

LEHRBUCH

Olaf Musch

Design Patterns mit Java

Eine Einführung



Springer Vieweg

Design Patterns mit Java

Olaf Musch

Design Patterns mit Java

Eine Einführung

Olaf Musch
Braunschweig, Deutschland

ISBN 978-3-658-35491-6 ISBN 978-3-658-35492-3 (eBook)
<https://doi.org/10.1007/978-3-658-35492-3>

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Vieweg

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert durch Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2021

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Mit genehmigten Inhalten aus der 1. Auflage von „Florian Siebler-Guth, Design Patterns mit Java“.

Planung: Sybille Thelen und David Imgrund

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Herzlich willkommen!

Ich beginne mit einem Geständnis: Dieses Buch ist eigentlich gar nicht von mir.

Es ist die aktualisierte Fassung des Buchs „Design Patterns mit Java – Eine Einführung in Entwurfsmuster“ von Florian Siebler, das 2014 im Hanser-Verlag erschienen ist. Er war der Meinung, dass eine Aktualisierung aufgrund der vielen zwischenzeitlich erschienenen neuen Java-Versionen erforderlich sei. Aus Zeitgründen konnte er die Überarbeitung allerdings nicht selbst in Angriff nehmen. Daher fragte er mich, ob ich Lust hätte, sein Buch in eigenem Namen zu übernehmen und auf den aktuellen Stand zu bringen.

@Florian: Vielen Dank für das Vertrauen.

Das einführende Kapitel über den Begriff „Entwurfsmuster“ stammt als Gastbeitrag von ihm und ich habe es sehr gerne aufgenommen.

Und weil es auch meinem persönlichen Sprachstil entspricht, trete ich in diesem Buch auch in die Ich-Perspektive des Autors, aus der heraus ich zu Ihnen spreche. Ich möchte Ihnen mit diesem Buch eine auf die zu diesem Zeitpunkt aktuell gültige Java-Version angepasste Beschreibung der „ursprünglichen“ Design Patterns geben, wie sie auch im ursprünglich 1994 erschienenen Buch „Design Patterns – Elements of Reusable Object-Oriented Software“ der Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (auch genannt „Gang of Four“, kurz GoF, also „Viererbande“) Erwähnung finden (Gamma, Erich (2011): Design patterns. Elements of reusable object-oriented software. 39. printing. Boston: Addison-Wesley. ISBN 978-0201633610 (Addison-Wesley professional computing series)). In der Zwischenzeit wurden viele weitere Patterns zusammengestellt und veröffentlicht, und auch mit Antipatterns hat man sich beschäftigt: Den Dingen, die man so besser nicht machen sollte und an welchen typischen Strukturen man sie erkennt. Ich beschränke mich allerdings hier auf die insgesamt 23 Patterns des oben genannten Werks.

Dabei möchte ich Ihnen aber nach einer Begriffsklärung und einem kleinen Grundlagenkapitel keine trockene Theorie der Strukturen vorlegen, sondern anhand von konkreten Beispielen die möglichen Herangehensweisen an bestimmte Aufgaben diskutieren und das für diesen Zweck jeweils geeignete Pattern/Muster daraus ableiten, so dass Sie auch das Warum eines Musters verstehen.

Auch ein paar Programmiertricks aus der Java-Praxis werden Erwähnung finden, allerdings möchte ich bei Ihnen eine Grundkenntnis der Programmiersprache Java und der Objektorientierten Programmierung (OOP) voraussetzen. Zum Zeitpunkt der Entstehung dieses Buches habe ich die Beispiele mit NetBeans 12.3 und Java 16 auf einem PC mit Windows 10 erstellt, und dort funktionieren sie so wie beschrieben.

Noch ein Wort zu meiner Verwendung der männlichen Bezeichnungen Programmierer, Designer oder Entwickler. Ich verwende diese Begriffe ausschließlich, um einen durchgängigen Lesefluss zu ermöglichen. Selbstverständliche sollen sich damit unabhängig ihres Geschlechts alle Menschen angesprochen fühlen, die programmieren, designen oder entwickeln.

Zum Abschluss dieses Vorworts möchte ich noch den wichtigsten Personen danken, die direkt und indirekt zum Entstehen dieses Buches beigetragen haben:

Florian Siebler-Guth, der mich mit seinem Buch erst wieder auf die Idee gebracht hat, mich nach langer Zeit wieder mit der Java-Programmierung zu befassen und mir dann auch noch die Möglichkeit gegeben hat, meine Anmerkungen selbst in eine Nachfolgeaufgabe einzuarbeiten.

Benjamin Sigg, der mir als Fachgutachter mit seinen großen Java-Erfahrungen zur Seite stand.

Beim Springer-Verlag Sybille Thelen, die uns tatkräftig bei der Umwidmung der Autorenschaft unterstützt hat, und David Imgrund, der mich dann als Lektor immer sehr offen und freundlich beraten hat.

Meiner Frau Christine und unserer Tochter Jessica, die meine Zurückgezogenheit während der Schreibzeiten ertragen mussten (oder durften – eine Frage der Sichtweise).

Braunschweig, Deutschland

Olaf Musch

Inhaltsverzeichnis

1	Der Begriff „Entwurfsmuster“	1
1.1	Was sind Entwurfsmuster?	1
1.1.1	Geschichtlicher und gedanklicher Hintergrund	1
1.1.2	Vor- und Nachteile von Mustern	3
1.1.3	Ein Muster ist kein Code ist kein Muster	5
1.1.4	Muster in der Praxis	5
1.2	Entwurfsmuster kategorisieren und beschreiben	6
1.2.1	Verschiedene Kategorien von Mustern	7
1.2.2	Ein Muster-Template definieren	8
1.3	Zusammenfassung	10
2	Objektorientierte Programmierung und Entwurfsprinzipien	13
2.1	Objektorientierte Programmierung	13
2.2	Gegen Schnittstellen programmieren	16
2.3	Single Responsibility Principle (SRP)	18
2.4	Inheritance may be evil	19
2.5	Open/Closed Principle (OCP)	21
2.6	Prinzip des rechten Augenmaßes	21
3	Singleton	23
3.1	Die Aufgabe des Singleton Patterns	23
3.2	Eine erste Version des Singletons	24
3.3	Den Zugriff synchronisieren	26
3.4	Double checked locking	27
3.5	Early instantiation – frühes Laden	28
3.6	Singleton – Das UML-Diagramm	29
3.7	Antipattern	30
3.7.1	Kritik am Singleton	30
3.7.2	Ist Singleton ein Antipattern?	31
3.8	Zusammenfassung	31
	Zweckbeschreibung	31

4 Template Method	33
4.1 Die Arbeitsweise von Template Method	33
4.1.1 Ein erster Ansatz	33
4.1.2 Der zweite Ansatz	34
4.1.3 Das Hollywood-Prinzip	35
4.1.4 Einführen von Hook-Methoden	36
4.2 Das Interface „ListModel“	37
4.3 Template Method – Das UML-Diagramm	38
4.4 Zusammenfassung	39
Zweckbeschreibung	39
5 Observer	41
5.1 Einleitung	41
5.2 Eine erste Realisierung	42
5.3 Den Ansatz erweitern	43
5.4 Observer in der Java Klassenbibliothek	45
5.5 Nebenläufiger Zugriff	46
5.5.1 Zugriffe synchronisieren	47
5.5.2 Die Datenbasis kopieren	48
5.5.3 Einsatz einer threadsicheren Liste	49
5.6 Observer als Listener	50
5.7 Listener in der GUI-Programmierung	51
5.8 Das Model-View-Controller Pattern	57
5.9 Observer – Das UML-Diagramm	58
5.10 Zusammenfassung	59
Zweckbeschreibung	59
6 Chain of Responsibility	61
6.1 Ein Beispiel aus der realen Welt	61
6.2 Erstes Code-Beispiel: Lebensmitteleinkauf	62
6.2.1 Die benötigten Lebensmittel	62
6.2.2 Die Verkäufer	63
6.2.3 Der Client	64
6.3 Ein Beispiel aus der Klassenbibliothek	67
6.4 Chain of Responsibility – Das UML-Diagramm	68
6.5 Zusammenfassung	68
Zweckbeschreibung	70
7 Mediator	71
7.1 Abgrenzung zum Observer Pattern	71
7.2 Aufgabe des Mediator Patterns	72
7.3 Mediator in Aktion – ein Beispiel	72

7.3.1	Definition eines Consumers	73
7.3.2	Definition eines Producers	74
7.3.3	Interface des Mediators	75
7.3.4	Test des Mediator Patterns	76
7.4	Mediator in Aktion – das zweite Beispiel	77
7.4.1	Mediator in der GUI-Programmierung	77
7.4.2	Aufbau der GUI	78
7.5	Mediator – Das UML-Diagramm	80
7.6	Kritik an Mediator	81
7.7	Zusammenfassung	81
	Zweckbeschreibung	81
8	State	83
8.1	Exkurs: Das Enum Pattern	83
8.1.1	Einen Zustand durch Zahlenwerte darstellen	83
8.1.2	Einen Zustand durch Objekte darstellen	84
8.1.3	Umsetzung in der Java-Klassenbibliothek	85
8.2	Den Zustand eines Objekts ändern	86
8.2.1	Ein erster Ansatz	86
8.2.2	Ein zweiter Ansatz	87
8.3	Das Prinzip des State Pattern	89
8.3.1	Die Rolle aller Zustände definieren	89
8.3.2	Das Projekt aus der Sicht des Clients	91
8.3.3	Veränderungen des Projekts	92
8.4	Das State Pattern in der Praxis	94
8.5	State – Das UML-Diagramm	95
8.6	Zusammenfassung	96
	Zweckbeschreibung	96
9	Command	97
9.1	Befehle in Klassen kapseln	97
9.1.1	Version 1 – Grundversion	98
9.1.2	Weitere Anbieter treten auf	102
9.1.3	Einen Befehl kapseln	103
9.2	Command in der Klassenbibliothek	105
9.2.1	Beispiel 1: Nebenläufigkeit	105
9.2.2	Beispiel 2: Event-Handling	106
9.3	Befehlsobjekte wiederverwenden	107
9.3.1	Das Interface „Action“	108
9.3.2	Verwendung des Interface „Action“	109
9.4	Undo und redo von Befehlen	110
9.4.1	Ein einfaches Beispiel	110

9.4.2	Ein umfangreicheres Beispiel	113
9.4.3	Besprechung des Quelltextes	114
9.4.4	Undo und redo grundsätzlich betrachtet	116
9.5	Command – Das UML-Diagramm	116
9.6	Zusammenfassung	116
	Zweckbeschreibung	117
10	Strategy	119
10.1	Ein erster Ansatz	119
10.2	Strategy in Aktion – Sortieralgorithmen	121
10.2.1	Das gemeinsame Interface	121
10.2.2	Der Selection Sort	121
10.2.3	Der Merge Sort	123
10.2.4	Der Quick Sort	124
10.2.5	Der Kontext	125
10.2.6	Bewertung des Ansatzes und mögliche Variationen	126
10.3	Das Strategy Pattern in der Praxis	127
10.4	Strategy – Das UML-Diagramm	128
10.5	Abgrenzung zu anderen Mustern	129
10.6	Zusammenfassung	129
	Zweckbeschreibung	130
11	Iterator	131
11.1	Zwei Möglichkeiten, Daten zu speichern	131
11.1.1	Daten in einem Array speichern	131
11.1.2	Daten in einer Kette speichern	133
11.2	Die Aufgabe eines Iterators	135
11.3	Das Interface Iterator in Java	136
11.3.1	Der Iterator der Klasse MyArray	136
11.3.2	Der Iterator der Klasse MyList	138
11.4	Das Interface Iterable	140
11.5	Iterator – Das UML-Diagramm	141
11.6	Zusammenfassung	142
	Zweckbeschreibung	142
12	Composite	143
12.1	Prinzip von Composite	143
12.2	Umsetzung 1: Sicherheit	144
12.3	Umsetzung 2: Transparenz	147
12.4	Betrachtung der beiden Ansätze	150
12.5	Einen Schritt weiter gehen	152
12.5.1	Einen Cache anlegen	152
12.5.2	Die Elternkomponenten referenzieren	155
12.5.3	Knoten verschieben	158

12.6	Composite – Das UML-Diagramm	158
12.7	Zusammenfassung	158
	Zweckbeschreibung	159
13	Flyweight	161
13.1	Aufgabe des Patterns	161
13.2	Die Realisierung	163
13.3	Ein komplexeres Projekt	164
13.3.1	Der erste Ansatz	164
13.3.2	Intrinsischer und extrinsischer Zustand	166
13.4	Flyweight in der Praxis	168
13.5	Flyweight – Das UML-Diagramm	169
13.6	Zusammenfassung	170
	Zweckbeschreibung	170
14	Interpreter	171
14.1	Die Aufgabe in diesem Kapitel	171
14.2	Der Scanner	173
14.2.1	Die definierten Symbole	173
14.2.2	Der Scanner wandelt Strings in Symbole um	174
14.3	Der Parser	177
14.3.1	Abstrakte Syntaxbäume	177
14.3.2	Expressions für den Parser	179
14.3.3	Strichrechnung parsen	180
14.3.4	Punktrechnung parsen	182
14.3.5	Klammern berücksichtigen	184
14.4	Interpreter – Das UML-Diagramm	187
14.5	Diskussion des Interpreter Patterns	188
14.6	Zusammenfassung	188
	Zweckbeschreibung	189
15	Abstrakte Fabrik (Abstract Factory)	191
15.1	Gärten anlegen	191
15.1.1	Der erste Versuch	192
15.1.2	Der zweite Versuch – Vererbung	193
15.1.3	Der dritte Ansatz – die abstrakte Fabrik	193
15.1.4	Vorteile der abstrakten Fabrik	195
15.1.5	Einen neuen Garten definieren	196
15.2	Diskussion des Patterns und Praxis	197
15.3	Gespenster jagen	198
15.3.1	Die erste Version	198
15.3.2	Die zweite Version des Projekts	205
15.3.3	Version 3 – Einführung einer weiteren Fabrik	207
15.3.4	Version 4 – das Geisterhaus	209

15.4	Abstract Factory – Das UML-Diagramm	211
15.5	Zusammenfassung	212
	Zweckbeschreibung	212
16	Factory Method	213
16.1	Ein erstes Beispiel	213
16.2	Variationen des Beispiels	215
16.3	Praktische Anwendung des Patterns	216
16.3.1	Rückgriff auf das Iterator Pattern.	216
16.3.2	Bei der Abstract Factory	217
16.4	Ein größeres Beispiel – ein Framework	218
16.4.1	Und noch ein Kalender	218
16.4.2	Die Schnittstellen für Einträge und deren Editoren	219
16.4.3	Die Klasse „Kontakt“ als Beispiel für einen Eintrag	220
16.4.4	Die Klasse FactoryMethod als Client	221
16.5	Unterschied zur Abstract Factory	222
16.6	Factory Method – Das UML-Diagramm	223
16.7	Zusammenfassung	223
	Zweckbeschreibung	224
17	Prototype	225
17.1	Objekte klonen	225
17.1.1	Kritik an der Realisierung	226
17.1.2	In Vererbungshierarchien klonen	230
17.2	Ein größeres Projekt	233
17.2.1	Besprechung der ersten Version	233
17.2.2	Die zweite Version – deep copy	236
17.2.3	Eigene Prototype definieren	238
17.3	Prototype – Das UML-Diagramm	239
17.4	Zusammenfassung	240
	Zweckbeschreibung	241
18	Builder	243
18.1	Ein Objekt erzeugt andere Objekte	243
18.1.1	Telescoping Constructor Pattern	244
18.1.2	JavaBeans Pattern	245
18.1.3	Builder Pattern	246
18.2	Ein komplexerer Konstruktionsprozess	249
18.2.1	XML-Dateien in ein TreeModel konvertieren	251
18.2.2	XML-Dateien als HTML darstellen	254
18.3	Builder – Das UML-Diagramm	257
18.4	Zusammenfassung	258
	Zweckbeschreibung	258

19	Visitor	259
19.1	Ein einfaches Beispiel	259
19.1.1	Das Aggregat	259
19.1.2	Der Visitor	261
19.1.3	Der Client	263
19.1.4	Ein weiterer Visitor	263
19.1.5	Kritik am Projekt	264
19.2	Visitor – Das UML-Diagramm	265
19.3	Zusammenfassung	266
	Zweckbeschreibung	266
20	Memento	267
20.1	Aufgabe des Memento Patterns	267
20.1.1	Öffentliche Datenfelder	267
20.1.2	Das JavaBeans Pattern	268
20.1.3	Default-Sichtbarkeit	268
20.2	Eine mögliche Realisierung	269
20.3	Ein größeres Projekt	271
20.4	Memento – Das UML-Diagramm	273
20.5	Zusammenfassung	273
	Zweckbeschreibung	274
21	Fassade	275
21.1	Ein Beispiel außerhalb der IT	275
21.2	Die Fassade in einem Java-Beispiel	276
21.2.1	Einführung einer Fassade	278
21.2.2	Der Begriff „System“ genauer betrachtet	279
21.3	Die Fassade in der Klassenbibliothek	280
21.4	Das „Gesetz von Demeter“	281
21.5	Fassade – Das UML-Diagramm	282
21.6	Zusammenfassung	282
	Zweckbeschreibung	284
22	Adapter	285
22.1	Ein einleitendes Beispiel	285
22.2	Ein klassenbasierter Entwurf	286
22.3	Ein objektbasierter Entwurf	287
22.4	Kritik am Adapter Pattern	288
22.5	Ein Etikettenschwindel	290
22.6	Adapter – Das UML-Diagramm	290
22.7	Zusammenfassung	291
	Zweckbeschreibung	291

23	Proxy	293
23.1	Virtual Proxy	294
23.2	Security Proxy	294
23.3	Smart Reference	296
23.3.1	Die Grundversion	296
23.3.2	Einführung eines Proxys	298
23.3.3	Einen zweiten Proxy einführen	299
23.3.4	Dynamic Proxy	301
23.4	Remote Proxy	305
23.4.1	Aufbau von RMI grundsätzlich	305
23.4.2	Der RMI-Server	306
23.4.3	Der RMI-Client	308
23.4.4	Das Projekt zum Laufen bringen	310
23.5	Proxy – Das UML-Diagramm	311
23.6	Zusammenfassung	311
	Zweckbeschreibung	312
24	Decorator	313
24.1	Autos bauen	313
24.1.1	Ein Attribut für jede Sonderausstattung	314
24.1.2	Mit Vererbung erweitern	314
24.1.3	Dekorieren nach dem Matroschka-Prinzip	315
24.2	Praxisbeispiele	318
24.2.1	Die Klasse „JScrollPane“	318
24.2.2	Streams in Java	318
24.3	Decorator – Das UML-Diagramm	322
24.4	Zusammenfassung	323
	Zweckbeschreibung	323
25	Bridge	325
25.1	Zwei Definitionen	325
25.1.1	Was ist eine Abstraktion	325
25.1.2	Was ist eine Implementierung	326
25.1.3	Ein Problem beginnt zu reifen	328
25.2	Das Bridge Pattern im Einsatz	329
25.2.1	Erster Schritt	330
25.2.2	Zweiter Schritt	331
25.3	Diskussion des Bridge Patterns	333
25.3.1	Die Bridge in freier Wildbahn	333
25.3.2	Abgrenzung zu anderen Patterns	334
25.4	Bridge – Das UML-Diagramm	335
25.5	Zusammenfassung	335
	Zweckbeschreibung	337

26 Muster kombinieren	339
26.1 Das Beispiel	339
26.2 Singleton: Protokollierung	340
26.3 Abstrakte Fabrik: Bau der Sensoren	341
26.4 Observer: Die Auslösung	343
26.5 Adapter: Benachrichtigungen verschicken	344
26.6 Command: Aktoren und Fernbedienung	345
26.7 Ein Sensor löst aus	347
26.8 Klassendiagramm	348
26.9 Der Client	350
26.10 Überlegungen	353
26.11 Zusammenfassung	354
Schlussbemerkungen	355
Stichwortverzeichnis	357



Der Begriff „Entwurfsmuster“

1

In diesem Kapitel gebe ich einen Überblick darüber, was Entwurfsmuster sind. Ich zeige Ihnen den geschichtlichen Hintergrund, also wo die Muster ihren Ursprung finden. Außerdem mache ich Sie auf Vor- und Nachteile von Mustern aufmerksam. Ferner erfahren Sie, wie Sie Muster kategorisieren können. Und schließlich stelle ich das Template vor, nach dem die Muster bei Gamma et al. beschrieben werden.

1.1 Was sind Entwurfsmuster?

In diesem Abschnitt beschreibe ich, was Entwurfsmuster sind, und wie Sie von ihnen profitieren können.

1.1.1 Geschichtlicher und gedanklicher Hintergrund

Der Architekt Christopher Wolfgang Alexander beschreibt im Jahr 1977 in seinem Buch „A Pattern Language: Towns, Buildings, Construction“ (Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray; Jacobson, Max (1977): A pattern language. Towns, buildings, construction. 41. print. New York, NY: Oxford Univ. Press. ISBN 978-0195019193 (Center for Environmental Structure series, 2)) verschiedene Muster für Häuserbau und Stadtentwicklung. Eines seiner Muster behandelt zum Beispiel die Frage, wie weit über dem Fußboden eine Fensterbank zu planen ist. Fenster haben die Aufgabe, die Menschen im Haus mit der Außenwelt zu verbinden. Die Verbindung gelingt am besten, wenn Sie etwa einen halben Meter vom Fenster entfernt stehen und sowohl den Himmel als auch die Straße sehen. Wenn die Fensterbank zu hoch ist, können Sie vielleicht nicht hinausschauen.

Wenn die Fensterbank zu niedrig ist, besteht die Gefahr, dass das Fenster mit einer Tür verwechselt wird, was ein mögliches Sicherheitsrisiko ist. Alexander schlägt daher vor, die Fensterbank zwischen 12 und 14 Zoll über dem Boden vorzusehen. In höheren Stockwerken soll die Fensterbank aus Sicherheitsgründen etwa 20 Zoll über dem Boden sein.

An diesem Beispiel können Sie verschiedene Merkmale von Mustern erkennen:

- Muster werden durch einen Satz an Informationen beschrieben. Der **Kontext** sagt etwas darüber aus, wann das Muster Anwendung findet: *Sie wollen in einer Wand ein Fenster haben*. Das **Problem** benennt einen Zielkonflikt: *die Fensterbank darf nicht zu hoch und nicht zu niedrig sein*. Die **Lösung** zeigt auf abstrakter Ebene, wie das Problem in der Vergangenheit erfolgreich gelöst wurde: *die Fensterbank befindet sich auf einer Höhe zwischen 12 und 20 Zoll*.
- Ein Muster ist kein Dogma, das unbedingt befolgt werden muss. Es kann vielmehr Gründe geben, bewusst gegen die 12-20-Zoll-Regel zu verstößen. Beispielsweise werden in bestimmten Räumen oft nur Oberlichter eingebaut: in Gefängnissen oder in Duschräumen von Schwimmbädern. Umgekehrt besteht bei dekorierten Schaufenstern nicht die Gefahr, dass sie mit Türen verwechselt werden; ihre Fensterbänke können also niedriger als 12 Zoll sein.
- Muster geben keine konkreten Handlungsempfehlungen. Im Beispiel schlägt Alexander vor, die Fensterbank auf 12 bis 14 Zoll oder bei etwa 20 Zoll zu planen. Ein Architekt findet beim Entwurf eines Bauplanes also eine große Spanne, die er kontext-abhängig berücksichtigen kann. Muster sind folglich keine Algorithmen oder konkrete Implementierungen, sondern abstrakte Lösungsvorschläge, die auf das aktuelle Problem angepasst werden müssen.

Zehn Jahre später, also 1987, entwickeln Kent Beck und Ward Cunningham in entsprechender Weise fünf Muster für die GUI-Entwicklung und stellen sie auf der OOPSLA 1987 vor (<http://c2.com/doc/oopsla87.html>). **Wichtig zu wissen:** Die eben genannten Merkmale von Mustern gelten für die Muster der Softwareentwicklung in gleicher Weise.

1994 formulieren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides in ihrem Buch „Design Patterns – Elements of Reusable Object-Oriented Software“ insgesamt 23 Muster für die Softwareentwicklung. In diesem Buch findet der Hype um Muster seinen Anfang. Neben dem Musterkatalog von Gamma et al. gibt es mittlerweile zahlreiche weitere Musterkataloge für verschiedene Programmiersprachen und unterschiedliche Anwendungsbereiche. Gamma et al. sind sich dessen bewusst. Sie schreiben, dass ihre Sammlung nicht vollständig und festgeschrieben sei, vielmehr sei sie eine Momentaufnahme ihrer Gedanken. Die Sammlung enthält ihrer Auffassung nach nur einen Teil dessen, was Experten wissen könnten, insbesondere fehlen Muster für Nebenläufigkeit, verteilte Programmierung oder Echtzeitprogrammierung.

- Um ein babylonisches Sprachgewirr zu vermeiden, beachten Sie bitte: Wenn im folgenden Text die Rede von „Design Pattern“, „Pattern“, „Muster“ oder „Entwurfsmuster“ ist, dann sind damit Entwurfsmuster der Softwareentwicklung gemeint.

1.1.2 Vor- und Nachteile von Mustern

Muster beschreiben, so habe ich eben geschrieben, keine konkreten Lösungen, keine Algorithmen und erst recht keine prêt-à-porter-Implementierungen, die man einfach per Copy-and-Paste in seine Anwendungen übernehmen könnte. Warum sollten Sie sich dennoch mit Mustern beschäftigen? Welchen Profit haben Sie, wenn Sie sich durch den in diesem Buch beschriebenen Musterkatalog durcharbeiten? Die Antwort möchte ich in diesem Abschnitt geben.

1.1.2.1 Muster transportieren und konservieren Wissen

Die Muster im vorliegenden Musterkatalog wurden nicht von Forschern in weißen Laborkitteln oder weisen Gurus mit langen Bärten erfunden. Die Muster wurden vielmehr gefunden. Ähnlich wie die Zahl Pi – niemand hat Pi jemals definiert. Die Kreiszahl werden Sie finden, wenn Sie zahlreiche Kreise auf das Verhältnis von Durchmesser zu Umfang untersuchen. Entsprechend untersuchen Gamma et al. viele Softwaresysteme und fanden dabei Übereinstimmungen. Wenn eine bestimmte Lösung in drei oder mehr Systemen auftritt, wird sie als Muster betrachtet (sogenannte „Rule of Three“). Gamma et al. geben diesem Design einen eingängigen Namen und beschreiben ihre Beobachtungen folgendermaßen: „*Keines der Entwurfsmuster [...] beschreibt neuartige Entwürfe. Wir haben lediglich solche Entwürfe berücksichtigt, die mehrfach angewendet wurden und sich in unterschiedlichen Systemen bewährt haben.*“ Wenn Sie sich also mit Mustern beschäftigen, erfahren Sie, wie andere Entwickler bestimmte Probleme gelöst haben. Sie können sich ansehen, welche Ansätze sie verfolgt haben, und welche Lösungen sich bewährt haben.

1.1.2.2 Muster schaffen ein gemeinsames Vokabular

Stellen Sie sich ferner vor, Sie und Ihr Team arbeiten an einem großen Softwaresystem. Ihre Aufgabe ist es, ein bestimmtes Subsystem zu entwickeln. In diesem Subsystem bestehen zahlreiche Abhängigkeiten. Ihr Kollege, der an einem anderen Teil des Systems arbeitet, möchte Ihr Subsystem nutzen. Er hat jedoch Schwierigkeiten, die Abhängigkeiten aufzulösen, und bittet Sie, eine **Fassade** (eines der Muster, die ich Ihnen später vorstellen werde) dafür zur Verfügung zu stellen. Da Sie sich mit Mustern beschäftigt haben, wissen Sie, dass Ihr Kollege einen vereinfachten Zugriff auf Ihr Subsystem haben möchte, in dem die dahinterstehenden Abhängigkeiten bereits aufgelöst sind. Muster definieren also ein Vokabular, das es Ihnen und Ihren Kollegen ermöglicht, auf einem höheren Abstraktionslevel zu kommunizieren. Einen Eindruck vom Umfang des Vokabulars gibt Ihnen Tab. 1.1.

Tab. 1.1 Kategorien von Entwurfsmustern nach Gamma et al.

	Klassenbasiert	Objektbasiert
Creational Patterns (Erzeugungsmuster)	Factory Method	Abstract Factory Builder Prototype Singleton
Structural Patterns (Strukturmuster)	Adapter	Adapter Bridge Composite Decorator Facade Flyweight Proxy
Behavioral Patterns (Verhaltensmuster)	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

1.1.2.3 Muster helfen, eine Programmiersprache besser zu verstehen

Muster beschreiben sprachunabhängig abstrakte Lösungsansätze. Das bedeutet, dass jede Entwicklergeneration gefordert ist, eigene Implementierungen zu entwickeln. Gamma et al. haben ihre beispielhaften Implementierungen unter anderem in Smalltalk geschrieben. Implementierungen in Java werden möglicherweise ganz anders aussehen. Viele Muster sind bereits in der Klassenbibliothek von Java verankert. Es ist ausgesprochen spannend, die Muster dort zu suchen und zu identifizieren. Wenn Sie sich mit Entwurfsmustern in Java beschäftigen, werden Sie nebenbei viel über Java lernen.

An dieser Stelle möchte ich eine Unschärfe ansprechen. Die Frage, ob Sie es mit einem Entwurfsmuster zu tun haben oder nicht, hängt von Ihrem Standpunkt ab. Für einen prozedural geprägten Programmierer wären Konzepte wie Vererbung, Kapselung oder Polymorphie Entwurfsmuster. Wenn Sie jedoch in objektorientierten Programmiersprachen heimisch sind, werden Sie die genannten Begriffe als feste Sprachbestandteile betrachten.

1.1.2.4 Muster können zu unangemessenen Entwürfen verleiten

Muster können einen Entwickler begeistern und motivieren. Er könnte vielleicht versuchen, möglichst viele Muster in seinen Programmen zu realisieren. Das kann jedoch zu Schwierigkeiten führen. Viele Muster bedingen, dass der Entwickler eine zusätzliche Abstraktionsebene einführt. Das kann dazu führen, dass ein Entwurf unnötig kompliziert wird oder die Performance leidet. „*Ein Entwurfsmuster sollte nur angewendet werden, wenn die gebotene Flexibilität auch wirklich benötigt wird,*“ lesen Sie bei der Gang of

Four dazu. Dass Sie Muster einsetzen können, heißt nicht, dass sie dies auch immer müssen. Es sind Werkzeuge, die sinnvoll eingesetzt werden sollen. Vor Jahren habe ich mal von einem britischen Kollegen den Spruch „*A fool with a tool is still a fool*“ gehört, der auch hier beachtet werden sollte. Nicht überall, wo man ein Werkzeug (hier Design Patterns) einsetzen könnte, ist es auch sinnvoll, dies zu tun. Im Abschlusskapitel zur Kombination von Mustern gehe ich darauf noch mal ein.

1.1.3 Ein Muster ist kein Code ist kein Muster

Bevor Sie anfangen, ein Pattern umzusetzen, lassen Sie uns die Frage diskutieren, was überhaupt ein Pattern ist. Wenn Sie im Internet zum Beispiel nach Singleton suchen, werden Sie für jede Programmiersprache eine eigene Implementierung finden. Auch für Java gibt es eine Implementierung, die vielleicht so etwas wie ein „Standard“ ist. Doch diese Implementierung ist nicht das Singleton Pattern selbst. Ein Pattern selbst beschreibt nur, was das Ziel ist, worauf man bei der Realisierung des Patterns achten soll und welche Vorteile und Nachteile sich daraus ergeben. Die GoF stellt in ihrem Buch für jedes Pattern ein Code-Beispiel vor. Dennoch bleibt es Ihnen allein überlassen, wie Sie das Pattern realisieren. Nur so können Patterns in alle Sprachen mit ihren jeweiligen Besonderheiten übertragen werden. Vergleichen Sie das mit dem Bauplan eines Hauses, der beschreibt, wo die Wände stehen, aber nicht, wie die Wände zu bauen sind.

Die meisten Bücher über Design Patterns zeigen konkrete Implementierungen, die sich als praktisch herausgestellt haben; auch ich stelle Ihnen diese „Standards“ vor. Dennoch ist es im Sinne des Patterns erlaubt, eine andere Lösung zu finden.

Was also sind „Entwurfsmuster“?

Was also sind Entwurfsmuster in der Softwareentwicklung? Entwurfsmuster beschreiben auf abstrakter Ebene Lösungen für wiederkehrende Probleme. Sie tragen das Wissen der Entwickler zusammen. Das erleichtert es, Erfahrung weiterzugeben und auf bewährte Lösungen zurückzugreifen, sie also wiederzuverwenden. Daneben erweitern sie das Vokabular der Entwickler.

Sie werden unter anderem das State Pattern und das Strategy Pattern kennenlernen. Im direkten Vergleich dieser beiden Muster werden Sie sehen, dass ein Muster sich immer in einem bestimmten Kontext bewegt, bzw. immer ein eindeutig definiertes Problem löst.

1.1.4 Muster in der Praxis

Wenn Sie dieses Buch durchgearbeitet haben, wie setzen Sie dann die Patterns in der Praxis um? Sie werden, wenn Sie dieses Buch gelesen haben, nicht jedes Muster mit allen

Einzelheiten im Kopf haben. Diesen Ehrgeiz sollten Sie gar nicht entwickeln. Sie haben aber gehört, dass es ein Muster gibt, das beschreibt, was in Situationen zu tun ist, wenn eine Ereignisquelle seine Beobachter über Zustandsänderungen informieren soll. Sie nehmen sich in der konkreten Situation Ihres Alltags mein Buch zur Hand und lesen die Struktur, die Vor- und Nachteile noch einmal nach. Passt das Muster auf Ihre konkrete Situation? Ist die Anwendung des Musters angemessen? Anhand dieser Fragen werden Sie Ihre konkrete Fragestellung lösen. Je mehr Muster Sie kennen und in der Vergangenheit angewandt haben, umso mehr steigt Ihr Erfahrungsschatz und umso flexibler und ziel-sicherer können Sie Muster einsetzen und bewerten. Mit Programmierern ist es wie mit Wein – je älter sie werden, umso besser werden sie.

- ▶ **Tipp** Setzen Sie Muster aber bitte mit Bedacht ein. Das Wissen über Design Patterns darf nicht dazu verleiten, dass Sie Ihre Software mit Patterns spicken. Die meisten Muster verlangen, dass Sie zusätzliche Klassen entwickeln. Wenn Sie versuchen, in einer überschaubaren Anwendung möglichst viele Muster zu realisieren, wird das Ergebnis ein schlecht wartbares, unnötig aufgeblähtes System sein. Haben Sie den Mut, auf ein Muster zu verzichten oder ein Muster sogar zu entfernen! Muster haben keinen Selbstzweck. Ihr Ziel als Programmierer sollte es sein, übersichtlichen und unspektakulären Programmcode zu erzeugen.

Lassen Sie mich in diesem Zusammenhang noch einen Punkt erwähnen: Erweiterungen. Eine Konstante der Anwendungsentwicklung ist, dass Anforderungen – und damit der Quellcode – wachsen. Es ist einerseits sinnvoll, Programme so zu gestalten, dass sie in Zukunft erweitert werden können. Doch die Ausrichtung auf künftige Anforderungen sollte andererseits sehr zurückhaltend stattfinden. Bedenken Sie, dass Sie im „Hier und Jetzt“ Programme schreiben, die im „Hier und Jetzt“ eingesetzt werden.

Auf der Webseite stackoverflow.com, einer seit 2008 bestehenden großen Entwickler-Community, habe ich folgende Formulierung gefunden, die mir sehr gut gefallen hat: „*Design Pattern is meant to be a solution to a problem, not a solution looking for a problem.*“¹

1.2 Entwurfsmuster kategorisieren und beschreiben

Entwurfsmuster werden in Kategorien zusammengefasst und nach einem bestimmten Template beschrieben. Diese Formalien zeige ich in diesem Abschnitt.

¹ https://stackoverflow.com/questions/11079605/adapter-any-real-example-of-adapter-pattern#comment14506747_11079605.

1.2.1 Verschiedene Kategorien von Mustern

Der Musterkatalog von Gamma et al. enthält 23 Muster. Diese werden in drei Kategorien unterteilt. Innerhalb dieser Kategorien wird weiter unterschieden, ob ein Muster eher objekt- oder eher klassenbasiert ist. Hier stelle ich Ihnen die drei Kategorien vor:

- **Creational Patterns (Erzeugungsmuster):** Sie verstecken den Erzeugungsprozess und helfen, ein System unabhängig davon zu machen, wie seine Objekte konkret erzeugt, zusammengesetzt und repräsentiert werden. Ein klassenbasiertes Erzeugungsmuster verwendet Vererbung, um die Klasse des zu erzeugenden Objektes zu variieren, während ein objektbasiertes Erzeugungsmuster die Erzeugung an ein anderes Objekt delegiert. Es gibt zwei immer wiederkehrende Leitmotive in diesen Mustern. Zum einen kapseln sie alle das Wissen um die konkreten vom System verwendeten Klassen. Zum anderen verstecken sie, wie Instanzen dieser Klassen erzeugt und zusammengefügt werden. Alles, was die Anwendung über die Objekte weiß, wird durch die definierten Schnittstellen bestimmt.
- **Structural Patterns (Strukturmuster):** Sie beschäftigen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen und Implementierungen zusammenzuführen. Objektbasierte Strukturmuster hingegen beschreiben Mittel und Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen.
- **Behavioral Patterns (Verhaltensmuster):** Sie befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Sie beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch die Muster der Interaktion zwischen ihnen. Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. Objektbasierte Verhaltensmuster verwenden Komposition anstelle von Vererbung. Muster dieser Kategorie beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachzuvollziehen sind. Dadurch verschiebt sich der Fokus weg vom Kontrollfluss hin zu der Art und Weise, wie die Objekte miteinander interagieren.

Tab. 1.1 zeigt Ihnen die Kategorien und die Muster im Überblick. Es fällt auf, dass es mehr objektbasierte Muster als klassenbasierte gibt. Das Muster „Adapter“ gibt es sowohl klassen- als auch objektbasiert. Vielleicht sind Ihnen nicht alle Muster geläufig. Das macht nichts – Aufgabe dieses Buches ist es, Ihnen alle Muster im Detail vorzustellen.

Andere Kategorisierungsschemata

Buschmann et al. [Buschmann, Frank; Löckenhoff, Christiane (2000): Pattern-orientierte Softwarearchitektur. Ein Pattern-System. 1., korr. Nachdr. München: Addison-Wesley.] kritisieren, dass „*eine Unterscheidung zwischen struktur- und verhaltensorientierten Mustern zu ungenau*“ sei, da sie „*die speziellen Problembereiche, mit denen es ein Entwickler bei der Konstruktion von Software zu tun hat*“ nicht benennen. Sie stellen neben Entwurfsmustern auch Architekturmuster und Idiome vor und entwickeln andere Kategorien. Um Ihnen einen Eindruck von ihrer Kategorisierung zu geben, habe ich das Schema von Buschmann et al. in Tab. 1.2 auszugsweise hinterlegt. Sie finden dort auch mehr Muster als in Tab. 1.1.

Tab. 1.2 Kategorien von Entwurfsmustern nach Buschmann et al., Seiten 376 f.

	Architekturmuster	Entwurfsmuster	Idiom
Vom Chaos zur Struktur	Layers Pipes-and-Filters	Interpreter	
Erzeugung		Abstract Factory Prototype Builder	Singleton Factory Method
Strukturelle Zerlegung		Composite	
Organisation von Arbeit		Master-Slave Chain-of-Responsibility Command Mediator	
Zugriffskontrolle		Proxy Facade Iterator	
Variation von Diensten		Bridge Strategy State	Template Method
Erweiterung der Dienstleistung		Decorator Visitor	
Management		Memento	
Adaptation		Adapter	
Kommunikation		Publisher-Subscriber (Observer)	
Ressourcenverwaltung		Flyweight	

Architekturmuster kommen zu Beginn des Entwurfs zum Einsatz, „*wenn die grundsätzliche Struktur der Anwendung festgelegt wird*“ (Seite 360). Entwurfsmuster finden für Buschmann et al. Anwendung, wenn die grundsätzliche Struktur „*verfeinert und erweitert*“ wird (Seite 360). Idiome werden „*während der Implementierungsphase benutzt*“ (Seite 361).

In diesem Buch beziehe ich mich ausschließlich auf die Muster von Gamma et al. sowie deren Kategorisierungsschema. Ich möchte Ihnen jedoch einen Eindruck davon geben, dass Sie in Literatur und Praxis auf andere Muster und andere Musterkategorien stoßen werden. Die Unterscheidung nach Architekturmuster, Entwurfsmuster und Idiom ist in diesem Buch nicht relevant.

1.2.2 Ein Muster-Template definieren

Dokumentationen werden oft nach dem gleichen Template aufgebaut – denken Sie beispielsweise an das Dokumentations-Template arc42 für Softwarearchitekturen, das Sie unter www.arc42.com finden. Der Leser weiß, wenn er das Template kennt, sofort, an welcher Stelle er welche Information findet. Auch Muster werden nach einem einheitlichen Schema dokumentiert. Aus der Vogelperspektive betrachtet muss jedes Muster mindestens einen **Namen** haben, sowie je einen Abschnitt über das zu lösende **Problem**, die

konkrete **Lösung** selbst sowie zu möglichen **Konsequenzen** bei der Anwendung des Musters.

Um diesen Mindestsatz an Informationen einheitlich zu beschreiben, greifen Gamma et al. auf folgendes Template zurück:

- **Mustername:** Der Name des Musters ist ein Stichwort, das mit ein oder zwei Wörtern das Problem, die Lösung und die Auswirkungen benennt. Der Name ist Teil des Vokabulars und erlaubt uns, auf einem höheren Abstraktionsniveau zu kommunizieren und zu dokumentieren
- **Kategorie:** Die Kategorie greift auf die in Abschn. 1.2.1 beschriebene Klassifizierung zurück
- **Zweck:** Dieser Abschnitt beschreibt kurz Aufgabe und Ziel des Musters
- **Auch bekannt als:** Sofern es andere Namen für das Muster gibt, werden sie in diesem Abschnitt genannt
- **Motivation:** Anhand eines konkreten Szenarios wird ein Problem aufgezeigt und erläutert, wie das Muster dieses Problem löst. Das Szenario fördert das Verständnis für die hinter dem Muster stehenden abstrakten Zusammenhänge
- **Anwendbarkeit:** Dieser Abschnitt beschreibt, in welchen Situationen das Muster angewandt werden kann. Hier wird also der Kontext genannt
- **Struktur:** Die am Muster beteiligten Klassen und Objekte werden in Diagrammen dargestellt. Gamma et al. verwenden hierfür OMT, einen Vorläufer von UML
- **Teilnehmer:** Die beteiligten Klassen und Objekte werden beschrieben. Sie werden dort auch erfahren, welche Aufgaben sie haben
- **Interaktionen:** In diesem Abschnitt lesen Sie, wie die Teilnehmer zusammenarbeiten
- **Konsequenzen:** Jedes Muster hat Vor- und Nachteile; diese werden hier genannt. Sie erhalten Hinweise zur Implementierung, insbesondere Fallen und hilfreiche Tipps
- **Implementierung:** Code-Schnipsel demonstrieren die Realisierung des Musters sowie verschiedene Implementierungsvarianten
- **Bekannte Verwendungen:** Sie erfahren, wo das Muster eingesetzt wird. Das ist nützlich, um Beispiele für den Gebrauch eines Musters zu bekommen
- **Verwandte Muster:** Oft sehen Muster sehr ähnlich aus; Sie erfahren, wo die Unterschiede liegen, aber auch, wie zwei Muster zusammenarbeiten

Ich werde diese Abschnitte im Buch nicht so formal aufführen wie Gamma et al. Mir ist es wichtig, Ihnen die relevanten Informationen in leicht verständlicher Weise zu präsentieren – und dazu breche ich bewusst mit dem formalen Aufbau. Ich werde jedoch genau wie Gamma et al. in jedem Kapitel jeweils ein Muster beschreiben. An einer zweiten Stelle weiche ich vom Schema von Gamma et al. ab: Ich zitiere die Zweckbeschreibung erst am Ende eines Kapitels. Der Grund dafür ist, dass ich selbst die Erfahrung gemacht habe, dass ich, wenn ich die Zweckbeschreibung lese, zunächst nicht verstehen, was damit gemeint ist. Wenn ich mich vorher mit dem Muster beschäftigt habe, ist sie hingegen eingängig. Kurz

gesagt: wenn ich die Zweckbeschreibung verstehe, habe ich das Kapitel insgesamt verstanden. Und genau diese Möglichkeit der Lernkontrolle möchte ich Ihnen geben.

- Im nächsten Kapitel werde ich Ihnen objektorientierte Entwurfsprinzipien vorstellen. Anschließend, ab Kap. 3 werde ich die Muster beschreiben. Die Beispiele nutzen die aktuellen Sprachfeatures von Java 16 – auch mit einigen Preview Features. Soweit diese helfen, die Bedeutung eines Musters zu verstehen oder den Quelltext einfach zu halten, gehe ich genauer darauf ein. Zumindest erwähne ich dann die Namen der APIs oder Nummern betreffender „Specification Requests“, die Sie als Anhaltspunkte für eigene Recherchen zu diesen Features verwenden können. Allerdings setze ich im Wesentlichen voraus, dass Sie die Sprachfeatures von Java 16 mindestens grob kennen. Das Ziel dieses Buches ist es, die Muster zu erläutern. Manchmal helfen dabei die neueren Sprachfeatures von Java, manchmal machen sie die Sache aber auch „komplizierter“ bzw. den Code ggf. länger. In diesem Falle verwende ich sie nicht. Im Buch drucke ich nur die notwendigsten Aspekte der Beispiele ab. In dieser Form wären die Programme nicht lauffähig, und ich habe die Kürzungen auch nicht überall markiert. Auch Quelltext-Kommentare habe ich hier für den Abdruck entfernt, sie sind aber vorhanden. Den Link zum Zusatz-Material eines Kapitels finden Sie am Anfang jedes Kapitels in der Fußzeile.

Sie finden in der zip-Datei, die Sie downloaden, die Quelltexte als NetBeans-Projekte, die Sie direkt in NetBeans öffnen können. Die von mir persönlich bevorzugte Entwicklungsumgebung NetBeans selbst finden Sie bei <https://netbeans.apache.org/>. Natürlich können Sie auch jede andere Ihnen genehme Entwicklungsumgebung verwenden. Selbst mit einem einfachen Texteditor und einer Kommandozeile kann man Java-Anwendungen entwickeln.

Das von mir verwendete Java Development Kit (JDK) 16 finden Sie bei <https://jdk.java.net/archive/> (bei Verfügbarkeit dieses Buches im Handel dürfte es nicht mehr das allerneueste JDK sein). Die Beispiele sollten aber zumindest unter Java 17 ebenfalls lauffähig sein.

Wenn ich in einem Kapitel ein Beispielprojekt nenne, finden Sie es auch im Unterordner des jeweiligen Kapitels. Gleiches gilt für spezielle Dokumente oder Dateien, die im jeweiligen Kontext relevant sind.

1.3 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Design Patterns sind bewährte Lösungen für eine unbestimmte Zahl von Problemen, die in dieser oder einer ähnlichen Form immer wieder auftreten.
- Design Patterns schaffen ein Vokabular, das für Programmierer zur Allgemeinbildung gehört.

- Design Patterns kann man nicht erfinden, sondern nur entdecken – genau wie die Zahl Pi.
- Die GoF – Gang of Four – hat maßgeblich dazu beigetragen, Design Patterns in die Softwareentwicklung einzuführen.
- Das Buch der GoF umfasst 23 Patterns, es gibt jedoch sehr viele mehr.
- Die GoF hat ihre Patterns in drei Kategorien eingeteilt:
 - Erzeugungsmuster beschreiben, wie Objekte effizient erstellt werden können,
 - Strukturmuster beschreiben, wie Klassen zu sinnvollen größeren Einheiten zusammengesetzt werden können,
 - Verhaltensmuster beschreiben das Zusammenspiel einzelner Objekte.
- Eine Musterbeschreibung setzt sich aus den folgenden wesentlichen Elementen zusammen:
 - einem einprägsamen und prägnanten Namen,
 - einem Problemabschnitt, der zeigt, wann das Muster angewandt wird,
 - einem Lösungsabschnitt, der die beteiligten Einheiten und deren Zusammenhänge beschreibt,
 - einem Abschnitt zu den Konsequenzen, der auf die Vor- und Nachteile des Musters eingeht.



Objektorientierte Programmierung und Entwurfsprinzipien

2

Gute Softwaresysteme sind nicht nur korrekt implementiert, sondern zeichnen sich unter anderem auch dadurch aus, dass sie erweiterbar und verständlich sind. In diesem Kapitel beschreibe ich die objektorientierte Programmierung und einige objektorientierte Entwurfsprinzipien, welche die Korrektheit, Erweiterbarkeit und Verständlichkeit fördern.

2.1 Objektorientierte Programmierung

Man hört sehr oft, dass man Patterns nicht braucht, wenn man die Regeln der objektorientierten Programmierung beherrscht und anwendet. Stimmt das? Wie stehen Patterns und objektorientierte Programmierung in Beziehung?

Wenn Sie objektorientiert programmieren, werden Sie sicher mit Vererbung arbeiten – Sie haben etwas Allgemeines und leiten daraus etwas Spezielles ab. In den meisten Java-Einführungen wird Vererbung als das Nonplusultra vorgestellt. Dennoch hat Vererbung auch Nachteile und die möchte ich jetzt ansprechen. Sie haben eine Klasse und bilden eine Subklasse, die das in der Oberklasse definierte Verhalten erbt. Sie können nun weitervererben, aber je weiter unten in der Vererbungshierarchie eine Klasse sich befindet, umso geringer ist die Möglichkeit der Wiederverwendbarkeit. Außerdem schlägt jede Änderung, die Sie an der Superklasse vornehmen, auf sämtlichen Subklassen durch; die Kapselung wird dadurch geschwächt, was in der Regel auf ein ungeeignetes Design schließen lässt. Doch der wohl größte Nachteil von Vererbung ist, dass die Vererbungshierarchie sowohl in der Breite als auch in der Tiefe sehr rasch wächst. Ein Beispiel soll das veranschaulichen.

Lassen Sie uns über eine Software nachdenken, die benötigt wird, um die Tiere eines Zoos zu verwalten. An der Spitze der Vererbungshierarchie der Tiere steht sicher die Klasse `Lebewesen`. In dieser Klasse wollen Sie die Anzahl Beine speichern. Moment –

was ist, wenn Sie Fische in die Hierarchie mit aufnehmen wollen? Die müssten das Attribut `anzahlBeine` ebenfalls mit sich herumtragen, obwohl es keine Fische mit Beinen gibt. Gut, einigen wir uns auf den Kompromiss, dass Fische nicht in der Vererbungshierarchie berücksichtigt werden. Die Superklasse speichert also die Anzahl Beine. Fangen wir mit den Vögeln an – Vögel können fliegen, also müssen Sie eine Subklasse `Vogel` entwickeln, in der Aussagen über das Flugverhalten getroffen werden, also zum Beispiel wie schnell ein Vogel fliegen kann. Dabei fällt mir ein: Pinguine werden doch auch zu den Vögeln gezählt, obwohl sie gar nicht fliegen können. Nun haben Sie die Wahl, ob Sie Pinguine ebenfalls nicht in die Vererbungshierarchie aufnehmen, oder ob Sie zwei unterschiedliche Subklassen bilden wollen: `FlugVogel` und `NichtFlugVogel`. Aber das soll im Moment noch das geringste Problem sein. Ein Tier hat Beine, um laufen zu können. In die Superklasse `Lebewesen` nehmen Sie jetzt auch die Informationen über das Laufverhalten auf. Damit sind Sie in der Lage, neben Vögeln auch Hunde, Katzen, Kammele, Affen und Löwen abzubilden. Aber halt – einige Vögel können gar nicht laufen, sondern nur hüpfen. Schon wieder müssen Sie sich Gedanken über Einschränkungen in der Vererbungshierarchie machen. Wollen Sie hinnehmen, dass Subklassen Attribute und Methoden erben, die sie gar nicht brauchen? Oder wollen Sie hüpfende Vögel aus der Vererbungshierarchie ausschließen? Wenn nicht, müssen Sie konsequenterweise von der Subklasse `Vogel` neben den Subklassen `FlugVogel` und `NichtFlugVogel` folgende weitere Subklassen bilden: `LaufVogel` und `HuepfenderVogel`. Die meisten Vögel können jedoch sowohl laufen als auch fliegen. Wie könnte ich mit dieser Situation umgehen? Eigentlich müssten Klassen entwickelt werden, die sowohl laufen (alternativ hüpfen) als auch fliegen ermöglichen. Sie brauchen also eine Klasse `FlugLaufVogel`, eine Klasse `HuepfenderFlugVogel` usw.

Um die Sache noch ein wenig auf die Spitze zu treiben, könnten Sie anfangen, die Fressgewohnheiten der Tiere zu betrachten. Ein Löwe ist sicher ein Fleischfresser und da er sich außerhalb der komplizierten Vererbungshierarchie der Vögel befindet, klingt die Anforderung nicht sonderlich schwierig. Nehmen Sie also in der Superklasse `Lebewesen` das Attribut `mengeFleischProTag` auf – diese Information ist für den Zoo bestimmt wichtig! Aber Moment! Da sich in der Vererbungshierarchie der Vögel auch Tauben befinden, erben diese das Attribut ebenfalls. Tauben stehen aber nicht im Verdacht, sich von Fleisch zu ernähren, also sollten sie dieses Attribut eigentlich nicht mit sich herumtragen. Die Information, wie viel Fleisch pro Tag ein Tier frisst, darf also nicht in der Klasse `Lebewesen` gespeichert werden. Welche Lösung bietet sich an? Führen Sie unterhalb der Klasse `Lebewesen` eine Klasse `FleischfressenderNichtVogel` ein. Von dieser Klasse kann der Löwe eine Instanz bilden. Kühe fressen genau wie Tauben kein Fleisch. Also entwickeln Sie unterhalb der Klasse `Lebewesen` die Klasse `VegetarischerNichtVogel`. In dieser Klasse wird sicher die Information gehalten, wie viel fleischlose Nahrung der Zoo pro Tag zur Verfügung stellen muss. Sie merken schon, dass ein weiteres Problem zu reifen beginnt. Es gibt auch unter den Vögeln Fleischfresser wie Adler und Vegetarier wie die Tauben. Sie können also auf der Vererbungslinie der Vögel

erneut anfangen und zwischen Fleischfressern und Vegetariern unterscheiden. Adler werden dann Instanzen der Klasse `FleischfressenderFlugLaufVogel` sein.

Grafik Abb. 2.1 zeigt, wie das Klassendiagramm für solch ein Projekt aussehen könnte.

Das Beispiel ist aus Sicht der Objektorientierung sicher richtig; dennoch ist die Lösung nicht besonders praktisch. Sie enthält zu viele Kompromisse bzw. Einschränkungen und ist darüber hinaus zu unflexibel. Design Patterns schlagen Alternativen vor, die Ihnen helfen, mit den Methoden der objektorientierten Programmierung flexible und erweiterbare Software zu erstellen.

- ▶ Ich habe Ihnen das Beispiel bewusst überspitzt vorgestellt, weil ich Sie davon überzeugen möchte, dass Vererbung Sie in ein sehr starres System führen kann, in dem eine Hierarchie besteht, die weder in der Breite noch in der Tiefe überblickt werden kann. Die Klassen sind so speziell, dass sie an keiner anderen Stelle wiederverwendet werden können.

Das Beispiel wäre mit *Komposition* sicher sehr viel besser entwickelt worden. Konkret bedeutet das, dass Sie eine Schnittstelle `Fressverhalten` entwickeln, von der die Klassen `Fleischfresser` und `Vegetarier` abgeleitet werden. In diesen Klassen wird gespeichert, wie viel Fleisch bzw. wie viel Gras pro Tag vorgesehen ist. Die Kuh muss genauso wie die Taube das Attribut `Fressverhalten fressverhalten = new Vegetarier()` haben. Und folglich werden sowohl der Löwe als auch der Adler das Attribut `Fressverhalten fressverhalten = new Fleischfresser()` bekommen. Das relevante Verhalten wird also nicht vererbt, sondern in einer eigenen Klasse definiert. Das Klassendiagramm Abb. 2.2 zeigt das Zusammenspiel.

Was ist dadurch gewonnen? Ihre Vererbungshierarchie wird sehr viel übersichtlicher. Darüber hinaus können Sie **sehr leicht** neues Verhalten definieren. Unter den Begriff „leicht“ können Sie alles subsumieren, was gute Software ausmacht: Beispielsweise lassen

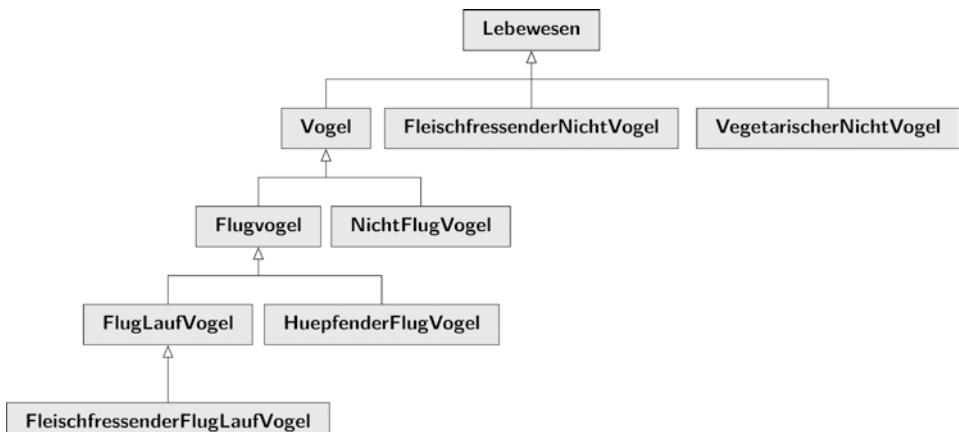


Abb. 2.1 Beispiel für ungeschicktes Design

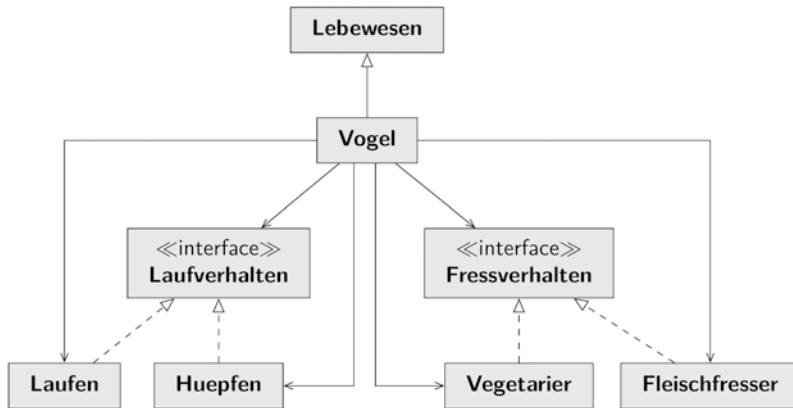


Abb. 2.2 Beispiel für besseres Design

sich Stellen, an denen Änderungen vorzunehmen sind, schnell identifizieren; außerdem muss bestehender Code nicht erneut getestet werden. Eine Änderung ist nicht nur wartungsfreundlicher, sondern auch weniger fehleranfällig durchzuführen. Die folgende Erweiterung veranschaulicht diese Aussage. Wenn man Schnecken und Würmer als Fleisch definiert, müssen Enten konsequenterweise Fleischfresser sein; sie fressen aber auch Gras, so dass Sie hier eine Mischform haben. Das ist kein Problem – Sie lassen die Klasse Allesfresser von der Schnittschnelle Fressverhalten erben und können ein neues Verhalten anbieten, ohne bestehenden Code aufzubrechen. Die Ente bekommt das Attribut Fressverhalten fressverhalten = new Allesfresser(). In entsprechender Weise könnte das Laufverhalten dargestellt werden.

