

HANSER



Leseprobe

zu

„Design Patterns mit Java“

von Florian Siebler

ISBN (Buch): 978-3-446-43616-9

ISBN (E-Book): 978-3-446-44111-8

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/978-3-446-43616-9>

sowie im Buchhandel

© Carl Hanser Verlag München

Inhalt

Mae govannen!	XIII
1 Singleton Pattern	1
1.1 Was Design Patterns sind	1
1.2 Die GoF und deren Verdienst	2
1.3 Die Musterkategorien der GoF	3
1.3.1 Erzeugungsmuster	3
1.3.2 Verhaltensmuster	3
1.3.3 Strukturmuster	3
1.3.4 Bewertung der Kategorisierung	4
1.4 Beschreibung der Muster	4
1.5 Was ist ein Pattern?	5
1.6 Objektorientierte Programmierung	6
1.7 Muster in der Praxis	9
1.8 Das Singleton Pattern	10
1.8.1 Aufgabe von Singleton und Beispiele	10
1.8.2 Realisierung des Patterns	11
1.8.3 Die Zugriffsmethode synchronisieren	12
1.8.4 Eine andere Lösung: Double-checked locking	13
1.8.5 Letzter Ansatz: early instantiation – frühes Laden	14
1.9 Antipattern	15
1.9.1 Kritik an Singleton	15
1.9.2 Ist Singleton ein Antipattern?	16
1.10 Zusammenfassung	16
2 Template Method Pattern	19
2.1 Die Arbeitsweise von Template Method	19
2.1.1 Der erste Ansatz	19
2.1.2 Der zweite Ansatz	20
2.1.3 Das Hollywood-Prinzip	21
2.1.4 Einführen von Hook-Methoden	21
2.2 Das Interface „ListModel“	23

2.3	Der patternCoder	24
2.3.1	Der patternCoder aus Anwendersicht	25
2.4	Zusammenfassung	34
3	Observer Pattern	35
3.1	Einleitung	35
3.2	Eine erste Realisierung	35
3.3	Den ersten Ansatz erweitern	37
3.4	Observer in der Klassenbibliothek	38
3.5	Nebenläufiger Zugriff	39
3.5.1	Zugriffe synchronisieren	40
3.5.2	Die Datenbasis kopieren	40
3.5.3	Einsatz einer thread-sicheren Liste	41
3.6	Observer als Listener	42
3.7	Listener in der GUI-Programmierung	43
3.8	Das Model-View-Controller Pattern	48
3.9	Zusammenfassung	49
4	Mediator Pattern	51
4.1	Abgrenzung zum Observer Pattern	51
4.2	Aufgabe des Mediator Patterns	51
4.3	Mediator in Aktion – ein Beispiel	52
4.3.1	Definition eines Consumers	52
4.3.2	Definition eines Producers	53
4.3.3	Interface des Mediators	54
4.3.4	Test des Mediator-Patterns	55
4.4	Mediator in Aktion – das zweite Beispiel	56
4.4.1	Mediator in der GUI-Programmierung	56
4.4.2	Aufbau der GUI	57
4.5	Kritik an Mediator	59
4.6	Zusammenfassung	59
5	Chain of Responsibility	61
5.1	Ein Beispiel aus der realen Welt	61
5.2	Beispiel 1: Lebensmitteleinkauf	61
5.2.1	Die benötigten Lebensmittel	62
5.2.2	Die Verkäufer der Lebensmittel	62
5.2.3	Der Client	64
5.2.3.1	Erweiterung des Projekts	64
5.2.3.2	Variationen des Patterns	65
5.3	Ein Beispiel aus der Klassenbibliothek	67
5.4	Zusammenfassung	68

6	State Pattern	69
6.1	Exkurs: das Enum Pattern	69
6.1.1	Einen Zustand durch Zahlenwerte darstellen	69
6.1.2	Einen Zustand durch Objekte darstellen	70
6.1.3	Umsetzung in der Java-Klassenbibliothek	71
6.2	Den Zustand eines Objekts ändern	72
6.2.1	Den Zustand ändern – erster Ansatz	72
6.2.2	Den Zustand ändern – zweiter Ansatz	73
6.3	Prinzip des State Patterns	75
6.3.1	Die Rolle aller Zustände definieren	75
6.3.2	Das Projekt aus der Sicht des Client	77
6.3.3	Veränderung des Projekts	78
6.3.3.1	State-Objekte zentral im Kontext verwalten	79
6.3.3.2	State-Objekte als Rückgabewerte von Methodenaufrufen	79
6.4	Das State Pattern in der Praxis	80
6.5	Zusammenfassung	81
7	Command Pattern	83
7.1	Befehle in Klassen kapseln	83
7.1.1	Version 1 – Grundversion	83
7.1.2	Weitere Anbieter treten auf	85
7.1.3	Einen Befehl kapseln	86
7.2	Command in der Klassenbibliothek	89
7.2.1	Beispiel 1: Nebenläufigkeit	89
7.2.2	Beispiel 2: Event-Handling	90
7.3	Befehlsobjekte wiederverwenden	91
7.3.1	Das Interface „Action“	91
7.3.2	Verwendung des Interface „Action“	91
7.4	Undo und redo von Befehlen	92
7.4.1	Ein einfaches Beispiel	93
7.4.2	Ein umfangreicheres Beispiel	95
7.4.3	Besprechung des Source Codes	96
7.4.3.1	Die beteiligten Klassen	96
7.4.3.2	Aufgabe der GUI	96
7.4.3.3	Arbeitsweise der Command-Klassen	97
7.4.3.4	Undo und redo	97
7.4.4	Undo und redo grundsätzlich betrachtet	97
7.5	Zusammenfassung	98
8	Strategy Pattern	99
8.1	Ein erster Ansatz	99
8.2	Strategy in Aktion – Sortieralgorithmen	101
8.2.1	Das gemeinsame Interface	101

8.2.2	Prinzip des Selection Sort	102
8.2.3	Prinzip des Merge Sort	103
8.2.4	Prinzip des Quick Sort	103
8.2.5	Der Kontext	104
8.2.6	Bewertung des Ansatzes und Variationen davon	105
8.3	Das Strategy Pattern in der Praxis	106
8.4	Abgrenzung zu anderen Mustern	107
8.5	Zusammenfassung	108
9	Iterator Pattern	109
9.1	Zwei Möglichkeiten, Daten zu speichern	109
9.1.1	Daten in einem Array speichern	109
9.1.2	Daten in einer Kette speichern	111
9.2	Aufgabe eines Iterators	113
9.3	Das Interface „Iterator“ in Java	114
9.3.1	Der Iterator der Klasse „MyArray“	114
9.3.1.1	Test des Iterators	115
9.3.1.2	Kritik am Iterator	115
9.3.2	Der Iterator der Klasse „MyList“	115
9.3.2.1	Test des Iterators	116
9.3.2.2	Nutzen des Iterators	117
9.4	Das Interface „Iterable“	117
9.5	Zusammenfassung	118
10	Objektorientierte Entwurfsprinzipien	121
10.1	Gegen Schnittstellen programmieren	121
10.2	Single Responsibility Principle (SRP)	123
10.3	Inheritance may be evil	124
10.4	Open/Closed Principle (OCP)	125
10.5	Prinzip des rechten Augenmaßes	126
11	Abstract Factory Pattern	127
11.1	Gärten anlegen	127
11.1.1	Der erste Ansatz	127
11.1.2	Der zweite Ansatz – Vererbung	129
11.1.3	Der dritte Ansatz – die abstrakte Fabrik	129
11.1.4	Vorteil der abstrakten Fabrik	131
11.1.5	Einen neuen Garten definieren	132
11.2	Diskussion des Patterns und Praxis	133
11.3	Gespenster jagen	134
11.3.1	Die erste Version	134
11.3.1.1	Der Spieler	134
11.3.1.2	Die vier Himmelsrichtungen	135

11.3.1.3	Die Bauteile des Hauses	136
11.3.1.4	Die Klasse „Haus“ steuert das Spiel	138
11.3.2	Die zweite Version des Projekts	140
11.3.3	Version 3 – Einführung einer weiteren Fabrik	142
11.3.3.1	Erweiterung in dieser Programmversion	142
11.3.3.2	Die neue Fabrik „TuerMitZauberspruchFabrik“	142
11.3.3.3	Das neue Bauteil „TuerMitZauberspruch“	143
11.3.4	Version 4 – das Geisterhaus	144
11.4	Zusammenfassung	145
12	Factory Method Pattern	147
12.1	Ein erstes Beispiel	147
12.2	Variationen des Beispiels	149
12.3	Praktische Anwendung des Patterns	150
12.3.1	Rückgriff auf das Iterator Pattern	150
12.3.2	Bei der Abstract Factory	151
12.4	Ein größeres Beispiel – ein Framework	152
12.4.1	Das Projekt, das erstellt werden wird	152
12.4.2	Die Schnittstellen für Einträge und deren Editoren	153
12.4.3	Die Klasse „Kontakt“ als Beispiel für einen Eintrag	154
12.4.4	Die Klasse „FactoryMethod“ als Client	155
12.5	Unterschied zur Abstract Factory	156
12.6	Zusammenfassung	157
13	Prototype Pattern	159
13.1	Objekte klonen	159
13.1.1	Kritik an der Realisierung	160
13.1.1.1	Das Interface „Cloneable“	160
13.1.1.2	Das Problem gleicher Referenzen	161
13.1.1.3	Was passiert beim Klonen?	162
13.1.2	In Vererbungshierarchien klonen	163
13.2	Ein größeres Projekt	166
13.2.1	Besprechung der ersten Version	167
13.2.2	Die zweite Version – deep Copy	169
13.2.3	Eigene Prototypen definieren	170
13.3	Zusammenfassung	171
14	Composite Pattern	173
14.1	Prinzip von Composite	173
14.2	Umsetzung 1: Sicherheit	174
14.3	Umsetzung 2: Transparenz	177
14.4	Betrachtung der beiden Ansätze	179
14.5	Einen Schritt weitergehen	181

