2010. 153, 217, 218, 226, 238

[vDMT10b] Detten, Markus von, Matthias Meyer und Dietrich Travkin: *Reverse Engineering with the Reclipse Tool Suite*. In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, Seiten 299–300. ACM Press, Mai 2010. 153, 218, 226, 238

[vGB02] Gurp, Jilles van und Jan Bosch: *Design erosion: problems and causes*. The Journal of Systems and Software, 61(2):105–119, 2002. 1, 20, 79, 145

[VKC96] Vlissides, John, Norman Kerth und James O. Coplien (Herausgeber): *Pattern Languages of Program Design*, Band 2 der Reihe *Software Patterns Series*. Addison-Wesley, August 1996, ISBN 978-0201895278. 24

[Vli98a] Vlissides, John: *Notation, Notation, Notation*. C++ Report, Seiten 48–51, April 1998. 223

[Vli98b] Vlissides, John: *Pattern Hatching – Design Patterns Applied*. Software Patterns Series. Addison-Wesley, Juli 1998, ISBN 978-0201432930. 25

[Vok04] Vokáč, Marek: *Defect Frequency and Design Patterns: An Empirical Study of Industrial Code*. IEEE Transactions on Software Engineering, 30(12):904–917, Dezember 2004. 2, 20

[Völ03] Völter, Markus: *A Catalog of Patterns for Program Generation*. In: *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP)*, Seiten B6–1–B6–34, Juni 2003. 24

[Völ06] Völter, Markus: *Software Architecture – A pattern language for building sustainable software architectures*. In: *Proceedings of the 11th European Conference on Pattern Languages of Programs 2006 (EuroPLoP)*, Seiten MT2–1 – MT2–33, März 2006. 24

[Völ09] Völter, Markus: *Handling Variability*. In: *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, Seiten E5-1–E5-12, Dezember 2009. Vers. 2.0. 24

[Wen05a] Wenzel, Sven: *An Approach to the Automatic Recognition of Design Forms*. Diplomarbeit, Universität Dortmund, September 2005. 216, 220, 238

[Wen05b] Wenzel, Sven: *Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams*. Nordic Journal of Computing, 12(4):379–394, Dezember 2005. 42, 216, 223, 238

[Wen07] Wendehals, Lothar: *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. Dissertation, Universität Paderborn, September 2007. 42, 218, 238

[YA03] Yacoub, Sherif M. und Hany H. Ammar: *Pattern-Oriented Analysis and Design – Composing Patterns to Design Software Systems*. Addison-Wesley, 2003, ISBN 978-0201776409. 216, 234

[ZB13] Zhu, Hong und Ian Bayley: *An Algebra of Design Patterns*. Transactions on Software Engineering and Methodology (TOSEM), 22(3):23:1–23:35, Juli 2013, ISSN 1049-331X. 209

[Zün01] Zündorf, Albert: *Rigorous Object Oriented Software Development*. Habilitation, Universität Paderborn, 2001. 33, 34, 35, 37, 38, 244