- ▶ Design Patterns nutzen die Mittel der objektorientierten Programmierung, um damit flexible, erweiterbare und wartungsfreundliche Lösungen zu entwickeln. Auch wenn Sie die Regeln der objektorientierten Programmierung perfekt anwenden können, sollten Sie sich unbedingt mit Entwurfsmustern beschäftigen. Sie werden weiter hinten – beim Builder Pattern – ein weiteres Beispiel dafür finden, dass der Werkzeugkasten der OOP Ihnen bei der Objekterzeugung keine befriedigende Lösung zur Verfügung stellt.

2.2 Gegen Schnittstellen programmieren

Im Kapitel über das Observer Pattern werde ich auf die Notwendigkeit hinweisen, gegen Schnittstellen zu programmieren. Was ist dadurch gewonnen? Was genau ist mit Schnittstelle gemeint? Die Rückfrage nach dem Begriff Schnittstelle ist geboten, weil es mir wichtig erscheint, ihn deutlich vom Begriff Interface abgrenzen.

Die Objektorientierung verlangt, dass Sie die Attribute eines Objekts, den Zustand eines Objektes, kapseln. Ein Zugriff von außen soll nicht möglich sein. Auf die Attribute darf nur indirekt über Methoden zugegriffen werden. Jede Methode hat einen Bezeichner, eine Parameterliste und einen Rückgabewert – das ist die Signatur einer Methode. Alle öffentlichen Methoden zusammen bilden die Schnittstelle eines Objekts. Der Typ ist der Name für diese Schnittstelle.

Methoden müssen nicht zwingend definiert sein, es genügt, dass sie nur die Signatur deklarieren; sie sind dann abstrakt. Es ist Aufgabe der abgeleiteten Klassen, diese Methoden zu überschreiben und konkret zu implementieren. Abgeleitete Klassen sind der Subtyp der abstrakten Klasse, die in einer Vererbungshierarchie den Supertyp darstellt. Klassen mit abstrakten Methoden sind selbst abstrakt. Abstrakte Klassen können jedoch neben abstrakten Methoden auch nichtabstrakte Methoden definieren. Sie können sogar alle Methoden definieren, so dass eine abstrakte Klasse gar keine abstrakten Methoden deklariert. Wenn eine Klasse ausschließlich abstrakte Methoden deklariert, können Sie als Java-Programmierer sie als Interface anlegen. Da Java keine direkte Mehrfachvererbung erlaubt, ist die Einführung von Interfaces ein wichtiges Sprachelement. Sie können beliebig viele Interfaces implementieren, so dass ein Objekt vielen Schnittstellen entsprechen – oder auch „viele Rollen einnehmen“ – kann. Auch wenn wir Java-Programmierer aus diesem Grund in der Regel Interfaces meinen, wenn wir von Schnittstellen sprechen, müssen Sie sich im Patterns-Umfeld immer vor Augen halten, dass Schnittstellen entweder Interfaces oder abstrakte Klassen sein können.

Hintergrundinformation

Seit Java 8 können in Interfaces statische und nichtstatische Methoden definiert werden. Es ist also möglich, von einem Interface Verhalten zu „erben“. Der Unterschied zwischen abstrakter Klasse und Interface wird unschärfer – auch vor diesem Hintergrund ist es wichtig, den Begriff der Schnittstelle im Kontext Design Patterns weit auszulegen.

Der Unterschied zwischen abstrakter Klasse und Interface ist, dass Interfaces Rollen definieren, während abstrakte Klassen die Aufgabe haben, eine Vererbungshierarchie zu begründen. Interfaces sind im Gegensatz zu abstrakten Klassen stateless, und wenn Sie in einem Interface eine Methode definieren, dann geben Sie der implementierenden Klasse ein Default-Verhalten mit. Sinnvoll ist das, wenn Sie ein Interface erweitern, aber nicht alle implementierenden Klassen anpassen wollen oder können.

Es ist in der Programmierung immer sehr arbeitsaufwändig, wenn ein Interface erweitert wird; alle Klassen, die dieses Interface implementieren, müssen angepasst werden. Mit den Default-Methoden ist es deutlich einfacher geworden.

Ich blende diesen Aspekt jedoch bewusst aus, um dieses Buch möglichst sprach- und versionsunabhängig zu halten.

Beim Observer Pattern und auch beim Mediator Pattern werden Sie eine beliebig große Anzahl von Objekten haben, die miteinander in Beziehung stehen. Schauen Sie sich dort die Weinhandels-Simulation an. Sie können dort die Consumer genauso austauschen wie die Producer. Für den Mediator dort ist es lediglich wichtig, dass er es mit Objekten vom Typ `ConsumerIF` oder `ProducerIF` zu tun hat. Welche konkrete Klasse sich dahinter ver-

birgt, ob ein privater Kunde oder ein Tante-Emma-Laden, spielt keine Rolle. Die Objekte sind lose gekoppelt und lose Kopplung spricht genau wie hohe Kohäsion für ein gutes Design.

Beim Observer Pattern werden Sie sowohl Studenten als auch Angestellte in die Rolle „Beobachter“ bringen. Für die Eventquellen wird es lediglich wichtig sein, dass sie mit Objekten kommunizieren, die der erwarteten Schnittstelle `JobObserver` entsprechen und die Methode `update()` definieren. Sowohl die Anzahl als auch die Art der konkreten Implementierung der Beteiligten können in beiden Projekten variieren. Kann ich Sie davon überzeugen, dass Ihre Implementierung dadurch, dass Sie gegen eine Schnittstelle programmiert haben, wesentlich an Unabhängigkeit und Flexibilität gewinnen wird?

Falls nein, möchte ich als letztes Beispiel auf das Kapitel zum Iterator verweisen. Sie werden sich dort mit Listen beschäftigen. Es gibt eine Vielzahl von Listen: `ArrayList`, `LinkedList` und viele weitere aus der Klassenbibliothek sowie natürlich die Listen, die Sie selbst programmieren. Alle diese Listen implementieren das Interface `List`. Wenn eine Methode nun eine Liste – allgemein gesprochen ein Objekt vom Typ `List` – zurückgibt, müssen Sie sich keine Gedanken über die dahinterstehende Implementierung machen. Die Methode `getList()` im unten stehenden Beispiel gibt irgendeine Liste zurück, deren Klasse Sie nicht kennen. Die Methode `testListe()` ruft die Methode `getList()` auf und lässt sich eine Liste zurückgeben.

Danach wird auf dem zurückgegebenen Objekt die Methode `size()` aufgerufen, die in der Schnittstelle `List` vorgeschrieben ist. An keiner Stelle innerhalb der Test-Methode ist es relevant, ob Sie sich die Größe einer `ArrayList`, einer `LinkedList` oder einer sonstigen anderen Liste zurückgeben lassen.

```
List liste;
void testeListe() {
    liste = getList();
    System.out.println("Anzahl Elemente:" + liste.size());
}

List getList() {
    // ... gibt irgendein Objekt vom Typ "List" zurück
}
```

- ▶ Sie entwickeln guten Code, wenn Sie sich an das Prinzip halten, das die GoF formuliert hat: „Programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung“.

2.3 Single Responsibility Principle (SRP)

Ein Prinzip der objektorientierten Programmierung sagt, dass eine Klasse nur für eine einzige Sache zuständig sein soll – nicht mehr, aber auch nicht weniger. Dieses Prinzip nennt sich Single Responsibility Principle (Prinzip der einzigen Verantwortung). Dieses Prinzip

geht zurück auf Robert Cecil Martin („Uncle Bob“), der das so formuliert hat: „*A class should have only one reason to change*“. Wenn Sie auf der Webseite von cleancoder.com dort auf den Link zu den „Old Articles“ klicken, gelangen Sie zu einer Übersicht vieler alter (und lesenswerter) Artikel von Bob Martin, darunter auch fast am Ende der Liste den Link zu seinem Artikel über das Single Responsibility Principle, der auf Google Docs frei zur Verfügung steht:

https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk

Dort finden Sie das obige Zitat auf Seite 2. Sie werden das Single Responsibility Principle im Kapitel über das Singleton-Pattern in der Praxis kennenlernen. Dieses Pattern wird von manchen Programmierern als Antipattern betrachtet. Einer der Gründe dafür ist, dass das Singleton genau gegen dieses Prinzip verstößt: Da gibt es eine Klasse, die sowohl für die Erzeugung ihrer einzigen Instanz verantwortlich ist als auch für ihre eigentliche Geschäftslogik.

Auch die Chain of Responsibility, die wir in Kap. 6 kennen lernen werden, verstößt gegen dieses Prinzip. In dem Beispiel dort ist es die Aufgabe eines Händlers, Produkte einzukaufen und zu verkaufen. Seine Aufgabe ist es nicht, den nächsten Händler über eine Kaufanfrage zu informieren. Aber warum werden Sie in diesen Beispielen so einfach hinnnehmen können, dass gegen das Prinzip verstoßen wurde?

Auf der einen Seite werden Sie das SRP schätzen, weil es verhindert, dass in einem System eine God-Class entsteht. Darunter versteht man allgemein Klassen mit sehr vielen Funktionen und Zuständigkeiten, die also übermäßig erscheinen – und dadurch sehr schwer zu überblicken sind. Wenn Zuständigkeiten dagegen getrennt sind, werden Ihnen Wartung und Test erleichtert. Auf der anderen Seite sagt niemand, dass Single Responsibility eine heilige Kuh ist, die niemals geschlachtet werden darf. Das Singleton und die Chain wären ohne großen Aufwand nicht zu realisieren gewesen, wenn Sie gegen das Prinzip nicht verstoßen hätten. Auch unabhängig von der Diskussion über Patterns ist es in der Praxis kaum durchführbar, einer Klasse wirklich nur eine einzige Aufgabe zuzuschreiben. Wenn Sie wissen, dass Sie gegen das Prinzip verstoßen und gute Gründe dafür haben, können Sie es auch gern verletzen. Streben Sie danach, Single Responsibility zu erreichen, aber haben Sie im Hinterkopf, dass das Prinzip keinen Selbstzweck hat. Es soll Ihnen helfen, die Kohäsion zu erhöhen und die Koppelung zu lockern. Es soll Sie aber keinesfalls dazu zwingen, ein umständliches Design zu wählen.

2.4 Inheritance may be evil

Diesen Abschnitt sollten Sie beim ersten Lesen des Buches vielleicht besser nur grob überfliegen. Ich erwähne hier viele Muster, die Sie ja erst in den folgenden Kapiteln kennenlernen werden. Schauen Sie hier aber unbedingt am Ende noch mal rein.

Code hat meistens Bestandteile, die sich nie oder selten ändern. Meistens gibt es aber auch Bestandteile, die sich ändern oder ändern können. **Identifizieren Sie die variablen Code-Teile und kapseln Sie diese.** Schauen Sie auf das Strategy Pattern. Sie werden dort

eine Vielzahl von unterschiedlichen Strategien zur Auswahl haben, um ein Array zu sortieren. In der allerersten Version verwenden Sie eine Vielzahl von if-Anweisungen oder eine switch-Anweisung, die für die konkrete Situation die passende Strategie ermitteln. Dieser Ansatz ist aus verschiedenen Gründen unbefriedigend: Sie schleppen in einer Klasse Code mit, den Sie im Zweifel niemals brauchen werden. Wenn Sie eine neue Strategie implementieren wollen, müssen Sie bestehenden, bereits getesteten Code ändern. Wenn eine Strategie geändert werden muss, beträfe diese Änderung den kompletten Code. Fehler werden sich garantiert einschleichen, wenn Sie an mehreren Stellen den Code ändern müssen.

Beim State Pattern bekommen Sie das gleiche Problem. In der Folge definieren Sie in beiden Fällen variable Code-Anteile in eigenen Klassen. Vergegenwärtigen Sie sich auch das Template Method Pattern. Hier müssen Teile des Algorithmus in Subklassen definiert werden. Oder denken Sie an das Command Pattern – Sie kapseln Befehle in eigenen Command-Klassen. Möchten Sie noch mehr Beispiele? Schauen Sie sich den Iterator an. Sie können keinen „Universal-Iterator“ schreiben, weil Sie niemals alle Objekte vom Typ List kennen werden. Also geben Sie die Verantwortung dorthin ab, wo sie hingehört, nämlich in die entsprechende Unterklasse. Anders formuliert: Sie kapseln den Zugriff auf ein Aggregat. Was passiert? Sie identifizieren variable Code-Teile und kapseln sie. In den meisten Fällen, Strategy und State zum Beispiel, definieren Sie die betreffenden Code-Teile in neuen Klassen. In einem anderen Fall – dem Template Method Pattern – wird der variable Code-Teil in eine Unterklasse verschoben.

Generell ist es sinnvoll, Vererbung sehr zurückhaltend einzusetzen. In der Einleitung habe ich Ihnen am Beispiel einer Zoo-Software gezeigt, dass Vererbung Sie unter Umständen in ein nicht überschaubares System führen kann. Als Alternative habe ich Ihnen Komposition vorgeschlagen. Das Vorgehen entspricht genau dem, was Sie beim Strategy Pattern und beim State Pattern noch sehen werden. Und tatsächlich greifen die meisten Patterns auf Komposition anstelle auf Vererbung zurück. Die GoF hat den Grundsatz aufgestellt: „*Ziehe Objektkomposition der Klassenvererbung vor.*“ Ihre Argumentation ist, dass „*Ihre Klassen und Ihre Klassenhierarchien [klein bleiben] und [...] sich mit geringerer Wahrscheinlichkeit zu unkontrollierbaren Monstern aus[wachsen].*“

Im Zusammenhang mit der Vererbung muss noch ein weiteres Prinzip angesprochen werden: das *Liskov Substitution Principle (LSP)*. Es besagt, dass sich Subtypen wie ihre Basistypen verhalten müssen: Der Basistyp muss durch den Subtyp ersetzt werden können. Dieses Prinzip klingt zunächst banal, doch seine Bedeutung darf nicht unterschätzt werden. Denken Sie an den Fall, dass eine Methode des Basistyps keine Exception wirft, die überschriebene Methode in der Subklasse schon. Wann kann diese Situation zum Problem werden? Ein Client wurde auf den Basistyp hin programmiert; er rechnet nicht damit, dass eine Exception geworfen werden könnte. Folglich kann er die Exception der Subklasse unter Umständen gar nicht sinnvoll verarbeiten. Oder – anderes Beispiel: Ein Punkt in einem dreidimensionalen Raum beschreibt seine Koordinaten anders als ein Punkt in einem zweidimensionalen Raum. Es ist daher nicht zulässig, hier eine Vererbungshierarchie zu bilden. Das LSP zwingt uns, jede Vererbungshierarchie genau zu überdenken und jede Subklasse

darauf zu überprüfen, ob sie tatsächlich ein Subtyp der Basisklasse ist: Ein Programmierer **IST EIN** Mensch. Aber: Ein Rechteck **IST KEIN** Quadrat. Das LSP erinnert uns daher auch wieder daran, dass wir mit Vererbung zurückhaltend umgehen müssen.

2.5 Open/Closed Principle (OCP)

Das Offen-Geschlossen-Prinzip geht auf Bertrand Meyer zurück (Meyer, Bertrand (2009): Object-oriented software construction. 2. ed., 15. print. Upper Saddle River, NJ: Prentice Hall PTR. ISBN 978-0136291558.), der fordert, dass Module sowohl offen als auch geschlossen sind. Offen müssen sie sein, um die Funktionalität zu erweitern. Geschlossen müssen sie in der Hinsicht sein, dass Code nicht mehr geändert werden darf, um neue Funktionen zu implementieren. Meyer hat dieses Prinzip vor fast dreißig Jahren beschrieben und dafür verschiedene Lösungsansätze entwickelt. Er entwirft in seinem Buch einige Lösungsansätze, verwirft sie und bietet als Lösung schließlich Vererbung an. Aus heutiger Sicht wird man Vererbung jedoch niemals als Lösung schlechthin betrachten. Wenn Sie heute Systeme entwickeln, die sowohl offen als auch geschlossen sein sollen, greifen Sie auf Komposition und Entwurfsmuster zurück: Sie programmieren gegen Schnittstellen und ziehen Komposition der Vererbung vor. Vor allem beschäftigen Sie sich mit Design Patterns, denn deren heiligste Aufgabe ist es, Ihnen Wege aufzuzeigen, dass Ihre Systeme dem OCP entsprechen.

Ich möchte Ihnen zwei Beispiele nennen, wie Patterns Ihnen helfen, Systeme zu entwickeln, die sowohl offen als auch geschlossen sind. Beim Template Method Pattern definieren Sie einen Algorithmus, der so sensibel ist, dass er nicht mehr geändert werden darf. Ihr Produktmanager würde Bauchschmerzen bekommen, wenn Sie den Code noch mal anfassen wollten. Die Klasse ist vielleicht sogar schon getestet und an die Kunden ausgeliefert. Kurz gesagt: Niemals nie und unter keinen Umständen werden Sie den Code mehr ändern wollen. Er ist geschlossen für Veränderung. Andererseits müssen Sie es Ihrem Kunden ermöglichen, Teile des Algorithmus neu zu definieren. Um das bestehende System zu erweitern, wird er eine Klasse schreiben, die von Ihrer Klasse erbt.

Vererbung ist aber nur ein Weg, um Systeme „offen und geschlossen“ zu halten. Beim Strategy Pattern legen Sie das System so an, dass jede Strategy in einer Klasse gekapselt ist. Wenn das System um einen neuen Sortieralgorithmus erweitert werden muss, ist es ausreichend, wenn Sie eine weitere Klasse definieren, die den gewünschten Algorithmus ausführt. Der Anwender kann den Algorithmus verwenden, ohne bestehenden Code zu ändern.

2.6 Prinzip des rechten Augenmaßes

Im ersten Kapitel habe ich bereits empfohlen, Patterns mit Bedacht einzusetzen, weil diese fast immer voraussetzen, dass Sie zusätzliche Klassen oder Schnittstellen definieren müssen. Es sollte aber das Ziel von Programmierern sein, unspektakulären übersichtlichen

Code zu produzieren. Sie werden regelmäßig entscheiden müssen, ob Sie ein Pattern einsetzen oder nicht. Bei dieser Entscheidung müssen Sie auf Ihre Erfahrung setzen. Es ist immer eine gute Idee, sich kritisch mit dem eigenen Code und dem Code anderer Programmierer zu beschäftigen und sich Gedanken darüber zu machen. Der Einsatz von Mustern ist meist dadurch motiviert, dass Sie künftige Änderungen berücksichtigen möchten. Änderungen sind die Konstante der Softwareentwicklung. Dass Ihr Code wachsen wird, ist quasi ein Naturgesetz. In der Literatur finden Sie daher oft den Rat, dass Sie Ihre Systeme immer mit Patterns aufbauen und zu allen Seiten offenhalten sollen, um für alle möglicherweise in der Zukunft auftretenden Eventualitäten gerüstet zu sein. Diesen Ansatz unterstütze ich nicht in dieser Absolutheit. Wenn ich meine Software für alle Änderungen vorbereiten möchte, ist das erste Release sicher weit davon entfernt, sich auf einfach wartbaren unspektakulären Code zu stützen. Schreiben Sie hier und jetzt gute Programme, die hier und jetzt eingesetzt werden können. Das ist auch in den allermeisten professionellen Situationen genau das, was ein Auftraggeber/Kunde zu bezahlen bereit sein wird. Produzieren Sie daher schlanken Code, der im Bedarfsfall schnell überblickt und geändert werden kann. Hier gilt das KISS-Prinzip: *Keep It Simple, Stupid*, in etwa: *So einfach wie möglich, so komplex wie nötig*.

Agile Softwareentwicklung widerspricht dabei auch nicht dem Einsatz von Entwurfsmustern – da, wo es sinnvoll ist. Das Problem auf der anderen Seite dabei ist, dass Sie ggf. mit zu einfacherem Code bei späteren Änderungen einen höheren Umarbeitungsaufwand bekommen. Mit den heutigen Möglichkeiten des Refactorings (Umbau), das praktisch jede moderne Entwicklungsumgebung meist gut unterstützt, hält sich das aber üblicherweise in Grenzen bzw. ist wenigstens in weiten Teilen „beherrschbar“. Lassen Sie sich dann aber auch von solchen Werkzeugen helfen. Nutzen Sie moderne Entwicklungsumgebungen. Bei dutzenden Klassen kommt man mit einem einfachen Texteditor schon sehr schnell an seine Grenzen, auch wenn es technisch natürlich möglich und legitim ist, ihn zu verwenden.

Jetzt aber genug der Vorrede, in den folgenden Kapiteln befassen wir uns mit den einzelnen Design Pattern, und am Ende zeige ich Ihnen noch in einem letzten Schritt, wie man beispielsweise mehrere Muster miteinander kombinieren kann.

► **Tipp** Probieren Sie's aus:

Sie könnten, wenn Sie ein wenig Zeit investieren möchten, die Muster in den folgenden Kapiteln daraufhin überprüfen, ob Sie die Entwurfsprinzipien wiederfinden.

Recherchieren Sie im Internet nach weiteren Entwurfsprinzipien – es gibt sehr viel mehr als die, die ich Ihnen vorgestellt habe. Werfen Sie mal einen Blick auf die Seite

<http://www.clean-code-developer.de>.

Im Kapitel zum Facade Pattern (Fassade) lernen Sie ein weiteres Entwurfsprinzip kennen.



Singleton

3

Wenn Sie sich in der Vergangenheit schon einmal mit Patterns beschäftigt haben, kennen Sie bestimmt das Singleton Pattern; es ist vermutlich das bekannteste Pattern. Daher beginnen wir einfach mit diesem Pattern, um auch einen „leichten“ Einstieg zu finden.

3.1 Die Aufgabe des Singleton Patterns

Stellen Sie sich vor, dass Sie eine Klasse haben, die etwas definiert, das so einzigartig ist, dass es nur eine Instanz davon geben darf. Auf diese eine Instanz soll über einen globalen Zugriffspunkt zugegriffen werden können. Nehmen Sie als Beispiel die Metaklasse einer Klasse. Für jede Klasse, die in die Java Virtual Machine (JVM) geladen wird, wird die Instanz einer Klasse angelegt, in der beschreibende Meta-Informationen hinterlegt sind. Dieses Objekt ist so einzigartig, dass es nur einmal in der JVM existieren darf. Sie können es mit `String.class`, aber auch mit `(new String()).getClass()` abfragen. Dass es sich beim Rückgabewert um ein und dasselbe Objekt handelt, können Sie beweisen, indem Sie die Referenzen vergleichen. Geben Sie in einer Java Shell folgende Zeile ein:

```
System.out.println(String.class == (new String().getClass()));
```

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_3

Auf der Konsole wird `true` ausgegeben, die Referenzen sind also gleich und damit ist die Objektgleichheit bewiesen. Das deckt sich mit Ihren Erwartungen – es wäre schlecht, wenn es zwei Meta-Klassen einer Klasse gäbe.

Das Singleton Pattern kommt noch an einer anderen Stelle zur Anwendung: Die Laufzeitumgebung Ihrer Applikation ist so einzigartig, dass sie nur einmal existieren darf. Betrachten Sie die API-Dokumentation der Klasse `Runtime`:

„Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method. An application cannot create its own instance of this class.“

Sie werden selbst auf ein Singleton zurückgreifen, wenn Sie die zum Beispiel Benutzerrichte Ihrer Applikation verwalten oder vielleicht einen Cache programmieren. Eine typische Anwendung kann auch ein Thread-Pool sein. Auch Ressourcen wie Icons oder Bilder, die Sie in Ihrem Programm verwenden, müssen nur einmal geladen werden und können in einem Singleton gespeichert werden. Schauen wir uns ein paar Beispiele an, wie Sie das Singleton Pattern realisieren können.

3.2 Eine erste Version des Singletons

Wie lässt sich das Singleton Pattern nun umsetzen? Sie sollen zunächst sicherstellen, dass es nur ein Exemplar der Klasse gibt. Um in Java zu verhindern, dass Anwender Instanzen einer Klasse erzeugen, müssen Sie zunächst den Konstruktor auf `private` setzen.

```
public class MySingleton {  
    private MySingleton() {  
    }  
}
```

Jetzt ist ein Zugriff auf den Konstruktor nur innerhalb der Klasse oder durch andere Instanzen dieser Klasse möglich. Da es keine (andere) Instanz der Klasse gibt, müssen Sie aber zusehen, dass die eine Instanz innerhalb der Klasse erzeugt wird. Definieren Sie also eine Methode, die eine Instanz erzeugt, sofern es noch keine Instanz gibt; die Instanz der Klasse wird an den Aufrufer der Methode zurückgegeben. Diese Methode muss natürlich öffentlich sein. Halten Sie eine Referenz auf diese Instanz in einer statischen Variablen:

```
public class MySingleton {  
    private static MySingleton instance;  
  
    private MySingleton() {  
    }
```

```
public static MySingleton getInstance() {  
    if (instance == null)  
        instance = new MySingleton();  
    return instance;  
}  
}
```

Das war's! Damit ist das Singleton Pattern auf einfache Art und Weise realisiert! Die Instanz der Klasse wird zum spätestmöglichen Zeitpunkt erzeugt, und zwar dann, wenn sie zum ersten Mal gebraucht wird. Daher nennt man dieses Verfahren *Lazy instantiation*, also *verzögertes Laden*.

Sie finden den Code dazu im Beispielprojekt Singleton_1.

In der Anwendung holen Sie sich dann mit `MySingleton single = MySingleton.getInstance()` die Referenz auf die eine Instanz und rufen anschließend mit `single.doSomething()` die eigentliche Funktion des Singletons auf:

```
public static void main(String[] args) {  
  
    var singleton1 = MySingleton.getInstance();  
    singleton1.doSomething();  
  
    var singleton2 = MySingleton.getInstance();  
    singleton2.doSomething();  
}
```

Und hier verwende ich ein in Java 10 hinzugekommenes Feature für lokale(!) Variablen: Die sogenannte „Local-Variable Type Inference“.

Anstatt

```
MySingleton singleton1 = MySingleton.getInstance();
```

Ist Java seit Version 10 mit dem JDK Enhancement Proposal (JEP) 286 in der Lage, für lokale Variablen den Typ selbstständig und ohne weitere Vorgabe durch den Programmierer aus dem Kontext zu ermitteln. Das passiert zur Übersetzungszeit, gefährdet also nicht die Typsicherheit von Java. Im Beispiel hier ist aus dem Kontext ja schon für den Leser ersichtlich, dass wir eine Instanz eines MySingleton haben wollen. Das zusätzliche vorangestellte MySingleton ist damit überflüssig geworden. Es reicht ein

```
var singleton1 = MySingleton.getInstance();
```

um die Variable singleton1 als MySingleton zu erkennen und entsprechenden Code zu erzeugen.

NetBeans schlägt die Umstellung auf var-Deklaration auch in einem Hinweis vor, wenn man mit dem Cursor in einer Deklarationszeile steht, bei der das möglich ist. Dieses Feature erleichtert in vielen Fällen die Lesbarkeit des Codes, wenn man an die zum Teil sehr langen Klassenbezeichnungen in Java denkt:

`BufferedInputStream inputStream = new BufferedInputStream(...);` wird dann einfach zu

```
var inputStream = new BufferedInputStream(...);
```

Ich werde dieses Feature im Folgenden häufig im Beispielcode verwenden, ohne weiter darauf einzugehen. Es ist ziemlich eingängig und nur dann erlaubt, wenn der Typ der lokalen Variablen auch wirklich eindeutig ermittelbar ist. Daher sollte es Ihnen im weiteren Verlauf auch keine Probleme machen.

3.3 Den Zugriff synchronisieren

Die oben beschriebene Klasse läuft wunderbar, solange Sie nicht mit Nebenläufigkeit arbeiten. Was kann passieren? Gehen Sie von folgendem Fall aus: Sie haben zwei Threads, die beide auf die Methode `getInstance()` zugreifen. Der erste Thread kommt bis zu Zeile 9 und stellt fest, dass es noch keine Instanz gibt. Dann entzieht die Virtuelle Maschine ihm die Rechenzeit und lässt Thread 2 auf die Methode zugreifen. Thread 2 darf bis zum Ende der Methode arbeiten und erzeugt eine Instanz der Klasse, die zurückgegeben wird. Dann bekommt Thread 1 wieder Rechenzeit zugewiesen. Das Letzte, woran er sich erinnern kann, ist, dass die Referenz gleich null ist. Er wird in den Block eintreten und ebenfalls eine Instanz erzeugen. Und jetzt kommt genau das, was Sie verhindern wollten – Sie haben zwei Instanzen der Klasse.

Immer, wenn Sie verhindern wollen, dass Code von zwei Threads gleichzeitig ausgeführt werden kann, sperren Sie den betroffenen Code mit einem Lock:

```
public class MySingleton {
    // ... gekürzt
    public static synchronized MySingleton getInstance() {
        if (instance == null)
            instance = new MySingleton();
        return instance;
    }
}
```

Diesen Code finden Sie im Beispielprojekt `Singleton_2`. Die Methode kann erst dann von einem Thread betreten werden, wenn kein anderer Thread (mehr) damit arbeitet. Das Problem ist damit gelöst. Synchronisierung ist jedoch aufwendig, so dass dieses Vorgehen eine unnötige Bremse ist. Stellen Sie sich vor – jedes Mal, wenn Sie sich die eine Instanz

der Klasse holen möchten, wird ein Lock auf die Methode gesetzt. Es muss also eine Alternative geben.

3.4 Double checked locking

Sperren Sie nicht die gesamte Methode, sondern nur den kritischen Teil! Fragen Sie zunächst ab, ob bereits eine Instanz erzeugt wurde. Wenn die Methode zum zweiten Mal aufgerufen wird, ist das Ergebnis des Vergleichs `false`. Daher ist ein Lock nur beim ersten Aufruf, wenn der Vergleich `instance == null` ein `true` zurückliefert, erforderlich. Also legen Sie innerhalb der if-Anweisung einen Block an, der synchronisiert wird. In diesem Block fragen Sie – diesmal threadsicher – ab, ob es eine Instanz der Klasse gibt. Wenn nicht, erzeugen Sie eine. Zum Schluss geben Sie die Instanz zurück. Schauen Sie dazu ins Beispielprojekt `Singleton_3`.

```
public class MySingleton {  
    private static volatile MySingleton instance;  
    // ... gekürzt  
    public static MySingleton getInstance() {  
        if (instance == null) {  
            synchronized (MySingleton.class) {  
                if (instance == null)  
                    instance = new MySingleton();  
            }  
        }  
        return instance;  
    }  
}
```

In NetBeans erhalten Sie übrigens für Zeile 6 („`synchronized (MySingleton.class)`“) einen Hinweis. NetBeans erkennt Double Checked Locking und schlägt noch den Einsatz einer lokalen Variablen vor, um die Performance zu verbessern. Wenn Sie NetBeans nutzen, schauen Sie sich das auch noch mal an. Der Code wird dadurch etwas länger, deswegen lasse ich es hier im Beispiel weg. Dieser Ansatz des Double Checked Lockings ist in der Theorie zwar richtig. Unter Umständen laufen Sie aber auf ein Problem. Stellen Sie sich folgendes Szenario vor: Sie haben zwei Threads; Thread 1 ruft die Methode `getInstance()` auf. In Zeile 5 stellt er fest, dass noch keine Instanz der Klasse erzeugt wurde, und lässt sich den Lock auf den folgenden Block geben. In Zeile 7 prüft er – threadsicher – noch einmal, ob wirklich noch keine Instanz der Singleton-Klasse erzeugt wurde. Falls nicht, erzeugt er mit dem `new`-Operator eine Instanz (Zeile 8). In Abhängigkeit von der Laufzeitumgebung passiert bei der Instanziierung Folgendes: Zuerst wird Speicher angefordert, der Speicher wird dann an die Variable `instance` übergeben, die jetzt ungleich null ist. Erst im nächsten Schritt wird der Konstruktor der Klasse

aufgerufen. Und jetzt beginnt unser Problem zu reifen: Thread 2 betritt vor dem Konstruktorauftruf durch Thread 1 die Methode `getInstance()`. Er stellt fest, dass `instance != null` ist, lässt sich die Instanz zurückgeben und arbeitet mit dieser. Erst später führt Thread 1 den Konstruktor aus, der die Instanz in einen gültigen Zustand versetzt.

Sie können dies verhindern, indem Sie die Variable `instance` als `volatile` deklarieren.

```
public class MySingleton {
    private static volatile MySingleton instance;
    // ... gekürzt
}
```

Die Probleme rund um double-checked locking dürften schon fast eine ganze Dissertation rechtfertigen. Ich möchte Sie an dieser Stelle nur darauf hinweisen, dass diese Lösung zu Laufzeitfehlern führen kann. Es ist nicht gewährleistet, dass das Schlüsselwort `volatile` in jeder virtuellen Maschine korrekt umgesetzt ist. Eine JVM in einer Version unter 1.5 kann diesen Ansatz beispielsweise überhaupt nicht richtig umsetzen. In solchen Fällen sollten Sie dringend überlegen, auf aktuellere JVM-Versionen zu wechseln. Ich hoffe aber, dass ich im Jahre 2021, wo wir bei Java 16 angelangt sind, niemanden mehr überzeugen muss, mindestens auf Java 8 (2014), besser 11 (2018) zu migrieren. Java 11 ist bis zum Erscheinen von Java 17 (für den Herbst 2021 geplant) die aktuellste „Long Term Support“-Version.

3.5 Early instantiation – frühes Laden

Um allen Problemen in nebenläufigen Systemen aus dem Weg zu gehen, können Sie zum frühen Laden greifen. Sie legen – wie gehabt – eine statische Variable an, die die Referenz auf die einzige Instanz hält. Sie initialisieren sie gleich beim Laden. Eine statische Methode gibt diese Instanz zurück. Der Code ist wirklich sehr einfach. Sie finden ihn im Beispielprojekt `Singleton_4`:

```
public class MySingleton {
    private static final MySingleton INSTANCE = new MySingleton();

    private MySingleton() { }

    public static MySingleton getInstance() {
        return INSTANCE;
    }

    public void doSomething() {
```

```
// Verhalten des Objektes  
// ... gekürzt  
}  
}
```

Da keine zweite Instanz der Klasse erzeugt werden soll und gar nicht erzeugt werden kann, ist auch folgende Lösung (im Beispielprojekt Singleton_5) denkbar: Machen Sie die Instanz öffentlich, und Sie sparen sich dabei dann zusätzlich die getInstance-Methode:

```
public class MySingleton {  
    public static final MySingleton INSTANCE = new MySingleton();  
    private MySingleton() {}  
    public void doSomething() {  
        // Verhalten des Objektes  
        // ... gekürzt  
    }  
}
```

Sie gehen mit diesem Ansatz allen Problemen aus dem Weg, die aus der Nebenläufigkeit resultieren. Aber Sie müssen darauf vertrauen, dass die Virtuelle Maschine zuerst die Instanz erzeugt, bevor sie anderen Threads den Zugriff darauf erlaubt – und genau darauf können Sie sich gemäß der Spezifikation auch verlassen.

Dennoch gibt es auch hier drei Dinge zu beachten:

1. Die Instanz wird erzeugt, wenn die Klasse geladen wird. Angenommen, Sie haben eine umfangreiche Initialisierungsroutine, dann wird diese in jedem Fall durchlaufen, auch wenn Sie das Singleton hinterher gar nicht brauchen.
2. Sie haben keine Chance, dem Konstruktor zur Laufzeit Argumente mitzugeben.
3. Das Singleton Pattern sichert zu, dass die Klasse nur einmal pro Klassenlader instanziert wird. Bei einem Webserver kann es sein, dass die Klasse in mehr als einem Klassenlader innerhalb einer Virtuellen Maschine geladen wird; dann haben Sie doch wieder mehr als eine Instanz der Singleton-Klasse. Ohne näher darauf eingehen zu wollen, begegnet man solchen Sachverhalten heute mit Dependency Injection.

3.6 Singleton – Das UML-Diagramm

In Abb. 3.1 sehen Sie das (recht einfache) UML-Diagramm für das Singleton aus dem Beispielprojekt Singleton_4, also mit der getInstance-Methode.

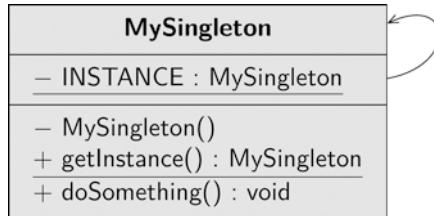


Abb. 3.1 UML-Diagramm des Singleton Pattern (Beispielprojekt Singleton_4)

3.7 Antipattern

Sie kennen jetzt das Singleton Pattern und wissen, wie Sie es umsetzen können. Sie kennen die Vor- und Nachteile der verschiedenen Realisierungen. Es gibt aber grundlegende Einwände gegen das Singleton Pattern. Ich meine damit Einwände gegen das Pattern als Definition und nicht gegen die gezeigte Realisierung. Suchen Sie in der Suchmaschine Ihres Vertrauens nach dem Begriff „evil singleton“. Ich erhalte dabei über 2.000.000 Treffer.

3.7.1 Kritik am Singleton

Ein paar der Kritikpunkte möchte ich Ihnen vorstellen.

- Jede Klasse soll sich auf genau eine Aufgabe konzentrieren: Single Responsibility Principle. Singleton verletzt dieses Prinzip dadurch, dass die Klasse sich sowohl um die Geschäftslogik kümmern muss als auch um die eigene Objekterzeugung.
- Genau wie bei globalen Variablen ist die Abhängigkeit nicht sofort ersichtlich. Ob eine Klasse von einer Singleton-Klasse Gebrauch macht, lässt sich nicht aus der Schnittstelle erkennen, sondern nur aus dem Code.
- Die Kopplung wird erhöht.
- Ein zu extensiver Gebrauch von Singleton verleitet den Programmierer dazu, prozedural zu programmieren.
- Sofern die Instanz eigene Attribute hat, sind diese in der kompletten Anwendung verfügbar. Es ist fraglich, ob es Daten gibt, die tatsächlich „überall“ gebraucht werden: in der View, im Controller, im Model und in der Persistenz-Schicht.
- Sie schränken sich beim Gebrauch von Singletons selbst ein: Sie können keine Subklassen von Singleton-Klassen erstellen. Wenn Sie eines Tages doch eine zweite Instanz brauchen, müssen Sie den Code wieder umschreiben.
- Singletons schaffen so etwas wie einen globalen Zustand. Damit erschweren Sie sich das Testverfahren.

In Anbetracht dieser Punkte halten viele Programmierer das Singleton Pattern für ein Antipattern.

3.7.2 Ist Singleton ein Antipattern?

Antipatterns beschreiben ungeeignete Vorgehensweisen, sozusagen Negativbeispiele. Jedes Antipattern kann genauso formal beschrieben werden wie ein Design Pattern. Sie erkennen daran, warum es nicht gut ist, auf eine bestimmte Art und Weise vorzugehen. Antipatterns helfen, aus Fehlern anderer zu lernen.

Die Argumente gegen die Verwendung des Singleton sind sicher nachvollziehbar. Es gibt aber Aufgaben, die ohne Singleton nur schwer zu lösen wären. Einer pauschalen Bewertung möchte ich mich nicht anschließen – Sie werden im Einzelfall entscheiden und sich bei jeder Singleton-Klasse überlegen müssen, ob diese wirklich erforderlich ist.

3.8 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Ein Singleton wird eingesetzt, wenn Sie eine Klasse haben, von der es nur eine Instanz geben darf.
- Für diese eine Instanz muss es einen globalen Zugriffspunkt geben.
- Verzögertes Laden (lazy instantiation) erzeugt die Instanz, wenn sie gebraucht wird.
- Gleichzeitige Zugriffe – Nebenläufigkeit – auf das Singleton führt insbesondere bei der Erzeugung zu Laufzeitfehlern und müssen entsprechend abgesichert werden.
- Das Frühzeitige Laden (early instantiation) vermeidet die Probleme der Nebenläufigkeit, hat aber andere Nachteile.
- Antipatterns beschreiben Negativbeispiele, also ungeeignete Vorgehensweisen.

Zweckbeschreibung

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

„Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.“



Template Method

4

Das Pattern „Template Method“ gehört zu den Verhaltensmustern und ist recht eingängig. Es hilft Ihnen, wenn Sie einen Algorithmus in Unterklassen unterschiedlich implementieren müssen, ihn aber andererseits nicht ändern dürfen.

4.1 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.

4.1.1 Ein erster 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 unten ab. Den Rest finden Sie im Beispielprojekt `Template_1`. Die Methode `run()` definiert den Algorithmus: eingeben, konvertieren, ausgeben. Der Quellcode der Klasse `Lowercase` ist identisch. Lediglich der fett gedruckte Teil unterscheidet sich.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_4

```

public class Uppercase {
    public final void run() {
        final var eingabe = textEingeben();
        final var konvertiert = convert(eingabe);
        drucke(konvertiert);
    }

    private String textEingeben() {
        final var message = „Bitte geben Sie den Text ein:“;
        return JOptionPane.showInputDialog(message);
    }

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

    private 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.

4.1.2 Der zweite Ansatz

Jetzt bauen wir eine abstrakte Oberklasse, die einerseits den Algorithmus definiert, andererseits die Methoden, die in beiden Klassen identisch sind. Da die Definition des Algorithmus für alle Unterklassen verbindlich sein soll, bietet es sich an, die Methode `run()` final zu deklarieren. Diesen Code finden Sie im Beispielprojekt `Template_2`.

```

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

```

```
private final String textEingeben() {  
    return JOptionPane.showInputDialog(„Bitte geben Sie den Text ein:“);  
}  
  
protected abstract String convert(String eingabe);  
  
private final void drucke(String text) {  
    System.out.println(text);  
}  
}
```

Die Unterkasse UppercaseConverter und LowercaseConverter überschreiben lediglich die abstrakte Methode convert(). Exemplarisch zeige ich hier die Klasse LowercaseConverter.

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

Im nächsten Abschnitt nehmen wir dieses Projekt in Betrieb.

4.1.3 Das Hollywood-Prinzip

Der Client kann jetzt durch Instanziierung aus der entsprechenden Unterkasse sehr leicht den gewünschten Converter auswählen:

```
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 eigentliche Funktionalität in der Unterkasse auf. Dieses Vorgehen nennt die GoF das Hollywood-Prinzip: „Rufen Sie uns nicht an – wir rufen Sie an!“

Beachten Sie auch, dass an dieser (oben fett markierten) Stelle die Deklaration der Variablen `eingabe` NICHT mit einem `var` funktioniert. Wir wollen hier eine lokale Variable vom Typ der Oberklasse haben, um dann Instanzen der diversen Unterklassen beliebig zuweisen zu können.

Wenn Sie an dieser Stelle `var` verwenden, erkennt der Compiler den Typ `LowerCaseConverter` und weist ihn der Variable zu. Bis dahin ist alles ok. Wenn Sie dann aber später mit der gleichen Variablen einen `UppercaseConverter` bauen wollen, laufen Sie auf einen Fehler. Zum Glück aber schon zur Compile-Zeit.

4.1.4 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 Beispielprojekt `Template_3` gibt es eine solche 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.

```
public abstract class Eingabe {  
    public final void run() {  
        var eingabe = textEingeben();  
        var 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:

```
public class UppercaseConverter extends Eingabe {  
    // ... gekürzt  
  
    @Override  
    protected boolean speichern() {  
        return true;  
    }  
}
```

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

```
public class LowercaseConverter extends Eingabe {  
    // ... gekürzt  
    @Override  
    protected boolean speichern() {  
        var frage = „Soll der Text gespeichert werden?“;  
        var antwort = JOptionPane.showConfirmDialog(null, frage);  
        return antwort == JOptionPane.YES_OPTION;  
    }  
}
```

Hook-Methoden werden auch Einschub-Methoden genannt. Sie bieten 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.

4.2 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 Beobachter, als `ListDataListener`, beim Model registrieren und deregistrieren 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.

Im Verzeichnis zu diesem Kapitel finden Sie das Beispielprojekt `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 Deregistrieren 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 diese immer selbst definieren. Schauen Sie sich dazu auch das Beispielprojekt `AbstractListModel_Beispiel` noch einmal an.

4.3 Template Method – Das UML-Diagramm

Das UML-Diagramm des Template Method Patterns zum Beispielprojekt `Template_3` finden Sie in Abb. 4.1.

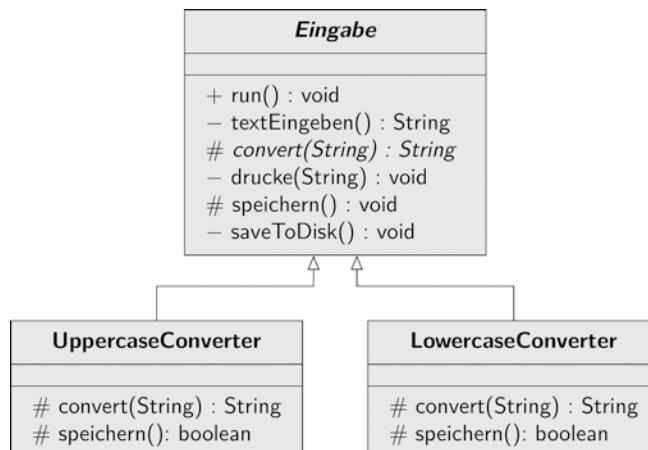


Abb. 4.1 UML-Diagramm des Template Method Pattern (Beispielprojekt `Template_3`)

4.4 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 „Template Method“ 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.“



Observer

5

Sie lernen jetzt ein Pattern kennen, das ähnlich leicht und intuitiv verständlich ist wie z. B. das Singleton oder das Template Method Pattern: Das Observer Pattern. Es geht darum, dass Sie ein Objekt haben, dessen Zustand sich ändert (Ereignisquelle); eine beliebige Anzahl anderer Objekte (Beobachter oder Listener) möchte über diese Zustandsänderungen informiert werden.

5.1 Einleitung

Wie werden Informationen zielgerichtet und adressatengerecht verteilt? Denken Sie an einen Newsletter-Verteiler. Für jedes Thema, das von Interesse sein könnte, gibt es einen Newsletter. Wenn jemand eine neue Wohnung sucht, meldet er sich im Verteiler A an; wenn jemand einen neuen Job sucht, meldet er sich im Verteiler B an. Jeder bekommt nur die Information, die ihn interessiert, und zwar exakt dann, wenn sie veröffentlicht wird. Dieses Prinzip, das auch Publish/Subscribe genannt wird, wird durch das Observer Pattern beschrieben.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_5

5.2 Eine erste Realisierung

Das Beispielprojekt Observer_01 führt Ihnen eine erste Realisierung des Patterns vor. Sie haben jemanden, der Nachrichten bekannt geben will, eine Ereignisquelle, im Beispiel eine Jobbörsse. Diese Klasse ist in der Lage, Objekte vom Typ `JobObserver` als Interessent, bzw. Beobachter, zu registrieren; dieser Typ wird durch das Interface `JobObserver` definiert, das die Methode `neuesAngebot()` vorschreibt.

```
public interface JobObserver {
    void neuesAngebot(String job);
}
```

Die Ereignisquelle, also die Jobbörsse, definiert drei wesentliche Methoden: Eine, um Observer zu registrieren, und eine, um sie zu deregistrieren. Außerdem eine Methode, die einen neuen Job speichert und alle Observer darüber informiert. Das Speichern der Jobs ist in der Beispielimplementierung eigentlich nicht erforderlich. Aber wenn Sie z. B. in der Jobbörsse Auswertungen über die gemeldeten Jobs hinzufügen möchten, brauchen Sie das natürlich.

```
public class JobBoerse {
    private final List<JobObserver> observerList = new ArrayList<>();
    private final List<String> jobList = new ArrayList<>();

    public void addJob(String job) {
        jobList.add(job);
        observerList.forEach(tempJobObserver -> {
            tempJobObserver.neuesAngebot(job);
        });
    }

    public void addObserver(JobObserver jobObserver) {
        observerList.add(jobObserver);
    }

    public void removeObserver(JobObserver jobObserver) {
        observerList.remove(jobObserver);
    }
}
```

Sie sehen in dem obigen Code, dass die Information an alle Beobachter über die Methode `forEach` läuft. Und für jeden `tempJobObserver` aus dieser Liste wird dann die über den Lambda-Ausdruck zugewiesene Methode `neuesAngebot()` aufgerufen. Sie können an dieser Stelle natürlich auch eine normale `for`-Schleife verwenden:

```
for(JobObserver temJobObserver : observerList)
    tempJobObserver.updateJob(job);
```

Sie bietet die gleiche Funktionalität. Der Unterschied ist, dass es sich bei der for-Schleife um einen externen Iterator (den Sie schreiben), bei der forEach-Methode um einen internen Iterator (den Sie nur aufrufen, weil er schon eingebaut ist) handelt. for Each funktioniert übrigens auch mit Sets, Queues und Maps.

Die Observer definieren, was passiert, wenn die Methode `neuesAngebot()` aufgerufen wird. Parameter dieser Methode ist der neue Job. Die Methode entscheidet im Beispiel zufallsgesteuert, ob der Beobachter sich um den Job bewirbt oder nicht. Unten sehen Sie das gekürzte Listing für die Klasse Student. Darüber hinaus finden Sie im Projekt außerdem die Klasse Arbeitnehmer; damit sollen Angestellte modelliert werden, die einen Nebenjob suchen. Auch sie müssen dazu das Interface `JobObserver` und damit eine eigene Version der Methode `neuesAngebot()` implementieren. Im Beispiel habe ich nur einen kleinen Unterschied gemacht: Studenten bewerben sich mit 80 % Wahrscheinlichkeit auf eine neue Stelle, Arbeitnehmer nur mit 50 % Wahrscheinlichkeit. Schließlich haben die ja schon eine Stelle.

```
public class Student implements JobObserver {
    @Override
    public void neuesAngebot(String job) {
        var zufallszahl = (int) (Math.random() * 10);
        var antwort = "Student " + name + " bewirbt sich ";
        if (zufallszahl <= 8)
            antwort = antwort + "um den Job";
        else
            antwort = antwort + "nicht um den Job";
        System.out.println(antwort);
    }
}
```

Das Testprogramm, das Sie im Projekt finden, legt zwei `JobObserver` an und testet, welcher von beiden unter welchen Umständen über einen neuen Job informiert wird. Die erwarteten Konsolenausgaben finden Sie in den Kommentaren der `main`-Methode des Testtreibers.

5.3 Den Ansatz erweitern

Drei Dinge sind im Leben eines Programmierers sicher: der Tod, die Steuern und Änderungen der Anforderungen an die eigene Software. Auch in diesem Projekt werden die Anforderungen wachsen.

- Tipp Um sich die größtmögliche Flexibilität zu erhalten, programmieren Sie immer gegen Schnittstellen, niemals gegen Implementierungen. Bitte beachten Sie: Der Begriff „Schnittstelle“ ist in der Sprache der Entwurfsmuster nicht ausschließlich mit einem Interface gleichzusetzen und erst recht nicht mit Schnittstellen im Sinne von Datenübertragung oder Funktionsaufrufen zwischen Systemen. Wenn von einer Schnittstelle die Rede ist, kann auch eine abstrakte Klasse gemeint sein. Folglich können Sie eine Schnittstelle „erweitern“ oder „implementieren“, was in der Patterns-Sprache den gleichen Vorgang beschreibt.

Diese Redeweise hat sich eingebürgert. Treffender wäre es sicher, wenn man von „Rolle einnehmen“ statt von „Schnittstelle implementieren“ spricht – die Klasse Student könnte dann die Rolle eines JobObserver einnehmen.

Sie haben im vorigen Projekt Observer_01 schon den Vorteil gesehen, dass die Jobbörse gar keine Ahnung hat, was für Objekte sie registriert. Für sie genügt es, dass die Objekte vom Typ JobObserver sind – das können Studenten oder Arbeitnehmer sein.

Welche Änderung könnte sich ergeben? Es könnte sein, dass die Studenten sich nicht nur bei der Jobbörse der Hochschule, sondern auch bei einer Jobbörse der Arbeitsagentur registrieren können sollen. Im Beispielprojekt Observer_02 finden Sie das Interface JobAnbieter, das von den Eventquellen implementiert wird. Für den Client, in unserem Fall die main-Methode der Testklasse, kann es unter Umständen sinnvoll sein, nicht an eine konkrete Implementierung (Jobbörse oder Arbeitsagentur) gebunden zu sein, sondern allgemein an Objekte vom Typ JobAnbieter. Er kann jetzt die konkreten Implementierungen austauschen und beispielsweise einen Job bei beiden Anbietern gleichzeitig einstellen:

```
var jobBoerse = new JobBoerse();
var arbeitsamt = new Arbeitsagentur();
var alleAnbieter = new JobAnbieter[] {
    jobBoerse, arbeitsamt
};
// ... gekürzt
var job = "Aushilfe im Theater";
System.out.println("\nNeuer Job: " + job);
for (var tempAnbieter : alleAnbieter)
    tempAnbieter.addJob(job);
```

Auch hier verwende ich die Typ-Inferenz mit „var“. Egal, ob dort „JobBoerse jobBoerse = new JobBoerse();“ oder „JobAnbieter jobBoerse = new JobBoerse();“ oder „var jobBoerse = new JobBoerse();“ steht, es wird immer ein Objekt des Typs JobBoerse angelegt. Das können Sie mit einem zwischendurch eingestreuten „System.out.println(jobBoerse.getClass());“ einfach überprüfen. Wie Sie sehen, funktioniert die Typ-Inferenz auch mit Feldern. Das Feld mit den beiden JobAnbietern wird ebenfalls korrekt erzeugt.

Eine Abwandlung dieses Beispiels ist noch denkbar: In den vorigen beiden Beispielen haben Sie einen Job als String übergeben und diesen an die Observer weitergereicht. Dieses Vorgehen wird Push-Methode genannt. Möglich wäre auch, dass die Eventquelle den Observern nur mitteilt, dass neue Informationen vorliegen, ohne diese gleich zu übergeben. Es ist dann Sache des Observers, die vollständigen Informationen bei der Eventquelle anzufordern. Dieses Verfahren wird Pull-Methode genannt.

Im Beispielprojekt `Observer_03` suchen die Studenten und Arbeitnehmer keine Jobs, sondern Wohnungen. Wohnungsangebote sind mit ihren Exposés und Grundrissen sehr umfangreiche Objekte, die nur auf Anforderung verschickt werden. Die Observer entscheiden zunächst zufallsgesteuert, ob sie die Informationen überhaupt anfordern. Danach entscheiden sie zufallsgesteuert, ob sie sich für die Wohnung interessieren. Bitte analysieren Sie das Projekt selbstständig.

5.4 Observer in der Java Klassenbibliothek

In der Klassenbibliothek von Java gibt es das Interface `Observer`. Dieses Interface definiert die Rolle eines Beobachters und schreibt die Methode `update()` vor. Als Parameter werden ein Objekt vom Typ `Observable` und ein Objekt vom allgemeinen Typ `Object` erwartet. Der Methode wird ein Link auf die Eventquelle übergeben, um den Beobachter in die Lage zu versetzen, bei der Eventquelle weitere Informationen anzufordern. Außerdem kann die Eventquelle auf ihren Datentyp (Methode `getClass()`) abgefragt werden. Der Parameter vom Typ `Object` enthält beispielsweise einen Job oder eine Wohnung.

Die Eventquelle ist also vom Typ `Observable`. Üblicherweise deutet die Endung `-able` auf ein Interface hin, hier jedoch nicht. `Observable` ist eine Klasse, die erweitert werden kann. Wie Sie es von Ihrer eigenen Implementierung gewöhnt sind, gibt es in der Klasse `Observable` Methoden, um `Observer` zu registrieren und zu deregistrieren. Im Unterschied zu Ihrer Implementierung muss, bevor Sie `Observer` informieren, die Methode `setChanged()` aufgerufen werden, die ein internes Flag setzt. Nur wenn dieses Flag gesetzt ist, übermittelt die Methode `notifyObservers()` die Änderungen tatsächlich an die `Observer`. Dieses Vorgehen kann sinnvoll sein, wenn viele Änderungen an der Datenbasis erfolgen, die `Observer` aber erst informiert werden sollen, wenn alle Änderungen abgeschlossen sind.

Das Interface `Observer` und die Klasse `Observable` sind allerdings als „deprecated“ – veraltet – gekennzeichnet und sollten möglichst nicht mehr verwendet werden. Es ist nicht typsicher und auch nicht thread-sicher.

Für spezielle `Observer`-Varianten gibt es in der Klassenbibliothek einmal das auf thread-sichere parallele Verarbeitung von Daten ausgelegte Package `java.util.Flow`, in dem Sie die Interfaces `Publisher` und `Subscriber` finden werden. Zusammen mit einem `Executor` und einer `CompletableFuture` lässt sich dann z. B. die Verarbei-

tung von vielen einzelnen Datenpaketen aus einer Quelle auf parallele Threads aufteilen, die ihr jeweiliges Arbeitspensum dann selbst anfordern können. Der Code dafür wird dann schon sehr aufwändig und sprengt den Rahmen für dieses Buch. Im Internet finden Sie dazu aber ausreichend viele nachvollziehbare Beispiele.

Zum anderen werden in der Programmierung von Benutzeroberflächen Ereignisse durch den Anwender ausgelöst, auf die die Anwendung reagieren muss: Das Drücken von Tasten, das Klicken mit der Maus auf Buttons oder Menüpunkte oder auch das Einfügen von Daten in Listen, die dann wieder eine Aktualisierung der Benutzeroberfläche veranlassen müssen. Speziell für letzteres gibt es in der JavaFX-Bibliothek seit der Version 2.0 auch spezielle Observable-Interfaces für Maps, Sets, Arrays oder Listen, z. B. javafx.collections.ObservableList.

Mit thread-sicheren Oberservern befassen wir uns im nächsten Abschnitt, und Benutzeroberflächen betrachten wir im Anschluss daran auch noch mal.

5.5 Nebenläufiger Zugriff

Die Lösung, die Sie im Projekt Observer_03 gesehen haben, funktioniert tadellos. Wenn Sie aber mit einem nebenläufigen System damit arbeiten möchten, stoßen Sie auf Probleme. Stellen Sie sich vor, ein Thread übergibt gerade eine neue Wohnung und alle Observer werden darüber informiert. Gleichzeitig versucht ein anderer Thread, einen Beobachter abzumelden. Es gibt dann zwei Threads, die gleichzeitig auf die Liste der Beobachter zugreifen möchten. Öffnen Sie das Beispielprojekt Observer_04. Ich habe es auf die nötigsten Klassen reduziert, so dass Sie beispielsweise nur Arbeitnehmer und keine Studenten haben. Starten Sie die main-Methode der Klasse ObserverSim. Diese Version startet kurz hintereinander zwei Threads (WohnungsThread und AbmeldenThread), die beide mit der Liste der Observer arbeiten können. Die geschilderte Situation, dass zwei Threads auf die Datenbasis gleichzeitig zugreifen möchten, wird hier bewusst provoziert. Es wird nach kurzer Zeit eine concurrentModificationException geworfen. Möglicherweise müssen Sie die main-Methode mehrmals starten.

Eine andere problematische Situation kann entstehen, wenn Observer über eine Änderung informiert werden und sich entscheiden, sich als Beobachter abzumelden. Diese Situation finden Sie im Beispielprojekt Observer_05. In der update-Methode wird zufalls gesteuert entschieden, ob der Beobachter weiterhin informiert werden möchte:

```
@Override  
public void updateFlat(WohnungsBoerse boerse) {  
    var zufallszahl = (int) (Math.random() * 10);  
    if (zufallszahl <= 6) {  
        var wohnung = boerse.getDetail();  
        if (zufallszahl < 5) {  
            // ... gekürzt
```

```

        boerse.removeObserver(this);
    }
}
// ... gekürzt
}

```

Auch in diesem Fall wird irgendwann eine `ConcurrentModificationException` geworfen, wie Abb. 5.1 zeigt. Das Problem hier ist, dass die Benachrichtigung aus dem Iterator kommt, der durch die Liste aller Observer geht. Wenn jetzt ein Observer währenddessen den Versuch startet, sich löschen zu lassen, veranlasst er die Methode `removeObserver` zu einer Manipulation der Liste, die aber gerade vom Iterator, der ja ggf. noch weitere Observer benachrichtigen muss, verwendet wird. Und genau da schlägt dann die Exception zu.

In den kommenden Abschnitten werde ich Ihnen zeigen, wie Sie Schwierigkeiten aus Nebenläufigkeit vermeiden.

5.5.1 Zugriffe synchronisieren

Das Beispielprojekt `Observer_06` bietet eine naheliegende Lösung, nämlich Zugriffe auf die Datenbasis zu synchronisieren.