14.5.1	Einen Cache anlegen	181
14.5.2	Die Elternkomponenten referenzieren	183
14.5.3	Knoten verschieben	186
14.6	Realisierung eines TreeModel	186
14.6.1	Die Methoden der Schnittstelle „TreeModel“	187
14.6.2	Knoten rendern und editieren	189
14.6.2.1	Knoten rendern	189
14.6.2.2	Knoten editieren	192
14.7	Zusammenfassung	195
15	Builder Pattern	197
15.1	Ein Objekt erzeugt andere Objekte	197
15.1.1	Umsetzung 1: Telescoping Constructor Pattern	197
15.1.2	Umsetzung 2: JavaBeans Pattern	199
15.1.3	Umsetzung 3: Builder Pattern	200
15.2	Ein komplexerer Konstruktionsprozess	202
15.2.1	XML-Dateien in ein TreeModel konvertieren	203
15.2.2	XML-Dateien als HTML darstellen	206
15.3	Zusammenfassung	209
16	Visitor Pattern	211
16.1	Ein einfaches Beispiel	211
16.1.1	Das Aggregat	211
16.1.2	Der Visitor	213
16.1.3	Der Client	214
16.1.4	Ein weiterer Visitor	215
16.1.5	Kritik am Projekt	215
16.2	Zusammenfassung	216
17	Memento Pattern	217
17.1	Aufgabe des Memento Patterns	217
17.2	Eine mögliche Realisierung	218
17.3	Ein größeres Projekt – der GrafikEditor	221
17.4	Zusammenfassung	222
18	Flyweight Pattern	225
18.1	Aufgabe des Patterns	225
18.2	Die Realisierung	226
18.3	Ein komplexeres Projekt: Pizza	228
18.3.1	Der erste Ansatz	228
18.3.2	Intrinsischer und extrinsischer Zustand	229
18.4	Flyweight in der Praxis	231
18.5	Zusammenfassung	233

19 Facade Pattern	235
19.1 Ein Beispiel außerhalb der IT	235
19.2 Die Fassade in einem Java-Beispiel	236
19.2.1 Einführung einer Fassade	238
19.2.2 Der Begriff „System“ genauer betrachtet	239
19.3 Die Fassade in der Klassenbibliothek	240
19.4 Das „Gesetz von Demeter“	241
19.5 Zusammenfassung	242
20 Adapter Pattern	243
20.1 Ein einleitendes Beispiel	243
20.1.1 Ein klassenbasierter Entwurf	243
20.1.2 Ein objektbasierter Entwurf	245
20.1.3 Kritik am Adapter Pattern	246
20.2 Ein Praxisbeispiel und falsche Namen	247
20.2.1 Ein Adapter im patternCoder	247
20.2.1.1 Objektbasierter Ansatz des Adapters	247
20.2.1.2 Klassenbasierter Ansatz des Adapters	248
20.2.2 Zum Schluss noch ein Etikettenschwindel	249
20.3 Zusammenfassung	250
21 Proxy Pattern	251
21.1 Virtual Proxy	251
21.2 Security Proxy	252
21.3 Smart Reference	253
21.3.1 Die Grundversion „Proxy_1“	253
21.3.2 Einführung eines Proxys im Projekt „Proxy_2“	255
21.3.3 Einen zweiten Proxy einführen	257
21.3.4 Dynamic Proxy	258
21.3.4.1 Das Reflection API	259
21.3.4.2 Der InvocationHandler	260
21.3.4.3 Die Proxy-Klasse	261
21.4 Remote Proxy	262
21.4.1 Aufbau von RMI grundsätzlich	262
21.4.2 Der RMI-Server	263
21.4.2.1 Die Schnittstelle „PIF“	263
21.4.2.2 Die Serverklasse „PiImpl“	264
21.4.2.3 Die Klasse „AppStart“ startet den Server	264
21.4.3 Der RMI-Client	265
21.4.4 Das Projekt zum Laufen bringen	267
21.5 Zusammenfassung	267

22	Decorator Pattern	269
22.1	Autos bauen – ein erstes Beispiel	269
22.1.1	Ein Attribut für jede Sonderausstattung	269
22.1.2	Mit Vererbung erweitern	270
22.1.3	Dekorieren nach dem Matroschka-Prinzip	270
22.1.3.1	Definieren der Grundmodelle	271
22.1.3.2	Definieren der Sonderausstattungen	271
22.1.3.3	Der Client steckt die Komponenten zusammen	272
22.1.4	Praxisbeispiele	273
22.1.4.1	Die Klasse „ScrollPane“	273
22.1.4.2	Streams in Java	274
22.2	Zusammenfassung	277
23	Bridge Pattern	279
23.1	Zwei Definitionen	279
23.1.1	Was ist eine Abstraktion?	279
23.1.2	Was ist eine Implementierung?	280
23.1.3	Ein Problem beginnt zu reifen	282
23.2	Das Bridge Pattern im Einsatz	283
23.2.1	Bridge Pattern – erster Schritt	284
23.2.2	Bridge Pattern – zweiter Schritt	285
23.2.2.1	Die Abstraktion erweitern	285
23.2.2.2	Die Implementierung erweitern	286
23.3	Diskussion des Bridge Patterns	288
23.3.1	Die Bridge in freier Wildbahn	288
23.3.2	Abgrenzung zu anderen Patterns	289
23.4	Zusammenfassung	289
24	Interpreter Pattern	291
24.1	Die Aufgabe in diesem Kapitel	291
24.2	Der Scanner	292
24.2.1	Die definierten Symbole	293
24.2.2	Der Scanner wandelt Strings in Symbole um	294
24.3	Der Parser	296
24.3.1	Abstrakte Syntaxbäume	296
24.3.2	Expressions für den Parser	297
24.3.3	Strichrechnung parsen	299
24.3.4	Punktrechnung parsen	301
24.3.5	Klammern berücksichtigen	303
24.4	Diskussion des Interpreter Patterns	304
24.5	Zusammenfassung	305
	Index	307

Mae govannen!¹

Design Patterns sind ein spannendes Thema!

Und Sie werden Spaß haben, wenn Sie dieses Buch durcharbeiten.

Wenn von Design Patterns die Rede ist, denken Sie sicher zuerst an das Buch von Erich Gamma und seinen drei Kollegen, die zusammen „Viererbande“ („Gang of Four“, „GoF“) genannt werden. Sie beschreiben 23 Design Patterns so grundlegend, dass ihr Buch seit vielen Jahren unverändert gedruckt wird.

Mein Buch hat die Aufgabe, Ihnen die GoF-Muster vorzustellen und Ihnen einen leichten Zugang zu ihnen zu ermöglichen. Dabei setze ich Java als Sprache für die Beispielcodes ein. Mein Ziel ist es jedoch, Ihnen nicht nur die Patterns zu zeigen; ich beabsichtige, Ihnen die dahinterstehenden Strukturen und die Wege zur Lösung zu zeigen. Ich gehe davon aus, dass Sie mit dem „Warum“ mehr anfangen können als mit dem „Wie“.

Mein Buch hat noch eine zweite Aufgabe – ich möchte ein paar Aspekte der Java-Programmierung beleuchten, die in vielen Java-Einführungen zu kurz kommen. Sie werden zum Beispiel einen eigenen LayoutManager schreiben und lernen, mit TableModel und TreeModel umzugehen. Ich zeige Ihnen ein paar Tricks aus meiner eigenen Java-Praxis; und schließlich werden Sie an vielen Stellen sehen, wie die Klassen der Klassenbibliothek arbeiten. Ich setze voraus, dass Sie Java bereits kennen und anwenden können. Sie müssen kein Java-Profi sein, um von diesem Buch zu profitieren. Sie sollten aber stabile Grundlagenkenntnisse haben und die vier Säulen der Objektorientierung (Abstraktion, Kapselung, Vererbung und Polymorphie) kennen. Außerdem sollten Sie mit den Sprachgrundlagen vertraut sein und zum Beispiel wissen, was eine innere Klasse ist. Wenn Sie die Grundzüge der GUI-Programmierung mit Swing kennen, ist das ganz optimal.

Ich stelle Ihnen in jedem Kapitel ein Muster der GoF vor. Da die Kapitel in sich abgeschlossen sind, können Sie gezielt das Kapitel lesen, das Sie am meisten interessiert. Dennoch greife ich in einigen Kapiteln auf Themen zurück, die weiter vorne besprochen wurden. Wenn Sie also mit Design Patterns bisher nichts oder nur wenig zu tun hatten, fangen Sie am besten vorne an und arbeiten sich nach hinten durch. Die Reihenfolge, in der ich Ihnen die Muster vorstelle, entspricht nicht der von Gamma gewählten Reihenfolge.

Um Muster aus der abstrakten Welt der Informatik herauszulösen, biete ich Ihnen als Einleitung an vielen Stellen Beispiele aus der realen Welt an, in denen ein Muster – ein Vorge-

¹ Frei übersetzt: „Herzlich willkommen!“ (Gruß der Elben)

hen – eingesetzt wird. Erst dann wird das Muster aus Sicht der IT besprochen. Ich möchte Ihnen damit die Sicherheit geben, dass Design Patterns kein Spezialwissen von irgendwelchen Gurus sind, sondern bewährte alltägliche Praxis beschreiben. Anschließend stelle ich Ihnen ein Problem aus der IT vor und biete dazu eine Lösung an. Diese Lösung wird funktionieren, aber sie ist bei weitem noch nicht perfekt. Also werde ich eine zweite Lösung entwickeln, die die erste verbessert. Sie werden also in fast jedem Kapitel mehr als ein Beispielprojekt finden. Wenn ein Pattern anhand zweier unterschiedlicher Projekte gezeigt wird, ist das erste Projekt recht einfach; es hat den Sinn, das Pattern zu beschreiben. Das zweite Projekt ist dann oft etwas aufwendiger und komplizierter; es zeigt, wie ein Pattern in einem größeren Kontext angewandt werden kann. Die meisten Quelltexte sind im Buch nur gekürzt abgedruckt. Sie finden die vollständigen Projekte unter

www.patternsBuch.de.

Die Codes habe ich mit NetBeans in der Version 8 auf Java 7 erstellt; Sie können die Quelltexte aber problemlos mit einer IDE Ihrer Wahl bearbeiten. Bitte öffnen Sie jedes Projekt und analysieren Sie es. Es ist sehr wichtig, dass Sie sich die Zeit nehmen, jeden einzelnen Schritt und jedes Projekt nachzuvollziehen. Das Buch ist nicht dafür gedacht, „konsumiert“ zu werden; es soll Sie vielmehr zu eigener Recherche motivieren.

Die GoF setzt in ihrem Buch die Zweckbeschreibung eines Patterns an den Anfang eines Kapitels. Da die Zweckbeschreibungen die zugehörigen Patterns auf einem sehr hohen abstrakten Niveau beschreiben, sind sie wenig hilfreich, wenn man das erste Mal mit einem Pattern zu tun hat. Daher gehe ich den umgekehrten Weg und erkläre erst die Grundzüge des Patterns und zitiere danach seine Zweckbeschreibung. Ich halte diese Reihenfolge auch deswegen für didaktisch sinnvoll, weil Sie anhand der Zweckbeschreibung kontrollieren können, ob Sie die Zielrichtung des gerade besprochenen Patterns verstanden haben. Sie sollten sich auch deshalb unbedingt mit den Zweckbeschreibungen beschäftigen, weil sie so etwas wie „offizielle“ Definitionen sind; Sie müssen sie einfach kennen.

Ich beteilige mich an dem Softwareprojekt „patternCoder“. Sie finden dieses Projekt auf der Homepage www.patternCoder.org. Auf www.patternCoder.de arbeite ich an der deutschen Übersetzung. Sie finden dort weiterführende Links zum Thema Design Patterns und deutschsprachige Musterbeschreibungen.

Wenn Sie ein Kapitel durchgearbeitet haben, sollten Sie das Muster unbedingt mit dem patternCoder umsetzen. Das Beispiel im patternCoder wird keine neuen Erkenntnisse liefern, sondern das besprochene Pattern wiederholen. Sie können damit das Gelernte aus einem anderen Blickwinkel und in einem anderen Kontext noch einmal betrachten. Der patternCoder kann Ihnen auch in der Praxis immer wieder als „Spickzettel“ dienen.



In allen Patterns-Büchern finden Sie eine Vielzahl von Klassendiagrammen. Ich habe mich bewusst entschieden, sehr sparsam mit Diagrammen umzugehen. Die Diagramme vieler Patterns gleichen sich zu sehr, um einen Erkenntnisgewinn daraus zu ziehen. Sie profitieren sehr viel mehr, wenn Sie die Beispiele im patternCoder nachvollziehen.

Die Klassendiagramme, die Sie im Buch finden, habe ich mit www.yUML.me erstellt. Auf die Angabe von Kardinalitäten habe ich verzichtet.

Ich möchte Sie im Buch direkt ansprechen. Dabei verwende ich ausschließlich die Anrede, die als „männliche“ Anrede gilt. Ich möchte den Text nicht unnötig aufblähen, indem ich permanent schreibe „Sie als Programmierer oder Programmiererin“. Ich halte es für unangemessen und unseriös, eine Klasse mit `class StudentIn` zu bezeichnen. Da die Bezeichner von Klassen nach der Konvention Nomen im Singular sein müssen, kann ich auch nicht in irgendwelche Behelfskonstrukte flüchten. Irgendwann muss ich `class Studierender` oder `class Studierende` coden. Noch unpassender finde ich es, aus Gründen der „Geschlechtergerechtigkeit“ grundsätzlich die weibliche Form zu verwenden, was konsequenterweise zu `class Klientin` und `class Serverin` führen würde. Daher meine Bitte: Sofern Sie eine Leserin sind, fühlen Sie sich nicht übergangen oder gering geschätzt.

Ich bedanke mich

- bei meiner Freundin **Martina Guth**. Ich habe zwei Jahre an diesem Buch gearbeitet und dabei sehr viel Energie und noch mehr Freizeit investiert. Martina hat mir in dieser Zeit sehr viel Kraft gegeben. Ich halte ihr Verständnis, ihre Zuverlässigkeit und ihre Unterstützung nicht für eine Selbstverständlichkeit – danke!
- bei **Benjamin Sigg**, der dieses Buch als Fachgutachter begleitet hat. Ich habe die Zusammenarbeit als ausgesprochen produktiv und motivierend erlebt. Benjamin hat mir immer sehr zeitnah ein umfassendes Feedback zu den Kapiteln gegeben. Darüber hinaus hat er mich mit sehr viel Programmierarbeit unterstützt; der `patternCoder` hätte ohne seine Unterstützung in dieser Form niemals zu einem Abschluss gebracht werden können. Vielen Dank!
- *Bu toil leam taing a thoirt do **Sheumas Paterson** (Ollamh aig Caledonian University Ghlaschu) airson an taic a fhuair mi bhuaithe. 'S iomadh latha a bha sinn ag obair air `pattern code` còmhla agus nach mi a dh'ionnsaich tòrr bho Sheumas chòir. Tha mi uamhasach fhèin taingeil airson a chomhairle is am fiosrachadh prìseil a bha na chuideachadh mòr dhomh agus a bha air leth cudthromach agus luachmhor airson an leabhair agam. Mile Taing agus Herzlichen Dank bhuan!*²

Erwähnen möchte ich schließlich auch meine Arbeitskollegen aus dem Referat Software-Entwicklung des Bundesamtes für Justiz. Wir haben ein ausgezeichnetes Arbeitsklima und das `patterns`Buch war beim Feierabendbier oft ein Thema. Auch von ihnen habe ich viel konstruktives Feedback und Input bekommen.

² Das ist Schottisch-Gälisch – ich bedanke mich hier bei James Paterson, Professor an der Glasgow Caledonian University.

2

Template Method Pattern

In diesem Kapitel werden Sie das Template Method Pattern kennenlernen. Es ist ein Verhaltensmuster und sehr leicht zu verstehen. Das Pattern hilft Ihnen, wenn Sie einen Algorithmus haben, der einerseits nicht geändert werden darf, andererseits aber in Unterklassen teilweise unterschiedlich implementiert werden muss.

■ 2.1 Die Arbeitsweise von Template Method

Nehmen Sie als Beispiel die klassische Reihenfolge von Eingabe, Verarbeitung und Ausgabe. Sie möchten einen String eingeben, diesen wahlweise in Groß- oder Kleinschreibung konvertieren und anschließend auf der Konsole ausgeben.

2.1.1 Der erste Ansatz

Auf den ersten Blick scheint die Aufgabe schnell gelöst zu sein, wenn Sie zwei Klassen definieren: Uppercase und Lowercase. Den Quelltext von Uppercase drucke ich folgend ab. Die Methode `run()` definiert den Algorithmus: eingeben, konvertieren, ausgeben. Der Quellcode der Klasse Lowercase ist identisch. Lediglich der fett markierte Part unterscheidet sich.

Listing 2.1 Klasse „Uppercase“ im Projekt „Template_1“

```
public class Uppercase
{
    public final void run()
    {
        final String EINGABE = textEingeben();
        final String KONVERTIERT = convert(EINGABE);
        drucke(KONVERTIERT);
    }
}
```

```

private final String textEingeben()
{
    final String MESSAGE = "Bitte geben Sie den Text ein: ";
    return JOptionPane.showInputDialog(MESSAGE);
}

private final String convert(String eingabe)
{
    return eingabe.toUpperCase();
}

private final void drucke(String text)
{
    System.out.println(text);
}

public static void main(String[] args)
{
    new Uppercase().run();
}
}

```

Wenn ein Projekt Quellcode hat, der per Copy and Paste übertragen wird, ist das in der Regel ein Hinweis auf ein schlechtes Design. Es bietet sich an, doppelten Code in eine gemeinsame Oberklasse auszulagern – und genau das machen Sie im folgenden Abschnitt.

2.1.2 Der zweite Ansatz

Die abstrakte Oberklasse definiert einerseits den Algorithmus, andererseits die Methoden, die in beiden Klasse identisch sind. Da die Definition des Algorithmus für alle Unterklassen verbindlich sein soll, bietet es sich an, die Methode `run()` `final` zu deklarieren.

Listing 2.2 Abstrakte Oberklasse im Projekt „Template_2“

```

public abstract class Eingabe
{
    public final void run()
    {
        String eingabe = textEingeben();
        String konvertiert = convert(eingabe);
        drucke(konvertiert);
    }

    private final String textEingeben()
    {
        final String MESSAGE = "Bitte geben Sie den Text ein: ";
        return JOptionPane.showInputDialog(MESSAGE);
    }

    protected abstract String convert(String eingabe);

    private final void drucke(String text)

```

```
{
    System.out.println(text);
}
```

Die Unterklasse `UppercaseConverter` und `LowercaseConverter` überschreiben lediglich die abstrakte Methode `convert()`. Exemplarisch wird die Klasse `LowercaseConverter` abgedruckt.

Listing 2.3 Die Subklasse „LowercaseConverter“ im Projekt „Template_2“

```
public class LowercaseConverter extends Eingabe
{
    @Override
    protected String convert(String eingabe)
    {
        return eingabe.toLowerCase();
    }
}
```

Im folgenden Absatz wird das Projekt in Betrieb genommen.

2.1.3 Das Hollywood-Prinzip

Der Client kann nun durch Unterklassenbildung sehr leicht den gewünschten Converter auswählen.

Listing 2.4 Client im Project „Template_2“

```
public class Client
{
    public static void main(String[] args)
    {
        Eingabe eingabe = new LowercaseConverter();
        eingabe.run();
        Eingabe neueEingabe = new UppercaseConverter();
        neueEingabe.run();
    }
}
```

Der Aufruf erfolgt immer von der Oberklasse aus. In der Oberklasse ist der Algorithmus definiert und die Oberklasse ruft die entsprechende Methode in der Unterklasse auf. Dieses Vorgehen nennt die GoF *Hollywood-Prinzip*: „Rufen Sie uns nicht an – wir rufen Sie an!“

2.1.4 Einführen von Hook-Methoden

Sie können das Projekt mit *Hook-Methoden* erweitern. Eine Hook-Methode ist eine Methode, die optional überschrieben werden kann. In der Oberklasse wird ein Standardverhalten – oder gar kein Verhalten – definiert. Die Unterklassen können – müssen aber nicht – dieses

Verhalten übernehmen. Im Projekt „Template_3“ gibt es eine Hook-Methode. Die Methode `speichern()` gibt zurück, ob der eingegebene Text auf der Festplatte gespeichert werden soll. In diesem Fall wird die Methode `saveToDisk()` aufgerufen. Standardmäßig wird `false` zurückgegeben.

Listing 2.5 Optional die Eingabe speichern – Projekt „Template_3“

```
public abstract class Eingabe
{
    public final void run()
    {
        String eingabe = textEingeben();
        String konvertiert = convert(eingabe);
        drucke(konvertiert);
        if ( speichern() )
            saveToDisk();
    }

    // ... gekürzt ...

    protected boolean speichern()
    {
        return false;
    }

    private void saveToDisk()
    {
        System.out.println("Eingabe wurde gespeichert.");
    }
}
```

Wenn die Unterklassen der Meinung sind, dass das Standardverhalten so nicht passt, steht es ihnen frei, dieses zu überschreiben. Der `UppercaseConverter` möchte beispielsweise, dass der Text immer gespeichert wird – er gibt also `true` zurück.

Listing 2.6 Klasse „UppercaseConverter“ im Projekt „Template_3“

```
public class UppercaseConverter extends Eingabe
{
    // ... gekürzt ...

    @Override
    protected boolean speichern()
    {
        return true;
    }
}
```

Der `LowercaseConverter` möchte den Anwender entscheiden lassen, ob der Text gespeichert wird.