```

public class WohnungsBoerse {
    private final List<WohnungsObserver> observerList = new ArrayList<>();
    // ... gekürzt
    public synchronized void addWohnung(Wohnung wohnung) {
        // ... gekürzt
    }

    public synchronized void addObserver(WohnungsObserver beobachter) {
        // ... gekürzt
    }

    public synchronized void removeObserver(WohnungsObserver observer) {
        // ... gekürzt
    }
}

```



Abb. 5.1 Nebenläufiger Zugriff kann eine Exception auslösen (Screenshot aus NetBeans)

Sind damit alle Probleme gelöst? Nein! Beobachter können sich nicht an- oder abmelden, solange eine Information an Beobachter übermittelt wird. Damit ist das Problem aus Projekt Observer_04 gelöst. Das Projekt Observer_05 würde weiterhin eine Exception werfen. Außerdem müssen Sie sich folgende Situation vorstellen: Sie informieren gerade alle Ihre Kunden über eine wichtige Neuerung. Da die Information sehr umfangreich ist, dauert es bestimmt eine halbe Stunde, bis die Liste aller Observer abgearbeitet ist; es wäre unschön, wenn sich in dieser Zeit keine neuen Kunden registrieren könnten. Es muss also noch eine andere Lösung geben. Übrigens synchronisiert die Klasse Observable, die ich Ihnen in Abschn. 5.4 vorgestellt habe, die Liste der Observer auf genau diese Weise.

5.5.2 Die Datenbasis kopieren

Das Beispielprojekt Observer_07 zeigt Ihnen eine andere Lösung: Die Liste der Observer wird vor der Benachrichtigung kopiert.

```
public void addWohnung(Wohnung wohnung) {  
    this.wohnung = wohnung;  
    List<WohnungsObserver> tempList;  
    synchronized (this) {  
        tempList = List.copyOf(observerList);  
    }  
    for (var tempObserver : tempList) {  
        tempObserver.updateFlat(this);  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException ex) {  
            // nichts tun  
        }  
    }  
}
```

Dadurch ist die Liste nur für einen kurzen Moment blockiert. Wenn Sie die Benachrichtigungen verschicken, wird immer die kopierte Liste informiert; wenn sich ein Beobachter zeitgleich bei der ursprünglichen Liste an- oder abmeldet, gibt es keinen Konflikt beim Zugriff. Nachteil dieser Lösung ist natürlich, dass die Observer-Liste bei jedem Update mehr oder weniger aufwendig kopiert werden muss.

An dieser Stelle verwende ich übrigens eine Funktion, die neben einigen anderen in Java 10 hinzugekommen ist: List.copyOf(). Dies ist die einfachste und schnellste Methode, eine Liste zu kopieren. Die Syntax ist sehr einfach, und gleiche Funktionen gibt es auch für die Klassen Set und Map: Set.copyOf() und Map.copyOf().

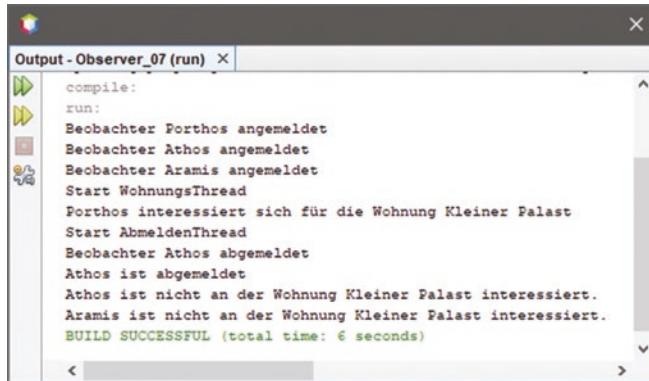


Abb. 5.2 Beobachter meldet sich ab und wird trotzdem noch einmal informiert

Wenn Sie das Projekt testen, werden Sie zudem feststellen, dass ein Beobachter, der sich gerade abgemeldet hat, trotzdem noch einmal informiert wird – lassen dazu die main-Methode ein paar Mal ablaufen (vgl. Abb. 5.2).

5.5.3 Einsatz einer threadsicheren Liste

Wenn Sie nicht selbst kopieren möchten, können Sie auch eine thread-sichere `CopyOnWriteArrayList` verwenden. Das Beispielprojekt `Observer_08` demonstriert den Einsatz dieser Klasse.

```
public class WohnungsBoerse {  
    private final List<WohnungsObserver> observerList =  
        new CopyOnWriteArrayList<>();  
    // ... gekürzt
```

Die Arbeit mit einer `CopyOnWriteArrayList` ist allerdings vergleichsweise teuer, weil immer dann, wenn ein neues Element eingefügt oder ein bestehendes gelöscht werden soll, die komplette Datenbasis kopiert wird. Insofern greift die gleiche Kritik wie im Projekt `Observer_07`. Allerdings gibt es keine triviale Lösung für dieses Problem, und immerhin ist die Implementation in der Bibliothek `java.util.concurrent` sehr effizient; Sie müssen sich selbst keine weiteren Gedanken um die genaue Implementierung machen. In der üblichen Praxis dürfte es die beste Lösung sein, auf eine `CopyOnWriteArrayList` zurückzugreifen.

5.6 Observer als Listener

Einen anderen Ansatz verfolgen Sie, wenn Sie das Ereignis in eine andere Klasse einpacken; das ist, wie wenn Sie einen Brief in einen Umschlag stecken und abschicken. Die serialisierbare Klasse `EventObject` aus dem Paket `java.util` erwartet als Parameter im Konstruktor eine Referenz auf dasjenige Objekt, das das Event abschickt. Die Methode `getSource()` gibt diese Quelle zurück. Die `toString`-Methode aus der Klasse `Object` wird sinnvoll überschrieben. Das Interface `EventListener` aus dem gleichen Paket schreibt keine Methoden vor, sondern definiert lediglich eine Rolle.

Im Beispielprojekt `ListenerDemo` gibt es die Klasse `JobEvent`, die `EventObject` erweitert. Sie erwartet als Parameter die Quelle und ein String-Objekt, das den neuen Job beschreibt. Selbstverständlich könnten Sie hier anstelle des Strings auch eine Wohnung vorsehen. Die Quelle wird an die Super-Klasse weitergereicht, der Job wird in einem eigenen Datenfeld gespeichert.

```
public final class JobEvent extends EventObject {  
    private final String job;  
  
    JobEvent(Object jobAnbieter, String job) {  
        super(jobAnbieter);  
        this.job = job;  
    }  
  
    @Override  
    public String toString() {  
        return job;  
    }  
}
```

Das Interface `JobListener` erweitert das Interface `EventListener` und schreibt die Methode `updateJob()` vor, der ein Objekt vom Typ `JobEvent` übergeben werden muss. Zwei Klassen – die Ihnen bekannten Studenten und Arbeitnehmer – implementieren auf jeweils individuelle Weise dieses Interface.

```
public interface JobListener extends EventListener {  
    void updateJob(JobEvent jobEvent);  
}
```

Die Klasse `Arbeitsamt` führt eine Liste von `JobListenern`. Wenn ein neuer Job eingestellt wird, erzeugt die Klasse ein Event vom Typ `JobEvent` und übermittelt es allen `JobListenern`.

```
public class Arbeitsamt {  
    private final List<JobListener> listener =  
        new CopyOnWriteArrayList<>();  
  
    public void addJob(String neuerJob) {  
        JobEvent jobEvent = new JobEvent(this, neuerJob);  
        listener.forEach((tempListener) -> {  
            tempListener.updateJob(jobEvent);  
        });  
        // ... gekürzt  
    }  
}
```

Schauen wir uns im nächsten Abschnitt an, wie diese Variante praktisch eingesetzt werden kann.

5.7 Listener in der GUI-Programmierung

Wenn Sie sich die Klasse `EventObject` in der API-Dokumentation betrachten, sehen Sie, dass sehr viele Event-Klassen, die in der GUI-Programmierung benötigt werden, hier von abgeleitet sind: `ListDataEvent`, `ListSelectionEvent`, `AWTEvent` und etliche mehr. Ein Event, das `TableModelEvent`, möchte ich jetzt vorstellen. Im letzten Kapitel habe ich kurz dargestellt, wie eine `JList` ihre Daten bei der Datenbasis abfragt. Das Zusammenspiel einer `JTable` und deren Datenbasis ist ähnlich: Sie haben eine Datenbasis, die Sie für das `JTable` so beschreiben müssen, dass es in der Lage ist, die Daten anzuzeigen. Die Brücke zwischen beiden ist das Interface `TableModel`. Das `JTable` erwartet ein Objekt vom Typ `TableModel`, das Sie ihm mit der Methode `setModel()` übergeben. Bei diesem Objekt meldet sich das `JTable` als Listener an, weshalb Sie die Methoden `addTableModelListener()` und `removeTableModelListener()` implementieren müssen. Mit `getColumnCount()` geben Sie zurück, wie viele Spalten angezeigt werden sollen. Die Methode `getColumnName()` returniert die Überschrift einer Spalte. Da Sie jeden beliebigen Datentyp zurückgeben können, gibt `getColumnClass()` Auskunft darüber, welche Klasse einem Objekt in einer bestimmten Spalte zugrunde liegt. Die Methode `getRowCount()` entspricht der Methode `getSize()` des `ListModel`; sie gibt die Anzahl Zeilen einer Spalte zurück. Eine Zelle in einer Tabelle kann grundsätzlich editiert werden; wenn Sie dies unterbinden wollen, muss die Methode `isCellEditable()` einen `false`-Wert liefern. Die Methoden `getValueAt()` und `setValueAt()` beschreiben einerseits, welcher Wert in einer bestimmten Zelle enthalten ist, andererseits wie die Datenbasis reagieren soll, wenn eine Zelle editiert wurde. Jede Zelle wird eindeutig über ihre Spalte und Zeile definiert. Im Beispielprojekt `MVC_1` werden die eingestellten Jobs in einer Tabelle angezeigt (Abb. 5.3).

Abb. 5.3 Anzeige der Jobs in einer Tabelle – Projekt MVC_1



Um diese Anzeige zu generieren, gibt es zwei Klassen, eine Anzeige und eine Datenbasis. Die Datenbasis beschreibt auf eine für die Klasse `JTable` verständliche Weise die Daten und die Spaltenüberschrift. Daneben gibt es zwei Methoden, die definieren, was passiert, wenn ein neuer Job eingestellt wird. Der Methode `addJob()` übergeben Sie einen String mit dem neuen Job; sie fügt den Job zur Job-Liste hinzu und ruft die private Methode `fireStateChanged()` auf. Diese Methode generiert ein `TableModelEvent`-Event und übergibt dieses an die Listener, die dann ihre Anzeige aktualisieren.

```
public class Datenbasis implements TableModel {
    private final List<TableModelListener> listener = new ArrayList<>();
    private final List<String> jobListe = new ArrayList<>();
    private final String[] headers = new String[] {
        "Jobs"
    };

    public void addJob(String job) {
        jobListe.add(job);
        this.fireStateChanged();
    }

    private void fireStateChanged() {
        TableModelEvent event = new TableModelEvent(this);
        listener.forEach((tempListener) -> {
            tempListener.tableChanged(event);
        });
    }
    // ... gekürzt
}
```

Wie werden die Daten angezeigt? Die Anzeige-Klasse baut ein `JFrame` auf und über gibt der `JTable` eine Instanz der Klasse Datenbasis als Tabellenmodell. Ferner stellt sie die Methode `addJob()` zur Verfügung, die einen Job entgegennimmt und an die Daten basis weiterreicht.

```
public class Anzeige {  
    private final JFrame mainFrame = new JFrame("Jobliste");  
    private final Datenbasis jobModel = new Datenbasis();  
  
    Anzeige() {  
        var pnlAnzeige = new JPanel();  
        pnlAnzeige.setLayout(new BorderLayout());  
        var tblJobs = new JTable();  
        tblJobs.setModel(jobModel);  
        pnlAnzeige.add(new JScrollPane(tblJobs), BorderLayout.CENTER);  
        mainFrame.getContentPane().add(pnlAnzeige);  
        mainFrame.setSize(500, 500);  
        mainFrame.setLocationRelativeTo(null);  
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        mainFrame.setVisible(true);  
    }  
  
    void addJob(String job) {  
        jobModel.addJob(job);  
    }  
}
```

Die `main`-Methode der Klasse `Anzeige` fügt der `Anzeige`-Instanz im Sekundentakt einen neuen Job hinzu. Vom Abdruck sehe ich ab, bitte analysieren Sie den Beispielcode dazu selbst.

Exkurs: Die Arbeit mit Tabellenmodellen

Eine Tabelle ist nur dann wirklich eine Tabelle, wenn sie auch mehrere Spalten enthält. Eine Tabelle mit einer Spalte ist eine Liste und die haben Sie im letzten Kapitel gesehen. Im Beispielprojekt `JTable_Demo` wird neben dem Job auch das angebotene Gehalt angezeigt. Da gute Arbeitnehmer nicht mehr so leicht zu finden sind, müssen Arbeitgeber sich in bestimmten Branchen noch eine Extragratisifikation einfallen lassen, um Bewerber zu finden. Das Bild Abb. 5.4 zeigt das Programm in Aktion. Zwei Dinge haben sich geändert: Zuerst gibt es zwei Spalten mehr: Das Gehalt und den Extrabonus. Außerdem wird das Gehalt rechtsbündig angezeigt, die beiden anderen Spalten linksbündig.

Die wichtigste Änderung des Projekts ist, dass ein Job jetzt nicht mehr einfach nur ein String ist, sondern ein eigener Datentyp `Job`. In ihm werden die Beschreibung des Jobs,



Jobs	Gehalt in €	Extra
Komparse beim Film	200	
Partylöwe	150,3	Cocktails
Buchhalter	400	
Kellner	200,9	eine Fliege
Programmierer	1.000,5	Pizza und Kaffee bis der Arzt kommt

Abb. 5.4 Screenshot „JTable_Demo“

das Gehalt und eine Extragratisifikation gespeichert. Den Job-Konstruktor habe ich überladen. Jobs, die begehrt sind, brauchen keine Extragratisifikation.

```
public class Job {
    final String beschreibung;
    final double gehalt;
    final String extra;

    // ... gekürzt

    Job(String beschreibung, double gehalt, String extra) {
        this.beschreibung = beschreibung;
        this.gehalt = gehalt;
        this.extra = extra;
    }
}
```

Die main-Methode der Startklasse erzeugt wieder einige Jobs und übergibt sie der Datenbasis:

```
public static void main(String[] args) {
    Job[] jobs = new Job[] {
        new Job("Komparse beim Film", 200),
        new Job("Partylöwe", 150.3, "Cocktails"),
        new Job("Buchhalter", 400),
        new Job("Kellner", 200.9, "eine Fliege"),
        new Job("Programmierer", 1000.5, "Pizza und Kaffee bis der
Arzt kommt")
    };
    // ... gekürzt
}
```

Die letzte Änderung betrifft das Datenmodell. Das Array mit den Spaltenüberschriften ist größer geworden. Wenn Sie den Wert einer bestimmten Zelle ausgeben wollen, lassen Sie sich zuerst den Job in dieser Zeile zurückgeben. Danach fragen Sie in Abhängigkeit von der Spalte den relevanten Wert ab.

```
public class Datenbasis implements TableModel {  
    private final List<TableModelListener> listener = new LinkedList<>();  
    private final List<Job> jobListe = new ArrayList<>();  
    private final String[] headers = new String[] {  
        "Jobs", "Gehalt in €", "Extra"  
    };  
  
    @Override  
    public Object getValueAt(int rowIndex, int columnIndex) {  
        Job job = jobListe.get(rowIndex);  
        return switch (columnIndex) {  
            case 0 -> job.beschreibung;  
            case 1 -> job.gehalt;  
            case 2 -> job.extra == null ? "" : job.extra;  
            default -> job.beschreibung;  
        }  
    }  
    // ... gekürzt  
}
```

Eine weitere wichtige Änderung bezieht sich auf die Angabe des Datentyps einer Spalte. Die zweite Spalte (Index = 1) ist vom Typ `double`. Alle anderen Spalten sind `Strings`. Ein `double`-Wert wird von `JTable` standardmäßig rechtsbündig angezeigt, `Strings` linksbündig.

```
@Override  
public Class<?> getColumnClass(int columnIndex) {  
    return switch (columnIndex) {  
        case 1 -> Double.class;  
        default -> String.class;  
    };  
}
```

Wie Sie sehen, steigt mit der Anzahl Spalten der Aufwand, die Datenbasis zu beschreiben. Schwierig ist es allerdings nicht, ein `TableModel` zu implementieren.

In den beiden obigen Programmauszügen sehen Sie eine weitere Java-Neuheit im Vergleich zu Java 8: Switch Expressions. Seit Java 12 sind sie als Preview Feature verfügbar, wurden mit Java 13 noch um das Schlüsselwort `yield` ergänzt (das ich Ihnen in einem späteren Kapitel noch zeigen werde) und sind seit Java 14 im Java-Standard enthalten.

Schauen Sie sich die beiden fett gedruckten Beispiele noch einmal genau an. Dabei fallen auf den ersten Blick mehrere Dinge auf:

1. `Switch` steht jetzt als Ausdruck (eben `expression`) zur Verfügung, in beiden Fällen jetzt im `return`-Statement einer Methode. Sie können aber genauso gut auf der rechten Seite einer Zuweisung oder eines Vergleichs stehen.
2. Der Doppelpunkt im `Switch Statement` wird jetzt durch einen Pfeil (wie bei Lambda-Ausdrücken) ersetzt. Dahinter steht dann in den beiden obigen Beispielen genau das, was auch auf der rechten Seite einer Zuweisung oder in einem Vergleich stehen würde.
3. Es gibt kein `break` (und auch kein `return`). Im Gegensatz zum `Switch Statement` endet die Verarbeitung im `Switch Expression` am Ende des jeweiligen Ausdrucks für den ausgewählten Fall. Die Verarbeitung „fällt“ nicht „durch“.
4. Der `default-Fall` ist – im ersten Code-Schnipsel – zwar identisch mit einem der normalen Fälle, muss aber dennoch in einer eigenen Fallunterscheidung stehen.

Das `Switch Statement` gibt es nach wie vor. Aber durch die neu hinzugekommenen `Switch Expressions` lässt sich geeigneter Code noch einmal deutlich eleganter und lesbarer formulieren. Es ist auch möglich, mehrere Statements in einem Fall abzuarbeiten, die dann mit geschweiften Klammern zusammengefasst werden. Um in diesem Fall dann den Rückgabewert des Ausdrucks zu definieren, wird der `yield`-Befehl (anstelle eines `return` oder eines `break`) verwendet.

Für die Fallunterscheidung von `enum`-Werten prüft der Compiler, ob alle Werte auch geprüft werden oder alternativ ein `default-Fall` vorhanden ist. Umgekehrt heißt das, dass bei der expliziten Prüfung aller Fälle auch der `default-Fall` entfallen kann. Für alle anderen Typen in der Fallunterscheidung muss aber ein `default-Fall` angegeben sein.

Die Klasse `AbstractTableModel` überschreibt die Methoden, die einer standardmäßigen Behandlung zugänglich sind: das Registrieren und De-Registrieren der Listener zum Beispiel. Standardmäßig ist auch vorgesehen, dass Zellen nicht editiert werden dürfen. Sie können (müssen aber nicht) diese Methoden überschreiben. Das ist das gleiche Prinzip wie beim Template Method Pattern, wo die Klasse `AbstractListModel` Ihnen einen Teil der Arbeit abgenommen hat. Im Verzeichnis zu diesem Kapitel finden Sie auch das Beispielprojekt `AbstractTableModel_Demo`; hier wird die Datenbasis von einem `AbstractTableModel` abgeleitet. Eine Besprechung des Codes erübrigt sich – die Datenbasis benötigt in dieser Version nur noch die Methoden, die erforderlich sind, um die anzueigenden Daten zu beschreiben. Alle weiteren Methoden erbt sie von der Oberklasse.

Warum habe ich das `TableModel` hier beschrieben? In den wenigsten Java-Einführungen finde ich Ausführungen zum Interface `TableModel`. Die meisten beschreiben die Klasse `DefaultTableModel`. Vielleicht ist das der Grund dafür, dass das `TableModel` in der Praxis eher ein Schattendasein führt. In der Praxis werden Sie sehr viel öfter Tabellenmodelle finden, die sich auf das `DefaultTableModel` stützen. Es ist besser dokumentiert und – jedenfalls auf den ersten Blick – einfacher zu verwenden. Tat-

sächlich aber ist die Klasse nicht ganz unproblematisch. Sie nutzt, um nur einen Kritikpunkt zu nennen, zur internen Repräsentation der Daten Vektoren – und das kostet unnötig Performance. Ich mache daher Werbung dafür, eigene Tabellenmodelle auf Basis von `TableModel` zu entwickeln, und hoffe, Sie ein wenig dafür interessieren zu können.

5.8 Das Model-View-Controller Pattern

Das Projekt `MVC_1` führt zu einer Diskussion über das MVC Pattern. MVC steht für Model, View und Controller. Jede dieser Einheiten hat klar definierte Aufgaben: Das Model enthält die Daten, die View ist für die Anzeige auf dem Bildschirm zuständig und der Controller vermittelt zwischen beiden. Der Anwender – im Beispiel die `main`-Methode – greift immer auf den Controller zu. Wenn die Datenbasis sich ändert, informiert sie die angemeldeten Listener, die View-Einheiten, über die Änderung. Die View fragt dann die Daten bei der Datenbasis ab und bringt sich so wieder in einen konsistenten Zustand. Nach diesem Prinzip arbeiten alle Komponenten, also beispielsweise auch `JList` und `JTree`.

- ▶ Was ist mit dieser Lösung gewonnen? Sie können beliebig viele, auch unterschiedliche Beobachter – Views – beim Datenmodell anmelden. Sowohl Datenmodell als auch View sind unabhängig voneinander und können ausgetauscht werden.

Wo lässt sich jetzt das MVC-Modell im Projekt `MVC_1` nachweisen? Die Datenbasis, also das Model, ist das `TableModelObjekt` mit seiner Job Liste. Die Klasse `Anzeige` nutzt eine Instanz der Klasse `JTable`, um die Daten auf dem Bildschirm anzuzeigen. Man könnte jetzt versucht sein, die Klasse `JTable` als View zu definieren. Diese Annahme ist jedoch falsch. Aber warum?

Die Datenbasis erwartet als Listener ein Objekt vom Typ `TableModelListener`. Wäre die Klasse `JTable` die View, müsste sie das Interface `TableModelListener` implementieren. Aus der API-Doku entnehmen Sie aber, dass `JTable` dieses Interface gar nicht implementiert. Wo also steckt der Listener? Wenn Sie ein Tabellenmodell mit `setModel()` an ein `JTableObjekt` übergeben, stellt dieses eine Verbindung der Datenbasis mit einer Instanz der inneren Klasse `AccessibleJTable` her. Diese innere Klasse implementiert das Interface `TableModelListener` und definiert die Methode `tableChanged()`, die Sie in der Datenbasis aufgerufen haben. Die Methode `tableChanged()` fragt beim `TableModelEvent` den Umfang der Änderung ab und weist die Superklasse von `JTable` an, sich neu zu zeichnen. Die View im Sinne von MVC ist also diese innere Klasse `AccessibleJTable`; die Klasse `JTable` hingegen ist der Controller. Da die View auf Daten und Methoden der sie umschließenden Klasse `JTable` zugreift, verschwimmen die Grenzen von Controller und View; man fasst diese zum Delegate zusammen. Java Programmierer sprechen in der Praxis auch eher von Model-Delegate statt von Model-View-Controller. In Java-Einführungen werden die Klas-

sen `JTable`, `JButton` usw. gern als View dargestellt. Das ist in Ordnung, um das dahinterstehende Prinzip zu erläutern. Genau betrachtet ist diese Sichtweise jedoch nicht richtig.

Hintergrundinformation

Im Verzeichnis zu diesem Kapitel finden Sie auch das Beispielprojekt `MVC_2`. Hier tauchen die Klassen `Student` und `Arbeitnehmer` wieder auf, die Sie von weiter oben noch kennen. Beide implementieren das Interface `JobListener`, das das Interface `TableModelListener` erweitert. Studenten und Arbeitnehmer nehmen also die Rolle der View ein und müssen die Methode `tableChanged()` überschreiben. Auf den Code möchte ich nicht weiter eingehen; er ist nicht schwierig zu verstehen.

Die meisten Bücher über Design Patterns – auch die GoF – beschreiben beim Observer Pattern eine Statistik (das Model), die einmal als Torten- und dann als Balkendiagramm angezeigt wird (View). In meinem Beispiel ist die Datenbasis die Jobbörse, die von zwei unterschiedlichen Views angezeigt wird. Das Prinzip ist jedoch das gleiche.

5.9 Observer – Das UML-Diagramm

In Abb. 5.5 zeige ich Ihnen beispielhaft das UML-Diagramm zum Beispielprojekt `Observer_08`.

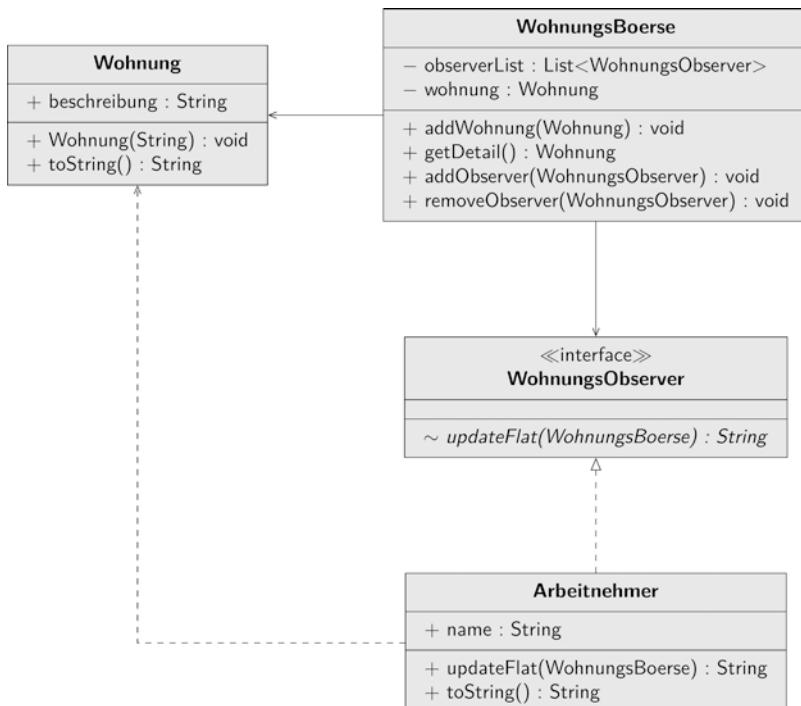


Abb. 5.5 UML-Diagramm des Observer Pattern (Beispielprojekt `Observer_08`)

5.10 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Das Observer Pattern findet immer dann Anwendung, wenn sich der Status eines Objekts ändert und beliebig viele weitere Objekte darüber informiert werden sollen.
- Es gibt eine Ereignisquelle und Beobachter.
- Beobachter müssen austauschbar sein.
- Zwei Threads können nicht gleichzeitig auf die Liste der Beobachter zugreifen.
- Es ist sinnvoll, vor der Information an die Beobachter die Liste zu kopieren oder eine `CopyOnWriteArrayList` zu verwenden.
- Die Klassenbibliothek von Java enthält (noch) die Klasse `Observable` und das Interface `Observer`, deren Verwendung nicht unproblematisch ist.
- Der GUI-Programmierung liegt das MVC-Muster zugrunde.
- MVC trennt die Zuständigkeiten in Model, View und Controller.
- Das Model enthält die Datenbasis.
- Die View zeigt die Daten an.
- Der Controller vermittelt zwischen Model und View.
- In Java sind die Grenzen zwischen View und Controller fließend, weshalb man vom Delegate spricht.

Zweckbeschreibung

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

„Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“



Chain of Responsibility

6

Das Verhaltensmuster Chain of Responsibility wird auch Zuständigkeitskette genannt. Auch hier geht es darum, eine Nachricht an eine Vielzahl von Objekten zu senden. Erinnern Sie sich – beim Observer Pattern haben Sie eine Nachricht an alle Objekte geschickt, die als Listener registriert sind. Über die Zuständigkeitskette wird eine Nachricht in eine Kette von Objekten weitergereicht; das erste Objekt, das diese Nachricht verarbeiten kann, bekommt die Zuständigkeit. Auch die Bearbeitung einer Nachricht durch mehrere Objekte nacheinander ist möglich, ebenso wie die Veränderung der Nachricht auf ihrem Weg durch die Kette.

6.1 Ein Beispiel aus der realen Welt

Stellen Sie sich vor, Sie wollen einen Führerschein beantragen. Sie gehen im Bürgeramt also in das erste Zimmer auf dem Flur. Dort fragen Sie nach, ob Sie einen Führerschein beantragen können. Der Beamte im ersten Zimmer bearbeitet keine Führerscheine, sondern Jagdscheine, ist also nicht zuständig; daher schickt er Sie in das zweite Zimmer. Auch dort fragen Sie nach und bekommen entweder Hilfe oder werden ein Zimmer weitergeschickt. Irgendwann werden Sie auf einen Beamten treffen, der für Führerscheine zuständig ist, und dieser wird Ihnen helfen.

Damit wird ein Merkmal des Patterns schon deutlich: Sie haben ein System aus Objekten, die **alle** die Nachricht möglicherweise verarbeiten könnten.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_6

6.2 Erstes Code-Beispiel: Lebensmitteleinkauf

Ihre Zukunftsvision nimmt Gestalt an: Sie haben einen Kühlschrank mit einem RFID-Empfänger. Der Füllstand Ihrer Bierflaschen wird regelmäßig überprüft. Auch wenn die Anzahl Wurst- und Käsescheiben sinkt, wird das registriert. Und wenn Eier und Brot zur Neige gehen, wird der Kühlschrank das ebenfalls wissen. Da Sie ein motivierter Programmierer sind, haben Sie natürlich keine Zeit, selbst in den Supermarkt zu gehen, um einzukaufen. Glücklicherweise haben die Läden in der Region sich zu einer elektronischen Lieferkette zusammengeschlossen. Wenn Kühlschränke melden, dass bei einem Kunden eine Ware zur Neige geht, prüft jeder der Händler, ob er dem Kunden neue Lebensmittel bringen kann. Wenn nicht, reicht er die Bestellung an den nächsten möglichen Händler weiter.

6.2.1 Die benötigten Lebensmittel

Im ersten Schritt definieren Sie, welche Lebensmittel von der elektronischen Lieferkette geliefert werden sollen. Im Beispielprojekt CoR_1 sollen das Brot, Käse, Wurst, Eier und Bier sein. Diese Waren definieren Sie in einer Enumeration.

```
public enum Ware {
    KAESE("Käse"), WURST("Wurst"), EIER("Eier"),
    BIER("Bier"), BROT("Brot");
    private final String beschreibung;

    private Ware(String beschreibung) {
        this.beschreibung = beschreibung;
    }

    @Override
    public String toString() {
        return beschreibung;
    }
}
```

- ▶ In diesem Projekt wird der Auftrag, das „Problem“, durch eine Variable aus der Enumeration übergeben. Erinnern Sie sich an das Observer Pattern? Dort haben Sie einen Listener und ein Eventobjekt verwendet. Dieses Prinzip passt auch bei der Chain of Responsibility: Sie legen ein Eventobjekt an und übergeben es an das erste Glied in der Kette. Dieses Glied entscheidet, ob es das Event verarbeiten kann; wenn nicht, wird das Event weitergeleitet. Dass die Java-Klassenbibliothek exakt so vorgeht, möchte ich Ihnen weiter unten zeigen.

Im nächsten Abschnitt lernen Sie die Verkäufer der Waren kennen.

6.2.2 Die Verkäufer

Als Händler definiere ich den Getränkehandel, die Bäckerei und den Hofladen von Bauer Huber, wo Sie frischen Käse, Wurst und Eier bekommen. Alle Händler müssen neben ihren eigentlichen Aufgaben (Einkauf, Herstellung usw.) identische Methoden haben: verkaufen und die Bestellung weitergeben, wenn sie nicht bedient werden kann. Sie definieren also eine abstrakte Oberklasse, die eine Referenz auf den jeweils nächsten Händler in der Kette hält. Die Methode `setNext()` rufen Sie auf, wenn Sie einen weiteren Händler der Kette hinzufügen möchten. Jeder Händler prüft zunächst, ob er selbst bereits auf einen folgenden Händler verweist. Ist dies der Fall, wird er den neuen Händler nicht speichern, sondern an den nächsten Händler weiterreichen, der nun prüft, ob er der letzte in der Kette ist. Wenn ja, wird er den neuen Händler als folgenden Händler speichern. Außerdem muss jeder Händler auch eine Methode `verkaufen()` anbieten – diese deklarieren Sie in der Oberklasse, definieren sie aber noch nicht.

```
public abstract class AbstractHaendler {  
    private AbstractHaendler next;  
  
    public abstract void verkaufen(Ware ware);  
  
    public void setNext(AbstractHaendler haendler) {  
        if (next == null)  
            next = haendler;  
        else  
            next.setNext(haendler);  
    }  
  
    protected void weiterleiten(Ware ware) {  
        if (next != null)  
            next.verkaufen(ware);  
    }  
}
```

Als Beispiel für einen konkreten Händler drucke ich hier nur die Bäckerei ab. Die anderen Händler finden Sie im Beispielprojekt. Die Methode `verkaufen()` steht stellvertretend für eine beliebig komplexe Problemlösung. Das Prinzip, das dahintersteht, ist jedoch immer gleich: Wenn ich das Problem lösen kann, löse ich es, sonst reiche ich es weiter.

```
public class Baeckerei extends AbstractHaendler {  
    // ... gekürzt
```

```

@Override
public void verkaufen(Ware ware) {
    if (ware == Ware.BROT)
        System.out.println(name + " verkauft " + ware);
    else
        weiterleiten(ware);
}
}

```

Lebensmittel sind definiert, die Händler stehen bereit. Es fehlt noch die Testklasse.

6.2.3 Der Client

Die main-Methode der Testklasse ist der Client, der den Umgang mit der Kette demonstriert. Zunächst legen Sie beliebig viele Händler an und verknüpfen diese:

```

public class Testklasse {
    public static void main(String[] args) {
        var hofladen = new Hofladen("Erwin Huber");
        var baeckerei = new Baeckerei("Schnellback");
        Getraenkehandel getraenke = new Getraenkehandel("Bier und
        Brause");

        baeckerei.setNext(getraenke);
        getraenke.setNext(hofladen);

        baeckerei.verkaufen(Ware.KAASE);
        hofladen.verkaufen(Ware.EIER);
        baeckerei.verkaufen(Ware.WURST);
        baeckerei.verkaufen(Ware.BROT);
        getraenke.verkaufen(Ware.BIER);
    }
}

```

Jetzt kann der Client einen Einkaufsauftrag an das erste Glied in der Kette übergeben. Der Auftrag wird nun so lange an einen Händler weitergereicht, bis einer seine Zuständigkeit erkennt und die gewünschte Ware liefert.

6.2.3.1 Erweiterung des Projekts

Wenn Sie das Projekt kritisch betrachten, fällt Ihnen sicher ein Schwachpunkt auf: Wenn es keinen Lieferanten für beispielsweise Eier gibt, wird die Anfrage gar nicht bearbeitet. Kommentieren Sie testweise die Zeile, in der der Hofladen erzeugt wird, aus und starten Sie das Projekt neu. Sie stellen fest, dass die Anforderung schlicht und ergreifend ignoriert

wird. Es bietet sich also an, ein Default-Verhalten vorzusehen. Dazu erweitern Sie die Klasse `AbstractHaendler`. Den Code dazu finden Sie im Beispielprojekt CoR_2.

```
public abstract class AbstractHaendler {  
    // ... gekürzt  
  
    private void druckeHinweis(Einkauf einkauf) {  
        System.out.println("Leider gibt es keinen, der " + einkauf  
                           + "liefern kann.");  
    }  
  
    protected void weiterleiten(Einkauf einkauf) {  
        if (next != null)  
            next.verkaufen(einkauf);  
        else  
            druckeHinweis(einkauf);  
    }  
}
```

6.2.3.2 Variationen des Patterns

Hätte es einen Unterschied gemacht, wenn Sie eine Liste – `ArrayList` oder `LinkedList` – angelegt hätten und die Händler darin gespeichert? Sie hätten mit einer Schleife über die Liste iterieren können und die Schleife ab dem Moment abbrechen, wo einer der Händler die Ware verkauft. Die Chain bietet aber sehr viel mehr Möglichkeiten als ein Aggregat.

6.2.3.2.1 Hierarchien gestalten und beliebigen Einstiegspunkt wählen

Im Händler-Beispiel gibt es viele Händler, die untereinander aber in keiner Hierarchie stehen. Lassen Sie uns ein ganz anderes Beispiel durchdenken. In einem Unternehmen wächst die Anzahl von hierarchischen Ebenen im Verhältnis zur Unternehmensgröße. Bei einem Unternehmen mit zehn Mitarbeitern dürfte der Draht zum Chef noch sehr direkt sein – flache Hierarchien sind tatsächlich flach. Wenn ein Unternehmen 100 Mitarbeiter hat, brauchen Sie mindestens ein Ebene Abteilungsleiter und darunter vielleicht auch eine Ebene für Referatsleiter. Unternehmen mit 500 Mitarbeitern haben in vermutlich dieser Reihenfolge: den Vorstand, die Abteilungsleiter, die Unterabteilungsleiter, die Referatsleiter, die Sachgebietsleiter, die Gruppenleiter und schließlich die Teamleiter. So ein Unternehmen benötigt strenge Regeln, die diese Hierarchie rechtfertigen. Ein Beispiel könnte sein: Wenn jemand eine Geschäftsreise macht, muss der Gruppenleiter die Reise bewilligen, wenn sie nicht länger als einen Tag dauert und sich im Nahbereich des Unternehmens abspielt. Wenn eine Reise länger als einen Tag dauert, muss der Referatsleiter die Reise abzeichnen. Wenn eine Fernreise ansteht, die länger als einen Tag dauert, muss der Abteilungsleiter beteiligt werden. Wenn Sie einfacher Mitarbeiter sind, werden Sie Ihren Reiseantrag immer an Ihren Teamleiter übergeben, der der erste in der Kette ist. Der Teamleiter prüft seine Zuständigkeit und wird Ihren Reiseantrag entweder genehmigen oder, wenn er

nicht zuständig ist, weiterleiten. Wenn Sie selbst aber Unterabteilungsleiter sind und zwei Tage verreisen wollen, werden Sie Ihren Reiseantrag wohl nicht bei einem Ihrer nachgeordneten Referatsleiter abgeben, sondern bei Ihrem nächsten Vorgesetzten, dem Abteilungsleiter.

Für die Chain of Responsibility heißt das: Jeder Client kann für seine Nachricht einen anderen Einstiegspunkt wählen.

6.2.3.2.2 Glieder der Kette modifizieren die Anfrage

Ich möchte aber noch zwei weitere Besonderheiten des Patterns herausarbeiten und dazu nochmal das Beispiel mit den Händlern aufgreifen. Sie können die verkaufen-Methode beliebig definieren. Es soll im folgenden Beispiel möglich sein, dass Sie ein Objekt vom Typ Einkauf übergeben. Die Klasse Einkauf ist eine Hilfsklasse. Sie hat zwei Attribute: die Ware selbst, wie Sie sie schon aus dem Projekt CoR_1 kennen, und die Menge. Wenn ein Händler die Ware verkauft, ruft er die Methode verkaufeWare() auf; die Methode willNochMehr() gibt zurück, ob der Kunde noch mehr kaufen möchte.

```
public class Einkauf {
    protected final Ware ware;
    private int menge;

    Einkauf(Ware ware, int menge) {
        this.ware = ware;
        this.menge = menge;
    }

    public void verkaufeWare() {
        menge--;
    }

    public boolean willNochMehr() {
        return menge > 0;
    }

    @Override
    public String toString() {
        return ware.toString();
    }
}
```

Ein Händler verkauft jetzt so lange die gewünschte Ware, bis entweder er selbst keine mehr anbieten kann oder die verlangte Menge gleich 0 ist. Die Frage, ob er überhaupt noch Ware hat, wird durch einen Zufallsgenerator entschieden. Am Beispiel der Bäckerei schauen wir uns die Änderungen an.

```
public class Baeckerei extends AbstractHaendler {  
    // ... gekürzt  
    @Override  
    public void verkaufen(Einkauf einkauf) {  
        if (einkauf.ware == Ware.BROT)  
            while (wareIstVorraetig() && einkauf.willNochMehr()) {  
                System.out.println(name + " verkauft " + einkauf.ware);  
                einkauf.verkaufeWare();  
            }  
        if (einkauf.willNochMehr())  
            weiterleiten(einkauf);  
    }  
  
    private boolean wareIstVorraetig() {  
        double zahl = Math.random() * 10;  
        return zahl >= 5;  
    }  
}
```

Jedes Glied in der Kette kann also jetzt die Anfrage – hier: den Einkauf – modifizieren.

6.3 Ein Beispiel aus der Klassenbibliothek

Wo lässt sich dieses Pattern in Java nachweisen? Schauen Sie sich das Beispielprojekt CoR_3 an. Die main-Methode baut ein neues Fenster auf. Eine Instanz der Klasse JFrame enthält eine Instanz der Klasse JPanel und diese wiederum enthält eine Instanz der Klasse JLabel. Ferner gibt es einen EventListener, der, wenn eine Komponente mit der Maus angeklickt wird, auf der Konsole ausgibt, auf welche Komponente geklickt wurde.

```
private final MouseAdapter adapter = new MouseAdapter() {  
    @Override  
    public void mouseClicked(MouseEvent e) {  
        Object quelle = e.getSource();  
        System.out.println(quelle.getClass());  
    }  
};
```

Allen drei Komponenten wird dieser EventListener zugewiesen:

```
myLabel.addMouseListener(adapter);  
myFrame.addMouseListener(adapter);  
myPanel.addMouseListener(adapter);
```

Führen Sie das Programm aus und beobachten Sie einmal, welche Quelle jeweils ausgegeben wird, wenn Sie auf das Fenster, das Panel oder das Label klicken. Wenn Sie das Label anklicken, wird das Label als Eventquelle genannt. Wenn Sie auf das Panel klicken, das Panel und sonst das Fenster.

Kommentieren Sie testweise die erste Zeile aus, so dass dem Label kein Eventhandler zugewiesen wird. Wenn Sie nun zur Laufzeit daraufklicken, wird das Panel als Eventquelle ausgegeben. Jede Komponente prüft also ihre Zuständigkeit und reicht das Event gegebenenfalls an den sie umschließenden Container weiter – bis zuletzt das Fenster seine Zuständigkeit nicht mehr abgeben kann. Hier lässt sich ein zweites Merkmal des Patterns ablesen: Die Verarbeitung geht **vom Speziellen zum Allgemeinen**, von den Blättern zur Wurzel. Sie werden weiter hinten das Composite Pattern kennenlernen, wo die Verarbeitung von der Wurzel zu den Blättern, also umgekehrt, geht.

Das Beispielprojekt CoR_4 geht einen Schritt weiter. Die GlassPane des Fensters wird aktiviert und bekommt den EventListener zugewiesen.

```
Component glass = myFrame.getGlassPane();
glass.setVisible(true);
glass.addMouseListener(adapter);
```

Nun ist es egal, wohin Sie klicken – es ist immer die GlassPane, die das MouseEvent verarbeitet. In der Praxis können Sie auf diese Weise Benutzereingaben und Mausklicks abfangen, wenn Ihr Programm zum Beispiel eine umfangreiche Rechnung durchführt. Sie decken praktisch die Benutzeroberfläche mit der Glasscheibe ab, um sie vor unerwünschtem Zugriff zu schützen. Sobald Sie die „Sichtbarkeit“ der Scheibe wieder auf false setzen, ist die abgedeckte Oberfläche wieder zugänglich. Die GlassPane ist ein Objekt vom Typ JPanel, so dass auf der Konsole immer dieser Typ als Eventquelle genannt wird. Dieses Beispiel fügt jetzt zwar nicht wirklich etwas Neues zum Thema des Patterns hinzu, ist aber ein praktischer Tipp, den ich hier gerne unterbringe.

6.4 Chain of Responsibility – Das UML-Diagramm

In Abb. 6.1 sehen Sie das UML-Diagramm zum Beispielprojekt CoR_2.

6.5 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Eine beliebige Anzahl Objekte kann potenziell ein Problem lösen,
- Diese Objekte werden verkettet oder in einem Baum organisiert.
- Eine Anfrage wird nur an ein Glied in der Kette übergeben,

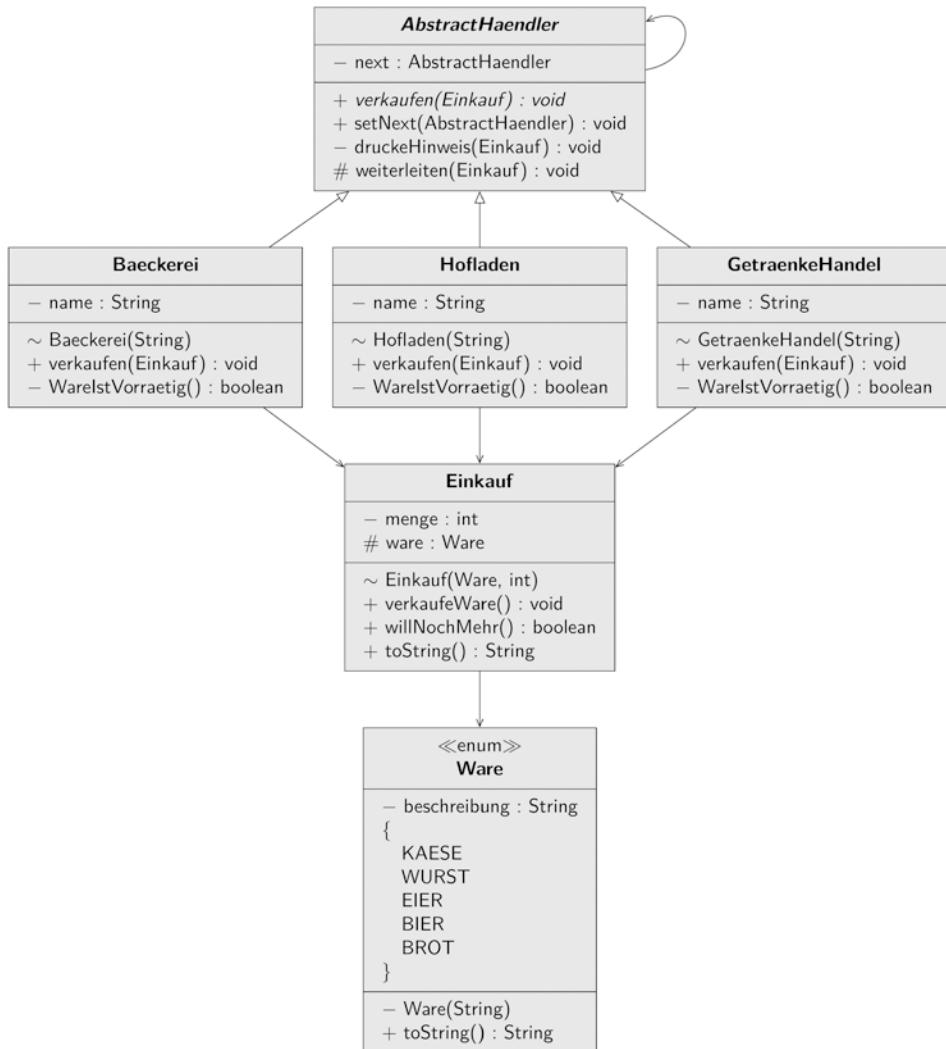


Abb. 6.1 UML-Diagramm des Chain of Responsibility Pattern (Beispielprojekt CoR_2)

- Kann das Objekt das Problem nicht lösen, reicht es die Anfrage weiter.
- Ein Objekt kann eine Anfrage verändern.
- Jedes weitere Objekt prüft seine Zuständigkeit; entweder es reagiert auf die Anfrage oder es leitet sie weiter.
- Es besteht die Gefahr, dass eine Anfrage vollkommen unbeantwortet bleibt; daher sollte immer ein Default-Verhalten implementiert werden.

Zweckbeschreibung

Die Gang of Four beschreibt den Zweck des Patterns „Chain of Responsibility“ wie folgt:

„Vermeide die Kopplung des Auslösers einer Anfrage an ihren Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.“



Mediator

7

Das Mediator Pattern ermöglicht das flexible Zusammenspiel mehrerer Objekte/Klassen, die sich untereinander ggf. nicht kennen. Die Vermittlungsfunktion kann sehr flexibel variiert werden, aber auch zu sehr umfangreichen Implementierungen führen. Im Gegensatz zum Observer oder zur Chain of Responsibility hat der Mediator nämlich auch eigene Aufgaben zu erledigen. Damit bekommt er eine zentrale Bedeutung. Allerdings besteht auch die Gefahr, dass er sehr umfangreich und sogar unübersichtlich wird.

7.1 Abgrenzung zum Observer Pattern

Zum Einstieg möchte ich Mediator und Observer gegenüberstellen. Sinn von Observer ist es, eine Ereignisquelle mit beliebig vielen Beobachtern in Beziehung zu setzen. Dabei war die Beziehung uni- oder bidirektional: Die Ereignisquelle kennt die Schnittstelle der Beobachter und vielleicht auch umgekehrt. Problematisch wird es aber, wenn jeder Beobachter auch gleichzeitig Ereignisquelle für jeden anderen Beobachter sein möchte. Das kann zum Beispiel der Fall sein, wenn Sie einen Chat programmieren. Es gibt eine unbestimmte Anzahl Teilnehmer, die alle an den Ereignissen – Nachrichten – der anderen interessiert sind. Würden Sie einen Chatroom mit dem Observer Pattern programmieren wollen, müsste jeder Teilnehmer eine Liste mit allen anderen Teilnehmern halten. Neue Teilnehmer – beim Mediator Pattern spricht man von Kollegen – müssten sich bei allen vorhandenen Teilnehmern anmelden und ihrerseits alle vorhandenen Teilnehmer referenzieren. Rechnen Sie hoch, wie viele Beziehungen Sie bei – sagen wir – 33 Teilnehmern, also Kollegen, haben. Dafür brauchen Sie eine andere Vorgehensweise.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_7

7.2 Aufgabe des Mediator Patterns

Wenn 33 Kollegen sich alle untereinander kennen, hätten Sie ein ineffizientes Spinnennetz von über 1000 Beziehungen. Der Mediator hat nun die Aufgabe, Ereignisquellen und Beobachter zu entkoppeln. Er vermittelt zwischen den Beteiligten, weshalb das Muster eben auch Vermittler genannt wird – der deutschen Bedeutung des (lateinischen) Wortes Mediator.

Der Mediator ist skalierbar. Ich möchte zur Veranschaulichung folgendes Szenario entwerfen:

Sie können eine Klasse in Ihrem Programm haben, die die Bestellungen eines Online-Shops verwaltet; anstelle oder sogar ergänzend zu dieser Klasse kann die Einheit „Bestellungen“ aber auch eine ganze Abteilung von Mitarbeitern sein, die im Telefonservice weitere Bestellungen erfassen. Die Einheit „Bestellungen“ meldet an den Mediator – der auch wieder entweder eine einzige Klasse oder eine Abteilung mit Mitarbeitern sein kann – einen neuen Auftrag zur Bearbeitung.

Der Mediator fragt bei der Kundenverwaltung – auch die kann wieder eine Klasse oder eine ganze Abteilung sein – nach, ob der Kunde bereits registriert ist. Wenn nicht, bittet der Mediator die Kundenverwaltung, einen neuen Datensatz anzulegen; ist der Kunde bereits im Bestand, fragt der Mediator die Buchhaltung (Sie wissen ja: Klasse oder Abteilung), ob der Kunde zuverlässig seine Rechnungen bezahlt hat. Abhängig von der Antwort der Buchhaltung weist der Mediator die Einheit „Versand“ an, die Ware entweder per Rechnung oder per Nachnahme zu versenden. Wenn nun eines Tages die Notwendigkeit besteht, eine weitere Einheit in das System einzuhängen, muss nur der Mediator angepasst werden. Lassen Sie den Fall eintreten, dass die Chefabteilung über alle Einkäufe ab einem bestimmten Verkaufswert informiert werden möchte. Sie müssen lediglich im Mediator eine entsprechende Meldung vorsehen; die anderen Einheiten würden davon gar nichts mitbekommen.

7.3 Mediator in Aktion – ein Beispiel

Die folgende Simulation zeigt Ihnen den Mediator in Aktion. Sie haben eine Einheit, die Wein produziert; diese Einheit wird vereinfacht Producer bezeichnet, in der Realität stehen Winzer, Weinküfer und Glykol-Industrie hinter dieser Einheit. Eine andere Einheit repräsentiert Verbraucher oder Einzelhändler, die Consumer. Die Consumer kaufen Wein zum jeweils günstigen Preis. Da es x Consumer und y Producer gibt, werden die Abhängigkeiten auf den Zwischenhändler, den Mediator, reduziert. Der Mediator nimmt die Anfragen der Consumer entgegen und leitet sie an die Producer weiter. Die Producer teilen dem Mediator schließlich mit, welchen Preis sie verlangen; der Mediator leitet diese Information an die Consumer weiter. Neben einer Methode zum Aushandeln des Preises braucht

der Mediator weitere Methoden zum Registrieren und De-Registrieren von Consumern und Producern. Den Beispiel-Code finden Sie im Projekt WeinSim.

7.3.1 Definition eines Consumers

Ein Consumer muss eine Methode haben, mit der er sich beim Mediator registrieren kann; auf die Möglichkeit, sich zu de-registrieren, verzichte ich, um das Beispiel nicht unnötig aufzublähen. Auf jeden Fall aber muss der Consumer eine Methode haben, Wein nachzufragen. Dieser Methode wird die Anzahl Einheiten – beispielsweise Flaschen – mitgegeben:

```
public interface ConsumerIF {  
    double preisErfragen(int einheiten);  
    void registrieren(MediatorIF mediator);  
}
```

Ein Consumer registriert sich beim Mediator und richtet seine Anfrage an diesen.

```
public class PrivaterKunde implements ConsumerIF {  
    private final String name;  
    private MediatorIF mediator;  
  
    public PrivaterKunde(String name) {  
        this.name = name;  
    }  
  
    // ... gekürzt  
  
    @Override  
    public double preisErfragen(int einheiten) {  
        System.out.  
            println(name + " fragt " + einheiten + " Flaschen Wein  
nach.");  
        double gesamtPreis = mediator.angebotErmitteln(einheiten);  
        return gesamtPreis;  
    }  
}
```

Im nächsten Abschnitt schauen wir uns die Produzenten des Weins mal an.

7.3.2 Definition eines Producers

Die Producer müssen zwei Methoden vorsehen – eine, um sich beim Mediator zu registrieren, und eine, an die der Mediator seine Anfragen richtet

```
public interface ProducerIF {  
    double angebotEinholen(int einheiten);  
    void registrieren(MediatorIF mediator);  
}
```

Ein Producer berechnet auf Basis einer Zufallszahl und in Abhängigkeit der angefragten Anzahl Einheiten den Preis für eine Einheit und teilt diesen Preis dem Mediator mit.

```
public class ProducerImpl implements ProducerIF {  
    private MediatorIF mediator;  
    private final String name;  
  
    public ProducerImpl(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public double angebotEinholen(int einheiten) {  
        double rabattFaktor = 1.0;  
        if (einheiten > 100) {  
            rabattFaktor = 0.7;  
        }  
        else {  
            if (einheiten > 50) {  
                rabattFaktor = 0.8;  
            }  
            else {  
                rabattFaktor = 0.9;  
            }  
        }  
        double preis = Math.random() * 9 + 1;  
        preis *= rabattFaktor;  
        String strPreis = NumberFormat.getCurrencyInstance().format(preis);  
        System.out.  
            println("Produzent " + name + " verlangt " + strPreis +  
" pro Flasche");  
        return preis * einheiten;  
    }  
}
```

Achten Sie darauf, dass hier der Producer zwar bei der Abgabe eines Angebots den Preis pro Flasche nennt (auf der Konsole ausgibt), aber dann an seinen Aufrufer den Gesamtpreis zurückgibt. Das muss der Mediator natürlich in seiner Kalkulation berücksichtigen. Um den kümmern wir uns nächsten Abschnitt.

7.3.3 Interface des Mediators

Die Methoden des Mediators werden im Interface MediatorIF deklariert.