Listing 2.7 Klasse „LowercaseConverter“ im Projekt „Template_3“

```
public class LowercaseConverter extends Eingabe
{
    // ... gekürzt ...
}
```



```

@Override
protected boolean speichern()
{
    String frage = "Soll der Text gespeichert werden?";
    int antwort = JOptionPane.showConfirmDialog(null, frage);
    return antwort == JOptionPane.YES_OPTION;
}

```

Hook-Methoden werden auch *Einschub-Methoden* genannt. Sie geben die Möglichkeit, auf den Ablauf des Algorithmus Einfluss zu nehmen.

Das war's dann auch schon – jetzt kennen Sie das Template Method Pattern. Trotz – oder gerade wegen – seiner trivial erscheinenden Formulierung ist das Template Method Pattern ausgesprochen wichtig für die Entwicklung von Frameworks wie der Java-Klassenbibliothek. Sie entwickeln ein Grundgerüst, den Algorithmus, und darauf bauen Sie Ihre Erweiterungen auf. Auf www.Sourcemaking.com habe ich Bild 2.1 gefunden, das, wie ich finde, das Prinzip sehr anschaulich zeigt.

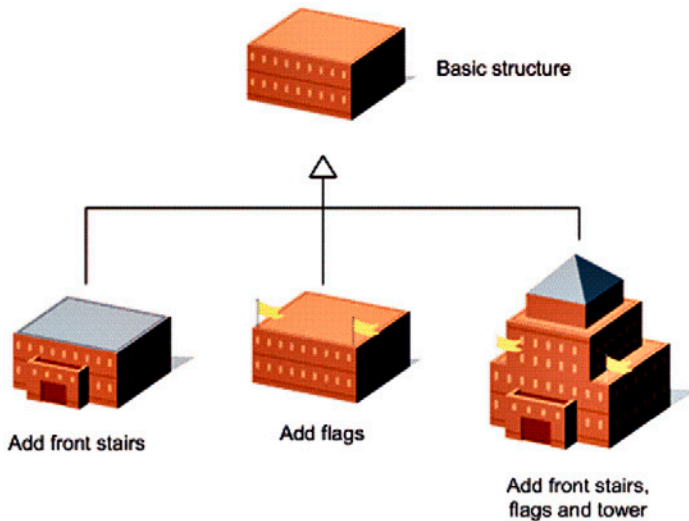


Bild 2.1 Prinzip des Template Method Patterns

■ 2.2 Das Interface „ListModel“

Das Template Method Pattern werden Sie überall dort identifizieren können, wo Sie auf abstrakte Klassen treffen. Ich möchte an dieser Stelle kurz auf den Algorithmus eingehen, den das Interface `ListModel` vorgibt; er beschreibt, was passieren muss und welche Daten vorliegen müssen, damit eine `JList` Daten anzeigen kann. Eine `JList` akzeptiert jedes Objekt als Datenmodell, das vom Typ `ListModel` ist. Das Interface `ListModel` schreibt vier Methoden vor, die für die Anzeige der Daten erforderlich sind. Eine `JList` muss sich als Beobach-

ter, als `ListDataListener`, beim Model registrieren und de-registrieren können; dafür müssen die Methoden `addListDataListener()` und `removeListDataListener()` implementiert sein. Die `JList` muss die Datenbasis außerdem fragen können, wie viele Elemente angezeigt werden sollen; das wird mit der Methode `getSize()` beantwortet. Und schließlich muss die `JList` das Element an einer bestimmten Stelle abfragen können; hierfür gibt es die Methode `getElementAt()`, der Sie den Index des gesuchten Elements übergeben. Unter www.patternsBuch.de finden Sie das Projekt „ListModel_Beispiel“. Dieses Projekt zeigt die Vorgehensweise mit einem sehr simplen Beispiel.

Wenn Sie seinen Quelltext analysieren, werden Sie sehen, dass die Methoden zum Registrieren und De-Registrieren von Listenern so allgemein sind, dass sie vermutlich in allen Situationen verwendet werden können. In der Klassenbibliothek gibt es die Klasse `AbstractListModel`, in der diese beiden Methoden implementiert sind. Sofern Sie eine andere als die Standardimplementierung benötigen, sind Sie frei, diese Methoden zu überschreiben. Methoden, die die Datenbasis beschreiben, also `getSize()` und `getElementAt()`, entziehen sich einer Standardimplementierung. Sie müssen sie selbst definieren.

■ 2.3 Der `patternCoder`

In diesem Abschnitt möchte ich Ihnen den `patternCoder` vorstellen. Der `patternCoder` war früher eine Erweiterung für BlueJ. Wir haben im Laufe der Zeit verschiedene Versionen erstellt, so dass der `patternCoder` mit gleichem Funktionsumfang als Stand-alone-Version, als Erweiterung für BlueJ und als Plug-in für Eclipse genutzt werden konnte. Im Hinblick auf dieses Buch haben wir 2013 angefangen, den `patternCoder` neu zu designen. Die Stand-alone-Version ist fertig, Sie finden sie unter www.patternsBuch.de. Die BlueJ-Version war bei Drucklegung noch nicht fertig erstellt, ist aber in Arbeit. Natürlich wird auch der Funktionsumfang des `patternCoder` weiterentwickelt und ausgebaut. Die neue Version des `patternCoder` ist unter Java 7 und höher lauffähig. Bitte schauen Sie ab und zu auf www.patternCoder.de vorbei; dort hinterlegen wir die jeweils neuen Versionsstände.

Den `patternCoder` installieren

Unter www.patternsBuch.de finden Sie eine zip-Datei, die das ausführbare jar-File und den Ordner `patternFiles` enthält. Entpacken Sie die zip-Datei, so dass das jar-File und der Ordner im gleichen Verzeichnis liegen. Wenn Sie den `patternCoder` starten, sucht er zunächst im gleichen Verzeichnis nach dem Ordner `patternFiles`. Wenn Sie diesen Ordner umbenennen oder an einen anderen Ort verschieben, prüft `patternCoder`, ob in der Registry ein anderer Pfad als Projektverzeichnis vorliegt. Ist das nicht der Fall, sind die meisten Funktionen ausgegraut und Sie müssen zuerst das Projektverzeichnis eingeben. Klicken Sie hierzu **strg + alt + s**. Den Pfad, den Sie eingeben, hinterlegt `patternCoder` in der Registry und greift beim nächsten Start darauf zu. Am einfachsten dürfte es jedoch sein, wenn Sie jar-File und `patternFiles`-Ordner im gleichen Verzeichnis ablegen.

Der Ordner „patternFiles“

Im Ordner *patternFiles* sind die Beschreibungen der Patterns hinterlegt. Jedes Pattern wird durch ein *patternFile*, eine XML-Datei, beschrieben. Im Ordner *behavioural* finden Sie beispielsweise die Datei *Template.patf.xml*. Sie beschreibt das Template Method Pattern. Bitte benennen Sie die Unterordner und die Dateien **nicht** um.

2.3.1 Der patternCoder aus Anwendersicht

Dieser Abschnitt beschreibt, wie der *patternCoder* funktioniert und was Sie von ihm erwarten können.

Die Ansicht „Übersicht“

Wenn Sie den *patternCoder* starten, sehen Sie auf der linken Seite einen Baum, in dem alle Patterns gelistet werden. Auf der rechten Seite lesen Sie eine Zusammenfassung, wie der *patternCoder* bedient wird.

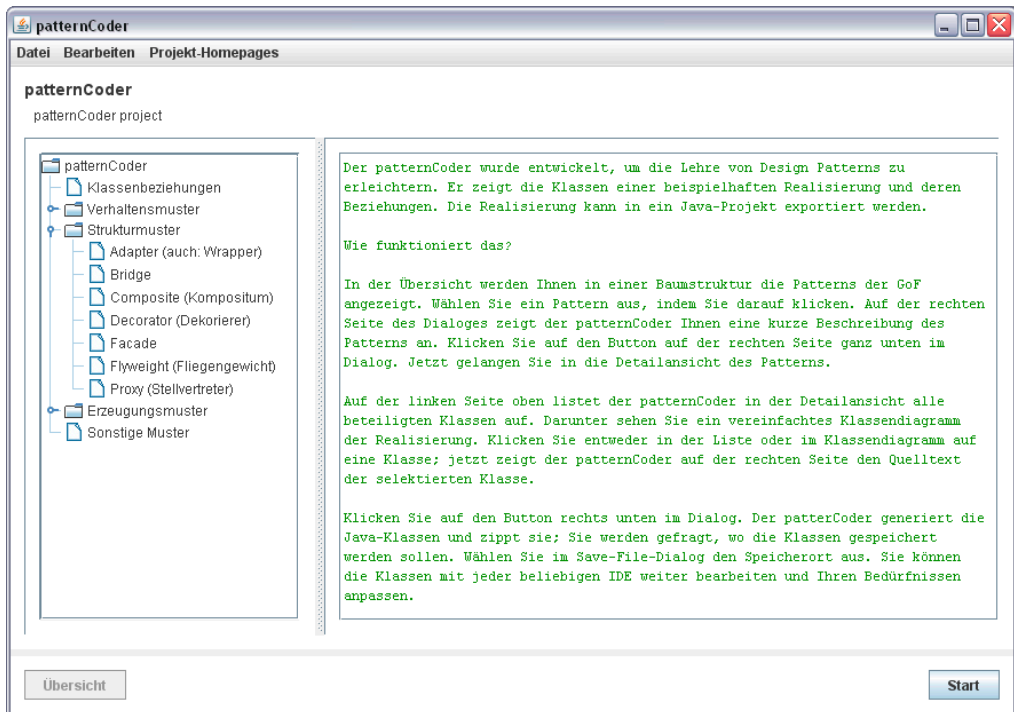


Bild 2.2 Der *patternCoder* nach dem Start

Wählen Sie aus der Kategorie Verhaltensmuster das Template Method Pattern. Klicken Sie einmal kurz auf den Eintrag. Jetzt wird auf der rechten Seite die Zweckbeschreibung des Patterns angezeigt. Klicken Sie nun auf den Button **Start**. Alternativ können Sie doppelt auf den Eintrag klicken, um in die Detailansicht zu wechseln.