```
public interface MediatorIF {  
    double angebotErmitteln(int einheiten);  
    void addProducer(ProducerIF producer);  
    void removeProducer(ProducerIF producer);  
    void addConsumer(ConsumerIF consumer);  
    void removeConsumer(ConsumerIF consumer);  
}
```

Ein konkreter Mediator, beispielsweise ein Großhändler, muss diese Methoden implementieren.

```
public class Grosshandel implements MediatorIF {  
    private final List<ProducerIF> producers = new ArrayList<>();  
    private final List<ConsumerIF> consumers = new ArrayList<>();  
  
    // ... gekürzt  
  
    @Override  
    public double angebotErmitteln(int einheiten) {  
        List<Double> angebote = new ArrayList<>();  
  
        for (ProducerIF tempProducer : producers) {  
            Double angebot = tempProducer.angebotEinholen(einheiten);  
            angebote.add(angebot);  
        }  
        double preis = Collections.min(angebote);  
        String strPreis = NumberFormat.getCurrencyInstance().format(preis);  
        System.out.  
            println("Die Großhandel kann als günstigstes Angebot für  
" + einheiten + " Flaschen anbieten: " + strPreis);  
        return preis;  
    }  
}
```

Im Beispiel wird die Liste der Consumer beim Mediator aktuell nicht verwendet. Aber Sie können ja mal überlegen, was Sie noch hinzufügen müssen, wenn ein Producer über den Mediator die Adressen der Kunden für die Zusendung einer Werbung erfragen möchte (Vorausgesetzt, die Kunden haben dazu ihr Einverständnis erteilt). Jetzt können wir das Projekt testen.

7.3.4 Test des Mediator Patterns

Wie können Sie das Zusammenspiel der Klassen testen? Sie erzeugen beliebig viele Consumer und beliebig viele Producer und melden alle beim Mediator an. Danach lassen Sie die Consumer den Preis für eine bestimmte Anzahl Flaschen anfragen. Der Mediator holt Angebote bei allen Producern ein, ermittelt das günstigste und leitet es an die Consumer weiter.

```
public class Testklasse {  
    public static void main(String[] args) {  
        // einen Mediator erzeugen  
        MediatorIF grosshandel = new Grosshandel();  
  
        // zwei Kunden erzeugen  
        ConsumerIF hans = new PrivaterKunde("Hans Kirch");  
        ConsumerIF heinz = new PrivaterKunde("Heinz Kirch");  
  
        // drei Anbieter erzeugen  
        ProducerIF winzer_1 = new ProducerImpl("Winzer 1");  
        ProducerIF winzer_2 = new ProducerImpl("Winzer 2");  
        ProducerIF winzer_3 = new ProducerImpl("Winzer 3");  
  
        // Consumer und Producer beim Mediator anmelden  
        hans.registrieren(grosshandel);  
        heinz.registrieren(grosshandel);  
        winzer_1.registrieren(grosshandel);  
        winzer_2.registrieren(grosshandel);  
        winzer_3.registrieren(grosshandel);  
  
        // Anfragen generieren und Angebote einholen  
        int einheit = 50;  
        double preis = hans.preisErfragen(einheit);  
  
        System.out.println("\n");  
  
        einheit = 10;  
        preis = heinz.preisErfragen(einheit);  
    }  
}
```

Das führt zu folgender Ausgabe (mit natürlich zufälligen Preisen):

Hans Kirch fragt 50 Flaschen Wein nach.
Produzent Winzer 1 verlangt 2,53 € pro Flasche
Produzent Winzer 2 verlangt 2,13 € pro Flasche
Produzent Winzer 3 verlangt 2,59 € pro Flasche
Das günstigste Angebot für 50 Flaschen ist: 106,56 €

Heinz Kirch fragt 10 Flaschen Wein nach.
Produzent Winzer 1 verlangt 4,71 € pro Flasche
Produzent Winzer 2 verlangt 2,59 € pro Flasche
Produzent Winzer 3 verlangt 8,00 € pro Flasche
Das günstigste Angebot für 10 Flaschen ist: 25,88 €

7.4 Mediator in Aktion – das zweite Beispiel

Das Mediator Pattern findet sich auch sehr häufig in der Implementierung von Benutzeroberflächen. Im Model-View-Controller Modell ist der Controller genau der Mediator, der zwischen der Ansicht/Verwendung und dem abstrakten Modell einer Anwendung vermittelt. Schauen wir uns dazu ein Beispiel an.

7.4.1 Mediator in der GUI-Programmierung

Sie haben eine GUI, auf der zwei Listen zu sehen sind. Die linke Liste zeigt alle Kontakte in Ihrem Adressbuch; rechts werden alle Kontakte gelistet, die zu der nächsten Party eingeladen werden. Sie benötigen vier Buttons. Der eine Button verschiebt einen Kontakt in die Liste der eingeladenen Kontakte, ein anderer Button verschiebt einen Kontakt wieder zurück in die allgemeine Liste aller Kontakte. Ein dritter Button löscht einen markierten Kontakt und der vierte Button beendet das Programm. Der Screenshot in Abb. 7.1 zeigt Ihnen, wie das fertige Programm aussehen soll. Den Quelltext dazu finden Sie im Projekt SwingBeispiel.

Verschiedene Abhängigkeiten sollen definiert werden:

- Der Button Exit ist unabhängig von den nachfolgend beschriebenen Abhängigkeiten immer aktiviert. Klickt man darauf, wird das Programm beendet.
- Wenn ein Kontakt auf der rechten Liste markiert ist, ist der Button nicht einladen aktiviert. Ein Klick auf diesen Button verschiebt den Kontakt in die linke Liste.
- Wenn ein Kontakt auf der linken Liste selektiert ist, ist der Button Einladen aktiviert.
- Wenn der Anwender auf diesen Button klickt, wird der selektierte Kontakt in die rechte Liste verschoben.
- Es kann in beiden Listen jeweils nur ein Kontakt markiert sein.

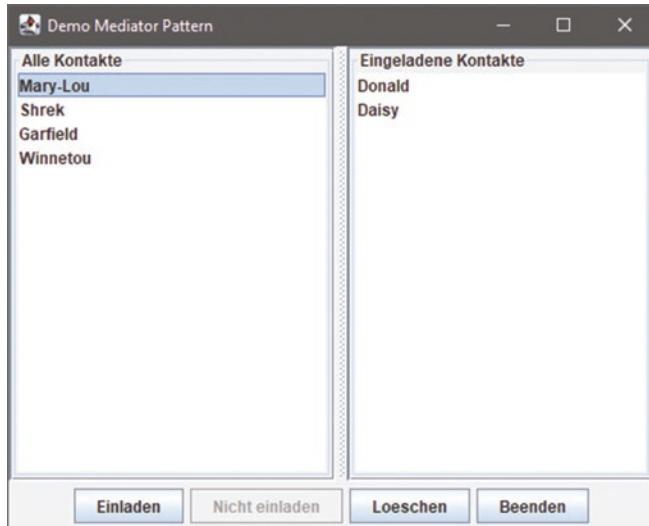


Abb. 7.1 GUI mit Mediator aus dem Beispiel SwingBeispiel

- Der Button Löschen ist aktiviert, wenn ein Eintrag in einer der Listen markiert ist.
- Wenn kein Kontakt markiert ist – weder in der rechten noch in der linken Liste – ist kein Button – außer Exit – aktiviert.
- Wenn ein Kontakt verschoben oder gelöscht wurde, werden alle Markierungen gelöscht und alle Buttons bis auf den Button Exit deaktiviert.

7.4.2 Aufbau der GUI

Den Quelltext des Projekts drucke ich nur gekürzt ab und beschränke mich auf die wesentlichen Eckpunkte. Bitte analysieren Sie den Quelltext selbstständig weiter. Die zwei Listen sind Instanzen der Klasse `JList`. Die vier Buttons sind Instanzen der Klasse `JButton`. Alle Komponenten registrieren sich bei einer Instanz der Klasse `Mediator`, die das Kernstück des Projekts darstellt. Sie speichert Referenzen auf alle beteiligten Komponenten. Außerdem definiert sie Methoden, die aufgerufen werden, wenn ein Event gefeuert wurde. Lassen Sie uns den Ablauf am Beispiel des Einladen-Buttons betrachten. Wenn der Einladen-Button aktiviert wird, wird die Methode `einladen()` des Mediators aufgerufen. Innerhalb dieser Klasse werden zunächst alle Buttons – bis auf den Beenden-Button – deaktiviert. Danach lässt sich die Methode die Modelle der beiden `JList`-Instanzen geben und verschiebt den markierten Eintrag in die Liste der eingeladenen Kontakte.

```
class Mediator {
    private JButton btnEinladen;
    private JButton btnAusladen;
```

```
private JButton btnLoeschen;
private JList alleKontakteList;
private JList eingeladeneKontakteList;

// ... gekürzt

void einladen() {
    btnEinladen.setEnabled(false);
    btnLoeschen.setEnabled(false);
    btnAusladen.setEnabled(false);

    String selectedItem = (String) alleKontakteList.getSelectedValue();

    Object tempModel = alleKontakteList.getModel();
    AlleKontakteModel alleKontakteModel = (AlleKontakteModel)
tempModel;

    tempModel = eingeladeneKontakteList.getModel();
    EingeladeneKontakteModel eingeladeneKontakteModel = (Eingelade-
neKontakteModel) tempModel;

    alleKontakteModel.removeData(selectedItem);
    eingeladeneKontakteModel.addData(selectedItem);

    alleKontakteList.clearSelection();
    eingeladeneKontakteList.clearSelection();
}

}
```

Der Einladen-Button wird nach dem Erzeugen deaktiviert und wird beim Mediator registriert. Der ActionListener, der an den Button übergeben wird, sieht vor, dass die Methode `einladen()` des Mediators aufgerufen wird, wenn der Button aktiviert wird.

```
public class GUI {
    private final JFrame frmMain = new JFrame();

    public GUI() throws Exception {
        // ... gekürzt

        JButton btnEinladen = new JButton("Einladen");
        btnEinladen.setEnabled(false);
        mediator.registerEinladenButton(btnEinladen);
        btnEinladen.addActionListener(ActionEvent e) -> {
```

```

        mediator.einladen();
    });
}
}
}

```

Beachten Sie, dass ich hier die Lambda-Schreibweise verwendet habe. Das erspart das Tippen von ...new ActionListener und @Override public void actionPerformed ... und vereinfacht die Lesbarkeit des Codes deutlich.

Auf entsprechende Weise werden alle Komponenten erzeugt und auf der GUI positioniert. Jede Komponente registriert sich beim Mediator und ruft dort eine bestimmte Methode auf.

7.5 Mediator – Das UML-Diagramm

Zum Beispielprojekt WeinSim sehen Sie das UML-Diagramm in Abb. 7.2.

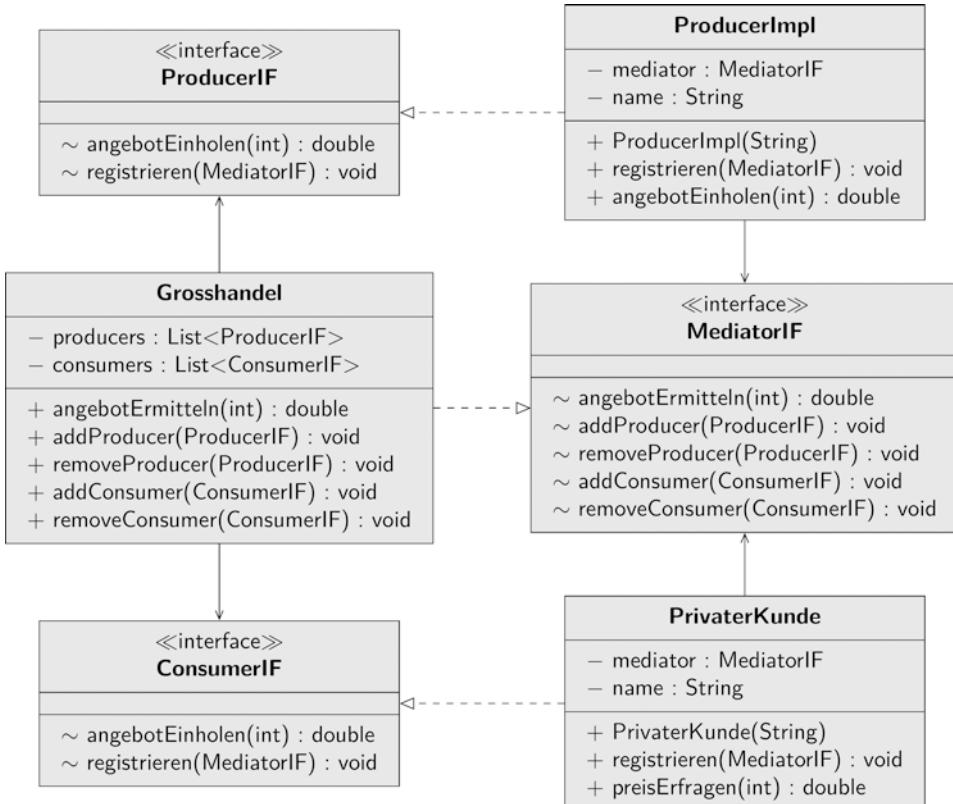


Abb. 7.2 UML-Diagramm des Mediator Pattern (Beispielprojekt WeinSim)

7.6 Kritik an Mediator

Lassen Sie uns das Projekt SwingBeispiel kritisch betrachten. Was fällt auf? Auf der einen Seite erkennen Sie wie bei der Weinhandelssimulation den Vorteil, dass die beteiligten Kollegen-Klassen sich nicht kennen – und sich auch nicht kennen müssen. Sie sind lose gekoppelt, was immer ein Hinweis auf ein gutes Design ist. Sie können jederzeit neue Kollegen einfügen – die Änderungen am Quelltext beschränken sich auf den Mediator.

Auf der anderen Seite soll aber auch der Nachteil nicht verschwiegen werden. Wenn Sie die Mediator-Klasse analysieren, werden Sie eine verhältnismäßig große Klasse finden. Und genau das ist die Gefahr des Mediators: Sie riskieren, eine „God-Class“¹ zu entwickeln, die mit steigender Anzahl beteiligter Komponenten rasch wächst und unübersichtlich wird.

7.7 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Das Mediator Pattern lockert die Bindung zwischen Objekten.
- Viele Beteiligte – Kollegen – hängen voneinander ab.
- Die Kollegen kennen sich untereinander nicht.
- Der Mediator vermittelt zwischen den Kollegen.
- Der Mediator erledigt eigene Aufgaben und kann sehr umfangreich werden.

Zweckbeschreibung

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

„Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Koppelung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es Ihnen, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren.“

¹ „God-Class“ heißt das deswegen, weil nur noch der liebe Gott den Überblick über den Code hat.



Das State Pattern kapselt Zustandsausprägungen in Objekten. Das kann nützlich sein, wenn ein Objekt abhängig von seinem Zustand unterschiedliches Verhalten zeigt. Denken Sie an ein Garagentor. Das Tor kann offen sein, es kann aber auch geschlossen sein. Sie können ein geschlossenes Tor öffnen; es ist jedoch wenig sinnvoll, ein offenes Tor öffnen zu wollen. Doch wie lässt sich verhindern, dass jemand versucht, ein offenes Tor zu öffnen oder ein geschlossenes Tor zu schließen? Das State Pattern löst das Problem dadurch, dass jeder Zustand durch ein eigenes Objekt repräsentiert wird; in der Folge kann jeder nur das tun, was in dem aktuellen Zustand erlaubt sein soll.

8.1 Exkurs: Das Enum Pattern

Lassen Sie uns zunächst einen Exkurs machen und das Enum Pattern betrachten, bevor wir das State Pattern angehen.

8.1.1 Einen Zustand durch Zahlenwerte darstellen

Möchte man zwei Zustandsausprägungen darstellen, kommt man schnell auf die Idee, diese Zustandsausprägungen durch Zahlen-Werte zu repräsentieren und diese Werte an finale Variablen zu übergeben. Innerhalb einer switch-Anweisung fragen Sie die Werte ab und reagieren darauf:

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_8

```

public class TorManager_1 {
    static final int OFFEN = 0;
    static final int GESCHLOSSEN = 1;

    private int zustand = OFFEN;
    void druckeZustand() {
        switch (zustand) {
            case OFFEN -> System.out.println("Tor ist offen");
            case GESCHLOSSEN -> System.out.println("Tor ist geschlossen");
            default -> System.out.println("!!! Fehlerhafter Zustand !!!");
        }
    }

    void setZustand(int zustand) {
        this.zustand = zustand;
    }

    // ... gekürzt
}

```

Achten Sie in diesem Code-Abschnitt auf das Switch Statement(!), bei dem ich hinter dem jeweiligen case die Pfeil-Syntax verwende, die mit Switch Expressions eingefügt wurde (vergleichen Sie meine ersten Erläuterungen zu Switch Expressions in Abschn. 5.7). Von dieser Syntax-Neuerung profitieren auch Switch Statements, denn damit spare ich hier die break-Befehle.

Wenn Sie ein Objekt dieser Klasse erzeugen, können Sie der Methode `setZustand()` entweder eine der definierten Konstanten übergeben oder aber auch fehlerhaft eine beliebige int-Zahl. In Abhängigkeit vom Zustand wird ein jeweils anderer Text auf der Konsole ausgegeben. Im Beispielprojekt Tormanager finden Sie die oben bzw. im folgenden beschriebenen Klassen `Tormanager_1` bis `Tormanager_5`. Ich habe sie der Einfachheit halber in einem gemeinsamen NetBeans-Projekt zusammengefasst. In `Tormanager_1` finden Sie innerhalb der Klasse eine main-Methode, die das oben beschriebene Vorgehen demonstriert.

8.1.2 Einen Zustand durch Objekte darstellen

Ein alternativer Ansatz ist, die Zustandsausprägungen durch Objekte eines eigenen Datentyps darzustellen. Dazu deklarieren Sie eine Klasse `Zustand`. Innerhalb dieser Klasse werden zwei statische finale Variablen von diesem Typ angelegt. Um zu verhindern, dass der Anwender weitere Objekte vom Typ `Zustand` erzeugt, wird der Konstruktor private deklariert – ein Zugriff ist jetzt nur noch innerhalb der Klasse möglich. Die Technik ist die

gleiche, wie Sie sie schon beim Singleton Pattern gesehen haben. Sie finden besagten Datentyp als statische innere Klasse in der Klasse TorManager_2.

```
public class TorManager_2 {  
    // ...  
    public static class Zustand {  
  
        public static final Zustand OFFEN = new Zustand("offen");  
        public static final Zustand GESCHLOSSEN = new  
                Zustand("geschlossen");  
        private final String beschreibung;  
  
        private Zustand(String beschreibung) {  
            this.beschreibung = beschreibung;  
        }  
        // ... gekürzt  
        private Zustand zustand = Zustand.OFFEN;  
  
        void setZustand(Zustand zustand) {  
            if (zustand != null)  
                this.zustand = zustand;  
        }  
        // ... gekürzt  
    }  
}
```

Jetzt können nur noch die in der Klasse Zustand vorgegebenen Zustandsausprägungen verwendet werden. Diese Lösung ist also typsicher geworden. Da dieser Ansatz sehr oft verwendet wurde, hat man ihn Enum Pattern genannt. Ein Nachteil ist, dass Sie die Zustandsausprägungen während der Verarbeitung nicht mehr in einer switch-Anweisung abfragen können.

8.1.3 Umsetzung in der Java-Klassenbibliothek

Programmierer stehen oft vor der Notwendigkeit, Zustandsausprägungen typsicher darzustellen. Dafür gibt es in Java die Enumerations. Der Compiler übersetzt Enums so, dass der Bytecode genau dem oben gezeigten Enum Pattern entspricht. Anstelle von `class` schreiben Sie jetzt `enum`. Außerdem wurde dem Programmierer ein wenig Schreibarbeit abgenommen; die Modifizierer `static final` und die explizite Objekterzeugung können entfallen; die verschiedenen Optionen werden durch Kommata abgetrennt. Sie finden diese Variante in der Klasse Tormanager_3.

```
enum Zustand {  
    OFFEN("offen"), GESCHLOSSEN("geschlossen");  
    private final String beschreibung;  
    private Zustand(String beschreibung) {  
        this.beschreibung = beschreibung;  
    }  
}
```

Enums erlauben es Ihnen wieder, Zustandsausprägungen in einer switch-Anweisung abzufragen:

```
void druckeZustand() {  
    switch (zustand) {  
        case OFFEN -> System.out.println("Das Tor ist offen");  
        case GESCHLOSSEN -> System.out.println("Das Tor ist ge-  
schlossen");  
    }  
}
```

Vermissen Sie in diesem Switch Statement den default-Fall? Da es sich bei dem überprüften Zustand um eine enum-Datentyp handelt, brauchen wir den nicht mehr, solange alle einzelnen Werte auch in den Fällen abgedeckt werden. Das ist hier der Fall, und damit entfällt die Notwendigkeit für den default-Fall. Eine weitere praktische Anpassung von Java, die mit den Switch Expressions eingeführt wurde und auch für die Switch Statements zur Verfügung steht.

Im nächsten Abschnitt werden Sie die Zustände eines Objekts ändern.

8.2 Den Zustand eines Objekts ändern

Im vorigen Absatz haben Sie gesehen, wie Sie Zustandsausprägungen typsicher definieren können. Es kann nun interessant sein, wie Sie einen Zustand in einen anderen überführen und wie das Verhalten eines Objekts dadurch geändert wird.

8.2.1 Ein erster Ansatz

In der Klasse Tormanager_4 betrachten Sie im ersten Schritt die Frage, wie Sie einen Zustand in einen anderen überführen können:

```
public class TorManager_4 {  
    enum Zustand {  
        // ... gekürzt
```

```
}

private Zustand zustand = Zustand.OFFEN;

private void oeffnen() {
    System.out.println("Das Tor wird geöffnet");
    this.zustand = Zustand.OFFEN;
}

private void schliessen() {
    System.out.println("Das Tor wird geschlossen");
    this.zustand = Zustand.GESCHLOSSEN;
}
}
```

Eigentlich ist diese Lösung doch optimal, oder? Wenn ein Tor geschlossen ist, machen Sie es auf, um es in den anderen Zustand zu überführen. Wenn es offen ist, schließen Sie es und haben dann auch einen anderen Zustand. Doch bei dieser Kodierung ist es möglich, dass der Anwender ein offenes Tor öffnet oder ein geschlossenes schließt. Und genau das sollte nicht passieren. Fragen Sie die Zustandsausprägungen also vorher ab!

8.2.2 Ein zweiter Ansatz

Fügen Sie in jede Methode eine if-Anweisung ein, die das Öffnen oder Schließen nur erlaubt, wenn das Objekt dadurch in einen anderen Zustand überführt wird. Wenn der Anwender versucht, ein geöffnetes Tor zu öffnen oder ein geschlossenes zu schließen, weisen Sie ihn mit einer Fehlermeldung darauf hin. Diesen Code finden Sie in der Klasse TorManager_5.

```
public class TorManager_5 {
    enum Zustand {
        // ... gekürzt
    }

    private Zustand zustand = Zustand.OFFEN;

    private void oeffnen() {
        if (zustand == Zustand.GESCHLOSSEN) {
            System.out.println("Das Tor wird geöffnet");
            this.zustand = Zustand.OFFEN;
        }
    }
}
```

```

        else {
            System.out.println("Das Tor ist bereits offen.");
        }
    }

private void schliessen() {
    if (zustand == Zustand.OFFEN) {
        System.out.println("Das Tor wird geschlossen");
        this.zustand = Zustand.GESCHLOSSEN;
    }
    else {
        System.out.println("Das Tor ist bereits geschlossen.");
    }
}
}

```

Diese Lösung funktioniert tadellos. Aber wo könnte sie Schwächen zeigen? Sie ist dann nicht mehr nützlich, wenn weitere Zustandsausprägungen hinzukommen und umfangreiche Abhängigkeiten zwischen ihnen bestehen.

Ich möchte das Beispiel dadurch erweitern, dass Ihr Auftraggeber einen neuen Zustand anfordert. Das Tor soll nicht nur geschlossen werden, sondern auch mit einem Schloss abgeschlossen werden können. Sie haben jetzt den dritten Zustand ABGESCHLOSSEN. Der Zustand ABGESCHLOSSEN darf nur gesetzt werden, wenn das Tor geschlossen ist. Umgekehrt kann der Zustand ABGESCHLOSSEN nur in den Zustand GE-SCHLOSSEN überführt werden. Abb. 8.1 zeigt die drei Zustandsausprägungen und deren mögliche Übergänge.

Sie müssen drei Zustandsausprägungen in Beziehung setzen und das Verhalten des Objekts davon abhängig machen. Wenn ein vierter Zustand hinzukommt, können Sie sich ausmalen, welche Arbeit auf Sie zukommen wird.

Wie für diesen Fall beispielsweise die Methode `oeffnen()` aussehen könnte, zeigt Ihnen das folgende Listing, das Sie im Beispielcode nur auskommentiert finden:

```

private void oeffnen() {
    switch (zustand) {
        case OFFEN -> System.out.println("Das Tor ist bereits offen.");
        case GESCHLOSSEN -> {
            System.out.println("Das Tor wird geöffnet");
            this.zustand = Zustand.OFFEN;
        }
        case ABGESCHLOSSEN -> System.out.println("Das Tor ist ver-schlossen");
    };
}

```

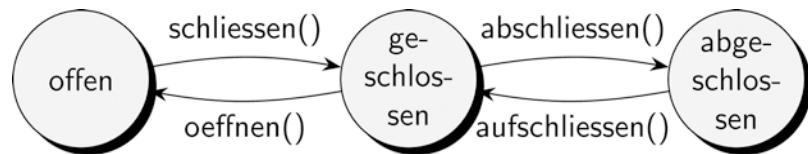


Abb. 8.1 Drei Zustandsausprägungen und deren Übergänge

In entsprechender Weise müssten die weiteren vier Methoden für jeden Zustand das richtige Verhalten definieren – selbst, wenn das Verhalten nur darin besteht, eine passende Fehlermeldung auszugeben. Der Umfang der Methoden nimmt mit jedem neuen Zustand zu. Die Klasse wird unübersichtlich und schwer zu warten. Fehler können sich sehr schnell einschleichen. Kurz gesagt: Es ist höchste Zeit für das State Pattern.

8.3 Das Prinzip des State Pattern

Die Definition der Zustandsausprägungen durch Enumerations reicht jetzt nicht mehr aus. Beschreiben Sie jeden Zustand durch eine eigene Klasse.

8.3.1 Die Rolle aller Zustände definieren

Um die Zustandsausprägungen austauschen zu können, müssen diese einen gemeinsamen Datentyp haben. Sie legen zuerst die abstrakte Klasse Zustand an, in der alle Methoden aus dem Zustandsdiagramm vorgesehen sind. Außerdem hält diese Klasse eine Referenz auf das Tor, die ihr im Konstruktor übergeben wird. Den folgenden Code finden Sie im Beispielprojekt StatePattern_1.

```

public abstract class Zustand {
    public final Tor tor;

    Zustand(Tor tor) {
        this.tor = tor;
    }

    abstract void oeffnen();
    abstract void schliessen();
    abstract void abschliessen();
    abstract void aufschliessen();
}
  
```

Jeder Zustand wird durch eine eigene Klasse repräsentiert und erweitert die abstrakte Klasse. Sie definieren also innerhalb des Zustands, was passieren soll, wenn eine bestimmte Methode, beispielsweise `oeffnen()`, aufgerufen wird. Wenn Sie einen Zustand definieren, müssen Sie sich überlegen, wie eine Methode in diesem Zustand aussehen muss. Lassen Sie uns das Beispiel anhand des Zustands Offen durchspielen. Wenn ein Tor offen ist und jemand versucht, es erneut zu öffnen, drucken Sie eine Fehlermeldung aus. Genauso wenig kann ein offenes Tor auf- oder abgeschlossen werden; daher drucken Sie auch in diesen Methoden Fehlermeldungen aus. Wenn der Anwender jedoch ein offenes Tor schließen möchte, ändern Sie den Zustand nach geschlossen.

```
public class Offen extends Zustand {  
    Offen(Tor tor) {  
        super(tor);  
    }  
  
    @Override  
    public void oeffnen() {  
        System.out.  
            println("Das Tor ist schon offen.");  
    }  
  
    @Override  
    public void schliessen() {  
        System.out.println("Das Tor wird geschlossen.");  
        tor.setZustand(new Geschlossen(tor));  
    }  
  
    @Override  
    public void abschliessen() {  
        System.out.println("Erst das Tor schließen.");  
    }  
  
    @Override  
    public void aufschliessen() {  
        System.out.  
            println("Das Tor ist nicht abgeschlossen.");  
    }  
}
```

Entsprechend werden die anderen Zustand-Klassen definiert.

Wie wirkt sich diese Änderung auf das Tor aus? Die Klasse `Tor` definiert ein Feld `zustand`, das den aktuellen Zustand referenziert. Auf diesem Zustand können alle Methoden, die Sie aus dem Zustandsdiagramm kennen, aufgerufen werden. Das Tor selbst kann

also eine Methode `oeffnen()` definieren, die den Aufruf an das Feld `zustand` weiterreicht. Hier drucke ich den gekürzten Quelltext der Klasse `Tor` ab.

```
public class Tor {  
    private Zustand zustand = new Offen(this);  
  
    public void setZustand(Zustand zustand) {  
        this.zustand = zustand;  
    }  
  
    public void oeffnen() {  
        zustand.oeffnen();  
    }  
    // ... gekürzt  
}
```

Das Tor ruft auf dem aktuellen Zustand die gewünschte Methode auf. Es hat jedoch keine Ahnung, welcher Zustand gerade hinterlegt ist. Wenn Sie zweimal dieselbe Methode auf ein- und demselben Objekt aufrufen, ist unter Umständen ein anderer Zustand aktiv, das Verhalten des Objekts ist ein anderes und man hat den Eindruck, dass man mit der Instanz einer anderen Klasse arbeitet.

8.3.2 Das Projekt aus der Sicht des Clients

In der Client-Klasse des Projekts wird das Tor zum Einsatz gebracht. Vergleichen Sie den Quelltext, den ich hier nur gekürzt abdrucke, mit der Konsolenausgabe:

```
public static void main(String[] args) {  
    TOR.oeffnen();  
    TOR.schliessen();  
    TOR.oeffnen();  
    TOR.aufschliessen();  
    TOR.abschliessen();  
    TOR.schliessen();  
    TOR.abschliessen();  
    TOR.oeffnen();  
    TOR.aufschliessen();  
    TOR.oeffnen();  
}
```

Die resultierende Konsolenausgabe finden Sie in Abb. 8.2.

Beachten Sie, wie oft die Methode `oeffnen()` aufgerufen wird: Insgesamt 4 mal. Bei jedem Aufruf reagiert der Kontext, also das Tor, anders. Für den Client ist diese

```

run:
Tor ist im Zustand Offen.
    Das Tor ist schon offen; es kann nicht noch einmal geöffnet werden.
Tor ist im Zustand Offen.
    Das Tor wird geschlossen.
Tor ist im Zustand Geschlossen.
    Das Tor wird geöffnet.
Tor ist im Zustand Offen.
    Das Tor ist nicht abgeschlossen und kann deshalb nicht aufgeschlossen werden.
Tor ist im Zustand Offen.
    Schließen Sie das Tor, bevor Sie es abschließen.
Tor ist im Zustand Offen.
    Das Tor wird geschlossen.
Tor ist im Zustand Geschlossen.
    Das Tor wird abgeschlossen.
Tor ist im Zustand Abgeschlossen.
    Das Tor ist verschlossen und kann nicht geöffnet werden.
Tor ist im Zustand Abgeschlossen.
    Das Tor wird aufgeschlossen.
Tor ist im Zustand Geschlossen.
    Das Tor wird geöffnet.
BUILD SUCCESSFUL (total time: 0 seconds)
|
```

Abb. 8.2 Konsolenausgabe des Beispielprojekts StatePattern_1

Lösung sehr komfortabel – er weiß nicht, wie das Tor intern arbeitet, er muss sich vor allem nicht mit den verschiedenen Zustand-Klassen beschäftigen.

Dennoch sollen zwei Nachteile nicht verschwiegen werden. Zunächst brauchen Sie für jeden Zustand eine eigene Klasse. Einige Zustandsausprägungen hängen „irgendwie“ voneinander ab, andere gar nicht. Dadurch besteht die Gefahr, dass das Projekt unübersichtlich wird. Ohne ein übersichtliches Zustandsdiagramm sind Sie unter Umständen aufgeschmissen.

Ein zweiter Punkt kann zum Problem werden. Sie erzeugen bei jedem Statuswechsel ein neues Objekt. Wenn Sie häufige Statuswechsel haben oder die Zustand-Klasse beim Instanziieren sehr kostenintensiv ist, ist diese Lösung nicht zu favorisieren. Lassen Sie uns im Folgenden alternative Implementierungen betrachten.

8.3.3 Veränderungen des Projekts

Zwei Optionen möchte ich Ihnen vorstellen. Die erste Alternative ist, die State-Objekte zentral im Kontext zu verwalten. Die zweite Alternative sieht vor, die State-Objekte in der Superklasse aller Zustand-Klassen anzulegen und bei jedem Methodenaufruf das richtige Objekt an den Aufrufer zu übergeben.

8.3.3.1 State-Objekte zentral im Kontext verwalten

Wenn Sie nicht bei jeder Statusänderung ein neues Objekt erzeugen möchten, muss eine alternative Lösung her. Lassen Sie den Kontext Variablen definieren, die jeden Zustand repräsentieren. Außerdem definiert der Kontext für jeden Zustand eine Methode, die das Feld Zustand aktuellerZustand auf den gewünschten Zustand setzt. Den Code dafür finden Sie im Beispielprojekt StatePattern_2:

```
public class Tor {  
    private final Zustand offenZustand = new Offen(this);  
    private final Zustand geschlossenZustand = new Geschlossen(this);  
    private final Zustand abgeschlossenZustand = new Abgeschlossen(this);  
    private Zustand aktuellerZustand = offenZustand;  
  
    public void setOffenZustand() {  
        aktuellerZustand = offenZustand;  
    }  
  
    public void oeffnen() {  
        aktuellerZustand.oeffnen();  
    }  
    // ... gekürzt  
}
```

Die Zustand-Klassen übergeben jetzt keine Zustand-Objekte mehr an den Kontext, sondern weisen den Kontext an, einen bestimmten Zustand einzunehmen. Wie er das macht, ist seine Sache. Dadurch wird der Kontext jedoch sehr stark an die Zustand-Klassen gebunden.

8.3.3.2 State-Objekte als Rückgabewerte von Methodenaufrufen

Eine andere alternative Implementierung sieht vor, dass Sie State-Objekte grundsätzlich als Rückgabewerte zurückgeben. Wenn der Zustand sich nicht ändert, gibt er eine Referenz auf sich selbst zurück, also `this`. Dazu definieren Sie in der abstrakten Superklasse Variablen, die die verschiedenen Zustandsausprägungen repräsentieren. Diese Variante finden Sie im Beispielprojekt StatePattern_3:

```
public abstract class Zustand {  
    protected static final Zustand OFFEN = new Offen();  
    protected static final Zustand GESCHLOSSEN = new Geschlossen();  
    protected static final Zustand ABGESCHLOSSEN = new Abgeschlossen();  
  
    abstract Zustand oeffnen();  
  
    // ... gekürzt  
}
```

Die finalen Variablen sind `protected` deklariert, um den Subklassen Zugriff darauf zu ermöglichen.

```
public class Offen extends Zustand {  
    @Override  
    public Zustand oeffnen() {  
        System.out.println("Das Tor ist schon offen.");  
        return this;  
    }  
  
    @Override  
    public Zustand schliessen() {  
        System.out.println("Das Tor wird geschlossen.");  
        return super.GESCHLOSSEN;  
    }  
    // gekürzt  
}
```

Der Kontext ist jetzt verantwortlich dafür, dass er das erhaltene Zustand-Objekt als aktuellen Zustand setzt. Damit ist das Projekt sehr flexibel, neue Zustand-Klassen können problemlos eingefügt werden. Dadurch, dass der Kontext jetzt keine `set`-Methode mehr anbieten muss, werden nicht versehentlich falsche Zustandsausprägungen übergeben oder gesetzt.

8.4 Das State Pattern in der Praxis

Sie werden das State Pattern in der Praxis recht oft wiederfinden. Beispielsweise wird bei Netzwerkprotokollen oft mit Zustandsausprägungen gearbeitet. Als Beispiel habe ich das RFC 1939 herausgegriffen, in dem das POP3 beschrieben wird. Sie finden den Text des RFC im Unterordner zum State Pattern.

Das RFC 1939 kennt drei verschiedene Zustandsausprägungen: AUTHORIZATION, TRANSACTION und UPDATE. Diese werden im RFC auch tatsächlich als State bezeichnet. Der POP3-Server wartet auf Anfragen. Wenn eine Anfrage kommt, wird eine TCP-Verbindung hergestellt. Die Session befindet sich zunächst im Zustand AUTHORIZATION, in dem der Name und das Passwort des Nutzers abgefragt werden. Wenn der Name identifiziert werden kann und das Passwort stimmt, geht die Session in den Zustand TRANSACTION über. In diesem Zustand können Befehle an den Server geschickt werden. Beispielsweise kann eine Liste der gespeicherten E-Mails abgefragt werden. Ferner kann eine E-Mail auch zum Löschen vorgemerkt werden. Wenn der Befehl QUIT kommt, wechselt die Session in den Zustand UPDATE. In diesem Zustand werden die zum Löschen markierten E-Mails tatsächlich gelöscht. Danach wird die Verbindung beendet.

Wenn der Befehl QUIT im Zustand AUTHORIZATION gegeben wird, wird die Session ohne UPDATE beendet. Der Aufruf von QUIT hat also in Abhängigkeit vom Zustand zwei vollkommen unterschiedliche Auswirkungen. Befehle, die sich auf die gespeicherten Nachrichten beziehen, sind nur im Zustand TRANSACTION zulässig.

Schauen Sie sich das Beispielprojekt POP3 an, das den Übergang dieser drei Zustandsausprägungen demonstriert. Ich habe im Package states vier Zustand-Klassen hinterlegt: Drei für die im Protokoll definierten Ausprägungen und eine Klasse für den Zustand START, der dann benötigt wird, wenn der Server auf Anfragen wartet. In diesem Package gibt es außerdem eine Klasse POP3_Session. Diese Klasse ist der Kontext, auf den die Clients zugreifen. Der Client „sieht“ nur den Kontext. Er hat keine Information darüber, dass eine Vielzahl von Zustand-Klassen im Hintergrund benötigt wird. Betrachten Sie unter diesem Aspekt die main-Methode der Klasse App1Start.

8.5 State – Das UML-Diagramm

Das UML-Diagramm zu State-Pattern aus dem Beispielprojekt StatePattern_3 sehen Sie in Abb. 8.3.

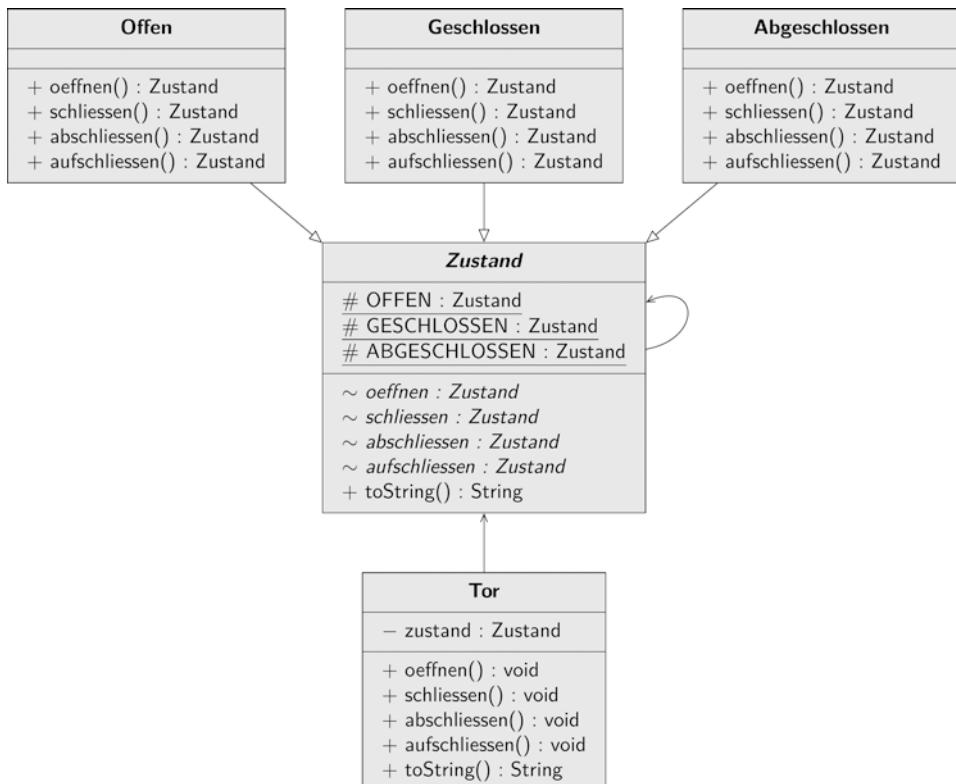


Abb. 8.3 UML-Diagramm des State Pattern (Beispielprojekt StatePattern_3)

8.6 Zusammenfassung

Gehen Sie das Kapitel nochmal stichwortartig durch:

Das Enum Pattern erlaubt es, verschiedene Zustandsausprägungen typischer zu definieren.

- Im Regelfall wird ein privater Konstruktor definiert.
- Instanzen der Klassen werden static und final deklariert.
- Das Enum Pattern wird seit Java 5 von der Enumeration umgesetzt.

Die wichtigsten Punkte zum State Pattern:

- Ein Objekt kann verschiedene Zustandsausprägungen haben.
- Abhängig vom Zustand zeigt das Objekt unterschiedliches Verhalten.
- Der Übergang von einem Zustand in einen anderen sollte in den Zustand-Klassen selbst festgelegt sein.
- Die Enumeration reicht hierfür in der Regel nicht aus.
- Für den Client sieht es so aus, als hätte er es mit der Instanz einer anderen Klasse zu tun.
- Die Zustandsausprägungen können entweder bei jedem Zustandswechsel neu erzeugt werden oder an zentraler Stelle gehalten werden. Welche Lösung Sie wählen, hängt davon ab, ob der Zustand sich voraussichtlich oft ändert oder die Erzeugung einer Instanz einer Zustand-Klasse kostenintensiv ist. In diesem Fall erzeugen Sie die Zustandsausprägungen beim Programmstart und speichern sie an einer zentralen Stelle.

Zweckbeschreibung

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

„Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.“



In diesem Kapitel möchte ich Ihnen ein Verhaltensmuster zeigen, das einen Befehlsaufruf von der Befehlsausführung löst. Was ist damit gemeint? Stellen Sie sich vor, dass der Chef Ihres Unternehmens in einer Abteilungsleiterrunde sagt: „Ich brauche die aktuellen Statistiken!“ Die Bosheit wird garantiert in der Form zuschlagen, dass Ihr eigener Abteilungsleiter den Finger hebt und sagt: „Ich kümmere mich darum!“ Vorhersehbar ist auch, dass er Sie mit den Worten: „Mach du mal, du kannst das doch!“ mit der Ausführung beauftragt. Was ist da passiert? Der Chef fordert eine Maßnahme an, Sie führen sie aus und dazwischen gibt es eine Instanz (Ihren Abteilungsleiter), die die Aufgabe weiterleitet. Anders ausgedrückt: Befehlsaufruf und Ausführung sind losgelöst voneinander und nur über ein Befehlsobjekt lose gekoppelt – und genau das ist das Ziel des Command Patterns.

9.1 Befehle in Klassen kapseln

Doch verlassen wir zunächst die Szene im Büro und überlegen uns, wie man eine Urlaubsreise bucht. Sie gehen ins Reisebüro und geben Ihre Reisewünsche an. Der Mensch im Reisebüro tippt eine Weile auf seinem Computer und Sie können beruhigt in Urlaub fahren. Tatsächlich aber führt nicht das Reisebüro die Reise durch, sondern ein Reiseveranstalter, der sich um den Flug und am Reiseziel um das Hotel kümmert. Das schauen wir uns mal genauer an, beschäftigen uns aber zusätzlich auch mit der in Java 8 hinzu gekommenen Date and Time API, die das alte java.util.Date ablöst.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_9

9.1.1 Version 1 – Grundversion

Im ersten Schritt, der mit dem Command Pattern noch gar nichts zu tun hat, werde ich ein Reisebüro und einen Reiseveranstalter programmieren. Das Reisebüro definiert die Methode `reiseBuchen()`. Ihr werden das Ziel, Abreise- und Rückreisetag als Argumente übergeben. Die Daten werden in ein Objekt der Klasse `Reise` gepackt und an den Reiseveranstalter übergeben, der die Reise dann real durchführt, also den Reise-Befehl ausführt. Sie finden das folgende Beispiel im Beispielprojekt `Reise_1`.

```
public class Reise {
    final String ziel;
    final String von;
    final String bis;
    final DateTimeFormatter dtFormatter =
        DateTimeFormatter.ofPattern("dd.MM.yyyy");

    Reise(String ziel, LocalDate von, LocalDate bis) {
        this.ziel = ziel;
        this.von = von.format(dtFormatter);
        this.bis = dtFormatter.format(bis);
    }

    @Override
    public String toString() {
        return "Reise nach " + ZIEL + " für die Zeit von " + VON + " bis "
            + BIS;
    }
}
```

Beachten Sie die Verwendung der Datumsfunktionen: Zunächst einmal bekommt die Methode `durchfuehren()` das Start- und Enddatum der Reise als Werte vom Typ `LocalDate`. Dazu muss die Klasse `Reise` den import `java.time.LocalDate` nutzen.

Dieses `LocalDate` gehört zur Date and Time API, die mit dem Java Specification Request JSR-310 in Java 8 implementiert wurde. Anlass, das bis dahin verwendete `java.util.Date` zu ersetzen, waren dessen Schwächen: Mangelnde Typsicherheit, mangelnde Erweiterbarkeit, unklare Zuständigkeiten (Date mit Zeitangabe gemeinsam, aber ohne Zeitzone) und einige andere.

Das Paket `java.time` und seine vier Unterpakete `chrono`, `format`, `temporal` und `zone` bieten dagegen weitgehend vereinheitlichte Befehle für diverse Datums- und Zeit-Typen, konsistente und klare Befehlsdefinitionen und threadsichere unveränderbare Objekte.

Schauen Sie sich den Beispielcode oben noch mal an:

Es geht im Konstruktor um das Umwandeln eines LocalDate in einen String. Das geschieht mittels eines DateTimeFormatter, der mit dem für uns in Deutschland gebräuchlichen Muster „dd.MM.yyyy“ für die Darstellung bzw. Umwandlung versehen wird. Diese Umwandlung ist nicht zwingend notwendig, aber ohne sie würde das Datum nach dem ISO-Standard ISO-8601 im Format yyyy-MM-dd ausgegeben. Entscheiden Sie selbst, was Ihnen mehr liegt.

Übrigens hat eine Variable vom Typ LocalDate keinerlei Zeit-Anteile. Sie beinhaltet Jahr, Monat und Tag sowie alle dafür notwendigen Methoden. Die Gegenstücke für das Verarbeiten von Zeit-Informationen behandeln wir in einem späteren Kapitel noch mal separat.

Die eigentliche Umwandlung in eine Zeichenkette ist dann auf zwei Arten möglich, die ich beide mal verwendet habe, um Sie ihnen vorzustellen

- Entweder verwendet man die `format`-Methode des LocalDate und gibt als Parameter den DateTimeFormatter an
- Oder man nutzt die `format`-Methode des DateTimeFormatter und übergibt ihm als Parameter das LocalDate

In beiden Fällen kommt aber der geeignet formatierte Text heraus. Wie man ein LocalDate erzeugt, sehen wir gleich bei der Erstellung der Testklasse.

Die Reiseveranstalter handeln mit Hotels, Fluglinien und lokalen Busunternehmen Verträge aus und führen die Reise durch. Jeder Veranstalter bekommt seine eigene Firma.

```
public class Reiseveranstalter {  
    private final String firma;  
  
    Reiseveranstalter(String firma) {  
        this.firma = firma;  
    }  
  
    void durchfuehren(Reise reise) {  
        System.out.println(firma + " führt folgende Reise durch: "  
+ reise);  
    }  
  
    @Override  
    public String toString() {  
        return firma;  
    }  
}
```

Eine Instanz des Reisebüros wird erzeugt, indem ihrem Konstruktor ein Objekt vom Typ Reiseveranstalter übergeben wird. Der Methode `durchfuehren()` werden die Daten der Reise als Parameter übergeben, die als Zeichenketten gespeichert werden.

```
public class Reisebuero {
    private final Reiseveranstalter veranstalter;

    Reisebuero(Reiseveranstalter veranstalter) {
        this.veranstalter = veranstalter;
    }

    void reiseBuchen(String ziel, LocalDate von, LocalDate bis) {
        var reise = new Reise(ziel, von, bis);
        veranstalter.durchfuehren(reise);
    }
}
```

In der main-Methode der Testklasse werden ein Reisebüro und ein Reiseveranstalter angelegt. Danach werden drei Reisen gebucht. Das schauen wir uns wegen der Date and Time API einmal genauer an. Die relevanten Code-Stellen habe ich fett hervorgehoben.

```
public class Testklasse {

    public static void main(String[] args) {
        var veranstalter = new Reiseveranstalter("ABC-Reisen");
        var reisebuero = new Reisebuero(veranstalter);
        LocalDate von, bis;

        // eine Reise buchen
        von = LocalDate.of(2021, Month.NOVEMBER, 4);
        bis = von.withDayOfMonth(15);
        reisebuero.reiseBuchen("Washington", von, bis);

        // eine weitere Reise buchen
        von = toDate("30.12.2021");
        bis = von.with(nextOrSame(DayOfWeek.TUESDAY));
        reisebuero.reiseBuchen("Rom", von, bis);

        // und noch eine Reise buchen
        von = toDate("02.10.2021");
        bis = von.plusWeeks(2);
        reisebuero.reiseBuchen("Peking", von, bis);
    }
}
```

```
private static LocalDate toDate(String datum) {  
    LocalDate tempDate;  
  
    try {  
        tempDate = LocalDate.parse(datum,  
                                    ofLocalizedDate(FormatStyle.MEDIUM));  
    } catch (DateTimeParseException ex) {  
        tempDate = LocalDate.now();  
        ex.printStackTrace();  
    }  
    return tempDate;  
}  
}
```

Beachten Sie, dass Sie für die Verwendung der Date and Time API einige imports angeben müssen. Für dieses Beispiel hier sind das

```
import java.time.DayOfWeek;  
import java.time.LocalDate;  
import java.time.Month;  
import java.time.format.DateTimeFormatter;  
import java.time.format.DateTimeParseException;  
import static java.time.temporal.TemporalAdjusters.nextOrSame;
```

Gehen wir die fett hervorgehobenen Zeilen in der Testklasse noch mal im Detail durch.

Das Startdatum der ersten Reise erzeuge ich mittels von = `LocalDate.of(...)`.

Beachten Sie, dass dabei kein new verwendet wird. Die Methode `of` erzeugt das Objekt für mich (genauer gesagt tut das eine private Methode `create`, die von `of` aufgerufen wird) und gibt es zurück. Diesen „Fabrik“-Ansatz werden wir in den Kapiteln zur Abstrakten Fabrik und zur Factory Method näher kennen lernen. Als Parameter für den Aufruf verweise ich Jahr, Monat und Tag jeweils separat. Dabei nutze ich für den Monat die im Standard vorhandene Enumeration `Month` mit den englischen Monatsnamen, in diesem Fall also `Month.AUGUST`. Die erste Reise beginnt also am 1.8.2021.

Die Kundin möchte aber am 15. desselben Monats wieder zurück sein, also lasse ich mir mittels `von.withDayOfMonth(15)` ein entsprechendes Enddatum einfach ausrechnen. Dabei erzeugt die Date and Time API jetzt eine neue `LocalDate` mit dem auf den 15. geänderten Tag.

Für die zweite und dritte Reise nutze ich die selbst geschriebene Methode `toDate`, die mir aus einem Text ein entsprechendes `LocalDate` ermittelt. Dafür verwende ich die `parse`-Methode des `LocalDate` und gebe ihr – wie bereits in der Klasse `Reise` erläutert – ebenfalls einen `DateTimeFormatter` mit. Der lässt sich nämlich in beide Richtungen – sowohl für das Parsen als auch für das Formatieren – verwenden. Da der Parser eine Exception werfen kann, wenn er mit dem Text nichts anfangen kann, muss hier ein ent-

sprechender try-catch-Block stehen. Für den Fall der Exception gebe ich `null` zurück. Sie sollten hier nicht in Versuchung geraten, z. B. das aktuelle Datum zu setzen. Das ist in der Folge möglicherweise Ursache für größere Probleme und lässt sich kaum nachvollziehen.

Die Dauer der zweiten Reise ist so eine Sache. Der Kunde möchte am „darauf folgenden Dienstag“ zurück sein. Anstatt jetzt selbst im Kalender zu blättern, kann ich das elegant mit dem Befehl `with(nextOrSame(DayOfWeek.TUESDAY))` lösen: Passe das Startdatum auf den „nächsten oder gleichen Wochentag Dienstag“ an und gib es mir als neues Datum zurück. Für die Wochentage gibt es ebenfalls eine Enumeration, die mir die Frage nach dem Start der Woche für den Index (war der Montag jetzt die 0 oder die 1?) einfacher erspart.

Die dritte Reise soll genau 2 Wochen dauern. Auch das lässt sich über `plusWeeks` einfach lösen. In diese Befehlskategorie fallen auch `plusDays`, `plusMonths` und `plusYears`.

In der Klasse `Reise` und in der Testklasse finden Sie noch auskommentierte alternative Codezeilen für den `DateTimeFormatter`. Schauen Sie sich dazu die Kommentare in der Klasse `Reise` einfach selbstständig an.

Nach diesem Ausflug in die Möglichkeiten der Date and Time API jetzt aber wieder zurück zum eigentlichen Thema: Was Sie bei der Analyse beachten sollten, ist, dass das Reisebüro Methoden des Reiseveranstalters direkt aufruft. Das wird sich aber gleich ändern!

9.1.2 Weitere Anbieter treten auf

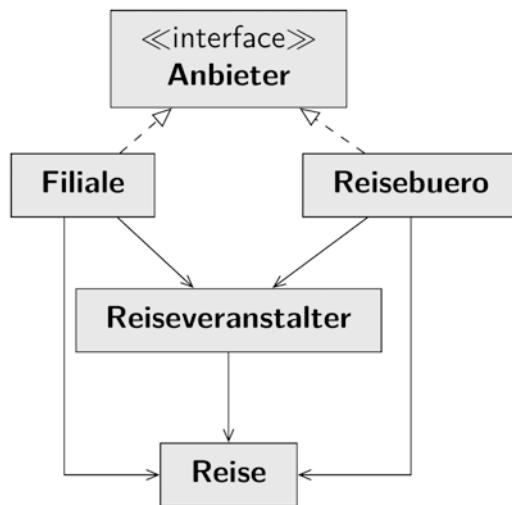
Das Geschäft mit den Reisen boomt – das Reisebüro muss noch eine Filiale eröffnen. Sowohl Reisebüro als auch Filiale habe ich unter der Schnittstelle `Anbieter` zusammengefasst. Das Interface schreibt die Methode `reiseBuchen()` vor, die Sie noch aus der letzten Version kennen. Das Beispielprojekt `Reise_2` simuliert diese Situation.

```
public interface Anbieter {
    void reiseBuchen(String ziel, LocalDate von, LocalDate bis);
}
```

Ansonsten hat der Code der Klassen sich nicht geändert. Wie werden die Klassen verwendet? Im Beispielcode der Testklasse finden Sie folgendes Vorgehen: Sie legen einen Reiseveranstalter an. Die Instanz übergeben Sie an den Konstruktor eines Anbieters. Sie rufen, um eine Reise zu buchen, die Methode `reiseBuchen()` des Anbieters auf – gegenüber der vorigen Version hat sich nichts geändert. Abb. 9.1 zeigt das Klassendiagramm dieser Projektversion.

Ich habe bei der Zusammenstellung der Reisedaten aber noch ein paar weitere Möglichkeiten der Datumsmanipulation genutzt, um Ihnen weitere Möglichkeiten der Date and Time API zu zeigen.

Abb. 9.1 Klassendiagramm des Beispielprojekts Reise_2



Wenn Sie das Projekt Reise_2 analysieren, stellen Sie fest, dass der Code der Klassen `Filiale` und `Reisebuero` nahezu identisch ist. Redundanter Code ist immer ein Hinweis auf ein ungeeignetes Programmdesign. Vor allem ist redundanter Code eine nicht zu unterschätzende Fehlerquelle. Es gilt also, doppelten Code zu eliminieren.

Wie würden Sie vorgehen, wenn Sie in der Mathematik viele gleiche Faktoren haben, also beispielsweise $5 * 3 + 5 * 2 + 5 * 9$? Sie ziehen den gleichen Faktor vor die Klammer: $5 * (3 + 2 + 9)$. Genau das Gleiche machen Sie mit dem redundanten Programmcode bei den Anbietern.

9.1.3 Einen Befehl kapseln

Im Beispielprojekt Reise_3 ziehen Sie jetzt gleichen Programmcode vor die Klammer, indem Sie die Klasse `ReiseBefehl` einführen, die zwischen den Anbietern und dem Reiseveranstalter liegt. Diese Klasse definiert einzig und allein den Befehl, der ausgeführt wird, um eine Reise zu buchen. Beim Erzeugen wird dem Konstruktor eine Instanz eines Reiseveranstalters übergeben, deren Referenz in einem Datenfeld gespeichert wird. Die Methode `buchen()` wird mit den gleichen Argumenten versorgt werden, die bereits im vorigen Projekt an die Methode `reiseBuchen()` übergeben wurden.

```

public class ReiseBefehl {
    private final Reiseveranstalter veranstalter;

    public ReiseBefehl(Reiseveranstalter veranstalter) {
        this.VERANSTALTER = veranstalter;
    }
}
  
```

```

public void buchen(String ziel, LocalDate von, LocalDate bis) {
    Reise reise = new Reise(ziel, von, bis);
    veranstalter.durchfuehren(reise);
}
}
}

```

Die Anbieter – ich zeige hier nur das Reisebüro – sind jetzt ziemlich schlank geworden. An deren Konstruktor wird eine Instanz der Klasse `ReiseBefehl` übergeben, deren Referenz in einem Datenfeld gespeichert wird. Die Methode `reiseBuchen()` greift nun nicht mehr selbst auf einen Reiseveranstalter zu, sondern auf die Instanz der Klasse `ReiseBefehl`.

```

public class Reisebuero implements Anbieter {
    private final ReiseBefehl reiseBefehl;

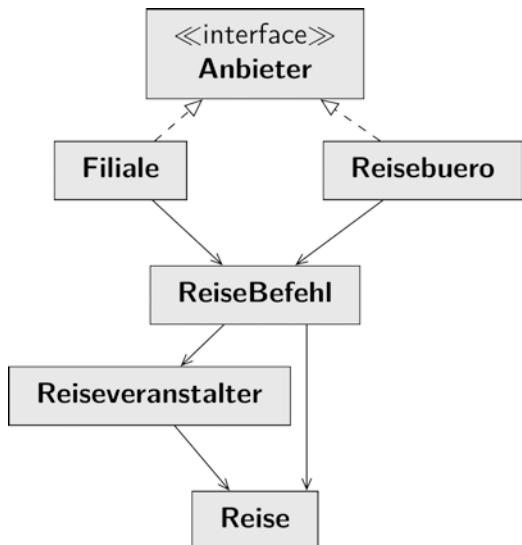
    Reisebuero(ReiseBefehl reiseBefehl) {
        this.reiseBefehl = reiseBefehl;
    }

    @Override
    public void reiseBuchen(String ziel, LocalDate von, LocalDate bis) {
        reiseBefehl.buchen(ziel, von, bis);
    }
}

```

Die Änderungen, die in dieser Programmversion hinzugekommen sind, sehen Sie in Abb. 9.2.

Abb. 9.2 Klassendiagramm des Beispielprojekts Reise_3



Doch welche Änderungen ergeben sich bei der Anwendung der Klassen? In der main-Methode der Testklasse, die Sie im Beispielprojekt Reise_3 finden, ist folgendes Vorgehen vorgesehen: Zuerst werden zwei Objekte vom Typ Reiseveranstalter erzeugt. Diese Objekte werden an die Konstruktoren zweier Instanzen der Klasse ReiseBefehl übergeben. Die ReiseBefehl-Instanzen werden schließlich an die Konstruktoren der Anbieter als Argumente übergeben. Für den Anbieter ist es unerheblich, welcher Veranstalter im Hintergrund arbeitet. Er ruft immer nur die Methode buchen() seiner ReiseBefehl-Instanz auf.

Was ist durch dieses Vorgehen gewonnen? Zunächst sieht es so aus, als wäre diese Lösung ziemlich umständlich – immerhin wird eine zusätzliche Klasse benötigt. Tatsächlich aber passieren verschiedene Dinge: Zunächst haben Sie doppelten Code soweit wie möglich eliminiert. Damit können Sie problemlos weitere Reiseanbieter hinzufügen. Sie können auch ein- und denselben Reisebefehl an verschiedene Reiseanbieter übergeben. Außerdem sind die Anbieter jetzt von den Reiseveranstaltern genauso entkoppelt wie Sie vom Chef Ihres Unternehmens mit seinen Statistiken.

- ▶ Das Auslagern eines Befehls in eine eigene Klasse ist der Kern des Command Patterns.

Das Command Pattern kennt verschiedene Begriffe, die ich zum Abschluss der Einführung formaler darstellen möchte. Es gibt den *Aufrufer* oder *Invoker*; das ist der Chef des Unternehmens, der eine Statistik anfordert, das ist aber auch ein Anbieter, der eine Reise verkauft. Der Aufrufer bedient sich eines *Befehls* oder auch eines *Command*; das ist zum Beispiel eine Instanz der Klasse ReiseBefehl; ein Command-Objekt ist aber auch Ihr Abteilungsleiter, der die Wünsche des Chefs an Sie weitergibt. Und schließlich gibt es diejenigen, die den Befehl ausführen, das sind die *Empfänger* oder *Receiver* – zum Beispiel die Unternehmen, die die Reisen tatsächlich durchführen.

9.2 Command in der Klassenbibliothek

In diesem Kapitel möchte ich Ihnen zwei Beispiele nennen, die die Verwendung des Command Patterns in der Java Klassenbibliothek zeigen

9.2.1 Beispiel 1: Nebenläufigkeit

Wenn Sie in einem Programm Nebenläufigkeit umsetzen wollen, legen Sie ein Objekt vom Typ Runnable an. Das Interface Runnable verlangt, dass Sie die Methode run() überschreiben. In dieser Methode wird sich der Code finden, der nebenläufig ausgeführt werden soll. Sie können die Ausführung des Befehls auf zwei Arten realisieren: Entweder führt das Runnable-Objekt den Code selbst aus oder es delegiert die Ausführung an ein

anderes Objekt. In unserem Beispiel hat das Reisebüro-Objekt den Befehl, eine Reise durchzuführen, an den Reiseveranstalter delegiert. Es ist jedoch nicht zwingend, den Befehl vom Empfänger ausführen zu lassen. Denkbar wäre auch, die gesamte Geschäftslogik oder zumindest einen Großteil davon vom Command ausführen zu lassen.

Das sähe beispielsweise dann so aus:

```
Runnable runnableDemo = new Runnable() {
    @Override
    public void run() {
        // ... nebenläufiger Code
    }
};
```

Sie erzeugen dann eine Instanz der Klasse Thread und übergeben eine Instanz dieser Klasse an den Konstruktor:

```
Thread threadDemo = new Thread(runnableDemo);
```

Und schließlich rufen Sie auf der Thread-Instanz die Methode start() auf, was dazu führt, dass der Code einer Klasse vom Typ Runnable ausgeführt wird. Die Thread-Klasse ist als Invoker mit dem Receiver genauso lose gekoppelt wie die Reiseanbieter mit den Reiseveranstaltern.

```
threadDemo.start();
```

- ▶ Es ist zwar offensichtlich, aber beachten Sie bitte, dass das Runnable kein Verhalten der Klasse Thread definiert. Das Interface Runnable und die vorgeschriebene Methode run() sind lediglich dafür geschaffen, dass ein Objekt einen Befehl eines fremden Objekts ausführen kann, von dessen Definition es keine Kenntnis hat. Wenn ich das Prinzip salopp formulieren darf: Die Steckdose liefert an einer definierten Schnittstelle Strom – ob damit eine Lampe, ein Computer oder eine Waschmaschine betrieben wird, ist ihr ziemlich schnuppe. Keines der genannten Geräte definiert irgendein Verhalten der Steckdose.

9.2.2 Beispiel 2: Event-Handling

Sie programmieren eine GUI. Auf der GUI soll es einen Button geben, der vom Anwender aktiviert werden kann. Sie übergeben dem Button einen ActionListener, der den Code enthält, der ausgeführt werden soll, wenn der Button aktiviert wird. Ein Objekt vom Typ ActionListener muss die Methode actionPerformed() überschreiben.

Wenn der Button aktiviert wird, wird diese Methode aufgerufen. Um das Vorgehen zu demonstrieren, legen Sie zunächst den Button an:

```
JButton btnBeispiel = new JButton("Non-Sense");
```

Dann folgt eine anonyme Klasse für den Code, der bei der Aktion ausgeführt werden soll:

```
ActionListener actionListener = new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        // ... irgendwas  
    }  
};
```

Und schließlich übergeben Sie das Command-Objekt, den ActionListener, an diesen Button:

```
btnBeispiel.addActionListener(actionListener);
```

Das Ganze funktioniert mit den in Java 8 hinzugekommenen Lambda-Ausdrücken auch sehr viel kürzer und übersichtlicher, bewirkt aber das Gleiche:

```
JButton btnBeispiel = new JButton("Non-Sense");  
btnBeispiel.addActionListener((ActionEvent e) -> {  
    // ... irgendwas  
})
```

Ist Ihnen aufgefallen, dass dieser Code strukturell dem der Nebenläufigkeit ziemlich ähnlich ist? Die Logik ist in der Tat die gleiche: Sie benötigen ein Objekt von einem bestimmten Typ, an dessen vorher definierte Schnittstelle (im einen Fall `run()`, im anderen Fall `actionPerformed()`) ein anderes Objekt Nachrichten schicken kann. Wenn der Invoker die definierte Methode des Command-Objekts aufgerufen hat, ist für ihn die Arbeit getan. Der Programmierer hat festgelegt, was passieren soll, wenn der Invoker aktiv geworden ist. Der Invoker hat aber keine Kenntnis davon, wie es passiert – und genau diese Kenntnis muss und darf er auch gar nicht haben.

9.3 Befehlsobjekte wiederverwenden

Die Klassen `JButton` und `JMenuItem` haben die gleiche Oberklasse `AbstractButton`. Sie können den `ActionListener` aus dem vorigen Abschnitt also wiederverwenden und gleichzeitig an das `MenuItem` im Hauptmenü Ihrer Oberfläche einfügen:

```
JMenuItem mtmBeispiel = new JMenuItem("Non-Sense");  
mtmBeispiel.addActionListener(actionListener);
```

Im nächsten Absatz gehen Sie noch einen Schritt weiter.

9.3.1 Das Interface „Action“

Das Interface Action erweitert das Interface ActionListener. Sowohl Action als auch ActionListener können Sie als Command an einen Aufrufer übergeben. Die Bindung zwischen Aufrufer und Command-Objekt ist bei einem Objekt vom Typ Action sehr viel enger. Das Interface Action sieht unter anderem die Methode setEnabled() vor, die festlegt, ob das Action-Objekt aktiviert oder deaktiviert ist. Außerdem können Sie mit einem Action-Objekt Text und Icon des Aufrufers bestimmen.

- ▶ Erinnern Sie sich an das, was beim Template Method Pattern gesagt wurde? In diesem Kontext habe ich Ihnen die Klasse AbstractListModel als schablonenhafte Implementierung des ListModels vorgestellt. Die Klasse AbstractListModel überschreibt alle Methoden, für die ein Standardverhalten sinnvoll realisiert werden kann. Die Methoden, die kontextabhängig sind, um beispielsweise die Datenbasis zu beschreiben, werden an Subklassen delegiert. Genau das Gleiche findet auch hier statt. Das Interface Action gibt den Algorithmus vor und die Klasse AbstractAction implementiert diesen teilweise. Die Logik, wie der Status zu ändern ist, ist bereits implementiert; wenn Sie den enabled-Status des Action-Objekts

Abb. 9.3 Gleiche Beschriftungen und gleiche Aktionen



ändern, wird der Status des Invoker ebenfalls geändert. Dieses Verhalten dürfte in den meisten Fällen sinnvoll sein; Sie können es jedoch überschreiben. Einem Standardverhalten entzieht sich allerdings die Methode `actionPerformed()`, die Sie zwingend überschreiben müssen.

9.3.2 Verwendung des Interface „Action“

Ich möchte Ihnen mit dem folgenden Beispiel die Wiederverwendbarkeit von Commands demonstrieren. Schauen Sie in das Beispielprojekt Action. Darin werden zwei Buttons und zwei Menü-Items angelegt. Jeweils einem Button und einem Menü-Item – zwei unterschiedlichen Aufrufen also – wird das gleiche Action-Objekt – ein Command-Objekt – übergeben.

Zunächst legen Sie einen Button und ein Menü-Item an:

```
JButton btnAusblenden = new JButton();
JMenuItem mnAusblenden = new JMenuItem();
```

Diese werden einer GUI hinzugefügt. Außerdem wird ein Command-Objekt, ein Objekt vom Typ `Action`, angelegt. Der Konstruktor der Klasse nimmt einen String entgegen, der den Anzeigetext des Aufrufers enthält. Die einzige Methode, die überschrieben werden muss, ist die Methode `actionPerformed()`. Wenn sie ausgelöst wird, deaktiviert sie das Objekt. Die Standardimplementierung der Klasse `AbstractAction` sieht vor, dass die Invoker ebenfalls deaktiviert werden.

```
AbstractAction actAusblenden = new AbstractAction("Deaktivieren") {
    @Override
    public void actionPerformed(ActionEvent evt) {
        this.setEnabled(false);
    }
};
```

Sowohl Button als auch Menü-Item werden mit demselben Command-Objekt versorgt.

```
btnAusblenden.setAction(actAusblenden);
mnAusblenden.setAction(actAusblenden);
```

Der Text, den Sie dem Konstruktor übergeben haben, wird von beiden Komponenten als Anzeigetext verwendet. Wenn Sie entweder den Button oder das Menü-Item anklicken, werden sowohl Button als auch Menü-Item deaktiviert. Abb. 9.3 zeigt Ihnen, wie die GUI aussieht.

9.4 Undo und redo von Befehlen

In diesem Abschnitt beschreibe ich einen letzten Aspekt des Command Patterns. Befehle können rückgängig gemacht und wiederhergestellt werden. Sie finden dazu zwei Beispiele. Das erste Beispiel ist ziemlich einfach gehalten. Das zweite ist ein wenig umfangreicher; ich werde den Quelltext nur in groben Zügen vorstellen – ich möchte Ihnen mit diesem Beispiel etwas zum Tüfteln für lange Winterabende anbieten.

9.4.1 Ein einfaches Beispiel

Das Beispielprojekt RadioCommand zeigt, wie ein Radio (die Älteren unter uns werden sich erinnern) mit dem Command Pattern bedient werden kann. Es gibt neben der Frequenz-Verstellung, die ich hier mal außen vor lasse, vier sehr simple Befehle: anschalten, ausschalten, lauter stellen und leiser stellen. Alle Befehle implementieren das Interface Command, das zwei Methoden vorschreibt. Die Methode `execute()` führt den Befehl aus, die Methode `undo()` gibt einen Befehl zurück, der ausgeführt werden muss, um die eigene `execute`-Methode rückgängig zu machen.

```
public interface Command {
    void execute();
    Command undo();
}
```

Den Befehl zum Anschalten des Radios erläutere ich hier stellvertretend für alle anderen Befehle. Dem Konstruktor des Command wird ein Objekt vom Typ Radio übergeben. Auf diesem Objekt wird der Befehl ausgeführt. Wenn also die Methode `execute()` aufgerufen wird, schaltet der Befehl das Radio an. Wird die Methode `undo()` aufgerufen, wird das Radio ausgeschaltet, der Anschalten-Befehl gibt also einen Ausschalten-Befehl zurück.

```
public class AnschaltenCommand implements Command {
    private final Radio radio;

    public AnschaltenCommand(Radio radio) {
        this.radio = radio;
    }

    @Override
    public void execute() {
        System.out.println("Das Radio wird angeschaltet.");
        radio.anSchalten();
    }
}
```

```
    @Override
    public Command undo() {
        return new AusschaltenCommand(radio);
    }
}
```

Das Radio muss nun die entsprechenden Methoden zur Verfügung stellen, damit die Commands ausgeführt werden können.

```
public class Radio {
    private int lautstaerke = 0;

    public void anschalten() {
        lautstaerke = 1;
        System.out.println(">Radio: Ich bin an.");
    }

    public void ausschalten() {
        lautstaerke = 0;
        System.out.println(">Radio: Ich bin jetzt aus.");
    }

    public void leiserstellen() {
        if (lautstaerke >= 1)
        {
            lautstaerke--;
            System.out.
                println(">Radio: Ich spiele jetzt leiser: " + lautstaerke);
        }
    }

    public void lauterstellen() {
        lautstaerke++;
        System.out.
            println(">Radio: Ich spiele jetzt lauter: " + lautstaerke);
    }
}
```

Das Radio ist in der Terminologie des Command Patterns der Receiver, der den Befehl ausführt. Invoker ist eine Klasse Logbuch, an die der Kontext den Befehlsaufruf und das Undo schickt. Der Invoker protokolliert alle Befehlsaufrufe. Dies ermöglicht dem Kontext, den jeweils letzten Befehl rückgängig zu machen.

```
public class Logbuch {  
    private final List<Command> history = new ArrayList<>();  
  
    public void execute(Command command) {  
        history.add(command);  
        command.execute();  
    }  
  
    public void undo() {  
        int groesse = history.size();  
        if (groesse > 0) {  
            Command command = history.remove(groesse - 1);  
            Command undoCommand = command.undo();  
            System.out.println("\tundo: " + undoCommand);  
            undoCommand.execute();  
        }  
    }  
}
```

Die Testklasse erzeugt ein Objekt vom Typ Radio. Außerdem werden die Commands erzeugt und mit dem Radio parametrisiert. An das Logbuch werden die Befehle dann übergeben und zur Ausführung gebracht.

```
Radio radio = new Radio();  
  
Command einschaltenCommand = new AnschaltenCommand(radio);  
Command ausschaltenCommand = new AusschaltenCommand(radio);  
Command leiserCommand = new LeiserCommand(radio);  
Command lauterCommand = new LauterCommand(radio);  
  
Logbuch logbuch = new Logbuch();  
  
logbuch.execute(einschaltenCommand);  
// ... gekürzt  
logbuch.undo();
```

Lassen Sie uns im nächsten Abschnitt ein größeres Swing-Beispiel angehen!

9.4.2 Ein umfangreicheres Beispiel

Das Beispielprojekt Swing demonstriert die undo und die redo-Funktionalität. Auf der GUI des Programms finden Sie vier Buttons. Ein Button trägt die Beschriftung Red und einer die Beschriftung Blue. Wenn Sie einen dieser Buttons anklicken, werden rote bzw. blaue Linien auf die Leinwand gemalt. Der zuletzt ausgeführte Befehl wird in die Liste auf der linken Seite unten angehängt. Wenn Sie den Button undo klicken, wird der zuletzt ausgeführte Befehl rückgängig gemacht – gelöscht also. Er wird jedoch nicht wirklich gelöscht, sondern in die Liste der wiederherzustellenden Befehle rechts geschrieben. Wenn Sie redo klicken, wird der Befehl wiederhergestellt, was einem erneuten Aufruf entspricht. Wenn Sie einen Befehl der linken Liste markieren und auf undo klicken, wird der gewählte Befehl gelöscht. Genauso, wenn Sie einen Befehl der rechten Liste markieren und auf redo klicken, wird genau dieser Befehl wiederhergestellt. In der Abb. 9.4 sehen Sie, wie die GUI aussehen wird. Ich habe ein paar Linien gezeichnet und wieder gelöscht.

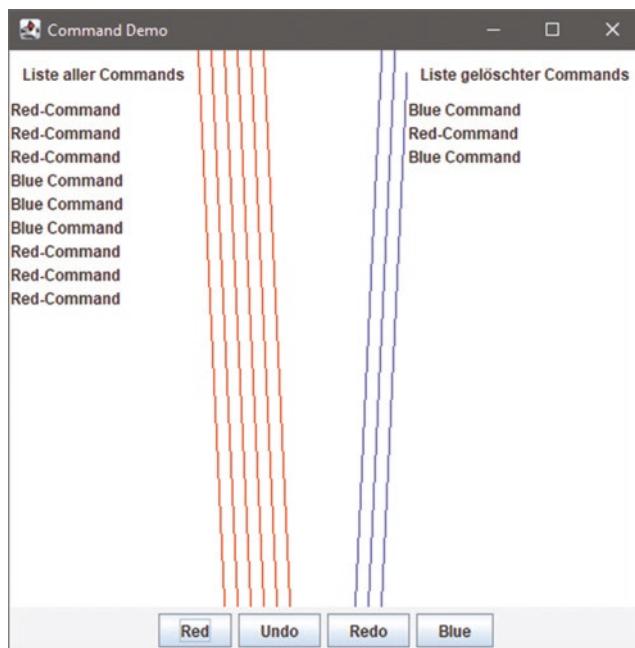


Abb. 9.4 GUI des Projekts „Swing“

Die Idee für dieses Projekt geht auf James W. Cooper (Cooper, James William (2000): Java design patterns. A tutorial. Reading, Mass.: Addison-Wesley. ISBN 0201485397.) zurück.

9.4.3 Besprechung des Quelltextes

Den Quelltext möchte ich Ihnen im Wesentlichen für die eigene Recherche überlassen. Ich stelle Ihnen daher nur die wesentlichen Eckpunkte vor. Das Klassendiagramm des Projekts finden Sie dann in Abb. 9.5.

9.4.3.1 Die beteiligten Klassen

Es gibt zwei Commands, das `CommandBlue` und das `CommandRed`. Die Commands haben teilweise gleichen Code, der in eine abstrakte Oberklasse ausgelagert wird. Und

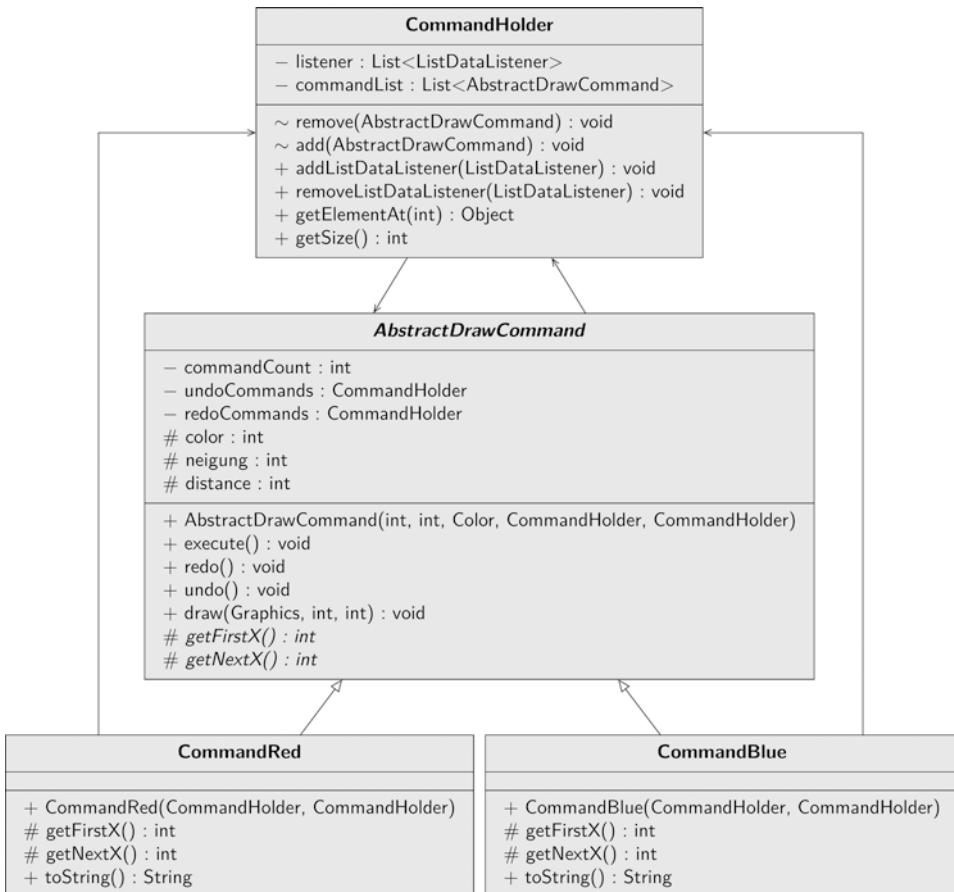


Abb. 9.5 UML-Diagramm des Command Pattern (Beispielprojekt Swing)

schließlich finden Sie die Klasse `CommandHolder`, in der die Commands in der Reihenfolge ihrer Abarbeitung gespeichert werden. Invoker ist die GUI.

9.4.3.2 Aufgabe der GUI

Die Klasse `GUI` erstellt die Komponenten: ein `JPanel` als Leinwand, auf der die Linien gemalt werden, zwei `JList`-Instanzen, die die ausgeführten und die gelöschten Commands anzeigen. Und schließlich finden Sie auf der GUI vier Buttons, Instanzen der Klasse `JButton`. Der `CommandHolder` implementiert das Interface `ListModel`, kann also als Model für die Listen verwendet werden. Die Klasse `GUI` erzeugt zwei `CommandHolder`-Instanzen, eine für die Undo-Liste und eine für die Redo-Liste, und übergibt beide an die Konstruktoren der Befehle `CommandRed` und `CommandBlue`.

```
private final CommandHolder undoCommands = new CommandHolder();
private final CommandHolder redoCommands = new CommandHolder();
private AbstractDrawCommand cmdRed =
    new CommandRed(undoCommands, redoCommands);
private AbstractDrawCommand cmdBlue =
    new CommandBlue(undoCommands, redoCommands);
```

Wenn der Button `btnRed` aktiviert wird, wird die Methode `execute()` der Instanz der Klasse `CommandRed` aufgerufen. Dieser zeichnet eine rote Linie auf die Leinwand – das Objekt `canvas` – und veranlasst sie, sich neu zu zeichnen.

```
btnRed.addActionListener((ActionEvent evt) -> {
    cmdRed.execute();
    canvas.repaint();
});
```

Der Listener für den Button `btnBlue` sieht entsprechend aus. Bevor ich auf die Listener für undo und redo eingehe, möchte ich die Command-Klassen näher betrachten.

9.4.3.3 Arbeitsweise der Command-Klassen

Die Klassen `CommandRed` und `CommandBlue` definieren zwei unterschiedliche Befehle: Der eine Befehl zeichnet eine rote Linie, die nach in die eine Richtung geneigt ist, der andere Befehl zeichnet eine blaue Linie, die in die andere Richtung geneigt ist. Dabei muss jedes Command zwei Aufgaben erledigen: Er muss zunächst die Parameter für die Linien festlegen und außerdem muss er in der Lage sein, seine Linien so oft auf die Leinwand zu zeichnen, wie er aufgerufen wurde. Die Parameter der Linie, also Farbe, Abstand und Neigung, werden in den Subklassen gespeichert. Die Oberklasse hält in einem nichtstatischen Datenfeld die Information, wie oft der Command aufgerufen wurde.

Wenn nun die `execute`-Methode ausgeführt wird, wird zunächst der Zähler inkrementiert. Dann wird eine Referenz auf das Command an den `CommandHolder` übergeben, der alle Commands in der Reihenfolge ihres Aufrufs speichert. Wenn die Linien gezeich-

net werden sollen, übergibt die Leinwand eine Referenz auf ihre Graphics-Instanz, den Stift, an das Command und veranlasst dieses, so viele Linien zu zeichnen, wie es aufgerufen wurde.

9.4.3.4 Undo und redo

Rechts und links neben der Leinwand sehen Sie zwei Listen, in der die Commands nacheinander gespeichert werden. Sie können ein beliebiges Command-Objekt selektieren und den undo-Befehl darauf aufrufen. Die Logik des undo ist im Befehl selbst definiert. Der Command dekrementiert seinen Zähler, löscht sich aus der undo-Liste und schreibt sich in die redo-Liste – zugegeben, im Gegensatz zu den Aufgaben realer Projekte ist diese Logik sehr einfach. Wenn Sie ein redo veranlassen, wird der Befehl aus der redo-Liste gelöscht und anschließend erneut ausgeführt, sprich die execute-Methode aufgerufen.

9.4.4 Undo und redo grundsätzlich betrachtet

In beiden Beispielen haben Sie eine vollkommene undo- und redo-Funktionalität. Das undo macht die Ausführung rückgängig und das redo entspricht exakt dem ursprünglichen execute-Befehl. In der Realität werden Sie auch auf Situationen treffen, in denen eine exakte Umkehrung gar nicht möglich ist. Wenn Sie beispielsweise in einem Supermarkt Schokolade gekauft haben und dann essen, können Sie diese sicher nicht mehr zurückgeben. Denkbar sind aber auch Situationen, in denen ein Befehl nur teilweise rückgängig gemacht werden kann. Vielleicht haben Sie ein Buch gekauft. Der Händler nimmt das Buch zurück, erstattet Ihnen aber nicht das Geld, sondern gibt Ihnen einen Gutschein dafür. Damit ist der Zustand, wie er vorher bestand, nur teilweise wiederhergestellt. Sie sind zwar das Buch los, haben aber Ihr Geld nicht zurück.

9.5 Command – Das UML-Diagramm

Das UML-Diagramm für das Command Pattern stammt aus dem Beispielprojekt Swing. Sie finden es in Abb. 9.5.

9.6 Zusammenfassung

Bevor ich die Kernpunkte des Patterns zusammenfasse, möchte ich Sie auf eines aufmerksam machen: die Ähnlichkeit zum Chain of Responsibility Pattern. Beide Patterns entkoppeln Befehlsaufruf und Befehlsausführung. Bei der Chain of Responsibility schickt der Aufrufer seinen Befehl an eine Kette von möglichen Empfängern; er kann jedoch nicht wissen, ob und wie der Befehl verarbeitet wird. Beim Command Pattern gibt es einen klar

definierten Befehlsausführer, den Receiver. Er ist mit dem Aufrufer, dem Invoker, lose gekoppelt.

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Befehle werden in eigene Klassen ausgelagert: Eselsbrücke: den Befehl „vor die Klammer ziehen“.
- Das Command Pattern entkoppelt Aufrufer (Invoker) und Empfänger (Receiver).
- Eine Befehlsklasse kann den Aufruf selbst ausführen oder an eine ausführende Einheit delegieren.
- Jede Befehlsklasse kann mit einer anderen ausführenden Einheit parametrisiert werden: zum Beispiel verschiedene Reiseveranstalter.
- Der Invoker sendet ausschließlich an das Command-Objekt Nachrichten; er muss den Receiver nicht kennen.
- Invoker und Command-Objekt tauschen über definierte Schnittstellen Ihre Nachrichten aus.
- Ein Command-Objekt kann zur Laufzeit ersetzt werden.
- Ein Befehl kann rückgängig gemacht werden.
- Ein rückgängig gemachter Befehl kann wiederhergestellt werden.
- Das Command Pattern ermöglicht das Führen einer Historie.

Zweckbeschreibung

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

„Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Objekte in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.“



Das Strategy Pattern ist ein Verhaltensmuster. Sie werden immer dann darauf zurückgreifen, wenn Sie eine Aufgabe mit unterschiedlichen Strategien lösen können; der Begriff Strategie ist in diesem Kontext ein Synonym für „Algorithmus“ oder für „Verhalten“. Ein Beispiel: Sie haben ein Urlaubsfoto und möchten es entweder im jpg-Format oder im bmp-Format speichern. Sie möchten zur Laufzeit entscheiden, in welchem Format Sie das Bild speichern. Das Strategy Pattern löst für Sie die Aufgabe, dass Sie ein- und dieselbe Aufgabe mit unterschiedlichen Algorithmen implementieren können – eben ein Bild in unterschiedlichen Formaten zu speichern. Sie können problemlos neue Algorithmen – Strategien – hinzufügen.

10.1 Ein erster Ansatz

Um in die Materie hineinzufinden, werden Sie ein Array sortieren. Es gibt sehr viele Sortieralgorithmen, die in unterschiedlichen Bereichen ihre Vorteile haben. In diesem Kapitel werde ich drei Algorithmen vorstellen: den SelectionSort, den MergeSort und den QuickSort. Lassen Sie uns mit einem sehr naiven Ansatz starten. Sie haben eine Klasse, in der sämtliche Algorithmen in verschiedenen Methoden definiert sind. Sie geben zur Laufzeit an, welchen Algorithmus Sie aufrufen möchten. Abhängig von der Eingabe wird die relevante Methode aufgerufen. Den Quellcode finden Sie im Beispielprojekt NaiverAnsatz. Die Sortieralgorithmen selbst habe ich nicht implementiert; es kommt hier erst einmal auf die Struktur der Anwendung an, nicht auf die Implementation von Sortieralgorithmen.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_10

```

public class SchlechterAnsatz {
    // ... gekürzt
    SchlechterAnsatz() {
        this.auswahlTreffen();
    }

    void auswahlTreffen() {
        var frage = "Wie sollen die Daten sortiert werden?";
        Object rueckgabe = JOptionPane.
            showInputDialog(null, frage, "selection sort");
        var antwort = (String) rueckgabe;

        switch (antwort) {
            case "selection sort" -> sortiereMitSelectionSort();
            case "merge sort" -> sortiereMitMergeSort();
            case "quick sort" -> sortiereMitQuickSort();
            default -> System.out.println("Unbekannte Auswahl");
        }
    }

    private void sortiereMitSelectionSort() {
        // ... der Selection Sort Algorithmus
    }

    private void sortiereMitMergeSort() {
        // ... der Merge Sort Algorithmus
    }

    private void sortiereMitQuickSort() {
        // ... der Quick Sort Algorithmus
    }
}

```

Beachten Sie, dass ich im switch Statement der Methode `auswahlTreffen()` die „Pfeil-Syntax“ aus den switch Expressions verwende, die wir in Abschn. 5.7 kennen gelernt haben. Damit entfällt jetzt auch im switch Statement das ansonsten erforderliche `break`. Der Code wird kürzer und einfacher lesbar. Sie können diesen Ansatz verändern, indem Sie eine if-Anweisung anstelle der switch-Anweisung verwenden. Dennoch kommen Sie nicht darum herum, Fallunterscheidungen zu treffen. Es ist also sehr unübersichtlich, neue Algorithmen zu implementieren. Außerdem haben Sie sämtliche Algorithmen in einer Klasse definiert. Die ohnehin schon große Klasse enthält daher viel Code, den Sie unter Umständen niemals verwenden werden. Sie erkennen, dass der Quelltext der Klasse sehr unflexibel ist; er ist der Albtraum eines jeden Wartungsprogrammierers. Lassen Sie uns also das Strategy Pattern betrachten, das diese Nachteile behebt.

10.2 Strategy in Aktion – Sortieralgorithmen

Das Prinzip des Strategy Patterns ist, dass Sie Algorithmen kapseln und sie austauschbar machen können. Wie könnte das geschehen? Definieren Sie drei Klassen, die die drei Sortieralgorithmen definieren. Wenn diese drei Klassen vom gleichen Datentyp sind, kann der Kontext diese beliebig austauschen. Abb. 10.1 zeigt das Klassendiagramm des Beispielprojekts Sortieren, das wir in diesem Abschnitt besprechen werden.

10.2.1 Das gemeinsame Interface

Ich habe Ihnen drei Klassen mit den genannten Sortieralgorithmen im Beispielprojekt Sortieren hinterlegt. Sie alle implementieren das Interface `SortStrategy`. Das Interface schreibt die Methode `sort()` vor. Dieser Methode wird das zu sortierende int-Array übergeben.

```
public interface SortStrategy {
    void sort(int[] zahlen);
}
```

Schauen wir uns jetzt in aller gebotener Kürze die Logik der Sortieralgorithmen an.

10.2.2 Der Selection Sort

Der Selection Sort durchsucht ein Array elementweise und sucht jeweils den kleinsten Wert. Dieser wird an den Anfang des noch unsortierten Rest-Arrays geschrieben und dann wird dieses sortiert.

Als Beispiel betrachten wir die untenstehende Zahlenfolge:



Im ersten Schritt wird die nullte Position, also der Wert 17, als kleinstes Element angenommen. Diese nullte Position wird mit jeder anderen folgenden Position verglichen. Da-

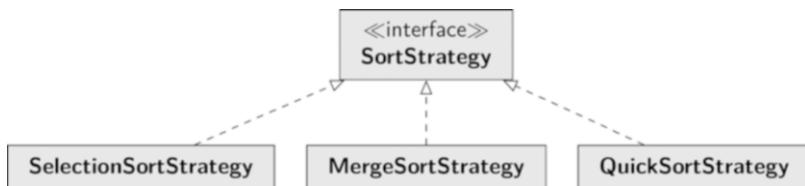


Abb. 10.1 Klassendiagramm des Beispielprojekts Sortieren

bei stellen Sie fest, dass es eine andere Position gibt, die den kleinsten Wert im Array enthält, nämlich die 3 an der vierten Position. Die beiden Positionen tauschen ihren Wert. Im zweiten Schritt haben Sie also folgendes Array:

3	45	21	99	17	20
---	----	----	----	----	----

Der kleinste aller Werte steht jetzt an der Position null. Sie fahren bei Position 1 fort. Der Wert 45 wird mit allen folgenden Werten verglichen, und dabei stellen Sie fest, dass die 17 an der vierten Position der kleinste Wert ist. Also tauschen Sie die Positionen 1 und 4. Nun sieht das Array wie folgt aus:

3	17	21	99	45	20
---	----	----	----	----	----

Jetzt vergleichen Sie die zweite Position, den Wert 21, mit allen folgenden Positionen und tauschen die 20 gegen die 21 und so weiter. Sie fahren fort, bis das Array insgesamt sortiert ist.

Der Code der Klasse `SelectionSortStrategy` kapselt diesen Algorithmus:

```
public class SelectionSortStrategy implements SortStrategy {
    @Override
    public void sort(int[] zahlen) {
        for (var i = 0; i < zahlen.length - 1; i++) {
            for (var j = i + 1; j < zahlen.length; j++) {
                if (zahlen[i] > zahlen[j]) {
                    var temp = zahlen[i];
                    zahlen[i] = zahlen[j];
                    zahlen[j] = temp;
                }
            }
        }
    }
}
```

Beachten Sie, dass diese Beispielimplementierung jede kleinere Zahl immer nach vorne holt, nicht nur die kleinste. Es werden also deutlich mehr Tauschoperationen fällig, als wirklich nötig. Da lässt sich sicher etwas optimieren. Man kann bei der Suche nach der kleinsten Zahl auch erst einmal das gesamte Feld durchgehen, bevor man dann wirklich tauscht. Damit ist eine Beschleunigung dieses Verfahrens um mehr als den Faktor 3 möglich. Ich habe Ihnen im Kontext schon mal (auskommentiert) die dafür notwendigen Anpassungen eingebaut. Sie werden sehen, dass es wirklich nur 3 Zeilen sind: Das erneute Klonen des Zahlenfeldes, die Zuweisung der Strategie und deren Ausführung. Ihnen bleibt jetzt noch die Implementierung der Klasse `Selection2SortStrategy` auf Basis ei-

ner Kopie der Klasse `SelectionSortStrategy`. Auch diese Kopie finden Sie bereits als fertige Klasse vor – aber noch unverändert gegenüber dem Original. Viel Spaß bei dieser Aufgabe.

10.2.3 Der Merge Sort

Der Merge Sort arbeitet nach dem Prinzip des „Teile und herrsche“. Die zu sortierende Menge wird in zwei Teile geteilt. Lassen Sie uns folgendes unsortiertes Array nehmen:

17	45	21	99	3	20	15	12
----	----	----	----	---	----	----	----

Wenn Sie dieses Array teilen, haben Sie zwei Teile, die Werte 17, 45, 21, 99 auf der linken und 2, 20, 15, 12 auf der rechten Seite. Wenn Sie diese zwei Hälften erneut teilen, haben Sie folgende vier Teile:

Liste 1	17	45
Liste 2	21	99
Liste 3	3	20
Liste 4	15	12

Es ist offensichtlich nicht sinnvoll, die einzelnen Listen weiter zu teilen, also sortieren Sie sie jetzt einzeln:

Liste 1	17	45
Liste 2	21	99
Liste 3	3	20
Liste 4	12	15

Jetzt werden die sortierten Teillisten paarweise zusammengefügt, und dabei darauf geachtet, dass die neuen Listen richtig sortiert sind:

Liste 1 und 2	<table border="1"><tr><td>17</td></tr></table>	17	<table border="1"><tr><td>21</td></tr></table>	21	<table border="1"><tr><td>45</td></tr></table>	45	<table border="1"><tr><td>99</td></tr></table>	99
17								
21								
45								
99								
Liste 3 und 4	<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>12</td></tr></table>	12	<table border="1"><tr><td>15</td></tr></table>	15	<table border="1"><tr><td>20</td></tr></table>	20
3								
12								
15								
20								

Und diese Listen werden jetzt wieder sortiert zusammengefügt zum Endergebnis

<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>12</td></tr></table>	12	<table border="1"><tr><td>15</td></tr></table>	15	<table border="1"><tr><td>17</td></tr></table>	17	<table border="1"><tr><td>20</td></tr></table>	20	<table border="1"><tr><td>21</td></tr></table>	21	<table border="1"><tr><td>45</td></tr></table>	45	<table border="1"><tr><td>99</td></tr></table>	99
3															
12															
15															
17															
20															
21															
45															
99															

Den Algorithmus dazu finden Sie in der Klasse MergeSortStrategy.

10.2.4 Der Quick Sort

Auch der Quick Sort arbeitet nach dem Prinzip „Teile und herrsche“. Neben wir das unsortierte Array von oben:

<table border="1"><tr><td>17</td></tr></table>	17	<table border="1"><tr><td>45</td></tr></table>	45	<table border="1"><tr><td>21</td></tr></table>	21	<table border="1"><tr><td>99</td></tr></table>	99	<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>20</td></tr></table>	20	<table border="1"><tr><td>15</td></tr></table>	15	<table border="1"><tr><td>12</td></tr></table>	12
17															
45															
21															
99															
3															
20															
15															
12															

Dieses Array wird wieder in zwei Teile geteilt. Sie berechnen einen Wert, ein Pivot-Element, so, dass etwa die Hälfte der Werte kleiner und die andere Hälfte größer als dieser Wert sind. Die kleineren Werte werden in die linke Liste geschrieben, die größeren in die rechte. Zur Demonstration nehme ich hier als Pivot-Wert den Wert 40. Sie haben also zwei Listen:

Linke Liste	<table border="1"><tr><td>17</td></tr></table>	17	<table border="1"><tr><td>21</td></tr></table>	21	<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>20</td></tr></table>	20	<table border="1"><tr><td>15</td></tr></table>	15	<table border="1"><tr><td>12</td></tr></table>	12
17												
21												
3												
20												
15												
12												
Rechte Liste	<table border="1"><tr><td>45</td></tr></table>	45	<table border="1"><tr><td>99</td></tr></table>	99								
45												
99												

„Rein zufällig“ enthält die rechte Liste nur zwei Werte – es ist also sinnlos, diese wieder zu teilen; sortieren Sie sie und Sie sind fertig damit. Betrachten Sie aber die linke Liste. Als Pivot-Wert könnten Sie 18 nehmen. Alle Werte, die kleiner als 18 sind, schreiben Sie links hin, die anderen rechts. Sie haben also:

Linke Liste	<table border="1"><tr><td>17</td></tr></table>	17	<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>15</td></tr></table>	15	<table border="1"><tr><td>12</td></tr></table>	12
17								
3								
15								
12								
Rechte Liste	<table border="1"><tr><td>21</td></tr></table>	21	<table border="1"><tr><td>20</td></tr></table>	20				
21								
20								
Fertig	<table border="1"><tr><td>45</td></tr></table>	45	<table border="1"><tr><td>99</td></tr></table>	99				
45								
99								

Die rechte Liste enthält „rein zufällig“ wieder nur zwei Elemente, die Sie sortieren. Sie dürfen sie anschließend mit den bereits fertig sortierten Werten zusammenfügen:

Linke Liste	17	3	15	12
Fertig	20	21	45	99

Die „linke Liste“ teilen Sie wieder nach einem vorgegebenen Pivot-Element und so weiter.

Sowohl Merge Sort als auch Quick Sort arbeiten rekursiv. Quick Sort ist dann schnell, wenn Sie es schaffen, in jedem Schritt einen Pivot-Wert so zu berechnen, dass etwa jeweils gleich viele Zahlen in der linken und der rechten Liste stehen. In meiner Implementierung der Klasse `QuickSortStrategy` wähle ich einfach das erste Element der Liste. Zu Sortieralgorithmen gibt es auch sehr viel weiterführende Literatur; soviel, dass ich hier keine Präferenz nennen möchte. Meine Beschreibungen hier sind nur sehr oberflächlich, aber auch für das Thema Design Patterns an sich nicht wirklich relevant.

10.2.5 Der Kontext

Alle Sortieralgorithmen sind vom Typ `SortStrategy`. Sie können sich also darauf verlassen, dass sie die Methode `sort()` definieren, die den Sortievorgang anstößt. Sie deklarieren innerhalb des Kontextes eine Variable, die eine Referenz auf die gewünschte Strategie hält. Eine Zuweisung legt die zu verwendende Strategie dann fest. Wenn Sie das Array sortieren möchten, rufen Sie einfach die `sort`-Methode der Strategie auf. Ein Ausschnitt aus der Testklasse `Kontext`:

```
public static void testeLaufzeit() {
    var feldgroesse = 100000;
    var wertebereich = 1000000;
    SortStrategy sortStrategy;

    var zahlen_1 = createArray(feldgroesse, wertebereich);
    var zahlen_2 = zahlen_1.clone();
    var zahlen_3 = zahlen_1.clone();

    System.out.
        println("Es wurden drei Arrays erzeugt, die die gleichen
unsortierten Zahlen enthalten");

    sortStrategy = new SelectionSortStrategy();
    executeStrategy(sortStrategy, zahlen_1);

    sortStrategy = new MergeSortStrategy();
    executeStrategy(sortStrategy, zahlen_2);

    sortStrategy = new QuickSortStrategy();
    executeStrategy(sortStrategy, zahlen_3);
}
```

```

private static void executeStrategy(SortStrategy s, int[] z) {
    Instant istart;
    Instant iende;
    Long idifferenz;

    System.out.println("Starte " + s);
    istart = Instant.now();

    s.sort(z);

    iende = Instant.now();
    Duration elapsed = Duration.between(istart, iende);
    idifferenz = elapsed.toMillis();
    System.out.println("Dauer " + s + ":" + idifferenz + " Millise-
kunden");
}

```

In Abschn. 9.1 haben wir einen ersten Blick in die Date and Time API geworfen und uns das LocalDate angeschaut. Sie erinnern sich an den Hinweis, dass es keinerlei Informationen zu Zeiten enthält? Im obigen Code sehen Sie das Gegenstück dazu: Die Klasse Instant. Diese Klasse enthält ausschließlich Zeitinformationen und entsprechend geeignete Methoden für Berechnungen. Eine einfache Anwendung sehen Sie hier: Ein Blick auf den Startzeitpunkt des Sortierens. Danach einen Blick auf die Uhr, wenn alles gelaufen ist, und anschließend das Berechnen der benötigten Dauer (als eigene Klasse Duration für die Länge von Zeitintervallen) und deren Ausgabe in Millisekunden.

Übrigens ist dieses Vorgehen des Benchmarkings sehr naiv und nur auf den ersten Blick aussagekräftig. Es reicht uns hier, um einen groben Vergleich der unterschiedlichen Algorithmen anzustellen. Aber die Startzeiten der Java Virtual Machine und das auch noch unter verschiedenen Umgebungsbedingungen werden hier nicht berücksichtigt. Wenn Sie ernsthafte Performance-Messungen von Java-Programmen im Blick haben, sollten Sie sich mit dem in Java 12 mit dem JEP 230 hinzugekommenen „Java Microbenchmarking Harness“ (JMH) befassen. Eine Beschreibung dieser kleinen Toolsammlung führt hier aber zu weit und am eigentlichen Thema des Buchs vorbei.

10.2.6 Bewertung des Ansatzes und mögliche Variationen

Was halten Sie von diesem Ansatz? Sicher springt Ihnen ein Vorteil gleich ins Auge: Der Kontext ist sehr viel übersichtlicher! Doch es gibt noch zwei weitere Vorteile:

Das Projekt kann um beliebig viele Sorteralgorithmen erweitert werden. Wenn Sie den Heap Sort implementieren möchten, definieren Sie eine neue Klasse, die das Interface SortStrategy implementiert. Der Kontext kann Ihre HeapSortStrategy sofort einsetzen. Beim Selection Sort habe ich eine Verbesserungsmöglichkeit vorgeschlagen. Auch die können Sie als Variante einer SortStrategy implementieren und gegen den von mir erstellten Selection Sort direkt vergleichen.

Prinzipiell muss der Kontext gar nicht wissen, wie sein Problem gelöst wird. Aber warum nur „prinzipiell“? Sie bieten eine Vielzahl von Algorithmen an, die alle die gleiche Aufgabe lösen. Damit ein Programmierer weiß, wann er welchen Algorithmus wählen soll, müssen Sie sehr genau dokumentieren, unter welchen Bedingungen welcher Algorithmus angemessen ist. Dabei werden Sie nicht vermeiden können, in der Dokumentation auch auf Implementierungsdetails einzugehen.

In der Implementierung oben haben Sie als Anwender festgelegt, welche Strategie Sie einsetzen möchten. Denkbar wäre aber auch, dass das Programm bei den Strategien nachfragt, wie gut sie eine bestimmte Aufgabe unter bestimmten Bedingungen lösen können. Ein Algorithmus, der die Daten im Hauptspeicher sehr effizient sortiert, kann versagen, wenn die Daten in einer Datei gespeichert sind und nicht vollständig in den Hauptspeicher geladen werden können. Der Kontext könnte also bei den Strategieklassen nachfragen: „Wie gut löst du die Aufgabe unter der Bedingung, dass die Daten auf der Festplatte gespeichert sind?“ Jede Strategiekasse gäbe eine Bewertungszahl zurück, beispielsweise im Bereich 0 bis 100. Der Kontext kann dann die Strategie auswählen, die die höchste Bewertungszahl zurückgibt.

Ein weiterer Vorteil des Strategy Patterns liegt in der Wiederverwendbarkeit der Algorithmen. Stellen Sie sich vor, Sie programmieren ein Office-Paket. Innerhalb der Tabellenkalkulation müssen Sie Daten sortieren können. Aber auch in der Textverarbeitung möchten Sie Daten sortieren. Die Algorithmen sind ja die gleichen, also können Sie sie wiederverwenden und sich bei der Tabellenkalkulation auf die Funktionalität beschränken, die typisch für die Tabellenkalkulation ist. Bei der Textverarbeitung programmieren Sie nur die Teile, die für eine Textverarbeitung typisch sind. Den Sortieralgorithmus können Sie wiederverwenden.

Im Beispiel oben habe ich vorausgesetzt, dass die zu sortierenden Daten an die Methode `sort()` übergeben werden. Diese Festlegung ist in dieser Konstellation sicher sinnvoll. Denken Sie aber an eine Familie von Algorithmen, bei denen eine Implementierung sehr viele Parameter benötigt, eine andere Implementierung aber deutlich weniger. Aufgrund der gemeinsamen Schnittstelle müsste der Kontext dennoch alle Argumente übergeben – ein vermeidbarer Aufwand.

10.3 Das Strategy Pattern in der Praxis

Wo lässt sich das Strategy Pattern in der Java-Klassenbibliothek nachweisen? Im Bereich der GUI-Programmierung finden Sie das Strategy Pattern an mehreren Stellen. Da gibt es beispielsweise `LayoutManager` oder auch das `Look and Feel`; Container werden standardmäßig mit einer bestimmten Strategy versehen, die aber vom Anwender ausgetauscht werden kann.

Instanzen der Klasse `JPanel` werden benötigt, um Komponenten einer GUI zusammenzufassen. Standardmäßig wird hierbei das `FlowLayout` verwendet. Sie können das nachvollziehen, indem Sie beispielsweise in der Java Shell einmal folgenden Code ausführen:

```
javax.swing.JPanel pnlLayoutTest = new javax.swing.JPanel();
java.awt.LayoutManager layout = pnlLayoutTest.getLayout();
System.out.println(layout);
```

Auf der Konsole wird dann `java.awt.FlowLayout [hgap=5, vgap=5, align=center]` ausgegeben. Sie können die Strategy `FlowLayout` ersetzen, indem Sie mit `pnlLayoutTest.setLayout(layoutManager)` eine neue Strategy übergeben. Die Variable `layoutManager` muss vom Typ `LayoutManager` sein.

Das Interface `LayoutManager` schreibt fünf Methoden vor. Die Methoden `addLayoutComponent()` und `removeLayoutComponent()` werden gebraucht, wenn Sie die Komponenten des `JPanel` mit einem String ansprechen möchten, sonst dürfen die Methodenrumpfe auch leer bleiben. Die Methode `preferredLayoutSize()` berechnet die optimale Größe des `JPanel` und gibt diese zurück, während `minimumLayoutSize()` die Mindestgröße des `JPanel` berechnet und zurückgibt. Die Positionierung der Komponenten wird durch die Methode `layoutContainer()` vorgenommen. In dieser Methode werden jeder Komponente mit `setBounds()` horizontale und vertikale Position sowie Breite und Höhe zugewiesen.

Im Quellcode zu diesem Buch finden Sie das Beispielprojekt `LayoutStrategy`, in dem ich den `LayoutManager` – die Strategie – `ZweiSpaltenLayout` implementiert habe. Wenn Sie auf dem `JPanel` Komponenten platzieren, werden diese in zwei Spalten angezeigt. Ziehen Sie die GUI in die Breite, werden die Komponenten der rechten Spalte entsprechend vergrößert. Abb. 10.2 zeigt Ihnen, wie eine GUI aussehen kann, die Sie mit dem `ZweiSpaltenLayout` erstellen.

Bitte beachten Sie, dass die Implementierung des `ZweiSpaltenLayout` ausgesprochen einfach gehalten ist – es ist sicher weit davon entfernt, als perfekt zu gelten. Ich möchte Ihnen mit diesem Beispiel lediglich demonstrieren, wie Sie eine eigene Strategie, einen eigenen `LayoutManager`, entwerfen und einsetzen können.

10.4 Strategy – Das UML-Diagramm

Aus dem Beispielprojekt Sortieren sehen Sie das UML-Diagramm in Abb. 10.3.

Abb. 10.2 Positionierung von Komponenten mit dem `ZweiSpaltenLayout`



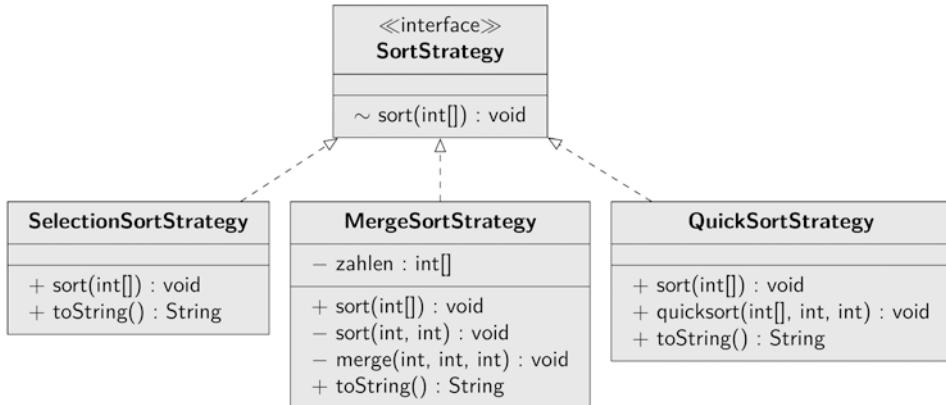


Abb. 10.3 UML-Diagramm des Strategy Pattern (Beispielprojekt Sortieren)

10.5 Abgrenzung zu anderen Mustern

Sie kennen das Command Pattern und das State Pattern und in diesem Kapitel haben Sie das Strategy Pattern kennengelernt. Bei allen drei Patterns wird Verhalten gekapselt. Die Klassendiagramme von State und Strategy sehen sogar ziemlich identisch aus. Sie behalten den Überblick, indem Sie sich die unterschiedlichen Ziele vergegenwärtigen:

Beim Command Pattern kapseln Sie Befehle. Sie benötigen es, um allen Buttons auf Ihrer GUI unterschiedliche ActionListener zu übergeben. Dabei gilt: Ein Befehl öffnet eine Datei, ein anderer speichert sie. Das Command Pattern beschreibt kein Verhalten des aufrufenden Objekts.

Das Strategy Pattern kapselt Algorithmen. Verhalten eines Objekts wird auf unterschiedliche Art umgesetzt: Es gibt viele Algorithmen, um Dateien zu komprimieren, zu verschlüsseln oder Daten zu sortieren. Sie werden aber immer nur einen einzigen Algorithmus brauchen. Also wählen Sie aus der Vielzahl von Strategien die eine aus, die Ihre Aufgabe am effizientesten löst.

Mit dem State Pattern kapseln Sie Zustandsausprägungen. Das Verhalten eines Objekts orientiert sich an seinem Zustand. Wenn ein Objekt einen bestimmten Zustand hat, zeigt es ein anderes Verhalten als in einem anderen Zustand. Denken Sie an das offene Tor – das Tor kann geschlossen, aber nicht abgeschlossen werden. Nur ein geschlossenes Tor kann auch abgeschlossen werden.

10.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Das Strategy Pattern wird eingesetzt, wenn Sie für eine Aufgabe mehrere Algorithmen haben.
- Jeder Algorithmus wird in einer eigenen Klasse definiert.
- Alle Strategy-Klassen implementieren das gleiche Interface.
- Der Kontext wird gegen die Schnittstelle programmiert.
- Zur Laufzeit wird ein Algorithmus an den Kontext übergeben.
- Der Kontext ruft die Lösungsstrategie, den Algorithmus auf, ohne zu wissen, welcher Algorithmus dahintersteht.
- Der Algorithmus kann wiederverwendet werden.

Zweckbeschreibung

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

„Definiere eine Familie von Algorithmen, kapsle jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.“



Java kennt verschiedene Sammlungen, beispielsweise die Klassen ArrayList und LinkedList. Die Daten werden in diesen Klassen intern unterschiedlich gespeichert. Für den Client ist es aber von Interesse, dass er ohne Kenntnis der internen Struktur über die Daten iterieren kann. In diesem Kapitel werden Sie sich ausführlich mit Sammlungen beschäftigen und die Klassen ArrayList und LinkedList nachbilden. Sie werden außerdem eine Schnittstelle erstellen, mit der Sie über diese Klassen iterieren können.

11.1 Zwei Möglichkeiten, Daten zu speichern

In den folgenden Abschnitten werde ich Ihnen zwei Möglichkeiten zeigen, wie Sie Sammlungen erzeugen können.

11.1.1 Daten in einem Array speichern

Die erste Sammlung, die Sie irgendwann einmal kennengelernt haben, war höchstwahrscheinlich das Array. In einem Array speichern Sie beliebig viele Elemente eines bestimmten Datentyps. Mit der Deklaration `int[] zahlen = new int[5]` legen Sie ein Array an, das fünf int-Zahlen speichern kann. Sie greifen sehr performant auf die einzelnen Speicherbereiche zu. Ein Array hat den Nachteil, dass es sich nicht vergrößern lässt.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_11

Das ist unpraktisch, wenn sich herausstellt, dass mehr Daten gespeichert werden sollen, als ursprünglich vorgesehen war.

Die Basis für unsere erste Sammlung soll ein Array sein. Die Elemente in diesem Array werden vom allgemeinen Typ Object sein. Initial sollen fünf Objekte referenziert werden können. Um verschiedene Datentypen typsicher speichern zu können, legen Sie die Klasse generisch an. Im Beispielprojekt Sammlungen_1 finden Sie folgenden Code:

```
public class MyArray<E> {
    private int zaehler = 0;
    private Object[] elemente = new Object[5];
}
```

Um ein neues Element einzufügen, definieren Sie die Methode `add()`. Ihr wird ein Argument vom generischen Typ übergeben. Dieses Element wird an der nächsten freien Stelle im Array gespeichert. Das Datenfeld `zaehler` speichert diese Position. Wenn bereits fünf Elemente gespeichert werden und ein sechstes hinzukommen soll, muss die Datenbasis erweitert werden. Da ein Array nicht vergrößert werden kann, bleibt nur der Weg, das Array neu zu definieren. Allgemein: Wenn die nächstfreie Position gleich der Anzahl Elemente ist, muss die Größe des Arrays um einen bestimmten Wert vergrößert werden. Im Beispiel soll die Arraygröße um weitere fünf Elemente vergrößert werden.

```
public void add(E e) {
    if (zaehler == elemente.length) {
        var tempArray = new Object[zaehler + 5];
        System.arraycopy(elemente, 0, tempArray, 0, zaehler);
        elemente = tempArray;
    }
    elemente[zaehler] = e;
    zaehler++;
}
```

Der Client möchte vielleicht erfragen, wie viele Elemente in der Sammlung gespeichert sind. Dafür reicht es, dass Sie die Position des nächstfreien Elements zurückgeben.