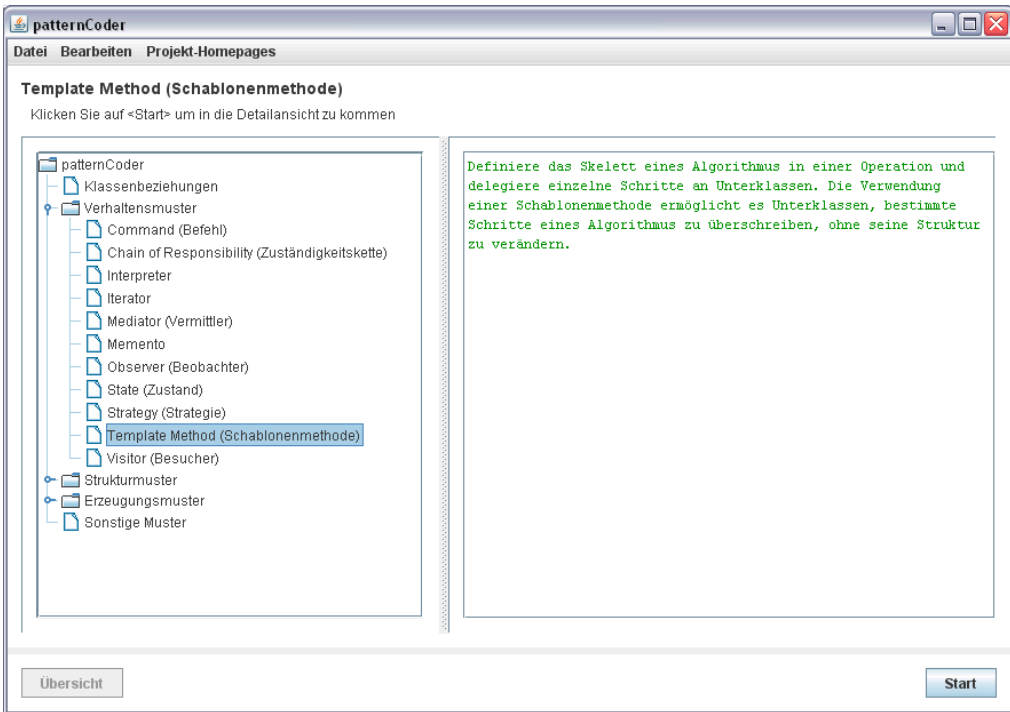


Bild 2.3 Übersichtsansicht des Template Method Patterns

Die meisten Texte sind auf Englisch hinterlegt. Wir werden sie nach und nach übersetzen. Auf www.patternCoder.de finden Sie die neuen patternFiles.

Die Ansicht „Detail“

Wenn Sie in die Detail-Ansicht wechseln, finden Sie im oberen linken Bereich eine Auflistung der am Pattern beteiligten Klassen. Im unteren linken Bereich sehen Sie das Klassendiagramm des hinterlegten Projekts. Auf der rechten Seite ist nach wie vor die Beschreibung des Patterns zu sehen. Wenn Sie auf eine Klasse klicken – entweder im Klassendiagramm oder in der Auflistung oben links – wird rechts der Quelltext der Klasse angezeigt. Die ausgewählte Klasse wird im Klassendiagramm grün markiert.

Oberhalb des Diagramms gibt es den Button **Klassendiagramm speichern**. Wenn Sie darauf klicken, wird das Diagramm in eine png-Datei exportiert; Sie werden von patternCoder aufgefordert, den Speicherort einzugeben. Der Button **Code generieren** ganz unten rechts exportiert die hinterlegten Java-Quelltexte. Sie werden gezippt und patternCoder lässt Sie eingeben, wo die zip-Datei gespeichert werden soll. Sofern Sie es vorziehen, dass öffnende Klammern ans Ende der vorigen Zeile geschrieben werden, entfernen Sie im Menü **Bearbeiten** das Häkchen bei **Öffnende Klammer in neuer Zeile**. Der Button **Übersicht** bringt Sie zurück zur Übersicht.

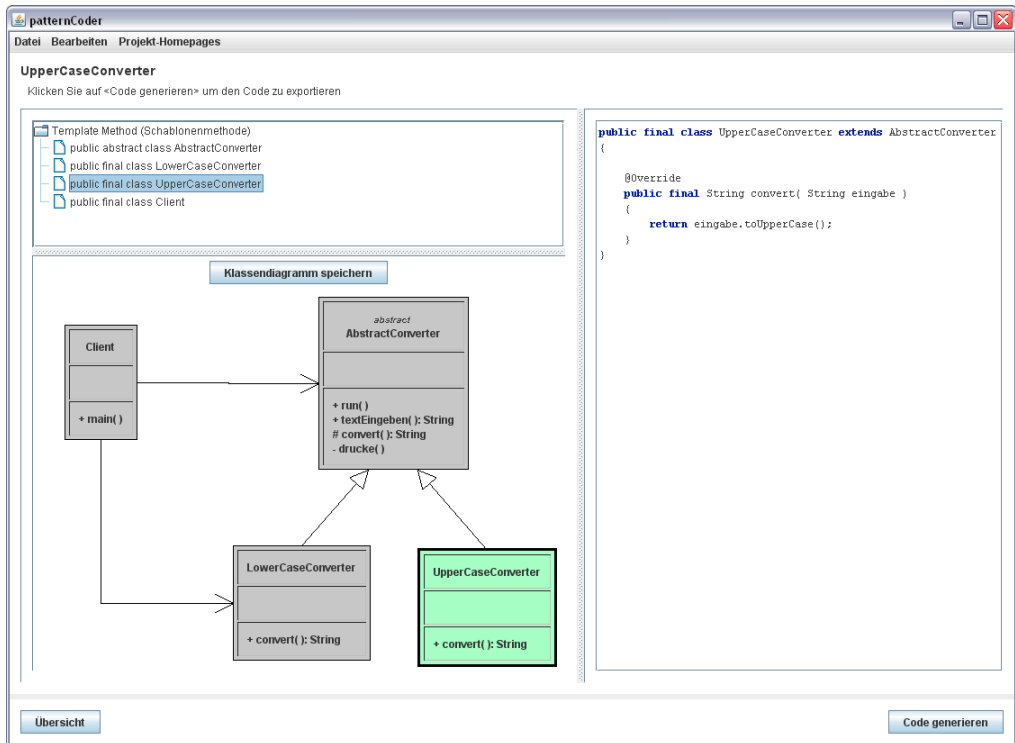


Bild 2.4 Detailansicht des patternCoder

Der patternCoder für Macher

Möchten Sie selbst patternFiles anlegen? Wechseln Sie zurück in die Übersicht. Setzen Sie im Menü **Bearbeiten** einen Haken bei **Pattern bearbeiten**.

Ein Pattern anlegen oder editieren

Wenn Sie nun einen rechten Mausklick auf den JTree setzen, erscheint ein Kontextmenü. Wenn Sie ein Pattern ausgewählt haben, können Sie es editieren oder löschen. Sie können auch ein neues Pattern anlegen. Egal, ob Sie ein bestehendes Pattern editieren oder ein neues anlegen, Sie arbeiten immer im Dialog „Ein Pattern editieren“. Wenn Sie ein Pattern anlegen, wählen Sie die Kategorie, zu der es gehört. Der Dateiname darf nur aus Buchstaben in Klein- und Großschreibung bestehen. Das Schlagwort ist der Text, der später im JTree angezeigt wird. In dem großen Feld „Beschreibung“ hinterlegen Sie die Beschreibung des Patterns. Die Kategorie und der Dateiname können später nicht mehr geändert werden.

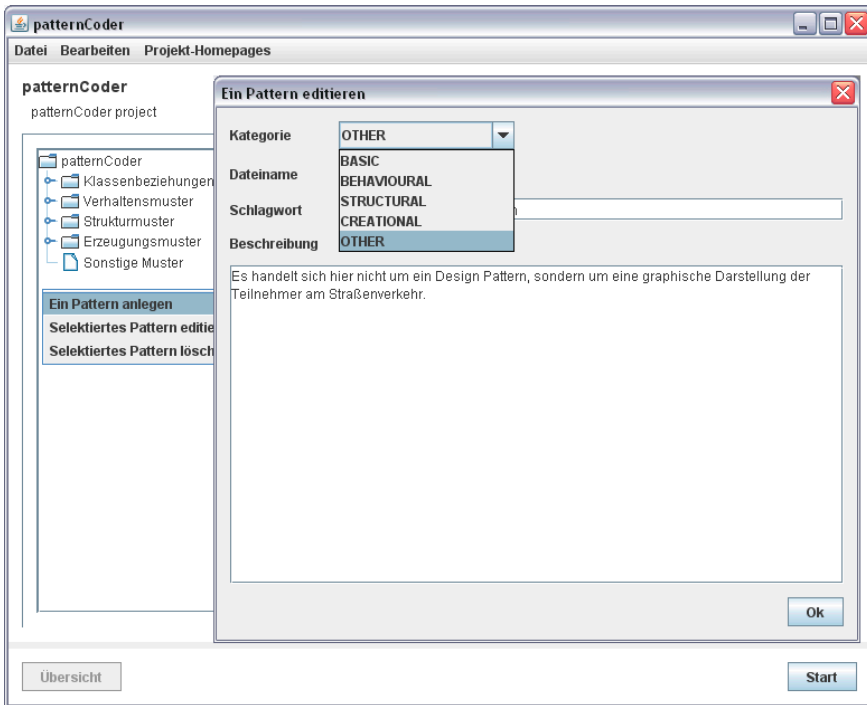


Bild 2.5 Ein Pattern anlegen oder editieren

Klicken Sie auf **Ok**, um die Änderungen zu speichern.

Eine Klasse anlegen

Markieren Sie das gerade angelegte Pattern und klicken Sie auf **Start**, um in die Detailansicht zu wechseln. Im Diagramm gibt es nun sechs ToggleButtons, die später gebraucht werden, um Beziehungen zwischen Klassen darzustellen. Klicken Sie auf die obere Hälfte der linken Seite. Das Kontextmenü bietet Ihnen an, eine Klasse zu erstellen, sie zu editieren oder sie zu löschen. Wählen Sie **Erstelle eine Klasse**, um eine neue Klasse zu erzeugen. Im Dialog „Eine Klasse editieren“, den patternCoder jetzt anzeigt, tragen Sie die Daten der Klasse ein. Sie wählen zunächst die Sichtbarkeit aus – standardmäßig ist `public` vorgegeben. Eine Klasse kann entweder `final` oder `abstract` sein – in den seltensten Fällen beides. Im patternCoder ist eine Prüfung hinterlegt, dass immer nur eine Option ausgewählt werden kann. Der Typ gibt an, ob Sie eine Klasse oder ein Interface anlegen. Geben Sie im nächsten Feld den Namen der Klasse ein. Im Eingabefeld *extends* geben Sie an, von welcher Klasse Ihre Klasse erbt – dazu später mehr. Die Combobox *implements* listet alle Interfaces auf, die Ihre Klasse implementieren soll. Einträge können Sie mit einem eigenen Dialog vornehmen, auch das wird gleich genauer beleuchtet. Die Combobox *imports* speichert alle Import-Anweisungen. Die Buttons **Editiere Attribute** und **Editiere Methoden** brauchen Sie, um Felder und Methoden anzulegen. Ein *UML comment* fügt Ihrer Klasse im Diagramm einen kurzen Kommentar hinzu. Wenn Sie die Checkbox *Erstelle Linien* selektiert lassen,

werden implements- und inherits-Beziehungen automatisch vom patternCoder in das Diagramm eingezeichnet. Diese Einstellung betrifft alle Klassen im Pattern. In das Textfeld *Documentation* tragen Sie die API-Dokumentation der Klasse ein.