```
public int size() {
    return zaehler;
}
```

Die Sammlung erfüllt ihren Sinn erst dann, wenn die einzelnen Elemente zurückgegeben werden können. Hierzu legen Sie die Methode `get()` an, die als Parameter einen Index erwartet, der die Stelle des gesuchten Elements in der Datenbasis beschreibt. Vor der Rückgabe wird der gespeicherte Wert auf den generischen Typ gecastet.

```
public E get(int index) {  
    return (E) elemente[index];  
}
```

Wenn Sie ein Element löschen möchten, muss zunächst der Zähler dekrementiert werden. Das Element wird dann dadurch gelöscht, dass Sie die nachfolgenden Elemente jeweils um eine Stelle nach vorne verschieben. Um keine „index out of bound“-Exception auszulösen, sind aber noch Prüfungen auf den Bereich zwischen 0 und zaehler erforderlich. Und um das entfernte Objekt am Ende des Feldes nicht im Speicher zu hinterlassen, muss es mit null überschrieben werden.

```
public void remove(int index) {  
    if ((index <= zaehler) && (zaehler > 0) && (index >= 0)) {  
        if (index != zaehler)  
            System.arraycopy(elemente, index + 1, elemente, index,  
                             elemente.length - 1 - index);  
        elemente[zaehler--] = null;  
    }  
}
```

Die Sammlung, die Sie gerade entwickelt haben, entspricht in ihrer Methodik übrigens der ArrayList der Klassenbibliothek. Sie ist optimal, wenn Sie auf einzelne Elemente über deren Index zugreifen müssen.

11.1.2 Daten in einer Kette speichern

Einen ganz anderen Ansatz verfolgen Sie, wenn Sie die einzelnen Elemente nicht in einem Array speichern, sondern in einer Kette. Jedes Element kennt seinen Nachfolger. Denkbar wäre auch, dass ein Element auch einen Vorgänger kennt; auf diese Möglichkeit gehe ich nicht weiter ein – das Projekt würde nur unnötig umfangreich, ohne das dahinterstehende Prinzip zu verändern.

Strings und alle anderen Objekte, die Sie in Ihrer Liste speichern möchten, nenne ich Elemente. Sie werden nicht in der Sammlung gespeichert, sondern in der Instanz einer inneren Klasse, die ich Node nenne. Die Klasse Node hat zwei Datenfelder, die das zu speichernde Element und das nachfolgende Node-Objekt speichern. Die Sammlung kann sich dann darauf beschränken, das erste Objekt (im Datenfeld header) zu referenzieren.

```
public class MyList<E> {  
    private int zaehler = 0;  
    private Node<E> header = null;  
  
    private class Node<E> {
```

```

    private final E element;
    private Node<E> nextNode;

    Node(E element, Node<E> next) {
        this.element = element;
        this.nextNode = next;
    }
}
}
}

```

Daten werden in die Sammlung eingefügt, indem ein neues Node-Objekt erzeugt wird. Dieses Objekt wird von der Variablen header referenziert und verdrängt das vorher als header gespeicherte Objekt. Das Feld nextNode des neuen header-Objekts referenziert den früheren header. Und schließlich muss der Zähler inkrementiert werden. Wenn Sie die Größe der Sammlung abfragen, wird der Zähler zurückgegeben.

```

@SuppressWarnings("unchecked")
public void add(E element) {
    header = new Node(element, header);
    zaehler++;
}

public int size() {
    return zaehler;
}

```

Um ein Element aus der Sammlung zu löschen, gehen Sie die Sammlung mit einer while-Schleife durch und prüfen, ob das referenzierte Element gleich dem gesuchten Element ist. In diesem Fall übergeben Sie die Referenz des nachfolgenden Node-Objekts an den Vorgänger des Node-Objekts, das das gesuchte Element referenziert. Anschließend dekrementieren Sie den Zähler. Die lokale Variable previous referenziert jeweils den Vorgänger des Node-Objekts, dessen Element gerade geprüft wird.

```

public boolean remove(E element) {
    Node<E> previous = null;
    var tempNode = header;

    while (tempNode != null) {
        if (equals(element, tempNode.ELEMENT)) {
            if (previous == null)
                header = tempNode.nextNode;
            else
                previous.nextNode = tempNode.nextNode;
            zaehler--;
        }
    }
}

```

```
        return true;
    }
    previous = tempNode;
    tempNode = previous.nextNode;
}

return false;
}
```

Die Methode `get()` soll die gleiche Aufgabe lösen wie die `get`-Methode der Klasse `MyArray`. Da die Datenbasis jedoch nicht indexbasiert ist, können Sie das x-te Element in der Sammlung nicht direkt erfragen. Sie müssen die gesamte Sammlung durchgehen, bis Sie das x-te Element gefunden haben.

```
public E get(int index) {
    if (index < 0 || index >= zaehler)
        throw new NoSuchElementException(index + " Größe " + zaehler);
    var tempNode = header;
    for (var i = 0; i < index; i++)
        tempNode = tempNode.nextNode;
    return tempNode.ELEMENT;
}
```

Die Sammlung, die Sie gerade entwickelt haben, entspricht in ihrer Methodik übrigens der `LinkedList` der Klassenbibliothek.

11.2 Die Aufgabe eines Iterators

Wenn Sie eine Sammlung anlegen, möchten Sie sicher auch über alle Elemente iterieren. Ein erster Ansatz könnte das Vorgehen der Testmethoden (der jeweiligen `main`-Methode) sein, die Sie zu beiden Klassen im Beispielprojekt `Sammlungen_1` finden. Bitte analysieren Sie diese und lassen Sie sie laufen. In beiden Testmethoden iterieren Sie mit einer `for`-Schleife über die Datensammlung.

```
for (var i = 0; i < myList.size(); i++)
    System.out.println(myList.get(i));
```

Sie greifen mit der Methode `get()` auf jedes Element der Sammlung zu. Bei der Klasse `MyArray` ist das durchaus sinnvoll. Die Performance der Klasse `MyList` bleibt bei einem indexbasierten Zugriff auf ihre Elemente jedoch weit unter ihren Möglichkeiten. Es ist also sinnvoll, den Algorithmus, wie über die Sammlung zu iterieren ist, in eine eigene Klasse, den Iterator, auszulagern. Den Iterator können Sie sich wie ein Lesezeichen

vorstellen, das Seite für Seite durch ein Buch geschoben wird. Der Iterator kennt die spezifischen Merkmale einer Sammlung und nutzt diese optimal.

Sie können einen Iterator als internen Iterator oder als externen Iterator entwerfen. Intern bedeutet, dass Sie die Aktion des Iterierens an den Iterator übergeben, der „selbstständig“ über alle Objekte iteriert. Wenn Sie einen externen Iterator programmieren, lassen Sie sich das jeweils nächste Element zurückgeben und fragen ab, ob noch weitere Elemente vorhanden sind, die Sie anfordern können; es ist also die Aufgabe des Clients, den Iterator voranzutreiben. Die größte Flexibilität erhalten Sie sich mit einem externen Iterator. Im Folgenden werde ich mich nur mit externen Iteratoren beschäftigen. Sie werden einen internen Iterator beim Composite Pattern wiederfinden.

11.3 Das Interface Iterator in Java

Die Klassenbibliothek kennt das Interface `Iterator`, das eine Schnittstelle für alle erdenklichen Iteratoren darstellt. Mit `hasNext()` lassen Sie sich zurückgeben, ob noch weitere Elemente in der Datensammlung enthalten sind. Die Methode `next()` liefert das nächste Element in der Sammlung. Wenn der Client auf ein Element zugreifen möchte, das es nicht gibt, werfen Sie eine `NoSuchElementException`. Und `remove()` schließlich löscht das aktuelle Element aus der zugrunde liegenden Datensammlung. Die Methode `remove()` muss gemäß der Spezifikation nicht implementiert werden, sie darf eine `UnsupportedOperationException` werfen.

11.3.1 Der Iterator der Klasse MyArray

Die einfachste Form eines Iterators wird von der Methode `iterator()` in der Klasse `MyArray` zurückgegeben. Der Iterator speichert intern die Position, an der das Lesezeichen gesetzt wird. Die Methode `next()` gibt das jeweils nächste Element zurück, die Methode `hasNext()` liefert `true` zurück, wenn es weitere Elemente gibt. Schauen Sie sich das Beispielprojekt Sammlungen_2 und dort die Klasse `MyArray` an.

```
public class MyArray<E> {
    // ... gekürzt
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int position = -1;

            @Override
            public boolean hasNext() {
                return (position < size()) && elemente[position + 1] != null;
            }

            @Override
            public E next() {
```

```
        position++;
        if (position >= size() || elemente[position] == null)
            throw new NoSuchElementException("Keine weiteren Da-
ten mehr");
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
};

}

// ... gekürzt
}
```

Der nächste Abschnitt zeigt, wie Sie den Iterator verwenden.

11.3.1.1 Test des Iterators

Sie legen in der main-Methode zunächst eine Sammlung vom Typ `MyArray` an und speichern einige Strings darin.

```
var myArray = new MyArray<>();
myArray.add("String 1");
// ... gekürzt
myArray.add("String 6");
```

Danach lassen Sie sich einen Iterator zurückgeben und fragen in einer while-Schleife so lange Daten ab, bis keine weiteren Daten mehr in der Sammlung enthalten sind. Um die Exception zu provozieren, rufen Sie gezielt ein Element mehr ab, als gespeichert ist.

```
var iterator = myArray.iterator();

while (iterator.hasNext()) {
    String temp = iterator.next();
    System.out.println(temp);
}
// wirft eine Exception
System.out.println(iterator.next());
```

Auf der Konsole wird nun jeder String ausgegeben. Anschließend wird die Exception geworfen.

11.3.1.2 Nutzen und Varianten des Iterators

Die Iteration wird nun sehr viel einfacher und da der Client-Code sich nicht ändert, sind die Listen austauschbar. Sie lassen sich zunächst einen Iterator geben und rufen so lange die Methode `next()` auf, wie die Methode `hasNext()` ein `true` zurückgibt. Die Beispiele sind sehr einfach gehalten, die Methode `iterator()` gibt als Iterator die Instanz einer anonymen Klasse zurück. Dieser Iterator geht die Datenbasis elementweise von vorne nach hinten durch. Diese Vereinfachung darf nicht darüber hinwegtäuschen, dass Sie als Programmierer frei sind, den Iterator so zu definieren, wie Sie es brauchen. Beispielsweise könnte der Iterator das Array von hinten nach vorne durchgehen. Alternativ wäre auch ein Iterator denkbar, der die Datenbasis zunächst kopiert und/oder sortiert, bevor er die einzelnen Elemente zurückgibt. Auch, wenn Sie eigene komplexe Strukturen (Baumstrukturen, ...) bauen, können Sie dafür einen „Standard“-Iterator nach Ihren Wünschen entwerfen. Natürlich müssen Sie den Iterator nicht als anonyme Klasse definieren – Sie könnten auch eine Klasse außerhalb des Namensraums der Sammlung entwerfen.

11.3.2 Der Iterator der Klasse MyList

Die Klasse `MyList` – ebenfalls im Projekt `Sammlungen_2` – ist intern anders aufgebaut. Die einzelnen Elemente werden nicht in einem Array gespeichert, sondern verlinkt. Welche Konsequenz hat das für den Iterator? Es wird im Gegensatz zu `MyArray` kein Zähler als Lesezeichen gespeichert, sondern das aktuelle Element. Initial wird das aktuelle Element auf den Header der Liste gesetzt.

Wenn Sie testen möchten, ob eine Sammlung weitere Elemente enthält, müssen Sie fragen, ob das aktuelle Element des Iterators ungleich null ist; in diesem Fall darf die Methode `hasNext()` ein `true` zurückgeben. Die Methode `next()` gibt den Inhalt des aktuellen Elements zurück und rückt den Zeiger auf das nächste Element um eine Stelle weiter.

```
public Iterator<E> iterator() {
    return new Iterator<E>() {
        private Node current = header;

        @Override
        public boolean hasNext() {
            return (current != null);
        }

        @Override
        public E next() {
            if (current == null)
                throw new NoSuchElementException("...");
            @SuppressWarnings("unchecked")
            }
```

```
        var value = (E) current.ELEMENT;
        current = current.nextNode;
        return value;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
};

}
```

Im nächsten Abschnitt testen wir diesen Iterator auch mal.

11.3.2.1 Test des Iterators

Um den Iterator zu testen, legen Sie eine Instanz der Klasse `MyList` an und speichern in ihr verschiedene Strings. Sie lassen sich einen Iterator zurückgeben und iterieren über die komplette Sammlung. Natürlich habe ich auch bei diesem Test eine Abfrage zu viel eingebaut, um das Auslösen der Exception zu provozieren.

```
var myList = new MyList<>();
myList.add("String 1");
// ... gekürzt
myList.add("String 6");

var iterator = myList.iterator();
while (iterator.hasNext())
    System.out.println(iterator.next());

System.out.println(iterator.next());
```

Wieder werden zunächst alle Strings und auf der Konsole ausgegeben; anschließend wird eine Exception geworfen.

11.3.2.2 Nutzen des Iterators

Den Besonderheiten der beiden Sammlungsklassen wird nun Rechnung getragen, ohne dass der Client das überhaupt mitbekommt. Sie erinnern sich, dass Sie beim Observer Pattern gesehen haben, dass Iteratoren anfällig sind für Probleme, die aus Nebenläufigkeit resultieren. Da Sie mehrere Iteratoren erzeugen können, ist es gar nicht so unwahrscheinlich, dass während der Iteration ein Element gelöscht, hinzugefügt oder ersetzt wird. Sie stehen dann schnell vor der Situation, dass ein Iterator auf ein Element zugreifen möchte, welches es gar nicht mehr gibt oder dass er auf ein neues Element noch gar nicht zugreifen kann. Die Sammlungsklassen der Klassenbibliothek werfen in solchen Fällen eine `ConcurrentModificationException`.

11.4 Das Interface Iterable

Eine while-Schleife ist mit der for-Schleife verwandt. Anstelle der while-Schleife

```
while (iterator.hasNext()) {
    // ... Aktion
}
```

Könnten Sie auch eine for-Schleife verwenden:

```
for (var iterator = myList.iterator(); iterator.hasNext(); ; ) {
    // ... Aktion
}
```

Seit Java 5 gibt es die for-each-Schleife. Iterieren wird sehr viel einfacher:

```
for (var tempString : liste)
    System.out.println(tempString);
```

Wie können Sie Ihre Sammlungen `MyList` und `MyArray` so vorbereiten, dass Sie sie in einer for-each-Schleife verwenden können? Sie müssen lediglich das Interface `Iterable` implementieren. Dieses Interface schreibt die Methode `iterator()` vor, deren Rückgabewert der Iterator der Sammlung ist. Das sehen Sie im Beispielprojekt Sammlungen_3 – hier am Beispiel der Klasse `MyArray`:

```
public class MyArray<E> implements Iterable<E> {
    public Iterator<E> iterator() {
        // ... wie vorher
    }
    // ... gekürzt
}
```

Jetzt können Sie die Klasse `MyArray` in einer erweiterten for-Schleife einsetzen:

```
var myArray = new MyArray<>();
myArray.add("String 1");
// ... gekürzt
myArray.add("String 6");

for (var tempString : myArray)
    System.out.println(tempString);
```

Bei der `MyList` funktioniert das entsprechend. Schauen Sie sich das bitte selbstständig noch mal an.

Suchen Sie in der API-Doku nach dem Interface `Iterable`. Ein Subinterface hiervon ist das Interface `Collection`. Dieses Interface wird von allen gängigen Sammlungen wie `List` und `Set` und von vielen weiteren implementiert. Daher dürfen Sie darauf vertrauen, dass alle Sammlungen die Methode `iterator()` definieren, die ein Iterator-Objekt zurückgibt.

Bei einer Map ist das Thema etwas vertrackter. Nehmen Sie als Beispiel eine `HashMap`, die aus drei Sammlungen besteht: einem `KeySet` für die Schlüssel, einer `Collection` für die Werte und einem `EntrySet` für die Verbindung zwischen beiden Sammlungen. Niemand kann im Voraus wissen, ob der Anwender über die Schlüssel oder über die Werte iterieren möchte; daher kann die `HashMap` keinen „Standard“-Iterator entwerfen.

Die `for-each`-Schleife ist für den Programmierer ein sehr nützliches Sprachkonstrukt. Doch obwohl dem Programmierer viel Arbeit abgenommen wird, hat der Compiler verhältnismäßig wenig Arbeit damit. Er schreibt die `for-each`-Schleife vor dem Übersetzen in eine `while`-Schleife um (Code-Rewriting) und lässt sich den Iterator geben. Wenn Sie mit der `for-each`-Schleife über ein Array iterieren, wird die Schleife in eine herkömmliche `for-next`-Schleife umgeschrieben und dann übersetzt.

Sie sehen an diesem Beispiel, wie Patterns Einzug in die Klassenbibliothek gefunden haben. Nicht nur der Name, sondern auch die Realisierung des Patterns entspricht der Beschreibung der GoF.

11.5 Iterator – Das UML-Diagramm

Aus dem Beispielprojekt `Sammlungen_3/MyList` sehen Sie das UML-Diagramm in Abb. 11.1.

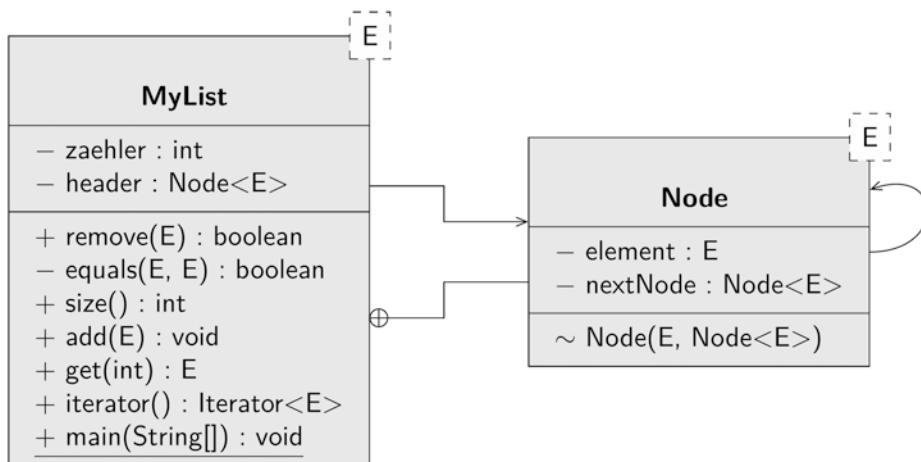


Abb. 11.1 UML-Diagramm des Iterator Pattern (Beispielprojekt `Sammlungen_3/MyList`)

11.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sammlungen können intern sehr unterschiedlich aufgebaut sein.
- Iteratoren sind Lesezeichen, die von einem Element zum nächsten verschoben werden.
- Sie ermöglichen es, ohne Kenntnis der internen Struktur über die Sammlung zu iterieren.
- Interne Iteratoren sind für das Voranschreiten des Iterators selbst verantwortlich.
- Bei externen Iteratoren ist der Client für das Voranschreiten verantwortlich.
- Ein Iterator hat, wenn er sich an der Java-Spezifikation orientiert, drei Methoden:
 - `remove()` löscht ein Element aus der Sammlung – die Methode muss nicht überschrieben werden,
 - `hasNext()` gibt `true` zurück, wenn die Sammlung weitere Elemente hat,
 - `next()` liefert das nächste Element der Sammlung zurück; gibt es kein Element, wird eine `NoSuchElementException` geworfen.
- Der `ListIterator` ist ein auf Listen spezialisierter Iterator und hat noch einige Methoden mehr, die auch eine Iteration in beide Richtungen erlauben.
- Der Iterator wird von der Methode `iterator()` zurückgegeben.
- Der Programmierer kann die Definition eines eigenen Iterators auf die Struktur seiner Sammlung bedarfsgerecht anpassen.
- Es kann mehrere Iterator-Klassen und mehrere Iterator-Instanzen geben.
- Während mindestens ein Iterator aktiv ist, darf die Datenbasis nicht geändert werden.
- Wenn eine Sammlung das Interface `Iterable` implementiert, kann mit der for-each-Schleife über sie iteriert werden.

Zweckbeschreibung

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

„Bietet eine Möglichkeit, um auf die Elemente eines zusammengesetzten Objekts sequenziell zugreifen zu können, ohne die zugrunde liegende Repräsentation offenzulegen.“



Das Composite Pattern, das ich jetzt vorstellen werde, gehört zu den Strukturmustern. Es beschreibt, wie Objekte zusammengesetzt werden, um größere sinnvolle Einheiten zu bilden. Eine zusammengesetzte Struktur lässt sich prima in einem JTree anzeigen. Ich werde in diesem Kapitel daher auch beschreiben, wie Sie eigene Datenmodelle für einen JTree definieren.

12.1 Prinzip von Composite

Sie haben eine Struktur, die aus mehreren Einheiten besteht. Diese Einheiten können optional kleinere Einheiten enthalten. Sie möchten auf allen diesen Einheiten bestimmte Methoden aufrufen können, ohne sich darüber Gedanken machen zu müssen, ob die Einheit weitere Einheiten enthält oder nicht. Was ist damit gemeint?

Falls Sie Haushaltsbuch führen, notieren Sie die einzelnen Positionen Ihrer Einnahmen und Ausgaben. Zu den Einnahmen gehört Ihr Gehalt aus Ihrer Haupttätigkeit. Vielleicht haben Sie ja auch einen Nebenjob, dann wird der ebenfalls unter die Rubrik der Einnahmen geschrieben. Zu den Ausgaben gehört bestimmt Ihre Miete oder Raten für eine eigene Immobilie. Daneben haben Sie auch Positionen, die sich unterteilen lassen. Unter die Rubrik „Essen“ fallen verschiedene Unterkategorien: Mittagessen, Essengehen mit Freunden usw. In diesen Unterkategorien stehen die einzelnen Positionen wie „Pizzeria 16,00 €“ oder „Kantine 4,00 €“. Sie können auf unterster Ebene fragen: „Was hat der Besuch in der Pizzeria gekostet?“ Sie können aber auch aufsummieren, indem Sie fragen: „Was habe ich

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_12

insgesamt für Lebensmittel bezahlt?“ Und schließlich können Sie auf oberster Ebene fragen: „Was habe ich insgesamt eingenommen und ausgegeben?“

Oder denken Sie an ein Unternehmen, das aus verschiedenen Abteilungen besteht, die sich in mehrere Hierarchiestufen gliedern; wenn Sie dort einen Mitarbeiter nach den Personalkosten fragen, nennt er sein eigenes Gehalt. Wenn Sie einen Abteilungsleiter nach seinen Personalkosten fragen, wird er erst die Personalkosten seiner unterstellten Mitarbeiter ermitteln, sein eigenes Gehalt addieren und Ihnen die gesamte Summe zurückgeben. Wenn Sie den Firmeninhaber fragen, gibt er zusätzlich zu seinem auch die aufsummierten Personalkosten aller Abteilungen zurück.

Ganz praktisch: Das Dateisystem Ihres Rechners gibt sowohl die Größe eines Ordners als auch einer einzelnen Datei zurück.

Kurz gesagt: Das Composite Pattern hilft Ihnen, Baumstrukturen darzustellen. Lassen Sie uns einige Begriffe klären! Jedes Element in einer Baumstruktur ist ein Knoten. Ein Knoten, der keine Unterknoten hat, ist ein Blatt (oder auch: leaf); das ist beispielsweise die einzelne Position „Pizzeria 16,00 €“. Ein Knoten, der Unterknoten hat, heißt Kompositum (oder auch: Composite); das ist zum Beispiel die Rubrik „Lebensmittel“. Der Knoten an der Spitze des Baums (der Firmeninhaber) ist die Wurzel (im Fachjargon auch: Root).

Es gibt zwei unterschiedliche Ansätze, das Composite Pattern zu realisieren.

12.2 Umsetzung 1: Sicherheit

Leafs und Composites müssen unterschiedliches Verhalten haben; sie sind folglich in unterschiedlichen Klassen zu definieren. Composites können jedoch sowohl andere Composites als auch Leaf als Knoten speichern; daher müssen Composites und Leaf eine gemeinsame Schnittstelle haben.

Im Beispielprojekt Haushalt_1 finden Sie einen ersten Entwurf. Die Klasse Knoten ist die gemeinsame Schnittstelle für Blätter und Komposita. Sie definiert ein Datenfeld, das die Einnahmen- bzw. Ausgabenposition beschreibt; dieses Feld wird sowohl bei Blättern als auch bei Komposita gebraucht. Und schließlich wird die Methode `print()` vorgeschrieben.

```
public abstract class Knoten {
    protected final String beschreibung;

    // ... gekürzt
    public abstract void print(int einrueckung);
}
```

Die Klasse Blatt erweitert die Klasse Knoten. Sie definiert zusätzlich ein Feld, in dem gespeichert wird, ob die Position erforderlich ist oder ob sie Luxus ist. Bei Einnahmen spielt dieses Flag sicher keine Rolle. Allerdings werden Sie sicher darüber nachdenken, ob

eine Ausgabe notwendig war oder nicht. Außerdem definiert sie ein Datenfeld, das den Betrag der Position speichert. Die print-Methode nimmt einen integer-Wert entgegen, der die Anzahl von Einrückungen beschreibt; entsprechend dieser Anzahl werden Tabulatoren eingefügt, bevor der Wert des Objekts ausgedruckt wird. Positionen, die erforderlich sind, wird ein Ausrufezeichen vorangestellt; Ausgaben, die nicht unbedingt erforderlich sind, werden nicht gesondert markiert.

```
public class Blatt extends Knoten {  
    private final boolean erforderlich;  
    private double betrag = 0.0;  
  
    // ... gekürzt  
  
    @Override  
    public void print(int einrueckung) {  
        for (var i = 0; i < einrueckung; i++)  
            System.out.print("\t");  
        System.out.println(this);  
    }  
  
    @Override  
    public String toString() {  
        var prefix = erforderlich ? "(!) " : "() ";  
        var tempBetrag = NumberFormat.getCurrencyInstance().format(betrag);  
        return prefix + beschreibung + ": " + tempBetrag;  
    }  
}
```

Die Klasse Kompositum definiert eine Liste, in der die Kindknoten gespeichert werden, und die erforderlichen Zugriffsmethoden. Dazu gehört beispielsweise eine Methode, die die Anzahl Kindknoten zurückgibt, und eine Methode, die das Kind an einer bestimmten Stelle returniert. Die print-Methode wird so überschrieben, dass entsprechend der Einrückung erst der Wert der toString-Methode ausgegeben wird und danach rekursiv alle Kindknoten.

```
public class Kompositum extends Knoten {  
    private final List<Knoten> kinder = new ArrayList<>();  
  
    // ... gekürzt  
    public Knoten getKind(int index) {  
        return kinder.get(index);  
    }  
}
```

```

public int getAnzahlKindKnoten() {
    return kinder.size();
}

@Override
public void print(int einrueckung) {
    for (var i = 0; i < einrueckung; i++) {
        System.out.print("\t");
    }
    System.out.println(this);
    kinder.forEach((knoten) -> {
        knoten.print(einrueckung + 1);
    });
}

@Override
public String toString() {
    return beschreibung;
}
}
}

```

Der Client legt Variablen für die Wurzel und verschiedene Ausgabenkategorien an, also beispielsweise Einnahmen und Ausgaben. Bei den Ausgaben gibt es unter anderem die Kategorie Bücher, wo zwei Bücher eingestellt werden. Das Listing drucke ich nur gekürzt ab.

```

final var root = new Kompositum("Haushaltsbuch");
final var januar = new Kompositum("Januar");
final var einnahmen = new Kompositum("Einnahmen");
final var ausgaben = new Kompositum("Ausgaben");
final var bucher = new Kompositum("Bücher");

januar.add(einnahmen);
januar.add(ausgaben);

root.add(januar);

einnahmen.add(new Blatt("Hauptjob", 1900.00, true));
einnahmen.add(new Blatt("Nebenjob", 200.00, true));

ausgaben.add(BUECHER);
ausgaben.add(new Blatt("Miete", -600.00, true));
bucher.add(new Blatt("Patternbuch", -29.9, true));
bucher.add(new Blatt("Schundroman", -9.99, false));

```

```
root.print(0);
System.out.println("\n\nNur die Ausgaben: ");
ausgaben.print(0);
```

Der Client kann sich nun darauf beschränken, `root.print(0)` aufzurufen. Dann werden die Einnahmen und Ausgaben für Januar übersichtlich auf der Konsole ausgegeben:

Haushaltsbuch

```
Januar
Einnahmen
(+ ) Hauptjob: 1.900,00 €
(+ ) Nebenjob: 200,00 €
Ausgaben
Versicherungen
(+ ) KfZ: -50,00 €
(+ ) BU: -100,00 €
Bücher
(+ ) Patternsbuch: -29,90 €
(- ) Schundroman: -9,99 €
(+ ) Miete: -600,00 €
```

Es ist möglich, dass Sie sich nur die Ausgaben im Januar ausdrucken lassen; rufen Sie dazu den `print`-Befehl auf dem Kompositum `ausgaben` auf: `ausgaben.print(0)`. Diese Lösung funktioniert tadellos. Sie erweist sich jedoch an bestimmten Stellen als zu unflexibel – und das werden Sie im folgenden Abschnitt sehen und korrigieren.

12.3 Umsetzung 2: Transparenz

Der Client-Code verstößt gegen den Grundsatz, dass man gegen Schnittstellen und nicht gegen Implementierungen programmieren soll: `final Kompositum root = new Kompositum("Haushalt")`. Das war unumgänglich, weil die Schnittstelle `Knoten` lediglich die Methoden deklariert, die auch wirklich sowohl von `Blättern` als auch von `Komposita` benötigt werden. Zu diesem „kleinsten gemeinsamen Nenner“ gehört `add()` nicht. Und hier beginnt ein gewaltiges Problem zu reifen – das Programm aus dem vorigen Ansatz konnte nur mit einem kleinen „Trick“ zum Laufen gebracht werden. Die `print`-Methode ruft sich rekursiv auf allen Objekten auf. Würden Sie allerdings mit einem externen Iterator auf die Liste zugreifen wollen oder sich ein einzelnes Element zurückgeben lassen, müssten Sie mit zahlreichen Vergleichen (`instanceof Kompositum`) und Downcasts arbeiten. Downcasts sind jedoch die Brechstange des Programmierers; wenn möglich sollte darauf verzichtet werden.

Um das Problem zu demonstrieren, möchte ich die Liste erneut auf der Konsole ausgeben, diesmal aber mit einem externen Iterator. Ich lösche im Beispielprojekt `Haushalt_2`

die print-Methode. Jetzt muss der Client selbst für die Iteration sorgen. Die wichtigste Änderung in diesem Projekt ist also im Bereich der Testroutine. Zunächst legen Sie wie im vorigen Beispiel einige Einnahmen und Ausgaben an. Dann rufen Sie innerhalb der Testklasse eine neu definierte print-Methode auf. In dieser Methode werden zuerst die benötigten Tabs ausgedruckt und dann das Objekt selbst. Anschließend wird geprüft, ob das auszugebende Element ein Kompositum ist. Wenn ja, wird das Knoten-Objekt auf ein Kompositum-Objekt gecastet. Und schließlich fragen Sie das Kompositum nach der Anzahl seiner Kindknoten und geben jeden Kindknoten auf der Konsole aus.

```
public static void main(String[] args) {
    final var root = new Kompositum("Haushaltsbuch");
    final var januar = new Kompositum("Januar");
    final var februar = new Kompositum("Februar");

    // ... gekürzt

    print(root, 0);
}

private static void print(Knoten knoten, int einrueckung) {
    for (var i = 0; i < einrueckung; i++)
        System.out.print("\t");
    System.out.println(knoten);
    if (knoten instanceof Kompositum kompositum)
    {
        // Kompositum kompositum = (Kompositum) knoten;
        var anzahlKinder = kompositum.getAnzahlKindKnoten();
        for (var j = 0; j < anzahlKinder; j++)
        {
            var kindKnoten = kompositum.getKind(j);
            print(kindKnoten, einrueckung + 1);
        }
    }
}
```

Finden Sie, dass das Beispiel ein wenig konstruiert ist? Keineswegs! Sie werden gleich eine Implementierung der Schnittstelle TreeModel kennenlernen – dort wäre eine Realisierung mit einem internen Treiber schwierig.

Fällt Ihnen bei dem instanceof-Befehl etwas auf? Hier habe ich ein in Java 14 neu hinzugekommenes Preview-Feature verwendet: Pattern Matching for instance of, zu finden unter dem Java Enhancement Proposal JEP 305. Sie können es nur nutzen, wenn Sie unter Java 14 oder 15 (JEP 375) die Preview-Features aktivieren. Das geht über die zusätzliche Compiler-Option – enable-preview, die Sie in den Compiling-Projekteinstellungen von

NetBeans oder eben beim Kommandozeilenaufruf des Java-Compilers javac hinzufügen müssen. Unter Java 16 (JEP 394) ist dieses feature endgültig final. Aber was bewirkt dieses Pattern Matching eigentlich?

Wir bekommen die Referenz auf den Knoten, der entweder ein Blatt oder ein Kompositum sein kann. Wenn wir abhängig von der Subklasse unterschiedlich agieren wollen, prüfen wir mit instanceof die Zugehörigkeit zur Klasse Kompositum und casten dann die Referenz „runter“ auf diese Klasse. Bisher waren das in Java zwei Schritte: Erst die Prüfung, dann das Casten. Jetzt funktioniert das in einem einzigen Schritt. Sie sehen im Code, dass hinter dem bisher verwendeten if (knoten instanceof Kompositum) jetzt noch einmal kompositum steht. Damit wird automatisch diese Variable eingeführt und der Knoten auf die Klasse Kompositum gecastet. Die zusätzliche Zeile Kompositum kompositum = (Kompositum) knoten; entfällt damit vollständig.

Solche Prüfungen werden in Java sehr häufig verwendet. Das Pattern Matching vereinfacht sie sowohl im Programmieraufwand als auch in der Lesbarkeit.

Das Beispielprojekt Haushalt_3 zeigt, wie Sie Allgemeingültigkeit herstellen. Sie definieren in der Klasse Knoten die folgenden zwei Methoden:

```
public abstract class Knoten {  
    protected final String beschreibung;  
  
    public Knoten(String beschreibung) {  
        this.beschreibung = beschreibung;  
    }  
  
    public Knoten getKind(int index) {  
        throw new RuntimeException("Ein Blatt hat keine Kindknoten");  
    }  
  
    public int getAnzahlKindKnoten() {  
        return 0;  
    }  
}
```

Die Klasse Kompositum überschreibt diese Methoden wie bisher. Neu in dieser Realisierung ist also, dass ein Blatt jetzt ebenfalls angeben kann, wie viele Kinder es hat – nämlich keines. Sofern ein Client dennoch versucht, ein Kind an einem bestimmten Index aufzurufen, fliegt ihm eine RuntimeException um die Ohren. Der Client muss sich also vergewissern, dass der auszugebene Knoten auch tatsächlich Kindknoten hat. Die print-Methode in der Testklasse wird allgemeingültiger formuliert.

```
private static void print(Knoten knoten, int einrueckung) {  
    for (var i = 0; i < einrueckung; i++)  
        System.out.print("\t");
```

```

        System.out.println(knoten);

        for (var j = 0; j < knoten.getAnzahlKindKnoten(); j++) {
            var kindKnoten = knoten.getKind(j);
            print(kindKnoten, einrueckung + 1);
        }
    }
}

```

Ein Blatt-Objekt gibt 0 als Anzahl Kinder zurück. Die Bedingung der for-Schleife ist daher nicht erfüllt, ein rekursiver Aufruf der print-Methode findet nicht statt.

12.4 Betrachtung der beiden Ansätze

Der erste Ansatz setzt auf eine schmale Schnittstelle. Ein Blatt kann nur, was es auch unbedingt können muss. Daher finden Sie Verwaltungsmethoden für die Liste der Kindknoten nur bei den Komposita. Die Sicherheit steht bei diesem Vorgehen im Vordergrund. Der Client muss zwar vergleichen und casten, er hat aber dafür die Gewähr, dass die Methoden, die er aufruft, auch sinnentsprechend ausgeführt werden. Das AWT beruht beispielsweise auf diesem Prinzip. Dort gibt es die abstrakte Klasse Component. In ihr werden Methoden definiert, die sowohl Blätter als auch Komposita verwenden: Registrieren von Listenern, Sichtbarkeit verändern usw. Von Component sind die verschiedenen Steuerelemente (Button, Label, CheckBox usw.) abgeleitet. Daneben erbt die Klasse Container von Component. Sie definiert Methoden, um Komponenten als Kindknoten zu verwalten. Von Container erben nur die Klassen, die in der Lage sein müssen, andere Komponenten aufzunehmen, also Frame und Panel.

Die Alternative ist der Weg über eine breite Schnittstelle: Blätter und Komposita können prinzipiell alles. Auf diesem Ansatz ist Swing aufgebaut. Es gibt dort die Klasse JComponent, die von Container erbt. Von JComponent erben alle Steuerelemente, also auch JLabel, JButton usw. Sie alle haben also die Methode add(), der Sie ein Component-Objekt übergeben können. In der Folge ist es möglich, ein JPanel in einem JLabel zu speichern, was Sie mit dem folgenden Code z. B. in der Java Shell von NetBeans überprüfen können.

```

javax.swing.JPanel pnlTest = new javax.swing.JPanel();
javax.swing.JLabel lblTest = new javax.swing.JLabel();
lblTest.add(pnlTest);
int anzahl = lblTest.getComponentCount();
System.out.println("Anzahl Kinder im JLabel: " + anzahl);
System.out.println("Parent des JPanel: " + pnlTest.getParent());

```

Auf der Konsole wird – gekürzt – ausgegeben:

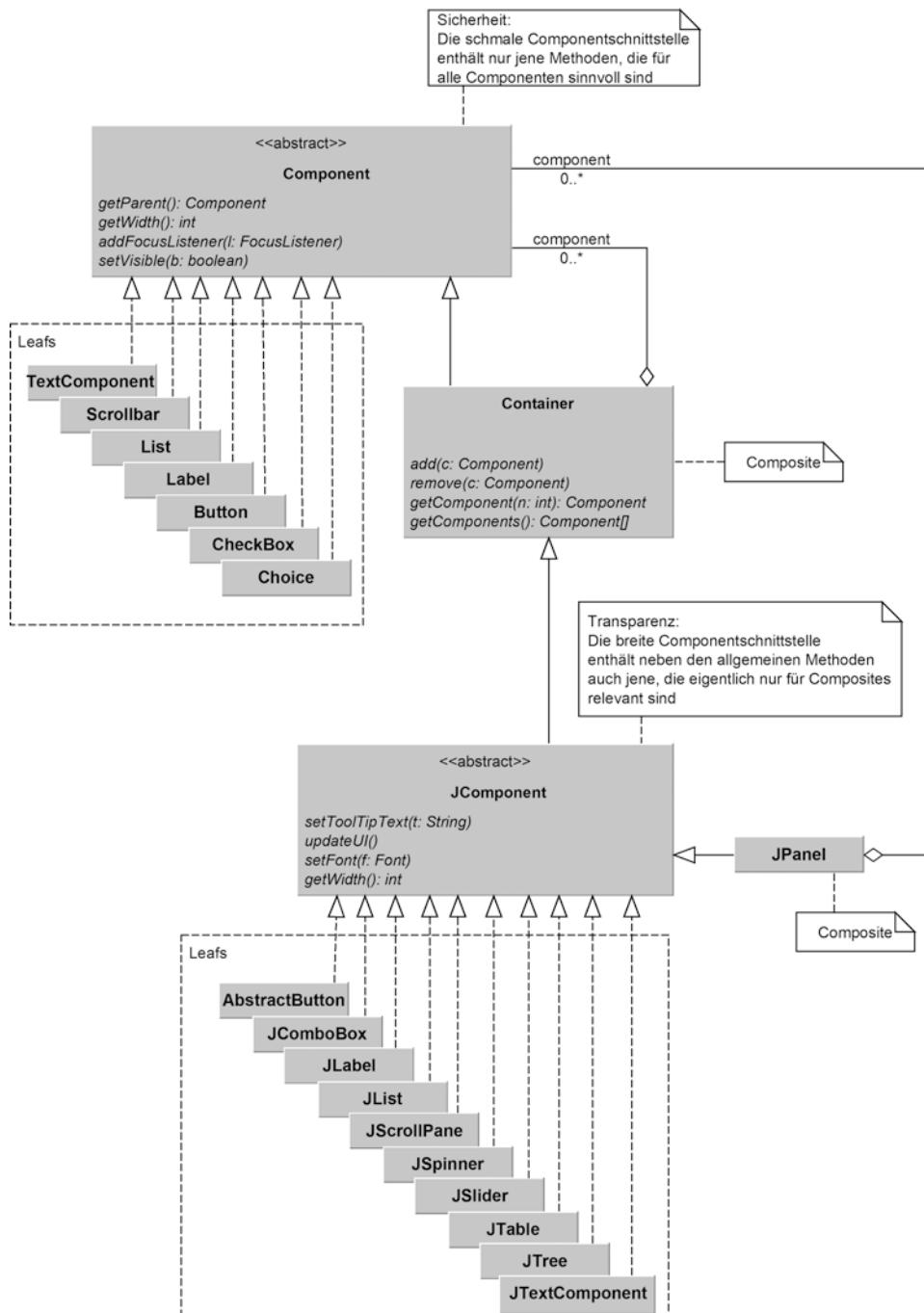


Abb. 12.1 Klassendiagramm AWT- und Swing-Komponenten, entlehnt von www.PhilippHauer.de

```
Anzahl Kinder im JLabel: 1
Parent des JPanel: javax.swing.JLabel...
```

Das Zusammenspiel von AWT und Swing wird in der Abb. 12.1 dargestellt. Ich habe das Diagramm bei Philipp Hauer entlehnt. Philipp beschäftigt sich auf seiner Seite www.PhilippHauer.de unter anderem mit Design Patterns. Es lohnt sich auf jeden Fall, seine Seite zu besuchen.

Eine zu breite Schnittstelle ist problematisch. Nehmen Sie Projekt Haushalt_3 mal gedanklich als Grundlage. Sie definieren dort Methoden, die spezifisch für ein Kompositum sind, in der Schnittstelle, um Allgemeingültigkeit zu erhalten. In der Testklasse fragen Sie die Anzahl Kindknoten ab und lassen sich einen Kindknoten zurückgeben. Das ist sicher kein Problem. Aber schwierig wird es, wenn Sie die Methode `addChild()` in die Schnittstelle aufnehmen müssen. Wie implementiert die Klasse Blatt diese Methode? Wie sieht die Default-Implementierung aus? Eine Lösung könnte sein, dass das Blatt einfach gar nichts macht `add() {}`. Eine leere Implementierung ist aber nicht unproblematisch – der Client möchte bestimmt wissen, dass sein Auftrag nicht ausgeführt werden kann. Der Client kommt also nicht umhin, im Vorfeld eine Fallunterscheidung zu treffen. Obwohl dieser Ansatz problematisch sein kann, wird er in der Praxis favorisiert, und auch die GoF spricht sich dafür aus.

12.5 Einen Schritt weiter gehen

Bauen wir das Projekt nun weiter aus.

12.5.1 Einen Cache anlegen

Es wäre doch eine sinnvolle Sache, wenn die Kategorien die einzelnen untergeordneten Positionen aufsummieren könnten, und das wird mit dem Beispielprojekt Haushalt_4 realisiert. Hierfür legen Sie in den Komposita einen Cache an, der die Summe der Kindknoten speichert. In der Schnittstelle deklarieren Sie eine abstrakte Methode `calculateCache()`, die von Blättern genauso wie von Komposita überschrieben werden muss. Außerdem sollen sowohl Blätter als auch Komposita in der Lage sein, den intern gespeicherten Wert zurückzugeben.

```
public abstract class Knoten {
    // ... gekürzt

    abstract double getValue();

    public abstract void calculateCache();
}
```

Die Definition in der Klasse Blatt ist recht einfach – ein Blatt muss keinen Cache anlegen, also kann die Methode leer überschrieben werden. Allerdings muss ein Blatt den gespeicherten Betrag nennen können:

```
public class Blatt extends Knoten {  
  
    // ... gekürzt  
  
    private double betrag = 0.0;  
  
    @Override  
    public void calculateCache() {  
    }  
  
    @Override  
    double getValue() {  
        return betrag;  
    }  
}
```

Die Klasse Kompositum gibt den Cache zurück, wenn die Methode `getValue()` aufgerufen wird. Die Methode `calculateCache()` ist so definiert, dass der Cache zunächst geleert wird. Anschließend bekommen alle Kinder rekursiv die Anweisung, ihren Cache zu berechnen. Schließlich wird von allen Kindern der gespeicherte Betrag abgefragt und zum eigenen Cache addiert.

```
public class Kompositum extends Knoten {  
    // ... gekürzt  
  
    private double cache = 0.0;  
  
    @Override  
    public void calculateCache() {  
        cache = 0;  
        for (var knoten : kinder) {  
            knoten.calculateCache();  
            cache += knoten.getValue();  
        }  
    }  
}
```

```

@Override
double getValue() {
    return cache;
}

@Override
public String toString() {
    var tempCache = NumberFormat.getCurrencyInstance().format(cache);
    return beschreibung + " (Summe: " + tempCache + ")";
}
}

```

Der Test des Programms unterscheidet sich nur minimal von den vorigen Beispielen. Der Baum wird wie gehabt aufgebaut. Anschließend bekommt der Wurzelknoten die Anweisung, seinen Cache zu berechnen.

```

final var root = new Kompositum("Haushaltsbuch");
// ...gekürzt
buecher.add(new Blatt("Patternbuch", -29.9, true));
buecher.add(new Blatt("Schundroman", -9.99, false));

ROOT.calculateCache();
print(ROOT, 0);

```

Wenn Sie das Programm starten, erhalten Sie die Zwischensummen der Kategorien.

```

Haushaltsbuch (Summe: 1.310,11 €)
    Januar (Summe: 1.310,11 €)
        Einnahmen (Summe: 2.100,00 €)
            (+) Hauptjob: 1.900,00 €
            (+) Nebenjob: 200,00 €
        Ausgaben (Summe: -789,89 €)
            Versicherungen (Summe: -150,00 €)
                (+) KfZ: -50,00 €
                (+) BU: -100,00 €
            Bücher (Summe: -39,89 €)
                (+) Patternbuch: -29,90 €
                (-) Schundroman: -9,99 €
                (+) Miete: -600,00 €
    Februar (Summe: 0,00 €)

```

Da das Projekt sehr überschaubar ist, wäre es sicher vertretbar gewesen, dass die Kategorien den Cache jeweils neu berechnen, wenn `getValue()` aufgerufen wird. Ein Cache ist jedoch dann sinnvoll, wenn der Aufruf von `getValue()` im Blatt hohe Kosten verursacht.

12.5.2 Die Elternkomponenten referenzieren

Im folgenden Schritt wird die Struktur als doppelt verlinkte Liste angelegt – die Eltern kennen ihre Kinder, aber die Kinder kennen auch ihre Eltern. Sie kennen das sicher von Swing – Sie rufen auf einer Komponente die Methode `getParent()` auf; zurückgegeben wird der übergeordnete Knoten. Im Beispieldatenprojekt Haushalt_5 pflegen Sie neue Einnahmen oder Ausgaben ein; die übergeordnete Kategorie wird darüber informiert und kann ihren Cache aktualisieren. Fügen Sie in der Klasse `Knoten` das Feld `Kompositum vater` mit den entsprechenden Zugriffsmethoden ein.

```
public abstract class Knoten {  
    // ... gekürzt  
  
    private Kompositum vater = null;  
  
    protected void setParent(Kompositum vater) {  
        this.vater = vater;  
    }  
  
    protected Kompositum getParent() {  
        return this.vater;  
    }  
}
```

Die Methode `add()` der Klasse `Kompositum` wird erweitert. Ein `Kompositum` über gibt sich selbst als Parent an jeden neu eingefügten Kindknoten. Wenn das neu hinzuzufü gende Kind bereits einen Vater hat, wird eine Exception geworfen – die Situation dürfte nämlich nicht vorkommen.

```
public class Kompositum extends Knoten {  
    // ... gekürzt  
  
    public void add(Knoten kind) {  
        kinder.add(kind);  
        var parent = kind.getParent();  
        if (parent == null)  
            kind.setParent(this);  
        else  
            throw new RuntimeException(kind + " hat schon einen Vater: " +  
                parent);  
    }  
}
```

Die Einführung einer Referenz auf den Vaterknoten wirkt sich in dieser Projektversion noch nicht aus. Der Vaterknoten wird im Beispielprojekt Haushalt_6 genutzt, um den Cache neu zu berechnen. Wenn Sie ein neues Blatt einfügen bzw. den Wert einer Einnahmen- oder Ausgabenposition ändern, geben alle betroffenen Komposita diese Information so lange an den Vater weiter, bis die Nachricht bei der Wurzel angekommen ist. Der Wurzelknoten ruft dann wie im vorigen Projekt rekursiv die Methode `calculateCache()` auf allen untergeordneten Knoten auf.

Sie möchten, dass nur die Komposita ihre Caches neu berechnen, die betroffen sind. Sie führen in der Klasse `Kompositum` ein Flag ein, das angibt, ob der Cache neu berechnet werden muss. Wenn einem Kompositum ein neuer Knoten hinzugefügt wird, ist der Cache nicht mehr gültig. Also veranlasst `add()`, dass der eigene Cache und der Cache des Vaterknotens neu berechnet werden. Das Gleiche gilt, wenn ein Knoten entfernt wird. Wenn die Methode `calculateCache()` den Cache gerade neu berechnet hat, ist dieser sicher korrekt und das Flag darf auf `true` gesetzt werden. Interessant ist die neu eingefügte Methode `setCacheIsValid()`. Wenn ihr `true` übergeben wird, wird das Flag korrigiert. Wird ihr `false` übergeben, wird zusätzlich der Vaterknoten angewiesen, den Cache als invalid zu markieren. Wenn es keinen Vater gibt – das gilt nur für den Wurzelknoten – werden die Kindknoten angewiesen, ihren Cache neu zu berechnen. Diese berechnen den Cache allerdings nur dann neu, wenn das Flag `isValid` auf `false` steht.

```
public class Kompositum extends Knoten {

    // ... gekürzt

    private boolean cacheIsValid = false;

    public void add(Knoten kind) {
        kinder.add(kind);
        var parent = kind.getParent();
        if (parent == null)
            kind.setParent(this);
        else
            throw new RuntimeException(kind + " hat schon einen Vater:
" + parent);
        this.setCacheIsValid(false);
    }

    public void remove(Knoten kind) {
        kinder.remove(kind);
        kind.setParent(null);
        this.setCacheIsValid(false);
    }
}
```

```
void setCacheIsValid(boolean isValid) {
    this.cacheIsValid = isValid;
    if (!isValid) {
        var parent = this.getParent();
        if (parent == null)
            this.calculateCache();
        else
            if (this != parent)
                parent.setCacheIsValid(isValid);
    }
}

@Override
public void calculateCache() {
    if (!cacheIsValid) {
        cache = 0;
        for (var knoten : kinder) {
            knoten.calculateCache();
            cache += knoten.getValue();
        }
        this.setCacheIsValid(true);
    }
}
```

Ich nehme die gleiche Testroutine wie im vorigen Projekt. Auf root.calculateCache() kann ich jetzt verzichten, weil die Struktur die Zwischenwerte „automatisch“ berechnet, wenn neue Positionen eingetragen werden. Um die neuen Funktionen detaillierter zu testen, füge ich bei den Versicherungen eine Riester-Rente ein, die mit 1000,00 € zu Buche schlägt. Ich lasse mir das Haushaltsbuch erneut ausgeben und stelle fest, dass die Zahlen stimmen. Mir fällt auf, dass die Riester-Rente nicht monatlich mit 1000,00 € anfällt, sondern jährlich. Also lösche ich die Versicherung und lasse mir das Haushaltsbuch erneut ausgeben – die Zahlen stimmen wieder. Ich füge die Riester-Rente erneut ein, diesmal aber auf gleicher Ebene mit den Monaten, also direkt unterhalb der Wurzel. Und wieder werden die richtigen Werte ausgegeben. Die Struktur des Patterns und der Berechnungsmethode ermöglichen hier also eine sehr einfach nutzbare automatische Aktualisierung.

```
final var root = new Kompositum("Haushaltsbuch");

// ... gekürzt

print(root, 0);
var riester = new Blatt("Riester", -1000.00, true);
```

```

versicherungen.add(riester);
print(root, 0);
versicherungen.remove(riester);
print(root, 0);
root.add(riester);
print(root, 0);

```

Fanden Sie, dass es etwas umständlich war, wie ich die Riester-Rente verschoben habe? Ich auch – lassen Sie uns das im nächsten Abschnitt korrigieren.

12.5.3 Knoten verschieben

Das Beispielprojekt Haushalt_7 beschäftigt sich mit der Frage, wie ein Knoten auf einen anderen Knoten verschoben wird. Da sowohl Blätter als auch Komposita verschoben werden müssen, wird in der Klasse `Knoten` die Methode `changeParent()` definiert, der Sie eine Referenz auf den neuen Vaterknoten übergeben. Die Methode lässt sich den aktuellen Vater zurückgeben und trägt sich dort als Kindknoten aus. Beim neuen Vater trägt der Knoten sich als Kindknoten ein. Da sowohl `add()` als auch `remove()` veranlassen, dass die Zwischenspeicher der Komposita neu berechnet werden, ist die gesamte Baumstruktur wieder aktuell und richtig berechnet.

```

public void changeParent(Kompositum neuerVater) {
    var parent = this.getParent();
    parent.remove(this);
    neuerVater.add(this);
}

```

12.6 Composite – Das UML-Diagramm

Aus dem Beispielprojekt Haushalt_7 finden Sie das UML-Diagramm in Abb. 12.2.

12.7 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Ein Composite repräsentiert hierarchische Strukturen.
- Die Strukturen werden aus Knoten (Blättern und Komposita) gebildet.
- Der Knoten, der keinen Vaterknoten hat, wird Wurzel genannt.
- Knoten können beliebig tief geschachtelt sein.

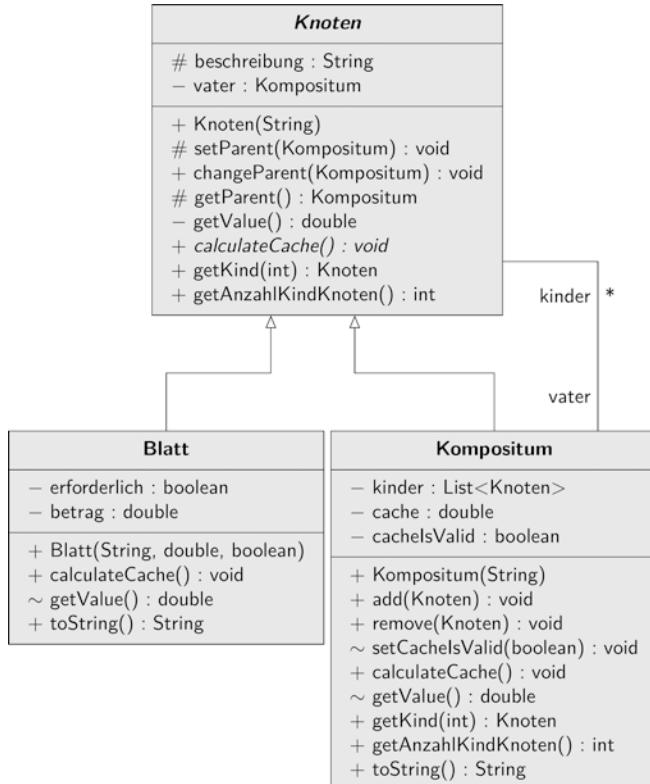


Abb. 12.2 UML-Diagramm des Composite Pattern (Beispielprojekt Haushalt_7)

- Blätter und Komposita verhalten sich für den Client identisch.
- Blätter und Komposita erben von einer gemeinsamen Schnittstelle.
- Die Breite der Schnittstelle bringt entweder Sicherheit oder Transparenz.
- Eine zu breite Schnittstelle zieht zu allgemeine Entwürfe nach sich.
- Knoten können optional eine Referenz auf ihren Vaterknoten halten.
- Komposita können Werte ihrer Kindobjekte cachen.

Zweckbeschreibung

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

„Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“



Das Flyweight Pattern ist ein Strukturmuster; um es verstehen zu können, müssen die Begriffe intrinsischer und extrinsischer Zustand definiert werden. Daher werde ich Ihnen in diesem Kapitel zwei Beispiele vorstellen. Das erste Beispiel verdeutlicht das Prinzip des Patterns, das zweite ist näher am Beispiel der GoF und beschreibt diese beiden Begriffe.

13.1 Aufgabe des Patterns

Das Fliegengewicht hat die Aufgabe, aus einem Elefanten eine Mücke zu machen. Stellen Sie sich vor, Sie programmieren eine Anwendung, die alle Geburten eines Jahres erfasst. Jeder Datensatz besteht aus einer ID1, dem Vornamen, dem Zunamen und dem Geburts- tag. Schauen Sie sich das sehr einfache Beispielprojekt Geburten an.

```
public class Kind {  
    private final long id;  
    private final String vorname;  
    private final String nachname;  
    private final Date gebTag;  
  
    public Kind(String vorname, String nachname, LocalDate gebTag) {  
        this.vorname = vorname;  
        this.nachname = nachname;
```

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_13

```

        this.gebTag = gebTag;
        id = (long) (((Long.MAX_VALUE * Math.random())
                      * (vorname.hashCode()
                      + nachname.hashCode()
                      + gebTag.getTime())));
    }
}

```

Ein Client legt eine Vielzahl von Kindern an:

Ein kritischer Blick auf den Clientcode zeigt, dass zwei Kinder Hans heißen. Zwei weitere tragen den Nachnamen Kirch. Und alle sechs sind am gleichen Tag geboren. Entsprechen diese sich wiederholenden Werte Ihren Erwartungen?