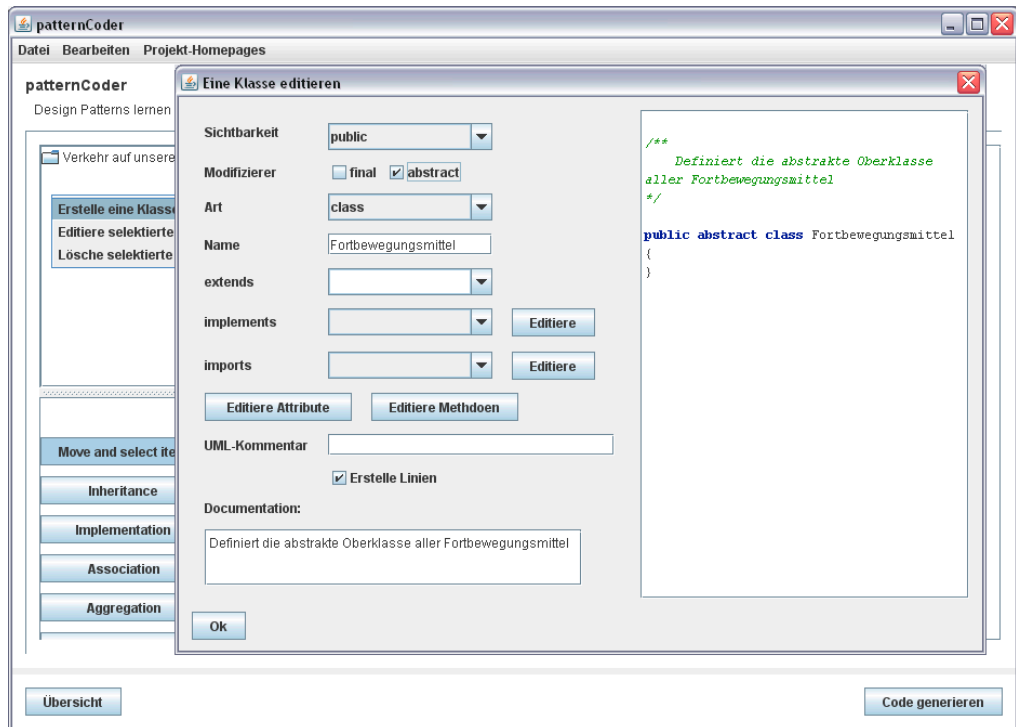


Bild 2.6 Eine Klasse anlegen oder editieren

Ein Datenfeld anlegen

Klicken Sie auf **Editiere Attribute**, um in den Dialog „Ein Attribut editieren“ zu gelangen. Klicken Sie auf **Neues Attribut**, um ein neues Feld zu erzeugen. Es wird in der Liste der Attribute auf der linken Seite des Dialogs angezeigt. Wieder geben Sie die Informationen des Felds an: die Sichtbarkeit, die Modifizierer, den Datentyp, den Bezeichner und die API-Dokumentation. Außerdem können Sie Ihr Datenfeld mit einem beliebigen Wert initialisieren.

Wenn Sie ein Attribut markieren, können Sie es mit **Auswahl löschen** wieder entfernen. Klicken Sie auf **Ok**, um zum Dialog „Eine Klasse editieren“ zurückzukehren.

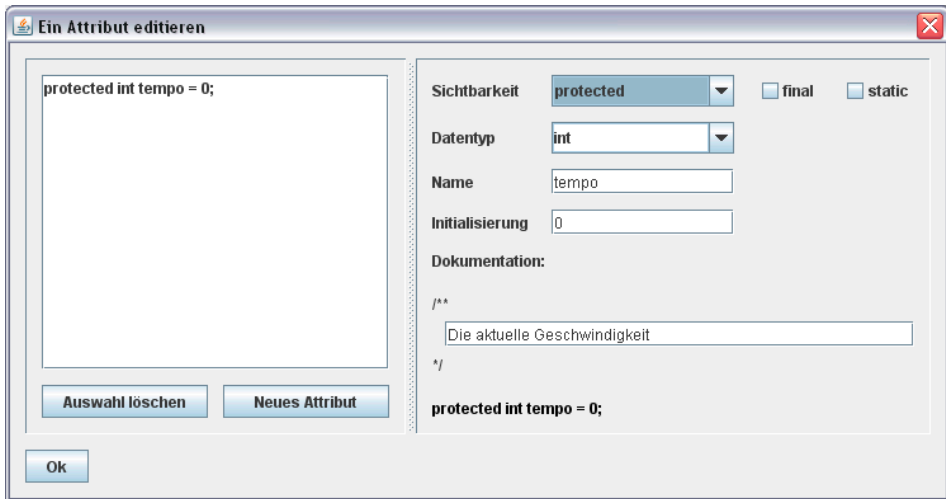


Bild 2.7 Ein Feld anlegen oder editieren

Eine Methode anlegen

Klicken Sie auf den Button **Editiere Methoden**. Der patternCoder öffnet den Dialog „Eine Methode editieren“. Auf der linken Seite werden alle deklarierten Methoden aufgelistet. Erstellen Sie mit **Neue Methode** eine neue Methode; sie wird dann links angezeigt. Sie können nun die Daten der Methode eingeben. Handelt es sich um einen Konstruktor? Überschreibt die Methode eine andere Methode? Welche Sichtbarkeit soll sie haben? Sie geben außerdem die Modifizierer an. Der Rückgabewert ist standardmäßig auf `void` gesetzt. In der Combobox sind verschiedene Vorschläge hinterlegt: `void`, die primitiven Datentypen, der Typ `String` und alle bereits angelegten Klassen im Pattern. Sie können die Einstellungen übernehmen, aber auch einen anderen Datentyp eintragen. Dokumentieren Sie den Rückgabewert und geben Sie der Methode einen Namen. Im Textfeld zwischen `/**` und `*/` tragen Sie die API-Dokumentation der Methode ein. Im unteren Textfeld hinterlegen Sie den Code der Klasse. Code können Sie nur eingeben, wenn die Methode nicht abstrakt ist und nicht in einem Interface deklariert wird.

Klicken Sie auf **Neuer Parameter**, um einen neuen Parameter anzulegen. In der Combobox wählen Sie den Datentyp. Geben Sie einen Namen für den Parameter ein und vergessen Sie die API-Doku nicht. Wenn Sie mehr als einen Parameter anlegen, können Sie die Reihenfolge per Drag and Drop ändern; markieren Sie einen Parameter und verschieben Sie ihn nach oben oder nach unten. Der Methodenkopf oberhalb des Code-Felds wird automatisch aktualisiert. Sowohl Parameter als auch Methoden können gelöscht werden. Klicken Sie dazu auf **Auswahl löschen** unterhalb der jeweiligen Liste.

In diesem Dialog sind verschiedene Abhängigkeiten hinterlegt, um Sie vor Fehleingaben zu schützen. Sie können keine zwei Parameter mit gleichem Namen anlegen. Methoden können überladen werden, müssen aber unterschiedliche Parameterlisten haben.

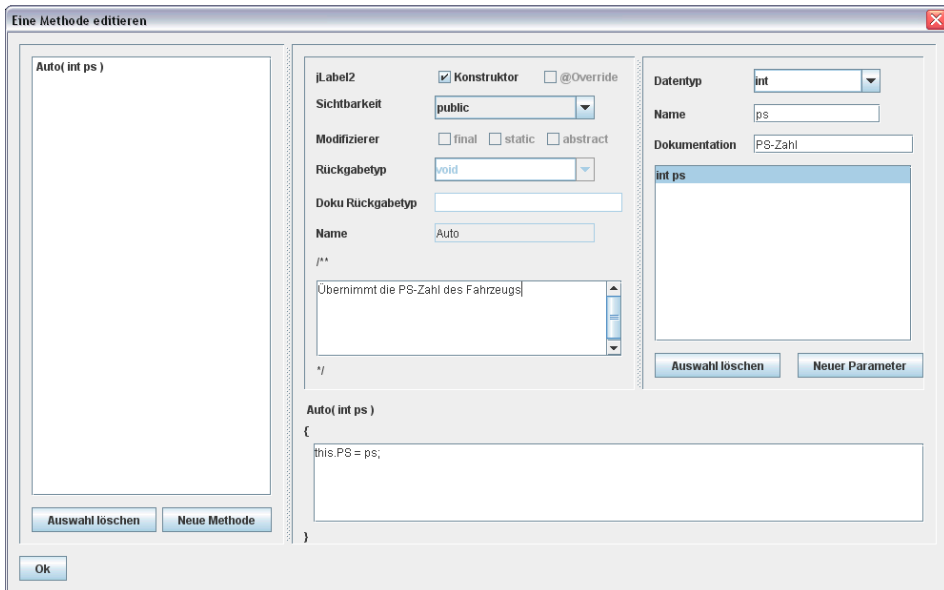


Bild 2.8 Eine Methode anlegen oder editieren

Klicken Sie auf **Ok**, um zum Klassendialog zurückzukehren. Sie werden sehen, dass patternCoder bereits den Code der Klasse erzeugt hat. Klicken Sie erneut auf **OK**, um zur Detailsicht zurückzukehren. Im Diagramm ist die Klasse bereits eingezeichnet.