```

public class ApplStart {

    public static void main(String[] args) {
        var kind_1 = new Kind("Hans", "Kirch", LocalDate.now());
        var kind_2 = new Kind("Peter", "Schmidt", LocalDate.now());
        var kind_3 = new Kind("Hans", "Dampf", LocalDate.now());
        var kind_4 = new Kind("Franz", "Müller", LocalDate.now());
        var kind_5 = new Kind("Peter", "Berger", LocalDate.now());
        var kind_6 = new Kind("Heinz", "Kirch", LocalDate.now());
    }
}

```

In Deutschland wurden nach Angaben des Statistischen Bundesamtes 2019 knapp 780.000 Kinder geboren. Gehen wir davon aus, dass die Geburten sich gleichmäßig über Jahr verteilen, dann haben über 2.100 Menschen am gleichen Tag Geburtstag. Nach Angaben der Gesellschaft für deutsche Sprache e.V. hatten 2019 ca. 2.52 % der neugeborenen Mädchen den Vornamen Marie, was also pro Tag etwa 54mal den gleichen neu zu vergebenden Vornamen ergäbe. Und auch eine Gleichheit von Nachnamen ist bei Neugeborenen an einem einzigen Tag sicherlich wahrscheinlich (Müller, Schmidt, Schneider, Fischer,).

Das Vorgehen von oben hätte also zur Folge, dass Sie über 2100 identische Date-Objekte haben; außerdem würden bei der bisherigen Programmierung tausende Strings mit gleichen Vornamen und gleichem Nachnamen angelegt. Und wir reden hier nur von einem Geburtsjahrgang. Für die Gesamtbevölkerung in Deutschland müssten Sie jeweils über 83 Millionen Vor- und Nachnamen einzeln verwalten.

Im oben gezeigten einfachen Beispielcode ist der Compiler intelligent genug, das zu erkennen und zu optimieren, aber wenn die Namen aus externen Quellen gelesen werden müssen, kann das der Compiler vorab nicht feststellen. Es würde nicht lange dauern, bis das Programm den verfügbaren Speicher sprengt.

Das Flyweight Pattern beschreibt nun, wie Objekte kleinster Granularität wiederverwendet werden können. Lassen Sie uns im nächsten Abschnitt betrachten, was damit gemeint ist.

13.2 Die Realisierung

Nehmen Sie noch mal als Negativbeispiel das Projekt Geburten. Gehen wir der Einfachheit halber davon aus, dass der Großteil der Kinder entweder Modenamen oder traditionelle Namen bekommt. Die Anzahl unterschiedlicher Vornamen wird bei den 780.000 Kindern im Verhältnis sehr gering sein. Ich unterstelle, dass es 200 unterschiedliche Vornamen gibt, die 2019 vergeben wurden. Es wäre sinnvoll, wenn nicht 780.000 String-Objekte angelegt werden, die ohnehin fast alle gleich sind. Sinnvoller wäre es, 200 unterschiedliche String-Objekte anzulegen. Kinder mit gleichem Vornamen können sich ein String-Objekt ja teilen.

Das Beispielprojekt Geburten_Flyweight demonstriert dies. Zunächst fällt auf, dass die KindObjekte nicht vom Client direkt aufgerufen bzw. erzeugt werden. Der Client ruft auf einer Fabrik die Methode `getKind()` auf und übergibt dieser Methode die Daten des Kindes.

```
public static void main(String[] args) {  
    var fabrik = new Fabrik();  
    var kind_1 = fabrik.getKind("Hans", "Kirch", new LocalDate());  
    var kind_2 = fabrik.getKind("Peter", "Schmidt", new LocalDate());  
    var kind_3 = fabrik.getKind("Hans", "Dampf", new LocalDate());  
    var kind_4 = fabrik.getKind("Franz", "Müller", new LocalDate());  
    var kind_5 = fabrik.getKind("Peter", "Berger", new LocalDate());  
    var kind_6 = fabrik.getKind("Heinz", "Kirch", new LocalDate());  
    fabrik.auswerten();  
}
```

Die Fabrik führt eine HashSet, in der alle Vornamen gespeichert werden. Wenn ein Kind angelegt werden soll, fügt der Aufruf der add-Methode den Namen nur dann dem HashSet hinzu, wenn er nicht schon enthalten ist. Mit den Nachnamen und den Geburtstagen wird genauso gearbeitet.

```
public class Fabrik {  
    private final HashSet<String> vornamenMenge = new HashSet<>();  
    private final HashSet<String> nachnamenMenge = new HashSet<>();  
    private final HashSet<LocalDate> datumsMenge = new HashSet<>();  
  
    Kind getKind(String vorname, String nachname, Date geburtstag) {  
        vornamenMenge.add(vorname);
```

```

nachnamenMenge.add(nachname);
datumsMenge.add(geburtstag);

return new Kind(vorname, nachname, geburtstag);
}

}

```

An der Kind-Klasse selbst ändert sich nichts. Die Überprüfung, ob ein Eintrag bereits in der Menge vorhanden ist, übernimmt hier die add-Methode des HashSet, so dass wir uns darum gar nicht kümmern müssen. Das macht den Code sehr einfach.

Der Vorteil dieses Vorgehens ist, dass Sie eine große Anzahl gleicher Objekte reduziert haben, indem Sie die Objekte geteilt haben.

Wenn Objekte geteilt werden, müssen Sie kritisch prüfen, ob sie Sie mutable oder besser immutable definieren. Wenn ein Kind von Peter nach Paul umbenannt wird, hätte die Änderung des Namens Auswirkung auf alle Geburten, die darauf verweisen. Da Strings jedoch immutable sind, erübriggt sich dieses Problem hier. Beim Geburtstag wäre die Situation problematischer; ein Date-Objekt könnte mit `datum.setTime()` auf einen anderen Wert gesetzt werden. Somit hätten alle Kind-Objekte, die das Datum referenzieren, auf einmal einen anderen Geburtstag.

Dieses Vorgehen ist besonders dann interessant, wenn diese geteilten Objekte entweder sehr groß und/oder aufwendig zu erstellen sind, beispielsweise durch eine Datenbankabfrage.

Dieses Beispiel ist ausgesprochen einfach. Im folgenden Abschnitt werden Sie noch ein wenig tiefer in die Materie eintauchen.

13.3 Ein komplexeres Projekt

Lassen Sie uns jetzt ein Beispiel betrachten, das etwas komplexer ist. Sie programmieren eine Software, die Bestellungen für eine Pizzeria aufnimmt.

13.3.1 Der erste Ansatz

Ihr erster Ansatz könnte vielleicht so aussehen, dass Sie eine Klasse Pizzabestellung definieren, die den Namen der Pizza und den Tisch, an den sie geliefert werden soll, speichert. Schauen Sie in das Beispielprojekt Pizza.

```

public class Pizzabestellung {
    public final String name;
    public final int tisch;

    public Pizzabestellung(String name, int tisch) {
        this.tisch = tisch;
    }
}

```

```
        this.name = name;
        System.out.println("Ich backe jetzt die " + name);
    }
}
```

Der Client legt neue Bestell-Objekte an und speichert sie in einer Liste. Wenn jeder Tisch eine Bestellung aufgegeben hat, werden die Pizzen serviert.

```
public class ApplStart {
    private static final List<Pizzabestellung> bestellungen =
        new LinkedList<>();

    private static void bestellungAufnehmen(int tisch, String pizza) {
        bestellungen.add(new Pizza(pizza, tisch));
    }

    public static void main(String[] args) {
        bestellungAufnehmen(1, "Pizza Hawai");
        bestellungAufnehmen(2, "Pizza Funghi");
        bestellungAufnehmen(3, "Pizza Carbonara");

        // ... gekürzt

        bestellungen.forEach(pizza -> {
            System.out.
            println("Serviere die " + pizza.name + " an Tisch " + pizza.tisch);
        });
    }
}
```

Wenn Sie den Code analysieren, sehen Sie, dass jede Pizza mehrfach bestellt wird. Natürlich ist es in Einzelfällen auch möglich, dass an einem Tisch zwei oder mehr Bestellungen aufgegeben werden. Welches Attribut werden Sie teilen – die Pizza oder die Tischnummer? Es ist sehr aufwendig, eine Pizza zu backen. Da Sie das Ziel haben, Ressourcen in Ihrer Bestellsoftware zu sparen, werden Sie also die Pizza teilen – aber Achtung: natürlich nur in der Software, jeder Gast bekommt natürlich am Ende die ganze Pizza seiner Wahl. Die Nummer des Tisches, von dem die Bestellung kommt, wird nicht geteilt – dafür wird gleich der Client verantwortlich gemacht. Sie haben also zwei unterschiedliche Zustände: den Zustand, der geteilt wird; und den, für den der Client verantwortlich gemacht wird. **Den geteilten Zustand nennen Sie intrinsisch; den anderen extrinsisch.** Objekte, die den intrinsischen Zustand definieren, sind die Fliegengewichte oder auch Flyweights. Sie sind von dem Kontext, in dem sie verwendet werden, unabhängig.

13.3.2 Intrinsischer und extrinsischer Zustand

Die Klasse Pizzabestellung hat zwei Attribute: die Beschreibung der Pizza und die Nummer des Tisches, an den sie geliefert werden soll. Sie zerlegen die Pizzabestellung-Klasse in einen intrinsischen und einen extrinsischen Zustand, wobei die Beschreibung der Pizza selbst der intrinsische Zustand sein soll. Sie ziehen also das andere Attribut aus der Klasse heraus und verlagern es in den Kontext. Übrig bleibt in der Pizza-Klasse nur das, was nötig ist, um die Pizza zu beschreiben. Es handelt sich hier nicht um die gebackene Pizza selbst, sondern um die Notiz, dass sie bestellt wurde. Im Beispielprojekt PizzaFlyweight soll es genügen, die Pizza durch ihren Namen zu definieren.

```
public class Pizza implements MenueEintrag {
    public final String name;

    public Pizza(String name) {
        this.name = name;
    }

    // ... gekürzt
}
```

Um später weitere Gerichte (Salat, Pasta, ...) zu ermöglichen, habe ich in diesem Zusammenhang gleich das Interface `MenueEintrag` eingeführt, das von der Klasse `Pizza` implementiert wird. Ich werde gleich noch genauer darauf eingehen; lassen Sie uns vorher den Client-Code betrachten.

Der Client verwaltet die unterschiedlichen Pizza-Objekte und die Tischnummern. Um eine eindeutige Zuordnung zu ermöglichen, werden die Bestellungen in einer Map gespeichert; der Key dieser Map ist die Tischnummer, der Value sind die bestellten Menüs. Um die Situation abzubilden, dass ein Tisch mehrere Bestellungen aufgeben kann, sind die Menüs als Arrays hinterlegt.

```
public class ApplStart {
    private static final Map<Integer, MenueEintrag[]> BESTELLUNGEN = new
    HashMap<>();
    private static final MenueFabrik MENUE_FABRIK = new MenueFabrik();

    public static void bestellungAufnehmen(int tisch, String... menue) {
        var bestellung = MENUE_FABRIK.getMenue(menue);
        BESTELLUNGEN.put(tisch, bestellung);
    }

    public static void main(String[] args) {
        bestellungAufnehmen(1, "Pizza Hawai");
        bestellungAufnehmen(2, "Pizza Funghi");
        bestellungAufnehmen(3, "Pizza Carbonara");
    }
}
```

```
bestellungAufnehmen(4, "Pizza Calzone", "Pizza Carbonara");
// ... gekürzt
}
```

Der Code der Fabrik, die eine Übersicht aller bestellten Varianten hält, ist nahezu identisch mit dem Code, den Sie bereits vom Beispielprojekt „Geburten“ kennen. Ich möchte nicht weiter darauf eingehen.

Wie wird die Pizza denn nun an den Tisch serviert? Das Interface Menue schreibt die Methode `serviere()` vor, der Sie den Tisch als Parameter übergeben. Die GoF beschreibt das mit den Worten: „*Hängen Operationen von extrinsischem Zustand ab, so erhalten sie ihn als Parameter.*“

```
public interface MenueEintrag {
    void serviere(int tisch);
}
```

Nun soll der Rest der Pizza-Klasse, die Methode `serviere()`, vorgestellt werden. Sie kann sich darauf beschränken, die Meldung auf der Konsole auszugeben, dass die Pizza an einen bestimmten Tisch serviert wurde.

```
public class Pizza implements MenueEintrag {
    // ... gekürzt

    @Override
    public void serviere(int tisch) {
        System.out.
        println("Die " + name + " wird an den Tisch Nr. " + tisch + " ge-
bracht.");
    }
}
```

Wenn der Client alle Bestellungen aufgenommen hat, geht er Tisch für Tisch die Bestellungen durch und serviert die Pizzen.

```
public class ApplStart {
    // ... gekürzt

    public static void main(String[] args) {
        // ... gekürzt
        bestellungAufnehmen(79, "Pizza Funghi");

        BESTELLUNGEN.keySet().forEach(tempTisch -> {
            var menues = BESTELLUNGEN.get(tempTisch);
            for (var tempMenue : menues)
```

```

        tempMenue.serviere(tempTisch);
    });
}
}

```

Im folgenden Absatz sehen Sie, wo das Fliegengewicht in der Praxis zum Einsatz kommt.

13.4 Flyweight in der Praxis

Wo finden Sie das Flyweight Pattern in freier Wildbahn? Wenn Sie ein Integer-Objekt mit dem new-Operator anlegen, vergleichen Sie die Referenzen der Objekte mit == auf Gleichheit. Führen Sie folgenden Code in einer Java-Shell aus:

```

Integer wert_1 = new Integer(1);
Integer wert_2 = new Integer(1);
boolean gleichheit = wert_1 == wert_2;
System.out.println("Objekte sind gleich: " + gleichheit);

```

Auf der Konsole wird ausgegeben: Objekte sind gleich: false

Legen Sie die Objekte testweise aber mal mit Integer.valueOf() an:

Jetzt wird auf der Konsole ausgegeben: Objekte sind gleich: true

Offensichtlich speichert die Klasse Integer die Werte im Sinne des Flyweight Patterns -zumindest, wenn die Objekte mittels valueOf angelegt werden. Und das ist auch die empfohlene Methode. Der Konstruktor Integer(int) ist seit Java 9 als „deprecated“, also überholt, markiert. Er wird in künftigen Java-Versionen irgendwann nicht mehr verfügbar sein.

Betrifft das alle Zahlen im Wertebereich von Integer? Testen wir das einfach mal!

```

for (int i = -130; i < 130; i++) {
    Integer wert_1 = Integer.valueOf(i);
    Integer wert_2 = Integer.valueOf(i);
    boolean gleichheit = wert_1 == wert_2;
    System.out.println(i + ": Objekte sind gleich: " + gleichheit);
}

```

Wenn Sie diesen Code ausführen, sehen Sie, dass alle Zahlen von – 128 bis + 127 geteilt werden:

```

-130: Objekte sind gleich: false
-129: Objekte sind gleich: false
-128: Objekte sind gleich: true
-127: Objekte sind gleich: true
...
-2: Objekte sind gleich: true
-1: Objekte sind gleich: true
0: Objekte sind gleich: true

```

```

1: Objekte sind gleich: true
2: Objekte sind gleich: true
...
126: Objekte sind gleich: true
127: Objekte sind gleich: true
128: Objekte sind gleich: false
129: Objekte sind gleich: false

```

Was passiert, wenn Sie Objekte via AutoBoxing, also dem automatischen Umwandeln von einfachen Datentypen in die objektorientierten Typen, anlegen? Machen Sie den Test:

```

Integer wert_1 = 3;
Integer wert_2 = 3;
boolean gleichheit = wert_1 == wert_2;
System.out.println("Objekte sind bei AutoBoxing gleich: " + gleichheit);

```

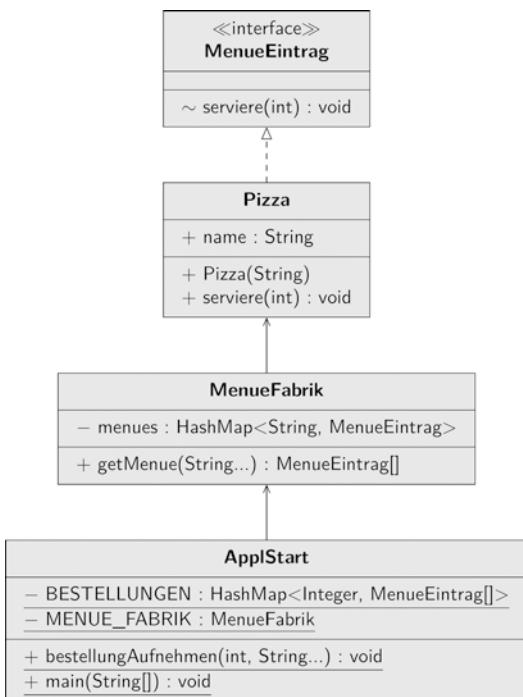
Auf der Konsole wird ausgegeben: Objekte sind bei AutoBoxing gleich: true

Sie können also mit Autoboxing oder valueOf arbeiten.

13.5 Flyweight – Das UML-Diagramm

Das UML-Diagramm aus dem Beispielprojekt PizzaFlyweight finden Sie in Abb. 13.1.

Abb. 13.1 UML-Diagramm des Flyweight Pattern
(Beispielprojekt PizzaFlyweight)



13.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sie haben eine Vielzahl von Objekten, die entweder groß sind, aufwendig zu erstellen sind oder einfach aufgrund ihrer Anzahl viel Systemressourcen verbrauchen.
- Diese Objekte zerlegen Sie in einen Teil, der geteilt werden kann, und einen Teil, der nicht geteilt wird.
- Die Attribute, die geteilt werden können, bilden den intrinsischen Zustand; sie sind Fliegengewichte.
- Der extrinsische Zustand wird in den Kontext verlagert.
- Der Kontext verwaltet den intrinsischen und den extrinsischen Zustand.
- Operationen werden ausgeführt, indem der Kontext den extrinsischen Zustand als Parameter an eine Methode des intrinsischen Zustands übergibt.
- Der intrinsische Zustand ist unabhängig von seinem Kontext.

Zweckbeschreibung

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

„Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können.“



Nach dem ersten Ausflug zu den Strukturmustern Composite und Flyweight machen wir wieder mit einem Verhaltensmuster weiter, dem Interpreter Pattern. Das Muster ist nicht wirklich schwierig zu verstehen, eröffnet aber tolle Möglichkeiten.

14.1 Die Aufgabe in diesem Kapitel

Ich möchte Ihnen gleich zu Beginn einen Eindruck davon geben, wo die Reise hingehen wird. Der Screenshot in Abbildung Abb. 14.1 zeigt das Programm aus dem Beispielprojekt Interpreter, das wir jetzt besprechen werden, in Aktion. Das Prinzip ist sehr einfach: Sie geben einen mathematischen Ausdruck in das Eingabefeld ein. Er kann beliebig lang und beliebig geschachtelt sein; die Eingabe schließen Sie mit einem Semikolon ab. Verarbeitet werden Vorzeichen-Plus und Vorzeichen-Minus, das Pluszeichen, das Minuszeichen, das Mal- und das Geteilt-Zeichen. Darüber hinaus können Sie runde Klammern setzen, um Prioritäten festzulegen. Eckige Klammern zeigen an, dass das Ergebnis als absoluter Wert weiterverarbeitet wird. Sie können beliebig viele Leerzeichen einfügen. Wenn Sie auf „**Ab geht's**“ klicken, wird die Eingabe gescannt, geparsst und berechnet.

Das Eingabefeld ist eine editierbare Combobox, in der bereits ein paar Ausdrücke hinterlegt sind. Der Aufbau der GUI soll hier nicht besprochen werden; der Code ist sehr einfach. Wenn Sie auf „**Ab geht's**“ klicken, wird von der Combobox der eingegebene Wert abgefragt. Dieser Wert wird an einen Scanner übergeben, der eine Liste von Symbolen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_14

Abb. 14.1 Ein Taschenrechner in Aktion, Projekt „Interpreter“



zurückgibt. Diese Liste schicken Sie an den Parser, der die Wurzel eines abstrakten Syntaxbaums erstellt. Auf dieser Wurzel rufen Sie die Methode `rechne()` auf und erhalten das berechnete Ergebnis.

```
final String eingabe = (String) cmbAusdruck.getSelectedItem();
try {
    var scanner = new Scanner(eingabe);
    edtScan.setText(scanner.toString());
    final var symbole = scanner.getSymbols();
    var parser = new Parser(symbole);
    var result = parser.getRoot();
    System.out.println(result);
    final var ergebnis = result.rechne();
    edtErgebnis.setText(Double.toString(ergebnis));
} catch (IllegalArgumentException ex) {
    JOptionPane.showMessageDialog(null, ex.getMessage());
}
```

Dieses Kapitel beschäftigt sich mit der Frage, wie Sie einen Ausdruck, den ein Anwender eingibt, so aufbereiten, dass ein Computer damit umgehen kann. Dabei werden die Werkzeuge des Interpreter Patterns, also der abstrakte Syntaxbaum, der Scanner und der Parser, besprochen.

Das Thema Compilerbau ist an der Uni Gegenstand einer eigenständigen Vorlesung im Rahmen eines Informatikstudiums. Fachbücher hierzu umfassen viele hundert Seiten. Die Implementierung, die ich Ihnen vorstellen werde, kann also nur ein Weg von vielen sein. Ich habe sie mit der Motivation entwickelt, einen verständlichen Einstieg anzubieten; Fragen nach der Effizienz habe ich außen vorgelassen.

14.2 Der Scanner

Der erste Verarbeitungsschritt wird vom Scanner durchgeführt. Er führt die lexikalische Analyse durch. Dabei wird die Eingabe in Sinneinheiten, die Tokens, oder auch Symbole, zerlegt. Symbole sind in diesem Beispiel Zahlen, Leerzeichen, Semikolon, die Klammern und die mathematischen Operatoren. Nehmen Sie beispielsweise an, Sie geben $5 + 14;$ ein. Der Scanner erkennt folgende Symbole:

```
Zahl ( 5 ) Leerzeichen ( ) Pluszeichen( + ) Leerzeichen ( )
Zahl( 14 ) Semikolon( ; )
```

Die Leerzeichen und das Semikolon möchte ich vom Scanner ausfiltern lassen. Der Scanner gibt also folgende Symbole zurück:

```
Zahl ( 5 ) Pluszeichen ( + ) Zahl ( 4 )
```

Es ist wichtig zu sehen, dass der Scanner noch keine syntaktische Prüfung vornimmt; er kann nicht erkennen, ob eine geöffnete Klammer auch wieder geschlossen wurde. Lassen Sie uns im folgenden Absatz betrachten, wie der Scanner einen String in Symbole zerlegt.

14.2.1 Die definierten Symbole

Im Package `symbole` legen Sie zunächst die Oberklasse `Symbol` an. Von ihr werden Symbolklassen abgeleitet, die die mathematischen Operatoren, die Klammern und die Zahlen repräsentieren. Betrachten Sie das Symbol `Plus` – es muss nicht mehr machen, als über sich selbst zu sagen „Ich bin ein Pluszeichen“.

```
public class Plus extends Symbol {
    @Override
    public boolean isPlus() {
        return true;
    }
}
```

Eine Zahl wird durch das `Numeral`-Symbol repräsentiert. Es muss darüber Auskunft geben, dass es ein Numeral ist. Außerdem speichert es den Wert, den es repräsentiert.

```
public class Numeral extends Symbol {
    final double VALUE;

    public Numeral(String zahl) {
```

```

        VALUE = Double.valueOf(zahl);
    }

    public Numeral(double zahl) {
        VALUE = zahl;
    }

    @Override
    public boolean isNumeral() {
        return true;
    }
}

```

Wenn die Grammatik die Eingabe von Strings erlauben würde, müssten Sie entsprechend dem Numeral-Symbol ein Literal-Symbol einführen.

Die Implementierung der `toString`-Methoden habe ich jeweils nicht mit abgedruckt. Sie erkennen, dass die Methoden `isNumeral()` und `isPlus()` jeweils eine Methode der Oberklasse `Symbol` überschreiben. In `Symbol` wird für alle abgeleiteten Symbole die `is`-Methode standardmäßig verneint.

```

public abstract class Symbol {
    // ... gekürzt

    public boolean isNumeral() {
        return false;
    }

    public boolean isPlus() {
        return false;
    }
}

```

Sie können also jedes Symbol fragen, ob es ein Pluszeichen ist, aber nur das Pluszeichen wird tatsächlich `true` zurückgeben. Im folgenden Absatz sehen Sie, wie der Scanner die Symbole erzeugt.

14.2.2 Der Scanner wandelt Strings in Symbole um

In der Klasse `Scanner` wird gemappt, welcher Character auf welches Symbol abgebildet wird. Außerdem wird eine Liste von Symbolen definiert, in der alle Symbole gespeichert werden; diese Liste wird vom Scanner später returniert.

```

public class Scanner {
    private final List<Symbol> symbole = new ArrayList<>();
    private final Map<Character, Symbol> operatoren = new HashMap<>();

    private Scanner() {
        operatoren.put('+', new Plus());
        operatoren.put('-', new Minus());
        operatoren.put('*', new Mal());
        operatoren.put('/', new Durch());
        operatoren.put('[', new AbsolutAnfang());
        operatoren.put(']', new AbsolutEnde());
        operatoren.put('{', new KlammerAuf());
        operatoren.put('}', new KlammerZu());
        operatoren.put('=', new Gleich());
    }

    // ... gekürzt
}

```

Der Konstruktor ist überladen. Sie übergeben ihm den eingegebenen String. Er entfernt die Leerzeichen und wirft eine Exception, wenn die Eingabe nicht mit einem Semikolon beendet wird. Außerdem prüft der Scanner, ob es mehr als ein Semikolon in der Eingabe gibt. Anschließend geht er Zeichen für Zeichen durch den String und versucht, die Zeichen in Symbole umzusetzen.

```

public Scanner(String eingabe) throws IllegalArgumentException {
    this();
    eingabe = eingabe.replaceAll(" ", "");
    var laenge = eingabe.length();
    var erstesSemikolon = eingabe.indexOf(';');
    if (erstesSemikolon != laenge - 1)
        throw new IllegalArgumentException(" ... ");
    var letztesZeichen = eingabe.charAt(laenge - 1);
    if (letztesZeichen != ';')
        throw new IllegalArgumentException(" ... ");
    else
        for (var i = 0; i < eingabe.length(); i++) {
            // ... gekürzt
        }
}

```

Betrachten Sie folgend die Arbeit innerhalb der for-Schleife. Das indizierte Zeichen wird aus dem String extrahiert. Da das Semikolon nicht berücksichtigt wird, muss der Scanner nur tätig werden, wenn das aktuelle Zeichen **kein** Semikolon ist. Der Scanner

prüft zuerst, ob das aktuell indizierte Zeichen in der Liste der Operatoren enthalten ist. In diesem Fall wird das entsprechende Symbol-Objekt ausgelesen und in der Ergebnisliste gespeichert. Danach prüft der Scanner, ob das aktuelle Zeichen ein Buchstabe, ein Literal, ist. Einen Buchstaben werden Sie brauchen, wenn Sie das Projekt weiterentwickeln und Variablen einführen oder eine Zuweisung abbilden wollen: $a = 5 + 7$.

Sobald der Scanner feststellt, dass das aktuelle Zeichen eine Zahl ist, legt er einen Zwischenspeicher an. Er liest in einer while-Schleife so lange das jeweils nächste Zeichen in den Zwischenspeicher ein, bis das folgende Zeichen keine Zahl mehr ist. Sofern das nächste aktuelle Zeichen ein Punkt oder ein Komma ist, wird es ebenfalls in den Zwischenspeicher kopiert – es ist davon auszugehen, dass der Anwender eine Double-Zahl eingegeben hat. Sofern es Nachkommazahlen gibt, werden diese ebenfalls in einer while-Schleife abgefragt und an den Zwischenspeicher übergeben. Der Zwischenspeicher wird schließlich verwendet, um ein Numeral-Symbol zu erzeugen, das in der Ergebnisliste gespeichert wird. Jetzt beginnt die Arbeit wieder am Anfang der for-Schleife mit dem nächsten aktuellen Zeichen. Hier der Code in der for-Schleife, den ich oben noch rausgekürzt hatte:

```
Character zeichen = eingabe.charAt(i);
if (!zeichen.equals(';'))
    if (operatoren.containsKey(zeichen))
        symbole.add(operatoren.get(zeichen));
    else if (Character.isLetter(zeichen))
        symbole.add(new Literal(zeichen));
    else {
        var zahlenBuilder = new StringBuilder();
        while (Character.isDigit(zeichen)) {
            zahlenBuilder.append(zeichen);
            zeichen = eingabe.charAt(++i);
        }
        if (zeichen == '.' || zeichen == ',') {
            zeichen = '.';
            zahlenBuilder.append(zeichen);
            i++;
            zeichen = eingabe.charAt(i);
            while (Character.isDigit(zeichen)) {
                zahlenBuilder.append(zeichen);
                zeichen = eingabe.charAt(++i);
            }
        }
        symbole.add(new Numeral(zahlenBuilder.toString()));
        i--;
    }
}
```

Damit ist die Arbeit des Scanners erledigt. Er überschreibt die `toString`-Methode und definiert die Methode `getSymbols()`, die die Ergebnisliste returniert.

14.3 Der Parser

Wenn Sie den String $2 + (6 + 4) * 4;$ eingeben, wird der Scanner Ihnen eine Reihe von Symbolen zurückgeben. Diese Symbole sind noch nicht daraufhin überprüft, ob sie syntaktisch korrekt sind, also ob beispielsweise alle Klammern geschlossen wurden. Den Symbolen wurde außerdem noch keine Bedeutung zugewiesen. Sowohl syntaktische als auch semantische Analyse sind Aufgabe des Parsers.

Die GoF setzt bei der Beschreibung des Interpreter Patterns voraus, dass der Ausdruck bereits geparst ist. Der Interpreter arbeitet auf dem geparsten abstrakten Syntaxbaum. Dennoch beschreibe ich in diesem Abschnitt den Parser, um Ihnen die Werkzeuge des Interpreters, die Expressions, vorzustellen.

14.3.1 Abstrakte Syntaxbäume

Scanner und Parser haben die Aufgabe, einen abstrakten Syntaxbaum zu erstellen. Was ist damit gemeint? Ein Syntaxbaum hat die Aufgabe, einen Satz in der definierten Grammatik als Baumstruktur darzustellen. Würden Sie alle Leerzeichen und das Semikolon in den Baum mit aufnehmen, hätten Sie einen *konkreten Syntaxbaum*. Ein konkreter Syntaxbaum ist nicht effizient zu verwalten. Daher gehen Sie, um einen abstrakten Syntaxbaum zu generieren, einen Schritt weiter und entfernen alle Symbole, die für die Ausführung nicht erforderlich sind.

Die Leerzeichen und das Semikolon werden bereits vom Scanner entfernt. Was ist mit Operatoren und Numeralen? Jeder Operator und jedes Numeral werden durch Objekte repräsentiert, die die Knoten des Syntaxbaums darstellen. Die Kanten des Baums beschreiben die Zusammensetzung der Knoten. Nehmen Sie das Beispiel von oben; der Anwender gibt ein: $5 + 4;$. Der Scanner entfernt die Leerzeichen und das Semikolon und returniert Zahl (5) Pluszeichen (+) Zahl (4). Die Grafik in Abb. 14.2 veranschaulicht, wie der Ausdruck in einen Syntaxbaum umgewandelt wird, dessen Knoten die Operatoren und Numeralen sind.

Die Priorität der Punktrechnung vor der Strichrechnung lässt sich durch die Stellung im Syntaxbaum berücksichtigen. Die Eingabe $10 + 5 * 4;$ würde den Syntaxbaum in Abb. 14.3 produzieren.

Klammern können die Priorität ändern. Die geänderte Priorität wird im Syntaxbaum berücksichtigt, so dass die Klammern entfernt werden können. Der Benutzer gibt beispielsweise ein: $(10 + 5) * 4;$. Diese Eingabe lässt sich durch den Syntaxbaum in Abb. 14.4 darstellen

Abb. 14.2 Abstrakter Syntaxbaum des Ausdrucks $5 + 10$

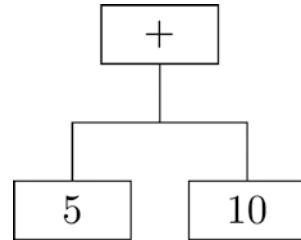


Abb. 14.3 Abstrakter Syntaxbaum des Ausdrucks $10 + 5 * 4$

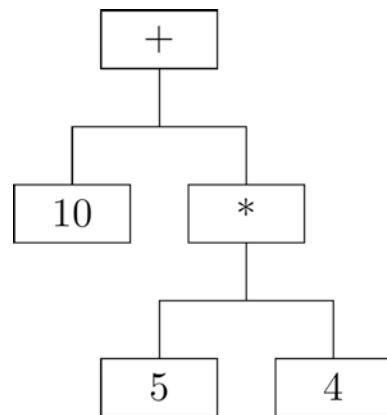
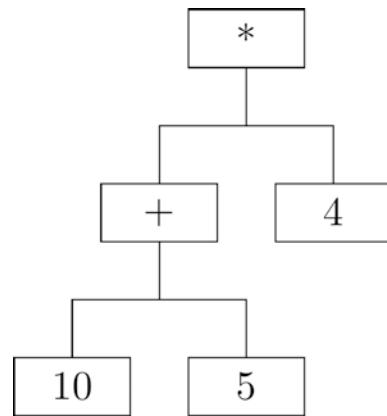


Abb. 14.4 Abstrakter Syntaxbaum des Ausdrucks $(10 + 5) * 4$



Im nächsten Abschnitt schauen wir uns an, wie Sie eine Folge von Symbolen in einen Syntaxbaum umwandeln.

14.3.2 Expressions für den Parser

Operatoren und Numerale sind Knoten des Syntaxbaums. Um sie gleich behandeln zu können, ist es erforderlich, dass beide vom gleichen Typ sind. Fassen Sie Numerale und Operatoren unter der Schnittstelle Expression zusammen, die die Methode `rechne()` vorschreibt.

```
public interface Expression {  
    double rechne();  
}
```

Ein Numeral referenziert keine weiteren Expressions. Es darf sich darauf beschränken, den Wert zu speichern und zurückzugeben, wenn `rechne()` aufgerufen wird.

```
public final class TerminalExpression implements Expression {  
    private final double value;  
  
    public TerminalExpression(double value) {  
        this.value = value;  
    }  
  
    @Override  
    public double rechne() {  
        return value;  
    }  
}
```

Operatoren arbeiten auf zwei Expressions. Sie sind nichtterminal und werden unter der abstrakten Klasse `NichtTerminalExpression` zusammengefasst.

```
public abstract class NichtTerminalExpression implements Expression {  
    protected final Expression links;  
    protected final Expression rechts;  
  
    protected NichtTerminalExpression(Expression links, Expression rechts) {  
        this.links = links;  
        this.rechts = rechts;  
    }  
}
```

Um zwei Zahlen addieren zu können, leiten Sie von dieser Klasse die Klasse `PlusExpression` ab. Ihr übergeben Sie die beiden Expressions, die addiert werden sollen. Die Methode `rechne()` ist so implementiert, dass die `PlusExpression` die beiden Expressi-

ons auffordert, ihre Werte zurückzugeben. Diese Werte werden addiert und die Summe wird zurückgegeben.

```
public class PlusExpression extends NichtTerminalExpression {
    public PlusExpression(Expression links, Expression rechts) {
        super(links, rechts);
    }

    @Override
    public double rechne() {
        return links.rechne() + rechts.rechne();
    }
}
```

Ihre Grammatik soll sich am Anfang darauf beschränken, dass zwei oder mehr Zahlen addiert werden können.

14.3.3 Strichrechnung parsen

Das Beispielprojekt Interpreter_1 ist eine abgespeckte Version des Projekts Interpreter. Der hier hinterlegte Parser beschränkt sich darauf, beliebig viele Zahlen zu addieren. Lassen Sie uns das Vorgehen ansehen. Dem Konstruktor übergeben Sie die vom Scanner generierte Liste von Symbolen. Der Parser lässt sich hiervon den Iterator geben. Die Methode `nextSymbol()` ermittelt aus dieser Liste das jeweils nächste Symbol, das im Attribut `currentSymbol` gespeichert wird. Wenn es keine weiteren Symbole gibt, wird an `currentSymbol` ein `EndSymbol` übergeben. Der Konstruktor übergibt an das Feld `root` eine Expression, die von der Methode `parseExpression()` zurückgegeben wird. Der Client kann `root` mit `getRoot()` abfragen.

Wir verwenden hier also tatsächlich schon mehrere Muster in Kombination: Den Iterator, den Sie in Kap. 11 gesehen haben, und das aus Kap. 12 bekannte Composite. Schlagen Sie dort gerne auch noch mal nach und versuchen, hier diese Strukturen wiederzufinden.

```
public final class Parser {
    private final Iterator<Symbol> iterator;
    private final Expression root;
    private Symbol currentSymbol;

    public Parser(List<Symbol> symbole) {
        iterator = symbole.iterator();
        nextSymbol();
        root = parseExpression();
    }
}
```

```

    private void nextSymbol() {
        currentSymbol = iterator.hasNext() ? iterator.next() :
            new EndSymbol();
    }

    // ... gekürzt
}

```

Die Methode `parseExpression()` ist der Einstiegsplatz für den Parser. Sie ruft die Methode `parseAddition()` auf, die ein Objekt vom Typ `Expression` zurückgibt; dieses Objekt wird zurückgegeben.

```

private Expression parseExpression() {
    return parseAddition();
}

```

Die Methode `parseAddition()` fragt im ersten Schritt den ersten Summanden ab und ruft dazu die Methode `parseNumber()` auf. Wenn `parseNumber()` feststellt, dass `currentSymbol` keine Zahl ist, wirft die Methode eine Exception, ansonsten wird der Wert von `currentSymbol` an eine `TerminalExpression` übergeben und returniert. Vorher wird `currentSymbol` auf das nächste Symbol in der Liste der Symbole aktualisiert. Die Variable `expression` in `parseAddition()` referenziert nun eine `TerminalExpression`. Die Methode `parseAddition()` prüft jetzt in einer Schleife, ob `currentSymbol` ein Pluszeichen ist. Ist dies der Fall, wird das Zeichen nicht mehr benötigt, an `currentSymbol` kann das nächste Symbol aus der Liste der Symbole übergeben werden. Von `parseNumber()` wird die nächste Zahl, eine `TerminalExpression`, abgefragt. Mit den beiden Expressions wird eine `PlusExpression` erzeugt und an `expression` übergeben. Die Variable `expression` wird schließlich zurückgegeben.

```

private Expression parseAddition() {
    var expression = this.parseNumber();
    while (currentSymbol.isPlus()) {
        nextSymbol();
        var rechts = this.parseNumber();
        expression = new PlusExpression(expression, rechts);
    }
    return expression;
}

private Expression parseNumber() {
    if (!currentSymbol.isNumeral())
        throw new IllegalStateException("Aktuelles Symbol ist keine
Zahl: " + currentSymbol);
    var value = Double.parseDouble(currentSymbol.toString());
}

```

```

    nextSymbol();
    return new TerminalExpression(value);
}

```

Im Beispielprojekt Interpreter_2 habe ich die Methode `addAddition()` in `addStrichrechnung()` umbenannt und die Subtraktion implementiert. Neu in dieser Version ist also auch die Klasse `MinusExpression`.

```

private Expression parseStrichrechnung() {
    var expression = this.parseNumber();
    while (currentSymbol.isPlus() || currentSymbol.isMinus()) {
        var addieren = currentSymbol.isPlus();
        nextSymbol();
        var rechts = this.parseNumber();
        expression = addieren
            ? new PlusExpression(expression, rechts)
            : new MinusExpression(expression, rechts);
    }
    return expression;
}

```

- ▶ **Tipp** Wenn Sie $5 + 7 - 2$; eingeben, erzeugt der Parser eine Reihe von Expressions, die teilweise geschachtelt sind. Versuchen Sie, mit einem Debugger oder einer geeigneten Konsolenausgabe, nachzuvollziehen, dass die Expressions wie folgt geschachtelt sind:

```

MinusExpression (
    PlusExpression (
        Terminal ( 5.0 ) Terminal ( 7.0 )
    )
    Terminal ( 2.0 )
)

```

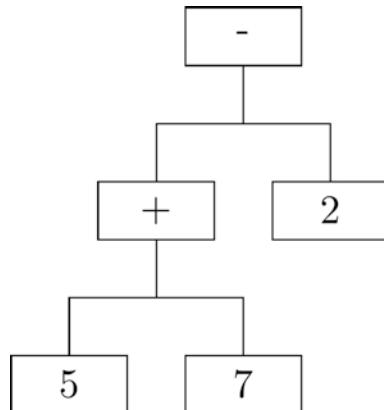
Den abstrakten Syntaxbaum dazu finden Sie in Abb. 14.5.

Im nächsten Abschnitt fügen wir die Punktrechnung ein, die vor der Strichrechnung berücksichtigt werden muss.

14.3.4 Punktrechnung parsen

Die Version im Beispielprojekt Interpreter_3 bezieht die Punktrechnung mit ein. Es kommen zwei Expressions hinzu: die `MalExpression` und die `DurchExpression`. Entsprechend kennt der Parser die Methode `parsePunktrechnung()`, die ähnlich aufgebaut ist wie

Abb. 14.5 Abstrakter Syntaxbaum für den Ausdruck
 $5 + 7 - 2$



`parseStrichrechnung()`. Die Punktrechnung hat Vorrang vor der Strichrechnung. Um diese Priorität herzustellen, muss die Methode `parseStrichrechnung()` erst `parsePunktrechnung()` aufrufen, bevor sie prüft, ob addiert oder subtrahiert werden soll.

```

private Expression parseStrichrechnung() {
    var expression = this.parsePunktrechnung();
    while (currentSymbol.isPlus() || currentSymbol.isMinus()) {
        var addieren = currentSymbol.isPlus();
        nextSymbol();
        var rechts = this.parsePunktrechnung();
        expression = addieren
            ? new PlusExpression(expression, rechts)
            : new MinusExpression(expression, rechts);
    }
    return expression;
}

private Expression parsePunktrechnung() {
    var expression = this.parseNumber();
    while (currentSymbol.isMal() || currentSymbol.isDurch()) {
        var multiplizieren = currentSymbol.isMal();
        nextSymbol();
        var rechts = this.parseNumber();
        expression = multiplizieren
            ? new MalExpression(expression, rechts)
            : new DurchExpression(expression, rechts);
    }
    return expression;
}
  
```

- **Tipp** Versuchen Sie, entweder mit einem Debugger oder mit einer Konsolenausgabe, nachzuvollziehen, dass $5 * 2 + 3$; in folgende Expression geparst wird:

```
PlusExpression {
    MalExpression (
        Terminal ( 5.0 ) Terminal ( 2.0 )
    )
    Terminal ( 3.0 )
}
```

Die Eingabe $5 + 2 * 3$; wird in eine andere Expression geparst:

```
PlusExpression (
    Terminal ( 5.0 )
    MalExpression (
        Terminal ( 2.0 ) Terminal ( 3.0 )
    )
)
```

Kommen wir jetzt zum Beispielprojekt Interpreter_4. Hier werden Sie zunächst ein Vorzeichen-Minus und ein Vorzeichen-Plus berücksichtigen. Ist ein Vorzeichen-Plus sinnvoll? Keine Ahnung – auf jeden Fall ist es aber mathematisch korrekt, und unsere Grammatik soll es erlauben. In der Priorität steht das Vorzeichen noch vor der Punktrechnung. Sie müssen sie also vor Division und Multiplikation prüfen. Die Methode `parseDivision()` muss, wenn sie aufgerufen wird, den Aufruf zuerst an die neue Methode `parseVorzeichen()` weiterleiten. Wenn `parseVorzeichen()` erkennt, dass ein Vorzeichen-Minus gesetzt ist, wird die nächste TerminalExpression in eine UnaerExpression gewrappt. Der Code ist nicht viel komplizierter als die Vorgängerversionen des Projekts; ich möchte ihn daher nicht näher erläutern. Schauen Sie sich einfach die genannten Methoden in der parser-Klasse an.

14.3.5 Klammern berücksichtigen

In diesem Abschnitt werden Klammern eingeführt. Jetzt schließen wir den Kreis und kommen wieder zum Beispielprojekt Interpreter vom Anfang dieses Kapitels. Runde Klammern legen die Priorität einer Rechnung fest; eckige Klammern bewirken, dass der geklammerte Ausdruck positiv ist. Die Klammern haben eine höhere Priorität als das Vorzeichen, also müssen sie vorher geparst werden. Die Methode `parseAbsolut()`

parst einen Ausdruck, der in eckigen Klammern steht, die Methode `parseKlammer()` parst runde Klammern. Die erste Änderung finden Sie in der Methode `parseVorzeichen()`: Wenn um einen Ausdruck eine Klammer gesetzt wird, kann dieser Ausdruck eine NumeralExpression sein oder ein geschachtelter Ausdruck, der aus Addition, Subtraktion, Multiplikation und Division bestehen kann. Eine UnaerExpression darf also nur erstellt werden, wenn das folgende Symbol eine Zahl repräsentiert.

```
private Expression parseVorzeichen() {
    Expression expression;
    var minus = currentSymbol.isMinus();
    if (currentSymbol.isMinus() || currentSymbol.isPlus())
        nextSymbol();
    if (currentSymbol.isNumeral())
        expression = this.parseNumber();
    else
        expression = parseAbsolut();
    if (minus)
        expression = new UnaerExpression(expression);
    return expression;
}
```

Die Methode `parseAbsolut()` legt eine Variable `Expression expression` an und prüft zuerst, ob `currentSymbol` eine öffnende eckige Klammer ist. In diesem Fall ist dieses Symbol „verbraucht“ und muss nicht mehr beachtet werden. Jetzt ruft `parseAbsolut()` die Methode `parseExpression()` auf, die eine Expression zurückgibt. Diese Expression wird in eine AbsolutExpression gewrapped und returniert. Die AbsolutExpression hat die Aufgabe, den Betrag der referenzierten Expression zu ermitteln. Ist in obiger Prüfung das `currentSymbol` keine öffnende eckige Klammer, fragt `parseAbsolut()` die Methode `parseKlammer()` nach einem Expression-Objekt.

```
private Expression parseAbsolut() {
    Expression expression;
    if (currentSymbol.isAbsolutAnfang()) {
        nextSymbol();
        expression = this.parseExpression();
        expression = new AbsolutExpression(expression);
        if (!currentSymbol.isAbsolutEnde())
            throw new RuntimeException("Ende Absolutwert fehlt");
        nextSymbol();
    } else
        expression = parseKlammer();
    return expression;
}
```

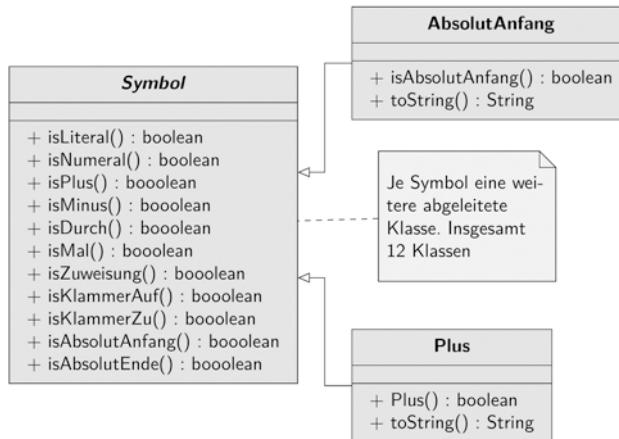


Abb. 14.6 UML-Diagramm des Interpreter Pattern (Beispielprojekt Interpreter_4, package symbole)

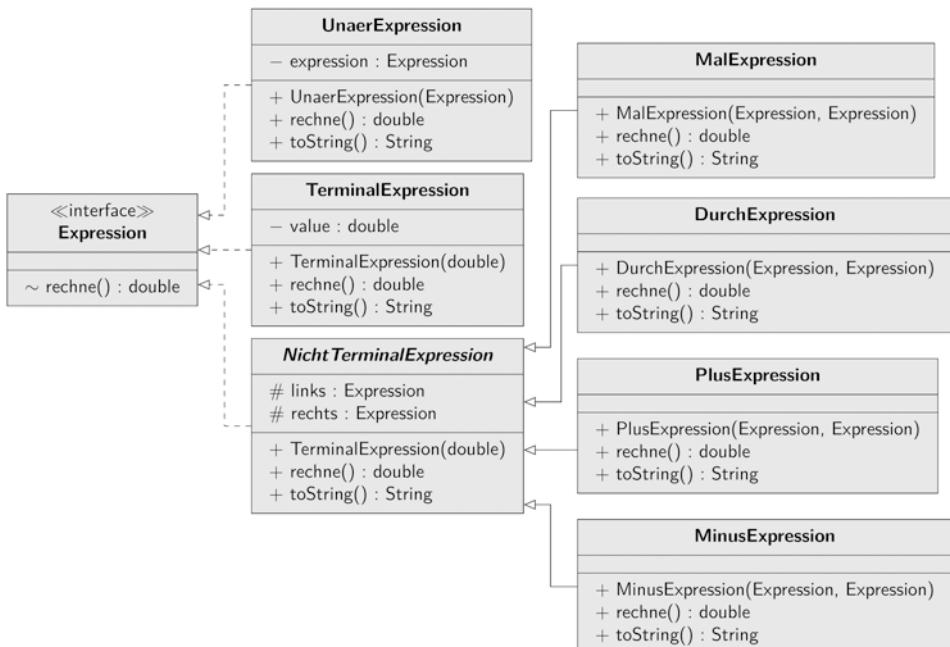


Abb. 14.7 UML-Diagramm des Interpreter Pattern (Beispielprojekt Interpreter_4, package expressions)

Wenn die Eingabe syntaktisch korrekt ist, muss `currentSymbol` beim Aufruf von `parseKlammer()` eine öffnende runde Klammer sein; ist `currentSymbol` ein anderes Zeichen, wirft die Methode eine Exception. Die Vorgehensweise von `parseKlammer()` entspricht der von `parseAbsolut()`: Die Methode `parseExpression()` wird rekursiv aufgerufen, um von dort eine Expression zu erhalten. Diese Expression wird zurückgegeben.

14.4 Interpreter – Das UML-Diagramm

Aus dem Beispielprojekt Interpreter_4 müssen wir das UML-Diagramm aufgrund seines Umfangs etwas aufteilen.

In Abb. 14.6 sehen Sie zunächst die Klassen aus dem Package `symbole`.

Die Klassen aus dem Package `expressions` finden Sie in Abb. 14.7.

Und die Klassen aus dem Package `interpreter` selbst finden Sie in Abb. 14.8.

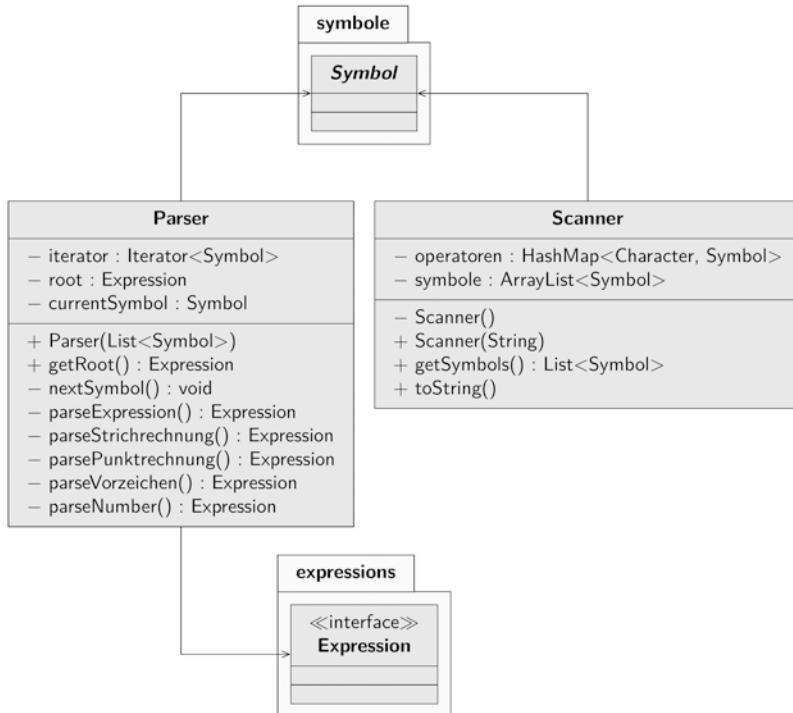


Abb. 14.8 UML-Diagramm des Interpreter Pattern (Beispielprojekt Interpreter_4, package `interpreter`)

14.5 Diskussion des Interpreter Patterns

Ich habe eingangs geschrieben, dass die GoF bei der Beschreibung des Interpreter Patterns voraussetzt, dass der Ausdruck bereits als fertiger Parse-Baum vorliegt. Was also ist das grundlegende Prinzip des Interpreters?

Sie haben die gemeinsame Schnittstelle Expression. Von dieser Schnittstelle werden terminale und nichtterminale Klassen abgeleitet. Jede nichtterminale Klasse repräsentiert eine Regel in der Grammatik; die Operanden – im oberen Beispiel die Zahlen – werden in Instanzen von terminalen Klassen gespeichert. Die nichtterminalen Klassen referenzieren Unterausdrücke, die selbst entweder terminal oder nichtterminal sind. Daraus folgt, dass jeder zu interpretierende Ausdruck sich als abstrakter Syntaxbaum darstellen lassen muss. Der Syntaxbaum wird aus Instanzen der definierten terminalen und nichtterminalen Klassen zusammengesetzt.

Der Client ruft auf der Wurzel des Syntaxbaums die in der gemeinsamen Schnittstelle definierte Methode auf. Jede NichtTerminalExpression ruft die gleiche Methode auf den referenzierten Expressions auf, verarbeitet die Antwort der Expressions und returniert das Ergebnis. Kommt Ihnen dieses Vorgehen vom Composite Pattern bekannt vor? Die Ähnlichkeit ist verblüffend, nicht wahr? Die GoF beschreibt, dass Composite und Interpreter „viele gemeinsame Implementierungsaspekte“ haben. Im Internet und in der Literatur werden Sie gelegentlich sogar die Meinung lesen, dass es sich beim Interpreter nur um einen Sonderfall von Composite handelt. Sie werden jedoch Unterschiede im Zweck und in der Implementierung feststellen. Das Composite ist ein Struktur-Muster, das eine Baumstruktur abbilden kann, der Interpreter dagegen ist ein Verhaltensmuster, das in unseren Beispielen über die Struktur hinweg Verarbeitungen vornimmt.

14.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sie haben eine einfache Grammatik.
- Ein Ausdruck in dieser Grammatik lässt sich als abstrakter Syntaxbaum darstellen.
- Die Syntaxbäume werden aus terminalen und nichtterminalen Ausdrücken zusammengesetzt.
- Jeder nichtterminale Ausdruck repräsentiert eine Regel in der Grammatik.
- Um einen abstrakten Syntaxbaum zu erstellen, muss die Eingabe gescannt und geparsst werden.
- Der Scanner wandelt den Eingabestring in eine Folge von Tokens – Symbolen – um.
- Der Parser prüft die syntaktische Korrektheit der Eingabe und baut den Syntaxbaum auf.

Zweckbeschreibung

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

„Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.“



Abstrakte Fabrik (Abstract Factory)

15

Erinnern Sie sich an das Singleton Pattern aus Kapitel Kap. 3? Sie haben dort ein Objekt erzeugt, ohne den new-Operator zu verwenden. Stattdessen konnte der Client sich mit einer statischen Zugriffsmethode die Instanz der Klasse holen. Die abstrakte Fabrik oder abstract factory delegiert das Erzeugen von Objekten an eine Methode, so dass Sie selbst als Anwender auf die eigentliche Erzeugung mit new verzichten können.

Anhand von zwei Beispielen schauen wir uns das genauer an. Wir beginnen mit einem schönen Garten

15.1 Gärten anlegen

In unserem Beispiel gibt es verschiedene Arten von Gärten; zunächst haben wir Klostergärten, in denen Kräuter angepflanzt werden. Aber es gibt auch Ziergärten, in denen Rosen wachsen. Die jeweilige Einfriedung ist natürlich unterschiedlich. Der Klostergarten ist von einer Steinmauer umgeben, während der Ziergarten von einer schön geschnittenen Hecke eingerahmt wird. Auch der Boden ist jeweils anders: Im Klostergarten liegen Steinplatten, im Ziergarten laufen Sie auf frischem Gras. Kloster- und Ziergarten sind also zwei unterschiedliche *Produktfamilien*. Jede Produktfamilie hat gleiche Merkmale (Pflanzen, Einfriedung, Boden), die aber jeweils unterschiedlich ausgestaltet sind; die jeweiligen Merkmale werden in der abstrakten Fabrik als *Produkte* bezeichnet.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_15

15.1.1 Der erste Versuch

Beginnen wir mit dem Beispielprojekt `Abstrakter_Garten`, dort mit dem Paket `Versuch1`, in dem wir eine Klasse `Garten` definieren, in der der Garten angelegt und gepflegt wird. Etwa so:

```
public class Garten {
    private enum Gartentyp {
        Klostergarten, Ziergarten
    };
    private Gartentyp garten = Gartentyp.Klostergarten;

    public void bodenLegen() {
        switch (garten) {
            case Klostergarten -> {
                // alte Steinplatten legen
            }
            default -> {
                // Gras säen
            }
        }
    }

    public void pflanzeSetzen() {
        switch (garten) {
            case Klostergarten -> {
                // Kräuter pflanzen
            }
            default -> {
                // Rosen setzen
            }
        }
    }

    public void einfrieden() {
        // ... wie bei den beiden vorigen Methoden
    }
}
```

Sie ahnen schon, dass dieser Ansatz irgendwie verkehrt wirkt. Aber warum stören Sie sich daran? Ganz einfach: Sie haben drei Methoden, in denen eine `switch`-Abfrage gebraucht wird. Wenn Sie einen weiteren Garten anlegen wollen – beispielsweise einen Nutzgarten mit Tomaten – müssen Sie in drei Methoden eine `switch`-Abfrage ändern. Das ist wenig wartungsfreundlich, aber dafür umso fehleranfälliger.

Der Garten muss sicher auch bewirtschaftet werden: Die Pflanzen wollen gegossen und geschnitten werden; außerdem müssen Sie regelmäßig das Unkraut jäten. Die Methoden zum Bewirtschaften des Gartens werde ich hier nicht abdrucken. Sie haben hier allerdings einen Verstoß gegen das Single Responsibility Principle: Das Anlegen des Gartens und dessen Pflege packen Sie in eine Klasse. Ich hatte zwar geschrieben, dass man ruhig dagegen verstoßen darf, wenn man sich darüber bewusst ist und einen guten Grund hat. Aber genau an diesem guten Grund fehlt es hier. Der Verstoß gegen SRP könnte aber problematisch werden, weil sowohl das Anlegen als auch das Bewirtschaften des Gartens sehr aufwendig sind; diese zwei Aufgaben sollten besser nicht vermischt werden. Der Ansatz oben implementiert also in einer einzigen Klasse sehr viel Code, der im Zweifel gar nicht gebraucht wird. Wenn Sie eine Zuständigkeit neu definieren, müssen beide Zuständigkeiten erneut getestet werden. Keine der Zuständigkeiten kann wiederverwendet werden. Hier ist die Kohäsion ausgesprochen schwach und das ist ein Indiz für ein unangemessenes Design.

15.1.2 Der zweite Versuch – Vererbung

Vielleicht wäre das Problem gelöst, wenn Sie Vererbung einsetzen? Im Paket Versuch2 definieren Sie eine abstrakte Klasse `AbstrakterGarten`, von dem die Unterklassen `Klostergarten`, `Nutzgarten` oder `Ziergarten` erben. Der Garten wird im Konstruktor der Unterklasse angelegt.