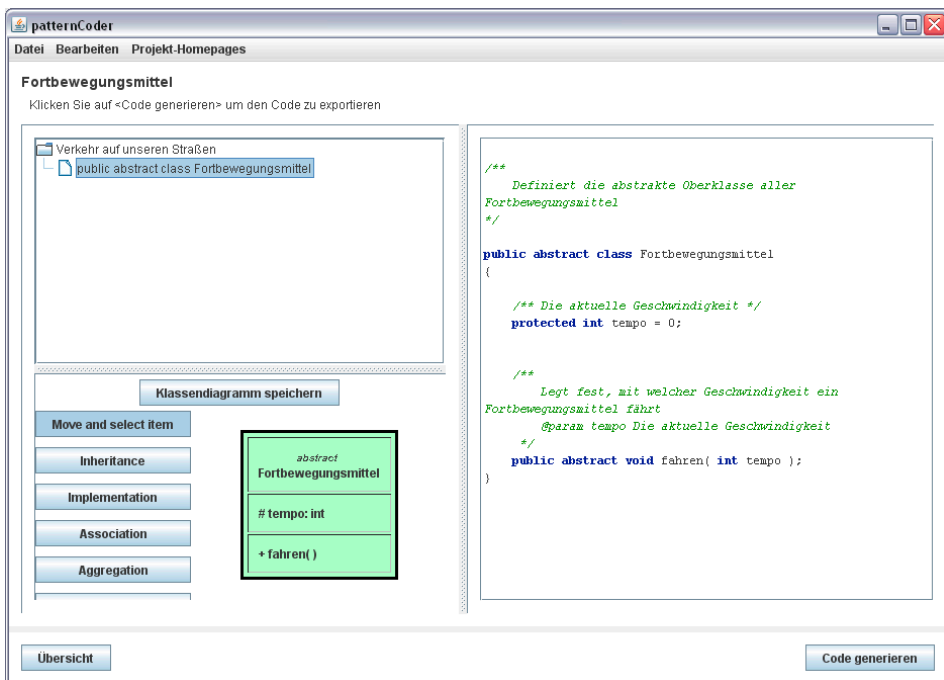


Bild 2.9 Das Pattern in der Detailsicht

Legen Sie eine neue Klasse `Auto` und das Interface `Versicherungspflichtig` an. Das `Auto` soll von `Fortbewegungsmittel` erben. Wenn Sie im Dialog „Eine Klasse editieren“ auf die ComboBox „extends“ klicken, sehen Sie, dass dort bereits alle bestehenden Klassen des Patterns gelistet sind. Natürlich können Sie die Einträge überschreiben. Ein Interface legen Sie an, indem Sie aus der ComboBox „Art“ den entsprechenden Eintrag wählen.

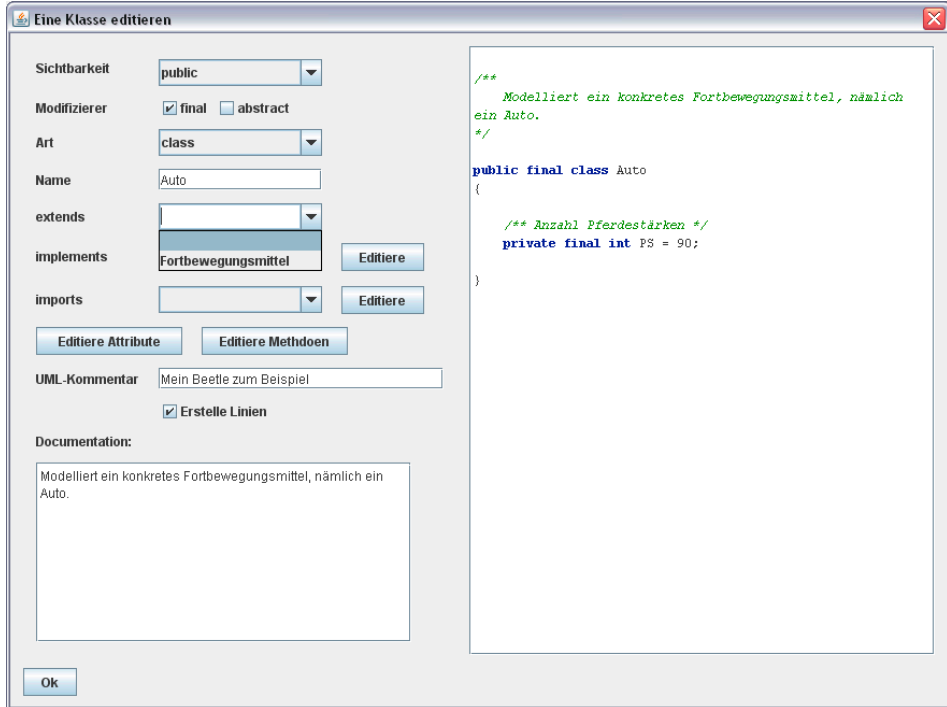


Bild 2.10 Eine andere Klasse erweitern

Ein Interface implementieren

Die Klasse `Auto` soll `Versicherungspflichtig` implementieren. Rufen Sie auf der Klasse `Auto` den Dialog „Eine Klasse editieren“ auf (Bild 2.11). Klicken Sie auf **Editiere** neben der Combobox „implements“. Der patternCoder zeigt einen neuen Dialog an, in dem Sie alle Interfaces aufgelistet finden, die von der Klasse implementiert werden; im Moment ist diese Liste leer. Klicken Sie auf das Pluszeichen, um im InputDialog ein zu implementierendes Interface einzutragen. Tragen Sie `Versicherungspflichtig` ein und schließen Sie den Dialog.

Der Dialog, um Imports einzutragen, ist identisch aufgebaut.

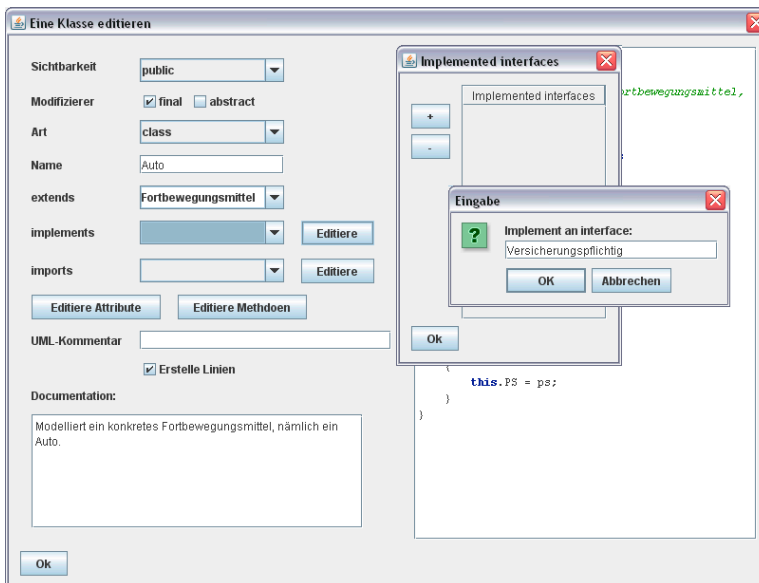


Bild 2.11 Ein zu implementierendes Interface eintragen

Das Klassendiagramm bearbeiten

Schließen Sie den Klassendialog. Der patternCoder hat die Klassen, deren Beziehungen und den UML-Kommentar in das Diagramm eingezeichnet. Vererbung und Realisierung werden zuverlässig erkannt. Mit den ToggleButtons links können Sie selbst Beziehungen zwischen Klassen einzeichnen (Bild 2.12). UML-Kommentare, Verbindungen und Klassen werden Items genannt. Das Kontextmenü ermöglicht es Ihnen, ein selektiertes Item zu löschen.

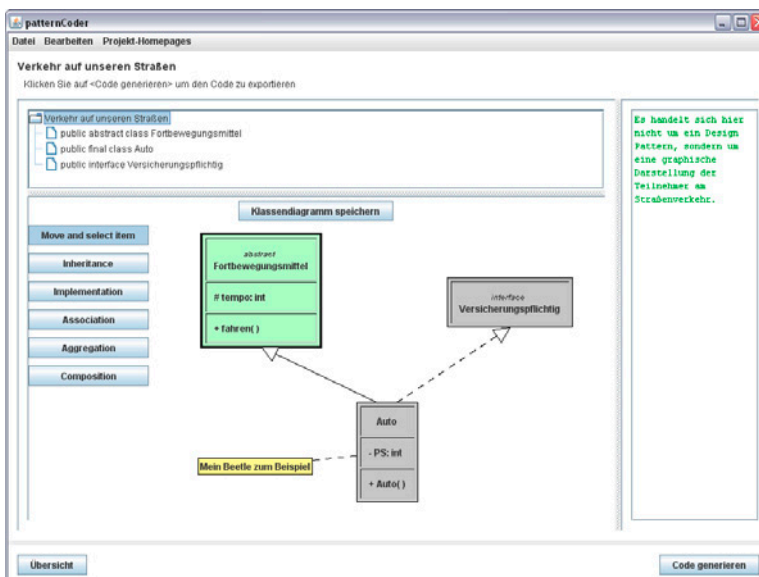


Bild 2.12 Das fertig angelegte Projekt



Um die Klassendiagramme generieren zu können, hat Benjamin Sigg die Bibliothek „GraphPainting“ entwickelt. Den Quelltext finden Sie auf <https://github.com/Benoker>.

■ 2.4 Zusammenfassung

Gehen Sie das Kapitel nochmal stichwortartig durch!

- Sie definieren einen bestimmten Algorithmus.
- Teile des Algorithmus können von der Klasse selbst ausgeführt werden, andere Teile werden den Subklassen vorgeschrieben, die die nicht abstrakten Teile implementieren.
- Den Algorithmus beschreiben Sie in einer finalen Methode der abstrakten Oberklasse.
- Die Unterklasse bzw. die Unterklassen überschreiben die abstrakten Methoden der Oberklasse.
- Optional können Hook-Methoden – Einschubmethoden – überschrieben werden.
- Hook-Methoden erlauben es Ihnen, den Algorithmus teilweise zu variieren.



Zweckbeschreibung:

Die Gang of Four beschreibt den Zweck des Patterns wie folgt:

„Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.“

Index

A

Abstract Factory Pattern

- Abstrakte Fabrik 130
- Abstraktes Produkt 129
- Abstraktionen 133
- Aufgabe 127
- Garten-Projekt 127
- Geisterhaus-Projekt 134
- Konkrete Fabrik 130
- Konkretes Produkt 129
- Konsistenz 131
- Nachteil 133
- Produkt 127
- Produktfamilie 127
- Vorteile 131
- Zweckbeschreibung 146

Abstrakter Syntaxbaum 296

Adapter Pattern

- Abgrenzung zur Fassade 246
- Adapter ist nicht wiederverwendbar 246
- Aufgabe 243
- Diskussion 246
- Gültigkeit für Unterklassen 246
- Klassenbasierter Ansatz 243, 248
- Lose Kopplung 246
- Objektbasierter Ansatz 245, 247
- Zweckbeschreibung 250

Antipattern 15

- Telescoping Constructor Pattern 199

B

Benjamin Sigg 34

Bridge Pattern 279

- Abgrenzung

- Adapter Pattern 289
 - Decorator Pattern 289
 - Facade Pattern 289
 - Proxy Pattern 289
 - Strategy Pattern 289
- Abstraktion 279
- reduzieren von Realweltausschnitten 279
 - weiterentwickeln 282, 285
- Aufgabe 283
- Implementierung 280
- weiterentwickeln 282, 286
- Praxisbeispiele 288
- Realisierung 284
- Trennung von Abstraktion und Implementierung 283
- Zweckbeschreibung 290

Builder Pattern

- Aufgabe 197
- Builder 1 – XML in TreeModel konvertieren 203
- Builder 2 – XML in HTML konvertieren 206
- Einfaches Beispiel 200
- JavaBeans Pattern 199
- Joshua Bloch 197
- Komplexes Beispiel 202
- Telescoping Constructor Pattern 197
- Zweckbeschreibung 209

C

Chain of Responsibility

- Aufgabe 61
- Beispiel aus der Klassenbibliothek 67

- Beispiel Lieferkette 62
- Beispiel Lieferkette - Abwandlung des Projektes 64
- Beispiel Lieferkette - der Client 64
- Beispiel Lieferkette - die Lebensmittel 62
- Beispiel Lieferkette - die Verkäufer 62
- Zweckbeschreibung 68

Christopher Alexander 2

Command Pattern

- Befehle wiederverwenden 91
- Befehl kapseln 86
- Beispiel GUI-Programmierung 96
- Beispiel Reisebüro 83
- Invoker 98
- Praxisbeispiel Eventlistener 90
- Praxisbeispiel Invoker 88
- Praxisbeispiel Nebenläufigkeit 89
- Praxisbeispiel Receiver 88
- Prinzip 83
- Receiver 98
- undo - einfaches Beispiel 93
- undo und redo 95
- Zweckbeschreibung 98

Composite Pattern 173

- Blatt 174
- Composite 174
- Einen Cache anlegen 181
- Knoten 174
- Knoten verschieben 186
- Kompositum 174
- Leaf 174
- Prinzip 173
- Vaterknoten referenzieren 183
- Wurzel 174
- Zweckbeschreibung 195
- Zwei Ansätze der Realisierung 174
 - Kritik an den Ansätzen 179
 - Prio Sicherheit 174
 - Praxisbeispiel AWT 179
 - Prio Transparenz 177
 - Praxisbeispiel Swing 179

D

Decorator Pattern

- Abgrenzung
 - Proxy Pattern 273
- Ändern der Schnittstelle 277

- Aufgabe 269
- Matroschka-Prinzip 270
- Praxisbeispiele 273
 - Klasse JScrollPane 273
 - Streams 274
 - Streams, die keine Decorator sind 276
- Realisierung 269
- Zweckbeschreibung 275, 278

E

Entwurfsmuster

- Beschreibung eines Musters 4
 - Definition 1
 - Definition Schnittstelle 37
 - Gang of Four 2
 - Kategorien 3, 156
- Entwurfsprinzipien 121
- Gegen Schnittstellen programmieren 37, 121, 245
 - Kapsle, was variiert 8, 124
 - Komposition vor Vererbung 124, 129
 - Law of Demeter 241, 246
 - Liskov Substitution Principle 125, 165, 246
 - Prinzip des rechten Augenmaßes 126
 - Probleme aus Vererbung 6
 - Single Responsibility Principle 123, 128
 - Zoo-Beispiel 6

Enum Pattern

- Ausgangssituation 69
- Klassenbibliothek 71
- Realisierung 71
- Typsicherheit 71
- Zustand als Objekt 70

Erich Gamma 2

F

Facade Pattern

- Abgrenzung Adapter Pattern 246, 249
- Aufgabe 235
- Aufgabe - Zusammenfassung 235
- Beispiel außerhalb der IT 235
- Java-Beispiel 236
- Klassenbibliothek 240
- Law of Demeter 241
- Reise-Beispiel 236
- Subsystem 239

- System 239
- Zweckbeschreibung 242

Factory Method

- Begriffe 148
- Fabrikmethode 147
- Praxisbeispiel 1
 - Iterator 150
- Praxisbeispiele 150
- Zweckbeschreibung 157

Flyweight Pattern

- Aufgabe 225
- Extrinsischer Zustand 229
- Intrinsischer Zustand 229
- Operationen des Flyweight 230
- Zweckbeschreibung 233

G

- Gang of Four 2

I

Interpreter Pattern

- Abstrakter Syntaxbaum 296, 301
- Beispiel: Rechner 291
- Besprechung 304
- Composite Pattern 304
- NichtTerminalExpression 298, 304
- Parser 296
 - Expressions 297
 - Klammern parsen 303
 - PlusExpression 298
 - Punktrechnung parsen 301
 - Strichrechnung parsen 299
 - Vorzeichen parsen 302
- Scanner 292
 - Konversion eines Strings 294
 - Symbole 293
- TerminalExpression 298, 304
- Zweckbeschreibung 305

Iterator Pattern

- Anwendung eines Iterators 115
- Aufgabe 109
- Die ArrayList nachbilden 109
- Die LinkedList nachbilden 111
- Interface "Iterable" 117
- Interface "Iterator" 114
- Iterator der Klasse "MyList" 115

- Iterator einführen 113
- Zweckbeschreibung 119

J

JList/ListModel

- addListDataListener() 24
- getElementAt() 24
- getSize() 24
- Interface ListModel 23
- Klasse AbstractListModel 24, 91
- ListDataListener 24
- removeListDataListener 24

John Vlissides 2

Joshua Bloch 197

JTree/TreeModel 186

- addTreeModelListener() 187
- ChangeEvent 194
- getChild() 188
- getChildCount() 188
- getIndexOfChild 188
- getRoot() 187
- Interface TreeCellEditor
 - cancelCellEditing() 194
 - CellEditorListener 193, 194
 - getCellEditorValue() 193
 - getTreeCellEditorComponent() 192
 - shouldSelectCell 194
 - stopCellEditing() 193, 194
- Interface TreeCellEditor 192
 - isCellEditable 193
- Interface TreeCellRenderer
 - getTreeCellRendererComponent() 190
- Interface TreeCellRenderer 189
- isLeaf() 187
- MVC 194
- removeTreeModelListener() 187
- TreePath 194
 - getLastPathComponent() 194
 - valueForPathChanged() 189, 194

K

Kategorien

- Erzeugungsmuster 3
- Klassenbasierte Muster 156
- Objektbasierte Muster 156
- Strukturmuster 3

- Verhaltensmuster 3
- Keine Regel ohne Ausnahme 242
- Klassendiagramm XVI
- Kovariante Rückgabetypen 165

L

LayoutManager 106

M

- Matthias Kleine 241
- Mediator Pattern
 - Aufgabe 51
 - Beispiel GUI-Programmierung 56
 - Beispiel Weinhandel 52
 - Kritik 59
 - Vergleich Observer Pattern 51
 - Zweckbeschreibung 59
- Memento Pattern
 - Aufgabe 217
 - Caretaker 218
 - Kapselung 217
 - Memento 218
 - Originator 218
 - Urheber 218
 - Zweckbeschreibung 223
- Model-Delegate 48
- Model-View-Controller
 - Model-Delegate 48
 - Swing 48

N

Nulla regula sine exceptione 242

O

- Observer Pattern
 - Beobachter 35
 - Ereignisquelle 35
 - GUI-Programmierung 43
 - In der Klassenbibliothek 38
 - Interface Observer 38
 - Klasse Observable 38
 - Listener 35
 - Listener-Konzept 42
 - Model-View-Controller 48

- Nebenläufiger Zugriff (Probleme) 39
- Publish/Subscribe 35
- Pull-Methode 38
- Push-Methode 38
- Zweckbeschreibung 49

P

- patternCoder XVI
 - Adapter Pattern
 - Klassenbasierter Ansatz 248
 - Objektbasierter Ansatz 247
 - Ansicht Detail 26
 - Ansicht Übersicht 25
 - Beschreibung 24
 - Datenfeld anlegen 29
 - GraphPainting 34
 - Installation 24
 - Interface implementieren 32
 - Klasse anlegen 28
 - Klassendiagramm 26
 - Methode anlegen 30
 - Pattern anlegen 27
 - patternFiles 25
- Philipp Hauer 179, 240
- Prototype Pattern
 - Aufgabe 159
 - ChlorBromIodidAlkohol 167
 - CloneNotSupportedException 166
 - Copy Constructor 163
 - Deep copy 162
 - Der Vertrag von clone() 161
 - Flache Kopie 162
 - GraphEditor 167
 - Interface Cloneable 159, 160
 - Klonen in Vererbungshierarchien 163
 - Methode clone() 159
 - Objekte klonen 159
 - Prinzip 159
 - Shallow copy 162
 - Tiefe Kopie 162
 - Unabhängigkeit von Klon und Prototyp 162
 - Zweckbeschreibung 171
- Proxy Pattern
 - Aufgabe 251
 - Dynamic Proxy 258
 - Dynamic Proxy 259

- InvocationHandler 260
- invoke() 260
- Proxy-Klasse 261
- Remote Proxy 262
 - Architektur von RMI 262
 - Der Client 265
 - Der Server 263
 - Der Skeleton als Proxy 263
 - Der Stub als Proxy 262
 - lookup() 265
 - RemoteException (Klasse) 263
 - Remote (Schnittstelle) 263
 - RMI-Registry 264
 - UnicastRemoteObject (Klasse) 264
- Security Proxy 252
 - UnmodifiableCollection 252
- Smart Reference 253, 257
 - Aufgabe 253
 - Zusätzlich ein Logging als Proxy 255
- Virtual Proxy 251
- Zweckbeschreibung 268

R

- Ralph Johnson 2
- Reflection 259
 - Class.forName() 259
 - Class-Objekt 259
 - instanceof 259
- RFC 1939 80
- Richard Helm 2

S

- Singleton Pattern
 - Antipattern 15
 - Double-checked locking 13
 - Early instantiation 14
 - Frühes Laden 14
 - Lazy instantiation 12
 - Praxisbeispiel (JDK) 11
 - Probleme aus Nebenläufigkeit 12
 - Realisierung 11
 - Verzögertes Laden 12

- Zweckbeschreibung 17
- State Pattern
 - Praxisbeispiel 80
 - Prinzip 69
 - Realisierung 75
 - Variation 78
 - Zustandsänderung 72
 - Zweckbeschreibung 81
- Strategy Pattern 99
 - Abgrenzung zu Command und State Pattern 107
 - Anwendung mit Sortier-Algorithmen 99
 - Aufgabe und Sinn 100
 - Kontext 104
 - LayoutManager 106
 - Strategie 1, der Selection Sort 102
 - Strategie 2, der Merge Sort 103
 - Strategie 3, der Quick Sort 103
 - Zweckbeschreibung 108

T

- Template Method Pattern
 - Abstrakte Oberklasse 20
 - Anwendungsbereich 20
 - Aufgabe 19
 - Einschubmethoden 21
 - Hollywood-Prinzip 21
 - Hook-Methoden 21
 - Konkrete Klasse 21
 - Zweckbeschreibung 34

V

- Visitor Pattern
 - accept() 213
 - Aufgabe 211
 - Beispiel Bauteile-Visitor 211
 - Beispiel Diagnose-Visitor 215
 - visit() 214
 - Visitor-Klasse 213
 - Vor- und Nachteile 215
 - Zweckbeschreibung 216