```
public class Klostergarten extends AbstrakterGarten {  
    Klostergarten() {  
        super.bodenLegen( new AlteSteinplatte() );  
        super.pflanzenSetzen( new Kraeutermischung() );  
        super.einfrieden( new Steinmauer() );  
    }  
}
```

Diese Lösung sieht zwar sauberer aus, tatsächlich ist dadurch das Problem der doppelten Zuständigkeit aber nicht gelöst: Sie haben immer noch ein Objekt, das den Garten aufwendig anlegt und ihn bewirtschaftet. Darüber hinaus verstößt dieser Ansatz gegen ein Prinzip, das ich als sehr viel wesentlicher erachte als das SRP: Ziehe Komposition der Vererbung vor. Lassen Sie uns also eine Alternative finden!

15.1.3 Der dritte Ansatz – die abstrakte Fabrik

Im Paket Versuch3 haben Sie unterschiedliche Gärten (*Produktfamilien*) mit jeweils unterschiedlichen *Produkten*, also unterschiedlichen Arten von Pflanzen, Bodenarten und Einfriedungen. Die Produkte variieren und werden jetzt gekapselt. Weiter vorne habe ich an-

gesprochen, dass es sinnvoll ist, gegen Schnittstellen zu programmieren. Sie erhalten sich hierdurch größtmögliche Flexibilität. Wie lassen sich die Produkte abstrahieren? Sowohl Kräuter als auch Rosen sind Pflanzen; sowohl die Steinmauer als auch die Hecke sind Einfriedungen; sowohl die Steinplatten als auch der Rasen sind Boden. Entwerfen Sie also die Schnittstellen, die unterschiedlichen abstrakten Produkte, und die Implementierungen, die unterschiedlichen konkreten Produkte. Als Beispiel zeige ich hier einmal die Pflanzen.

```
public interface Pflanze {
    // das abstrakte Produkt "Pflanze"
}
```

Es gibt mindestens zwei Implementierungen von Pflanzen:

```
public class Rose implements Pflanze {
    // das konkrete Produkt "Rose"
}

public class Heilkraut implements Pflanze {
    // das konkrete Produkt "Heilkraut"
}
```

Um die unterschiedlichen Gärten anzulegen, definieren Sie eine Fabrik. Für einen Klostergarten implementieren Sie eine `KlostergartenFabrik` und für den Ziergarten eine `ZiergartenFabrik`. Von beiden Fabriken erwarten Sie, dass sie die gleichen Methoden anbieten und die gleichen Produkte erstellen. Um dies zu gewährleisten, definieren Sie eine Schnittstelle (abstrakte Klasse oder Interface), die abstrakte Fabrik, von der die konkreten Fabriken erben. Und darin sind die Methoden enthalten, die jeweils in den konkreten Fabriken implementiert werden müssen.

```
public interface AbstrakteGartenfabrik {
    Pflanze pflanzeSetzen();
    Boden bodenLegen();
    Einfriedung einfrieden();
}
```

Die `KlostergartenFabrik` liefert die für einen Klostergarten typischen Elemente zurück.

```
public class KlostergartenFabrik implements AbstrakteGartenfabrik {
    @Override
    public Pflanze pflanzeSetzen() {
        return new Heilkraut();
    }
}
```

```
@Override  
public Boden bodenLegen() {  
    return new Steinplatte();  
}  
  
@Override  
public Einfriedung einfrieden() {  
    return new Steinmauer();  
}  
}
```

Und wie kann der Client nun damit umgehen, der einen Garten haben möchte? Er erzeugt eine Instanz der gewünschten GartenFabrik und lässt sich hiervon die einzelnen Elemente geben. Im folgenden Beispiel erstellen wir die Elemente für einen Klostergarten.

```
public class Gartenbau {  
    Gartenbau() {  
        AbstrakteGartenfabrik fabrik = new KlostergartenFabrik();  
        var boden = fabrik.bodenLegen();  
        var einfriedung = fabrik.einfrieden();  
        var pflanze = fabrik.pflanzeSetzen();  
    }  
}
```

Beachten Sie, dass ich für die Deklaration der Variablen `fabrik` explizit die Oberklasse `AbstrakteGartenfabrik` angebe. Die Verwendung von `var` für diese Deklaration würde eine Fabrik vom Typ `KlostergartenFabrik` erzeugen, was ich aber hier explizit nicht möchte.

Schauen wir uns an, was wir dadurch gewinnen.

15.1.4 Vorteile der abstrakten Fabrik

Das Klassendiagramm Abb. 15.1 zeigt zur besseren Übersicht das Projekt nur gekürzt – der Boden und die Klostergartenfabrik bleiben hier mal außen vor.

Die abstrakte Gartenfabrik kennt die abstrakte Deklaration der einzelnen Produkte, also die Schnittstellen Boden, Einfriedung und Pflanze. Der Client legt sich eine Variable vom Typ der abstrakten Fabrik an, die eine Instanz einer konkreten Fabrik referenziert: `AbstrakteGartenfabrik fabrik = new ZiergartenFabrik()`. Die konkrete Fabrik kennt ihre speziellen Produkte: Im Beispiel kennt der Ziergarten nur Rasen, Rose und Hecke – er hat keinen Bezug zu den Steinplatten, zum Heilkraut und zur Steinmauer. Wenn der Client Boden, Pflanzen und Einfriedung anfordert, ist die Fabrik in der Lage, ihre speziellen Produkte zurückzugeben. Da der Client sich auf Abstraktionen stützt, also

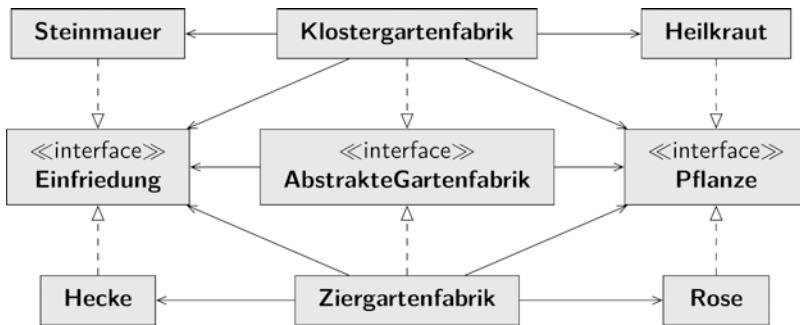


Abb. 15.1 Gekürztes Klassendiagramm des Gartenprojekts

die Schnittstellen, hat er keine Ahnung, welche konkreten Produkte er erhält – und dank der Polymorphie muss er sich auch keine Gedanken über den Laufzeittyp des Objekts machen.

Der Client ist frei, eine andere Fabrik zu instanzieren und sich so ganz andere Produkte geben zu lassen. Eine einzige Zeile Code ändert die Art des Gartens:

```
AbstrakteGartenfabrik fabrik = new ZiergartenFabrik();
```

Der Client ist unabhängig von Änderungen bestehender Fabriken. Wenn im Ziergarten anstelle von Rosen Lilien angepflanzt werden sollen, kann diese Änderung in der **ZiergartenFabrik** vorgenommen werden; der Client wird diese Änderung gar nicht bemerken, er muss nicht erneut getestet werden.

Sie stellen mit dieser Lösung sicher, dass der Garten konsistent ist. Sie beziehen die einzelnen Produkte von einer bestimmten Fabrik. Die Fabrik liefert einheitlich nur Produkte entweder für einen Klostergarten oder für einen Ziergarten. Die Gärten sind nun konsistent; es ist sichergestellt, dass im Klostergarten keine Rosen wachsen und im Ziergarten kein Heilkraut angepflanzt wird.

15.1.5 Einen neuen Garten definieren

Sie haben eine zusammengehörende Familie aus Produkten und Sie können sehr leicht neue Familien hinzufügen. Was müsste passieren, wenn Sie einen neuen Garten, beispielsweise einen Schrebergarten, erstellen wollten? Zunächst brauchen Sie eine neue konkrete Fabrik, die das Interface **AbstrakteGartenFabrik** implementiert. Schauen Sie dazu in das Beispielprojekt Schrebergarten:

```
public class SchrebergartenFabrik implements AbstrakteGartenfabrik {
    @Override
    public Pflanze pflanzeSetzen() {
        return new Tomate();
    }
}
```

```
@Override  
public Boden bodenLegen() {  
    return new Betonboden();  
}  
  
@Override  
public Einfriedung einfrieden() {  
    Return new Maschendrahtzaun();  
}  
}
```

Sie erstellen nun noch die konkreten Produkte, die Klassen Tomate, Betonboden und Maschendrahtzaun, die den Schnittstellen der abstrakten Produkte, Boden, Einfriedung und Pflanze, entsprechen müssen.

- Warum wird das Pattern „Abstract Factory“ genannt? Weil sowohl die Fabriken als auch die Produkte sich auf Abstraktion stützen.

15.2 Diskussion des Patterns und Praxis

Sie werden die abstrakte Fabrik immer dann einsetzen, wenn Sie eine Menge einzelner Produkte benötigen, die alle einer bestimmten Art entsprechen. Die Produkte bilden die Produktfamilie. Im vorigen Beispiel haben Sie die Produktfamilien der Gärten kennengelernt. Denkbar wäre aber auch folgendes Beispiel gewesen: Sie programmieren eine Adressverwaltung. Wenn Sie die Postleitzahl codieren, planen Sie für Deutschland fünf Ziffern ein. Der Aufbau einer Postleitzahl wird aber in Amerika oder in Russland ein anderer sein. Genauso variiert die Länge der Telefonnummer von Land zu Land. Vielleicht gibt es ja sogar einen unterschiedlichen Aufbau oder ein bestimmtes Muster dafür? Wenn Sie also einen Kontakt instanzieren, brauchen Sie für jedes Land einheitlich eine Menge von Produkten: eine länderspezifische Postleitzahl und eine länderspezifische Telefonnummer. Die AmerikaFabrik wird genauso wie die RusslandFabrik diese Produkte liefern, aber eben abgestimmt auf das betreffende Land. Der Kontakt, der von der Fabrik erzeugt wird, ist, wenn die Fabrik richtig programmiert wurde, konsistent. Darüber hinaus kann der Client so allgemein geschrieben werden, dass sein Code für alle erdenklichen Produktfamilien gültig ist.

Wo finden Sie in der Praxis die abstrakte Fabrik wieder? Denken Sie an die unterschiedlichen Look and Feels, die Java zur Verfügung stellt. Sie können ein bestimmtes Aussehen einstellen und zur Laufzeit ändern. Jede Komponente wird einheitlich mit dem gewählten Look and Feel gezeichnet. Drittanbieter können ihren eigenen Look and Feel entwickeln. Und da alle erzeugten Objekte aus derselben Familie kommen, können sie auch untereinander agieren, wenn es notwendig wird. Eine Klippe muss aber angesprochen werden: Wenn Sie die Produkte bestimmt haben, sollten Sie daran festhalten. Stellen

Sie sich vor, Sie wollen in Ihrem Garten auch einen Teich anlegen. Die Schnittstelle `AbstrakteGartenfabrik` muss eine entsprechende Fabrikmethode `getTeich()` vorschreiben. Anschließend müssen alle konkreten Fabriken eine Methode definieren, die ein Teich-Objekt zurückgibt. In unserem kleinen Beispiel ist das sicher kein Problem. Wenn Sie aber ein großes Framework erstellt haben, für das Ihre Kunden schon eigene Gartenfabriken erstellt haben, dann kann diese Änderung sehr wartungsintensiv und damit auch sehr kostspielig werden.

15.3 Gespenster jagen

Stellen Sie sich vor, sie bekommen den Auftrag, ein Spiel zu erstellen in dem man in einem alten Haus Gespenster jagt und magische Türen öffnen muss. Erinnern Sie sich noch an die alten Textabenteuer, die in den Anfängen der Homecomputer-Zeit sehr populär waren und auch heute noch Fans haben? Wir machen es ähnlich.

15.3.1 Die erste Version

Die erste Version, die Sie erstellen, soll ganz simpel sein. Sie lassen sich ein Haus erzeugen und erforschen es mit ein paar einfachen Befehlen. Die Befehle, die Sie eingeben, werden mit dem Größer-Zeichen markiert, das nach der Beschreibung der Spielsituation erscheinen soll. Ein Beispiel:

Beispiel

Du befindest Dich in der Diele.

Du siehst

- In nördlicher Richtung: eine geschlossene Tür
- In südlicher Richtung: eine Wand
- In westlicher Richtung: eine geschlossene Tür
- In östlicher Richtung: eine Wand

> open door west

Dir Tür wird geöffnet.

> enter door west

Du bist jetzt im Badezimmer.

> look

Du befindest Dich im Badezimmer.

Du siehst

- In nördlicher Richtung: eine Wand
- In südlicher Richtung: eine Wand
- In westlicher Richtung: eine Wand
- In östlicher Richtung: eine offene Tür

> exit ◀

Die erste Version hat mit Fabriken noch gar nichts zu tun. Sie soll Ihnen die Logik des Spiels und die Herangehensweise der Programmierung vorstellen. Wir beginnen mit dem Beispielprojekt Haus_1.

15.3.1.1 Der Spieler

Die Klasse Spieler hat die Aufgabe, die aktuelle Position des Spielers zu speichern. Wenn der Spieler einen anderen Raum betritt, wird das Datenfeld geändert.

```
public class Spieler {  
    private Raum aktuellerRaum;  
  
    public void setAktuellerRaum(Raum raum) {  
        aktuellerRaum = raum;  
    }  
  
    public Raum getAktuellerRaum() {  
        return aktuellerRaum;  
    }  
}
```

15.3.1.2 Die vier Himmelsrichtungen

Sie haben in dem kurzen Auszug oben gesehen, dass die vier Himmelsrichtungen eine wesentliche Rolle spielen. Sie müssen immer explizit angeben, welche von vier möglichen Türen Sie betreten oder öffnen möchten. Die Himmelsrichtungen werden in einer Enumeration gespeichert. Den Quelltext drucke ich im Folgenden nur gekürzt ab – Sie finden in den Quelldateien das Beispielprojekt Haus_1 und darin einige Erweiterungen. Beispielsweise spielt die Gegenrichtung eine wichtige Rolle: Wenn Sie in einem Raum eine Tür in nördlicher Richtung finden und diese Tür durchqueren, muss die Tür im neuen Raum zwangsläufig in südlicher Richtung sein. Außerdem gibt es eine statische Methode, die die eingegebenen Strings des Anwenders auf ein Objekt der Enumeration abbildet. Ich bitte Sie aber, diese Details selbst zu erkunden.

```
public enum Richtung {  
    NORDEN("in nördlicher Richtung"),  
    SUEDEN("in südlicher Richtung"),  
    WESTEN("in westlicher Richtung"),
```

```

OSTEN("in östlicher Richtung"),
UNBEKANNT("unbekannte Richtung");

private final String beschreibung;
private Richtung gegenrichtung;

private Richtung(String beschreibung) {
    this.beschreibung = beschreibung;
}

private void setGegenrichtung(Richtung gegenRichtung) {
    this.gegenrichtung = gegenRichtung;
}
// gekürzt

@Override
public String toString() {
    return beschreibung;
}
}

```

15.3.1.3 Das Haus bauen

Ein Haus besteht aus Türen und Wänden und bildet damit Räume. Die Klasse Bauteil beschreibt die Bauteile. Bitte beachten Sie, dass diese Klasse kein abstraktes Produkt ist und auch in den späteren Versionen keines werden wird. Der einzige Zweck dieser gemeinsamen Klasse ist es, die Methoden betreten() und beschreiben() zu deklarieren.

```

public abstract class Bauteil {
    protected abstract void betreten(Spieler spieler);

    protected abstract String beschreiben();
}

```

Eine Wand kann – jedenfalls in meiner Version – nicht betreten werden; aber vielleicht möchten Sie das Projekt ja weiter ausarbeiten und Wände erstellen, die sich verschieben lassen und den Weg zu einem geheimnisvollen Gang freimachen. Für meine Version kann die Wand sich darauf beschränken, eine Fehlermeldung auszugeben, wenn die Methode betreten() aufgerufen wird. Allerdings muss sie in der Lage sein, sich selbst zu beschreiben.

```

public class Wand extends Bauteil {
    @Override
    public void betreten(Spieler spieler) {

```

```
        System.out.println("Du läufst mit der Nase gegen die Wand.");
    }

    @Override
    protected String beschreiben() {
        return " eine Wand.";
    }
}
```

Eine Tür erhält im Konstruktor die Referenzen auf die zwei Räume, die sie verbindet. Wenn die Tür offen ist, kann sie auch betreten werden, sprich der Anwender geht durch sie hindurch.

```
public class Tuer extends Bauteil {
    private boolean istOffen = false;
    private final Raum raum1;
    private final Raum raum2;

    Tuer(Raum raum1, Raum raum2) {
        this.raum1 = raum1;
        this.raum2 = raum2;
    }

    void oeffnen() {
        System.out.println("Die Tür wird geöffnet");
        istOffen = true;
    }

    @Override
    public void betreten(Spieler spieler) {
        if (istOffen) {
            var aktuellerRaum = spieler.getAktuellerRaum();
            var neuerRaum = aktuellerRaum == raum1 ? raum2 : raum1;
            neuerRaum.betreten(spieler);
        } else
            System.out.println("Die Tür ist geschlossen.");
    }

    @Override
    protected String beschreiben() {
        return istOffen ? " eine offene Tür" : " eine geschlossene Tür.";
    }
    // gekürzt
}
```

Die Klasse Raum erwartet im Konstruktor einen Namen und eine Beschreibung. Beide werden gebraucht, um eine sinnvolle Textausgabe zu generieren. In einer EnumMap werden die Richtungen als Schlüssel gespeichert; die Values sind Bauteile, die an einer Seite angebracht sind: Türen oder Wände. Die überladene Methode bauteilEinfuegen() erlaubt es, an einer Seite des Raums eine Tür oder eine Wand einzufügen. Wenn ein Raum angelegt wird, werden alle Seiten zunächst mit Wänden vorbelegt., und eine Tür kann nur nachträglich an eine Stelle gesetzt werden, an der in beiden Räumen jeweils Wände stehen.

Das ist physikalisch nicht ganz korrekt, denn die Wand zwischen zwei Räumen existiert ja nur einmal. Hier im Beispiel betrachten wir sie von jedem Raum aus individuell.

```
public class Raum extends Bauteil {
    private final String beschreibung;
    private final String name;
    private EnumMap<Richtung, Bauteil> richtungen = new EnumMap<>(Richtung.class);

    void bauteilEinfuegen(Richtung richtung, Wand wand) {
        if (richtungen.get(richtung) == null)
            richtungen.put(richtung, wand);
        else
            System.out.println("Hier kann keine Wand gesetzt werden");
    }

    void bauteilEinfuegen(Richtung richtung, Tuer tuer) {
        var gegenrichtung = richtung.getGegenrichtung();
        var nachbar = tuer.getNachbar(this);
        var meineSeite = richtungen.get(richtung);
        var gegenSeite = nachbar.richtungen.get(gegenrichtung);
        if (gegenSeite instanceof Wand && meineSeite instanceof Wand) {
            richtungen.put(richtung, tuer);
            nachbar.richtungen.put(gegenrichtung, tuer);
        } else
            System.out.println("Hier kann keine Tür gesetzt werden");
    }
}
```

Selbstverständlich können Sie einen Raum auch betreten. Die Methode gibt eine Beschreibung des aktuellen Raums aus und informiert den Spieler über seine aktuelle Position.

```
@Override
public void betreten(Spieler spieler) {
    System.out.println("Du bist jetzt " + beschreibung);
    spieler.setAktuellerRaum(this);
}
```

Die weiteren Methoden der Klasse beschreiben den Raum. Wenn Sie das Projekt weiter ausbauen möchten, können Sie entsprechend diesem Vorgehen Falltüren oder eine Treppe oder auch einen Kamin vorsehen, durch die der Raum gewechselt werden kann.

15.3.1.4 Die Klasse „Haus“ zur Steuerung des Spiels

Die Klasse Haus steuert das Spiel. Die Methode betreten() fragt in einer while-Schleife Ihre Eingaben ab und reagiert darauf. Mit dem Befehl look schauen Sie sich um. Der Befehl exit beendet das Spiel. Um eine Tür zu öffnen, geben Sie open door und die Richtung, in der die Tür sich befindet, ein, also beispielsweise open door north. Um in einen anderen Raum zu gehen, betreten Sie die Tür mit enter door north. Sie können selbstverständlich nur eine geöffnete Tür durchqueren. Mehr Kommandos braucht unser Spiel nicht.

Die main-Methode erzeugt eine Instanz der Klasse Haus und ruft deren Konstruktor auf. Der Konstruktor ruft die Methode erzeugeHaus() auf. In dieser Methode werden Räume, Wände und Türen erzeugt. Anschließend werden die Bauteile zusammengestellt und die Diele als Einstiegspunkt definiert. Achten Sie im Beispielcode auch auf den Kommentar. Es gäbe für den Bau der Räume eine alternative Vorgehensweise hier an dieser Stelle und in der Klasse Tuer, bei der Sie – ebenfalls kommentiert – einen Vorschlag für einen zusätzlichen Konstruktor finden. Schauen Sie sich das bitte separat an.

Ach ja: Sie sind bei den Richtungsangaben zwischen den Räumen völlig frei. Niemand zwingt Sie, eine physikalisch überhaupt mögliche Anordnung von Räumen zu konstruieren. Schauen Sie sich die Räume und ihre Anordnung im Beispielcode an, und konstruieren dann einfach noch eine Tür von der Vorratskammer Richtung Osten ins Badezimmer. Aus jedem Raum gibt es in jede Richtung maximal eine Tür zu genau einem anderen Raum, und aus diesem geht es auch in Gegenrichtung nur wieder in den Ursprungsräum zurück. Aber ob das Herrenzimmer nach Norden noch eine Tür in die Küche oder ins Wohnzimmer enthalten darf, bleibt Ihnen als „Schöpfer“ überlassen. Verwirren Sie den Spieler nach Belieben.

Wenn die Bauarbeiten abgeschlossen sind, ruft die main-Methode die Methode betreten() auf, die das Spiel startet. Der Spieler sieht sich zunächst um und die Diele wird beschrieben. Danach werden in einer while-Schleife die Befehle des Spielers gelesen und verarbeitet.

Bei der Verarbeitung der Befehle greifen Sie auf das Command-Pattern zurück. Es gibt das Interface Command, das die Methoden matches() und handle() vorschreibt. Der Methode matches() wird die eingegebene Zeile übergeben; die Methode prüft, ob der eingegebene String dem Command-Objekt entspricht. Der eingegebene Befehl wird durch die statische innere Klasse CmdLine repräsentiert, die intern den Befehl in Tokens zerlegt.

```

public static class CmdLine {
    private final String[] tokens;

    public CmdLine(String nextLine) {
        this.tokens = nextLine.split(" ");
    }

    public boolean startsWith(String... tokens) {
        if (this.tokens.length < tokens.length)
            return false;

        for (var i = 0; i < tokens.length; i++)
            if (!this.tokens[i].equalsIgnoreCase(tokens[i]))
                return false;

        return true;
    }

    public boolean hasRichtung(int index) {
        try {
            var r = Richtung.getRichtung(tokens[index]);
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    public Richtung getAsRichtung(int index) {
        return Richtung.getRichtung(tokens[index]);
    }
}

```

Die Methode `handle()` des Interface `Command` beschreibt die durchzuführende Aktion des Commands. Exemplarisch drucke ich den Quelltext des `Exit`-Commands ab. Schauen Sie sich aber bitte auch noch im Beispielcode an, wie auf fehlende Richtungsangaben bei den Kommandos „`open door`“ und „`enter door`“ reagiert wird.

```

public class Exit implements Command {
    @Override
    public boolean matches(CmdLine cmdLine) {
        return cmdLine.startsWith("exit");
    }

    @Override
    public void handle(CmdLine cmdLine, Spieler spieler) {
        System.exit(0);
    }
}

```

```
}
```

Die oben erwähnte while-Schleife übergibt den jeweils eingegebenen Befehl an ein Objekt der Klasse CmdLine. Anschließend werden in einer for-each-Schleife alle Befehle darauf geprüft, ob sie dem eingegebenen Befehl entsprechen. Falls ja, wird dieser ausgeführt.

```
private void betreten() {
    System.out.println(eingang.beschreiben());
    var spieler = new Spieler();
    eingang.betreten(spieler);

    var scanner = new Scanner(System.in);
    String befehl;
    System.out.print(">");
    while (!(befehl = scanner.nextLine()) == null)) {
        var cmdLine = new CmdLine(befehl);
        for (var cmd : commands)
            if (cmd.matches(cmdLine)) {
                cmd.handle(cmdLine, spieler);
                break;
            }
        System.out.print(">");
    }
}
```

Bitte analysieren Sie unbedingt den vollständigen Quelltext der Klasse Haus im Beispielprojekt Haus_1.

15.3.2 Die zweite Version des Projekts

Die zweite Version des Projekts soll in erster Linie die Komplexität der Klasse Haus der ersten Version auflösen. Die Klasse hat drei Aufgaben: Sie muss sich die Bestandteile des Hauses beschaffen, sie muss die Bauteile zusammensetzen und sie muss die Spielsteuerung übernehmen. Die Vielzahl von Aufgaben verstößt gegen das SRP! Tatsächlich sollte es genügen, wenn die Klasse wüsste, in welchem Raum der Spieler sich gerade befindet, um auf ihm den aktuell eingegebenen Befehl anzuwenden.

Um die Komplexität zu brechen, wird in dieser Programmversion, die Sie im Beispielprojekt Haus_2 finden, der Konstruktionsprozess ausgelagert. Es gibt eine Klasse BauteilFabrik, deren Aufgabe es ist, aus den Bauteilen Räume, neue Wände und neue Türen zu erzeugen.

```
public class BauteilFabrik {
    Raum erzeugeRaum(String name, String beschreibung) {
        return new Raum(name, beschreibung);
    }
    // Türen und Wände entsprechend
}
```

Außerdem gibt es in dieser Programmversion die neue Klasse Architekt, die die Fabrik kennt und sich von dort Räume und Türen holt, um das Haus zu konstruieren und aus den einzelnen Bestandteilen zusammenzusetzen. Wenn der Grundriss in einer Datei hinterlegt ist, die gelesen und geparsst werden muss, kann die Klasse Architekt ziemlich komplex werden. Der Architekt bekommt im Konstruktor eine Referenz auf eine BauteilFabrik übergeben. Von dieser Fabrik lässt er sich die Bauteile geben und setzt sie zusammen. Danach gibt er den Raum, den Sie als Einstiegspunkt definiert haben, zurück.

```
public class Architekt {
    private final BauteilFabrik fabrik;

    Raum erzeugeHaus() {
        var diele = fabrik.erzeugeRaum("Diele", "in der Diele");
        var flur = fabrik.erzeugeRaum("Flur", "im Flur");
        // ... gekürzt

        var tuer = new Tuer[7];
        tuer[0] = fabrik.erzeugeTuer(diele, flur);
        // ... gekürzt
        tuer[6] = fabrik.erzeugeTuer(wohnzimmer, arbeitszimmer);

        diele.bauteilEinfuegen(Richtung.NORDEN, tuer[0]);
        // ... gekürzt

        return diele;
    }
}
```

Die Klasse Haus ist jetzt sehr schlank geworden. Sie kann sich bei der Konstruktion darauf beschränken, einen Architekten zu erzeugen, ihn mit einer BauteilFabrik zu parametrisieren und von ihm ein Haus bauen zu lassen. Die Klasse Haus muss nur den Eingang kennen und den nennt ihr der Architekt.

```
public class Haus {
    private final Raum eingang;
    private final Architekt architekt;
```

```
private Haus() {
    architekt = new Architekt(new BauteilFabrik());
    eingang = architekt.erzeugeHaus();
}

private void betreten() {
    System.out.println(eingang.beschreiben());
    var spieler = new Spieler();
    eingang.betreten(spieler);

    var scanner = new Scanner(System.in);
    String befehl;
    System.out.print(">");
    // Auf Befehlseingabe (Enter) warten
    while (!(befehl = scanner.nextLine()) == null))
    {
        // ... gekürzt
    }
}
```

In dieser Version haben wir eine Fabrik eingeführt; im folgenden Abschnitt erstellen wir eine zweite Fabrik.

15.3.3 Version 3 – Einführung einer weiteren Fabrik

In der dritten Version werden Sie das Prinzip des Abstract Factory Pattern wiederfinden. Allerdings ist die abstrakte Fabrik gar nicht so abstrakt, wie Sie gleich sehen werden. Schauen Sie jetzt in das Beispielprojekt Haus_3.

15.3.3.1 Zaubersprüche

Neu in dieser Version ist eine Klasse, die eine Tür definiert, die nur mit einem Zauberspruch geöffnet werden kann. Wenn Sie versuchen, eine Tür zu öffnen, wird diese Ihnen eine Frage stellen. Beispielsweise kann es vorkommen, dass Sie eine Tür öffnen wollen und diese Ihnen sagt: „SPRICH FREUND UND TRITT EIN“. Wenn Sie mit der einschlägigen Literatur vertraut sind, werden Sie natürlich wissen, dass Sie *mellan* eingeben müssen. Die Frage und Ihre Antwort werden in der statischen inneren Klasse *Frage* gespeichert.

```
public static class Frage {
    public final String frage;
    public final String antwort;

    private Frage(String frage, String antwort) {
```

```

        this.frage = frage;
        this.antwort = antwort;
    }
}

```

15.3.3.2 Die neue Fabrik für Türen mit Zauberspruch

Die Klasse TuerMitZauberspruchFabrik erbt von der Klasse BauteilFabrik und überschreibt nur die Methode erzeugeTuer(). Diese Methode entscheidet zufallsgesteuert, ob eine Tür einen Zauberspruch benötigt. Außerdem hat die Fabrik die Aufgabe, eine Liste mit Fragen zu generieren. Wenn eine Tür mit einem Zauberspruch erzeugt werden soll, wird aus dieser Liste eine Frage und das zugehörige Passwort ausgewählt.

```

public class TuerMitZauberspruchFabrik extends BauteilFabrik {
    public static class Frage {
        // ... s. voriger Quelltext
    }

    private final List<Frage> fragen = new ArrayList<>();

    // Initialisierer
    {
        fragen.add(new Frage("SPRICH FREUND UND TRITT EIN ", "mellan"));
        // weitere
    }

    @Override
    Tuer erzeugeTuer(Raum raum1, Raum raum2) {
        var zahl = (int) (Math.random() * 10);
        if (zahl > (fragen.size() - 1))
            return new Tuer(raum1, raum2);
        else
            return new TuerMitZauberspruch(raum1, raum2, fragen.get(zahl));
    }
}

```

Die Fabrik-Klasse ist also sehr unspektakulär. Lassen Sie uns jetzt die Tür mit Zauberspruch betrachten.

15.3.3.3 Das neue Bauteil: Eine Tür mit Zauberspruch

Die Klasse TuerMitZauberspruch erbt von der Klasse Tuer und überschreibt die Methode oeffnen(). An den Konstruktor übergeben Sie sowohl die betroffenen Räume als auch die Frage. Die Räume werden an die Superklasse übergeben, die Frage wird in einem Datenfeld gespeichert.

```
public class TuerMitZauberspruch extends Tuer {  
    private final Frage frage;  
  
    TuerMitZauberspruch(Raum raum1, Raum raum2, Frage frage) {  
        super(raum1, raum2);  
        this.frage = frage;  
    }  
  
    @Override  
    void oeffnen() {  
        if (super.istOffen())  
            System.out.println("Die Tür ist schon offen");  
        else {  
            var scanner = new Scanner(System.in);  
            System.out.print(frage.frage + " >> ");  
            var eingabe = scanner.nextLine();  
            if (eingabe.equalsIgnoreCase(frage.antwort))  
                super.oeffnen();  
            else  
                System.out.println("Die Tür bleibt zu. ");  
        }  
    }  
}
```

In der nächsten Version werden Sie eine weitere Fabrik einführen, mit der Räume generiert werden, in denen Sie Geister antreffen werden.

15.3.4 Version 4 – das Geisterhaus

Unser Ziel ist es, ein Geisterhaus zu programmieren. Dort gibt es Türen, die mit einem Zauberspruch geöffnet werden müssen. Außerdem gibt es einige Räume, in denen sich Geister aufhalten. Um die letzte Version zu programmieren, erstellen Sie also die Klasse GeisterhausFabrik, die von der Klasse TuerMitZauberspruchFabrik erbt. Das heißt, dass zufallsgesteuert Türen mit Zaubersprüchen erstellt werden. Die gleiche Logik wenden Sie erneut an, um Räume zu erstellen, in denen zufallsgesteuert Geister auftauchen können. In einem statischen Datenfeld wird gespeichert, wie sich die Erscheinungen bemerkbar machen können. Sie finden den Code im Beispielprojekt Haus_4.

```
public class GeisterhausFabrik extends TuerMitZauberspruchFabrik {  
    @Override  
    Raum erzeugeRaum(String name, String beschreibung) {  
        var zahl = (int) (Math.random() * 10);  
        if (zahl > 5)
```

```

        return new Raum(name, beschreibung);
    else
        return new RaumMitGeist(name, beschreibung);
}
}
}
```

Die Fabrik soll in der Lage sein, einen Raum mit Geist zu erzeugen. Die Klasse Raum-MitGeist erweitert die Klasse Raum. Wenn Sie den Raum betreten, wird zufallsgesteuert entschieden, ob der Geist erscheint oder nicht. In einem statischen Datenfeld ist gespeichert, wie sich die Erscheinungen bemerkbar machen können.

```

public class RaumMitGeist extends Raum {
    private static final List<String> geister = new ArrayList<>();

    static
    {
        geister.add("Ein Geist huscht durch die Tür.");
        geister.add("Du hörst merkwürdige Geräusche.");
    }

    RaumMitGeist(String name, String beschreibung) {
        super(name, beschreibung);
    }

    @Override
    public void betreten(Spieler spieler) {
        super.betreten(spieler);
        var zufallszahl = (int) (Math.random() * (geister.size() * 2));
        if (zufallszahl < geister.size() - 1) {
            System.out.println("\t*****");
            System.out.println("\t" + geister.get(zufallszahl));
            System.out.println("\t*****\n");
        }
    }
}
```

Zum Abschluss des Projekts möchte ich auf einen Unterschied zum Garten-Projekt eingehen. Dort gab es abstrakte Produkte, die Interfaces Pflanze, Boden und Einfriedung. Im Geisterhaus-Projekt sind die abstrakten Produkte, die Klassen Tuer, Raum und Wand, auch gleichzeitig konkrete Produkte. Weitere konkrete Produkte werden von diesen Produkten abgeleitet, z. B. die Tür mit Zauberspruch oder der Raum mit Geist. Genauso ist auch die BauteilFabrik eine abstrakte und gleichzeitig eine konkrete Fabrik – neue Fabriken werden von dieser Fabrik abgeleitet. Auch das ist eine mögliche Implementierung des Abstract Factory Patterns. Vererbung setze ich statt Komposition ein, weil hier tatsächlich Verhalten einer bestehenden Klasse erweitert wird. Ein RaumMitGeist **ist** ein Raum.

15.4 Abstract Factory – Das UML-Diagramm

In Abb. 15.2 sehen Sie das UML-Diagramm aus dem Beispielprojekt Haus_4.

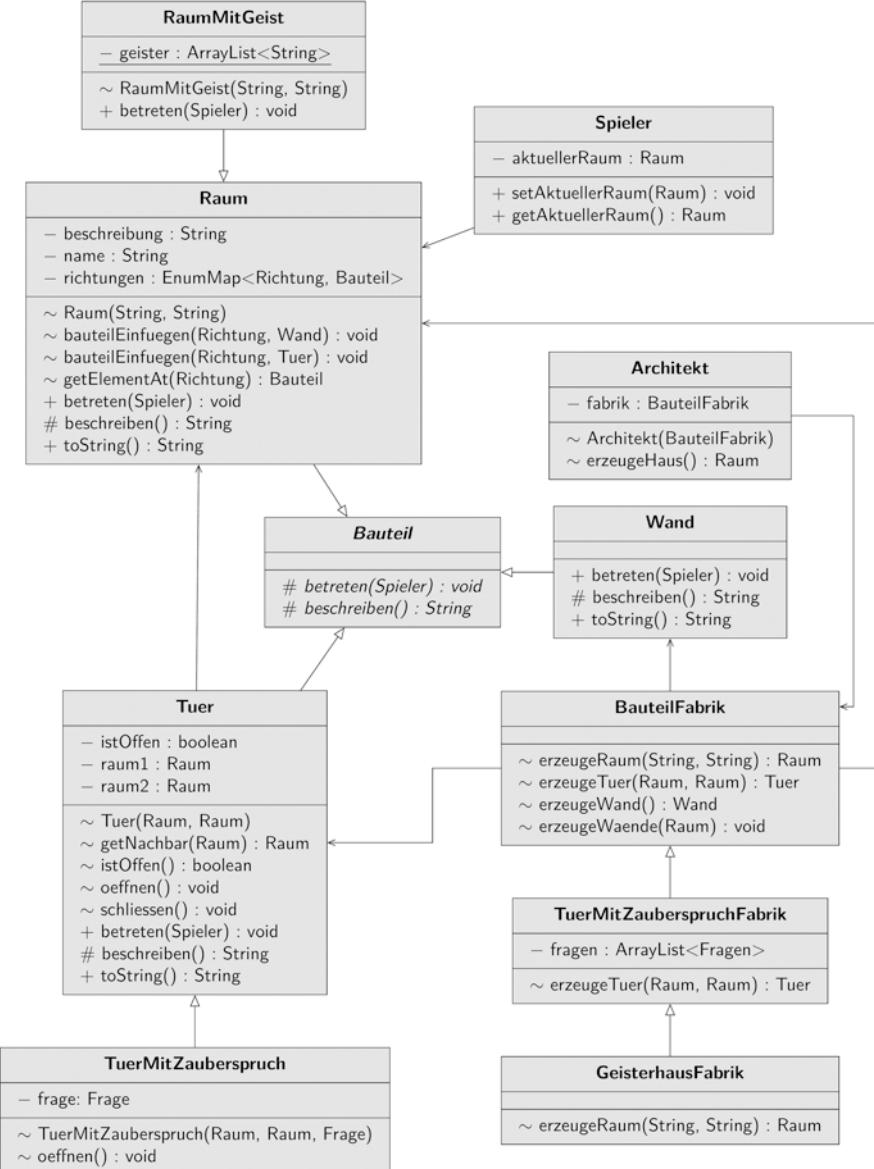


Abb. 15.2 UML-Diagramm des Abstract Factory Pattern (Beispielprojekt Haus_4)

15.5 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Mit der abstrakten Fabrik erstellen Sie eine Produktfamilie, die aus verschiedenen einzelnen Produkten besteht.
- Die Produkte bilden zusammen eine Einheit, z. B. ein einheitliches Look and Feel.
- Bei der Implementierung wird eine Schnittstelle für eine Fabrik angelegt: die abstrakte Fabrik.
- Der Client programmiert gegen Abstraktionen von Produkten, also gegen abstrakte Produkte.
- Die Abstraktionen können abstrakte Klassen oder Schnittstellen sein.
- Eine konkrete Fabrik erzeugt die konkreten Produkte, die zu einer bestimmten Familie gehören.
- Die Bindung zwischen den Produkten und dem Client wird gelockert.
- Das hat zur Folge:
 - Eine ganze Produktfamilie kann ausgetauscht werden, ohne dass der Client-Code betroffen ist,
 - Ein konkretes Produkt kann geändert werden, ohne dass der Client-Code hiervon betroffen wäre,
 - Neue Produktfamilien können hinzugefügt werden,
 - Produkte können wiederverwendet werden, z. B. in anderen Produktfamilien,
 - Die Produkte können untereinander kommunizieren, da sie ihre gegenseitigen Interfaces kennen.
- Ein Nachteil ist, dass eine Produktfamilie nur schwer um weitere Mitglieder erweitert werden kann.

Zweckbeschreibung

Die Gang of Four beschreibt den Zweck des Patterns „Abstract Factory“ wie folgt:

„Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.“



In Kap. 15 haben Sie die abstrakte Fabrik kennengelernt. Wenn Sie nun das Factory Method Pattern bearbeiten, können Sie viele Strukturen und Argumente hierher übernehmen. Während es bei beiden Patterns darum geht, Objekte zu erzeugen, unterscheiden sie sich in der Zielrichtung; die abstrakte Fabrik erstellt einheitliche Produktfamilien, die Factory Method erzeugt nur ein einzelnes Produkt. Sie unterscheiden sich auch in der Struktur – die Abstract Factory basiert auf Objektkomposition, die Factory Methode beruht, wie Sie gleich sehen werden, auf Vererbung.

16.1 Ein erstes Beispiel

Für das erste, einleitende Beispiel sollen Mahlzeiten zubereitet werden. Da im Moment noch alles sehr einfach gehalten ist, gibt es nur die Klassen `Doener` und `Pizza`. Beide implementieren das Interface `Mahlzeit`. Passend dazu gibt es eine `Pizzeria` und eine `Doenerbude`. Beide erben von der abstrakten Klasse `Restaurant`. Die abstrakte Klasse definiert die Methode `bestellen()`, die vorschreibt, dass der Gast erst nach seinem Wunsch gefragt werden soll, dann die gewünschte Mahlzeit zubereitet und diese schließlich serviert wird. Diese Methode ruft der Gast auf, wenn er eine Mahlzeit bestellen möchte. Die Methode `mahlzeitZubereiten()` ist eine Fabrikmethode. Ihre Aufgabe ist es, ein bestimmtes Produkt zu erstellen. Die konkrete Ausführung wird an Subklassen, also `Doenerbude` und `Pizzeria`, delegiert. Bitte beachten Sie, dass zwar die Methode `bestellungAufnehmen()` ebenfalls von den Subklassen definiert wird, dies aber für

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_16

die Diskussion des Patterns irrelevant ist; ich wollte mit dieser Codierung erreichen, dass der Gast in der Pizzeria nach seinen Pizza-Wünschen und in der Dönerbude nach seinem Döner-Wunsch gefragt wird. Sie finden den vollständigen Code im Beispielprojekt **Mahlzeit**.

```
public abstract class Restaurant {  
    protected abstract String bestellungAufnehmen();  
  
    protected abstract Mahlzeit mahlzeitZubereiten();  
  
    private void mahlzeitServieren(Mahlzeit mahlzeit) {  
        System.out.println("Essen ist fertig! Es gibt " + mahlzeit);  
    }  
  
    public final Mahlzeit bestellen() {  
        var bestellung = bestellungAufnehmen();  
        var mahlzeit = mahlzeitZubereiten();  
        mahlzeitServieren(mahlzeit);  
        return mahlzeit;  
    }  
}
```

Dönerbude und Pizzeria erstellen entweder einen Döner oder eine Pizza – jeweils nach Kundenwünschen variiert. Sämtliche Mahlzeiten implementieren die gleiche Schnittstelle. Der Gast – im Beispiel die Testklasse – erzeugt die Instanz einer Unterklasse der Klasse Restaurant und ruft auf dieser die Methode `bestellen()` auf.

```
public class Test {  
    public static void main(String[] args) {  
        Restaurant mammaMia = new Pizzeria();  
        mammaMia bestellen();  
  
        Restaurant istanbul = new Doenerbude();  
        istanbul bestellen();  
    }  
}
```

Auf der Konsole wird folgender Text ausgegeben:

```
Esse ist fertig! Es gibt Pizza Margharita  
Esse ist fertig! Es gibt Döner mit allem
```

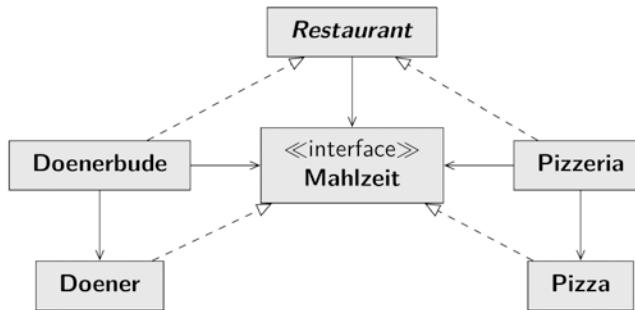


Abb. 16.1 Klassen-Diagramm des Beispielprojekts *Mahlzeit*

Wenn der Gast eine Pizza essen möchte, erzeugt er sich eine Instanz der Klasse `Pizzeria`, sonst eine Instanz der Klasse `Doenerbude`. An keiner Stelle gibt es eine `if`-Abfrage. Allein durch die Auswahl der „richtigen“ Unterklasse wird das gewünschte Produkt erzeugt.

Abb. 16.1 zeigt Ihnen das Klassendiagramm des Projekts. In der Begrifflichkeit des Patterns ist das Interface `Mahlzeit` ein Produkt. Die abgeleiteten Klassen `Doener` und `Pizza` sind konkrete Produkte. Die Schnittstelle `Restaurant` ist ein Erzeuger und die konkreten Lokalitäten (`Pizzeria` und `Dönerbude`) sind konkrete Erzeuger.

16.2 Variationen des Beispiels

Die Factory Method kann optional parametrisiert werden. Das Projekt ist schon auf diese Variation vorbereitet. Die Methode `bestellungAufnehmen()` gibt einen String mit der Bestellung zurück. Die vorige Projektversion ignoriert den Kundenwunsch, was nicht sehr freundlich ist. In der neuen Version wird die Bestellung an die Factory Method als Parameter übergeben, die dann die gewünschte Mahlzeit erstellt. Das Beispiel der Pizza demonstriert dieses Vorgehen. Der nachfolgende Code aus dem Beispielprojekt `Mahlzeit_Variation` zeigt die parametrisierte Fabrikmethode der `Pizzeria`.

```

protected Mahlzeit mahlzeitZubereiten(String bestellung) {
    if (bestellung == null || bestellung.isEmpty())
        return new Pizza();
    else
        return switch (bestellung) {
            case "Calzone" -> new Calzone();
            case "Hawaii" -> new Hawaii();
            default -> { System.out.
                println("Diese Pizza haben wir leider nicht im Angebot");
                yield null;
        };
}
  
```

Beachten Sie, dass ich das return-Statement im else-Zweig mit einem Switch-Expression versehen habe und dann nicht in jedem einzelnen Fall noch ein return schreiben muss, sondern den Ergebnis-Ausdruck jeder Fallunterscheidung an das return-Statement gebe. Im obigen Beispiel sehen Sie auch, dass man erstens in einer Fallunterscheidung (hier der default-Fall) auch mehrere Statements ausführen lassen kann, und zweitens das Schlüsselwort `yield`, das anstelle eines returns verwendet wird. Ein `return` ist ausschließlich für das Beenden einer Methode vorgesehen. Die Beendigung einer Befehlsfolge innerhalb eines Switch-Expressions geschieht mit `yield` und gibt den dann folgenden Wert als Wert des Ausdrucks zurück.

Mit einer parametrisierten Fabrikmethode können Sie leicht neue Produkte einführen. Ohne Parameter wären Sie darauf angewiesen, für jedes neue konkrete Produkt einen neuen konkreten Erzeuger zu erstellen. Da jedoch der Produktionsaufwand für die unterschiedlichen Pizzen ähnlich ist und die Calzone sogar von Pizza erbt, bietet es sich an, die Fabrikmethode zu parametrisieren.

16.3 Praktische Anwendung des Patterns

Sie haben das Pattern bereits im praktischen Einsatz gesehen.

16.3.1 Rückgriff auf das Iterator Pattern

In Kap. 11 haben Sie Iteratoren angelegt. Um eine Sammlung in einer erweiterten `foreach`-Schleife verwenden zu können, muss sie das Interface `Iterable` implementieren. Das Interface schreibt die Methode `iterator()` vor, die ein Objekt vom Typ `Iterator` zurückgibt. Das `Iterable` wird beispielsweise von der Klasse `ArrayList` implementiert, aber auch die zwei Sammlungen, die Sie im Kapitel über das Iterator Pattern erstellt haben, haben dieses Interface implementiert. Alle diese Klassen müssen ein Produkt, ein Objekt vom Typ `Iterator` zurückgeben können; der Iterator schreibt drei Methoden vor, mit denen Sie elementweise durch die Sammlung iterieren können. In den meisten Fällen dürfte es sich anbieten, den Iterator als anonyme Klasse zu definieren, also `return new Iterator{ ... }`. Die Grafik in Abb. 16.2 zeigt dieses Zusammenspiel.

Sie definieren eine Schnittstelle, das Interface `Iterable`, das eine Methode zum Erzeugen von Objekten vorschreibt: `iterator()`. Diese Methode ist die Fabrikmethode. Ein Objekt vom Typ der Schnittstelle `Iterable` muss ein Objekt vom Typ `Iterator` zurückgeben, das mit der eigenen Spezifikation zusammenarbeitet. Im Diagramm habe ich dieses Objekt hilfsweise `MyIterator` genannt. Das Beispielprojekt `Iterator` zeigt diese (grobe) Struktur.

Wenn die Methode `iterator()` eine Fabrikmethode ist, weil sie ein Objekt zurückgibt, könnte man doch im Umkehrschluss sagen, dass jede Methode, die ein Objekt zurückgibt, eine Fabrikmethode ist – was in der Objektorientierten Programmierung gar nicht so selten ist. Eine Fabrikmethode zeichnet sich

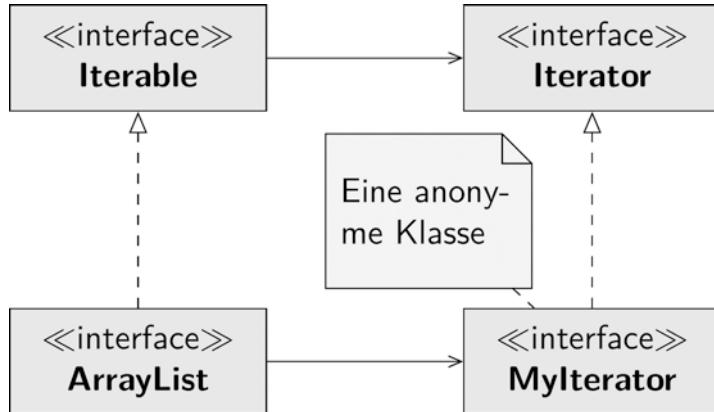


Abb. 16.2 Klassendiagramm Factory Method Pattern

dadurch aus, dass Unterklassen über die genaue Ausprägung des Objekts entscheiden. Der Client weiß nicht, aus welcher Klasse das Objekt, das er erhält, erzeugt wurde.

Sinnvoll ist das Factory Method Pattern, wenn Sie allgemeingültigen Code schreiben wollen. Eine for-each-Schleife, die ich im letzten Beispiel angesprochen habe, hat keine Idee davon, von welcher Klasse das Objekt vom Typ `Iterable` instanziiert wurde – sie benötigt dieses Wissen auch gar nicht. Genauso wenig ist es für die for-each-Schleife interessant, mit welchem konkreten Produkt sie arbeitet. Da die Bindung des konkreten Iterator-Objektes zur for-each-Schleife sehr lose ist, können Sie beliebige Sammlungsklassen definieren und in der for-each-Schleife einsetzen. Der konkrete Erzeuger ist verantwortlich für die Erzeugung eines passenden konkreten Produkts.

Wenn Sie ein neues Objekt mit dem `new`-Operator erzeugen, müssen Sie immer einen Konstruktor der Klasse aufrufen. Wenn Sie eine Fabrikmethode definieren, können Sie den Bezeichner hingegen beliebig wählen. Betrachten Sie beispielsweise das Vorgehen der Klasse `Color`. Hier gibt es zahlreiche statische Methoden, deren einzige Aufgabe es ist, aus bestimmten Parametern ein `Color`-Objekt zu erzeugen. Dabei gibt es beispielsweise die Methode `getHSBColor()`, die sehr viel prägnanter ihren Zweck beschreibt.

Wenn Sie das Factory Method Pattern bis hierher mit der Abstract Factory vergleichen, fällt Ihnen sicher auf, dass die Abstract Factory sehr viel komplexer ist. Tatsächlich beginnen viele Projekte mit der Factory Method und enden schließlich bei der Abstract Factory.

16.3.2 Bei der Abstract Factory

Im Kapitel zur Abstract Factory, als Sie das Geisterhaus programmiert haben, haben Sie mehrfach auf die Factory Method zurückgegriffen. Da gab es den Erzeuger, die Klasse `BauteilFabrik`. In dieser Klasse gibt es die Methoden `erzeugeRaum()`, `erzeuge-`

getür() und erzeugeWand(), die alle Türen, Räume und Wände erzeugen, die keinerlei Besonderheit, also weder einen Geist noch einen Zauberspruch, haben. Diese Klasse ist die Schnittstelle und nichts spricht dagegen, die Schnittstelle so zu definieren, dass sie ein Default-Verhalten mitbringt. Daher wurden von dieser Klasse weitere Klassen, sogenannte konkrete Erzeuger, abgeleitet; die eine Klasse erzeugt Türen mit Zauber- spruch. Die andere Klasse erzeugt Türen mit Zauber spruch und Räume mit Geistern. Die abgeleiteten Klassen überschreiben bestimmte erzeuge-Methoden der Oberklasse. Das sind genau die „Factory Methods“. Bei den Produkten sah es ähnlich aus. Produkte waren die Klassen Tür, Raum und Wand. Davon wurden weitere konkrete Produkte abgeleitet: Türen mit Zauber sprüchen und Räume mit Geistern.

16.4 Ein größeres Beispiel – ein Framework

Praktische Anwendung findet die Factory Method beim Erstellen von Frameworks.

16.4.1 Und noch ein Kalender

Sie erstellen ein Framework, das Termine und Kontakte verwaltet. Auf dem Screenshot in Abb. 16.3 sehen Sie, dass ich verschiedene Kontakte erstellt habe. Außerdem habe ich einen Termin mit meiner Lektorin am 22. Juli 2021 – den darf ich keinesfalls vergessen!

Der Rahmen des Programms stellt eine leere Oberfläche und das Menü zur Verfügung. Sie können Einträge (Verabredungen und Kontakte) erfassen. Wenn Sie einen neuen Eintrag erstellt haben, wird dieser auf einem neuen Karteireiter angezeigt. Das Programm soll flexibel bleiben, um in einer weiteren Ausbaustufe neue Einträge wie E-Mails oder ganze Adressbücher anzeigen zu können. Sie ahnen, dass die Reise dahin geht, dass der Rahmen und die Einträge voneinander getrennt sind und nur über lose Schnittstellen verfügen.

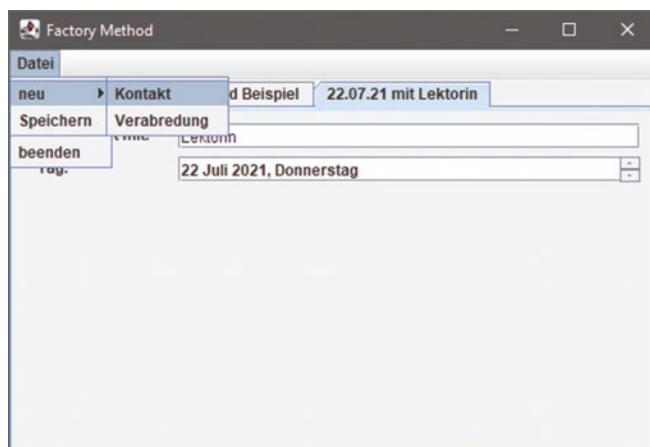


Abb. 16.3 Screenshot des fertigen Beispielprojekts Adressbuch

Lassen Sie uns die Verbindung von Rahmen und Eintrag genauer betrachten. Die Einträge sind für ihre Daten sicher selbst verantwortlich. Ein Kontakt besteht aus Vor- und Nachnamen sowie einem Geburtsdatum. Ein Termin besteht aus dem Namen des anderen Teilnehmers und dem Tag der Verabredung. Die Frage ist nun, welche Einheit die Steuerelemente für die jeweiligen Daten zusammensetzen. Wenn Sie diese Aufgabe an den Rahmen delegieren, setzt dies voraus, dass der Rahmen alle Daten aller denkbaren Einträge kennt. Sie schränken sich insofern ein, als ein bestehender Eintrag keine zusätzlichen Daten bekommen darf, weil der Rahmen sonst modifiziert werden müsste. Auch die Implementierung neuer Einträge wäre ausgesprochen aufwendig, weil die neuen Steuerelemente im Rahmen codiert werden müssten. Der Rahmen wurde jedoch schon ausgiebig getestet und an den Kunden ausgeliefert – Sie werden ihn unter keinen Umständen jemals erneut anfassen wollen. Also muss die Verantwortung für die Gestaltung des Editors an die Klasse delegiert werden, die am besten die benötigten Steuerelemente kennt: den Eintrag selbst.

Sie finden den folgenden Code im Beispielprojekt Adressbuch.

16.4.2 Die Schnittstellen für Einträge und deren Editoren

Die Einträge implementieren das Interface `Eintrag`. Es schreibt zwei Methoden vor; die eine fordert beim Eintrag eine textuelle Beschreibung an, die andere ruft ein Objekt vom Typ `EintragsPanel` ab. Der zurückgegebene `EintragsPanel` kann entweder für eine Verabredung oder für einen Kontakt optimiert sein – das ist so wie beim Iterator, der von jeder beliebigen Liste mit seinen eigenen Spezifika zurückgegeben wird.

```
public interface Eintrag {  
    EintragsPanel getEintragsPanel();  
    String getBeschreibung();  
}
```

Der Editor eines Eintrags ist vom Typ der Schnittstelle `EintragsPanel`. Diese Schnittstelle schreibt auch wieder zwei Methoden vor. Eine Methode fordert den Editor auf, den Eintrag zu speichern. Die andere Methode gibt ein `JPanel` zurück, auf dem die Steuerelemente angeordnet sind.

```
public interface EintragsPanel {  
    void speichern();  
    JPanel getEditor();  
}
```

Wie diese Schnittstellen realisiert werden, zeigt der folgende Abschnitt.

16.4.3 Die Klasse „Kontakt“ als Beispiel für einen Eintrag

Ich möchte als nächstes die Klasse Kontakt als Beispiel für einen möglichen Eintrag vorstellen. Die Klasse speichert in ihren Datenfeldern den Vornamen, den Nachnamen und das Geburtsdatum. Außerdem erstellt die Klasse ein Objekt vom Typ EintragsPanel. Dieses Objekt ist eine Instanz einer inneren Klasse, die weiter unten beschrieben wird.

```
public class Kontakt implements Eintrag {
    private String vorname = "<Vorname>";
    private String nachname = "<Nachname>";
    private Date geburtstag = new Date();
    private final EintragsPanel eintragsPanel = new MyEditor();

    @Override
    public EintragsPanel getEintragsPanel() {
        return eintragsPanel;
    }

    private class MyEditor implements EintragsPanel {
        // ... gekürzt
    }
}
```

Im Konstruktor der inneren Klasse wird das Editor-Panel zusammengesetzt. Ich verweise hier das ZweiSpaltenLayout, das Sie im Kapitel über das Strategy Pattern entwickelt haben. Ferner habe ich den Eingabefeldern einen FocusListener spendiert, der den gesamten Text markiert, wenn Sie in das Feld klicken. Und schließlich bekommt der JSpinner, in dem das Geburtsdatum eingetragen wird, einen eigenen Datumseditor und ein Datumsmodel. Bitte analysieren Sie den Code des Projekts. Natürlich ließe sich statt des einfachen Spinners auch ein DatePicker verwenden, aber den gibt es nicht nativ im JDK, wohl aber in sehr vielen kostenlosen Libraries, von denen Sie sich natürlich gerne eine aussuchen können. Für die Demo hier erfüllt der JSpinner aber seinen Zweck. Sie können das Datum auch manuell überschreiben, statt mühsam tageweise vor und zurück zu „spinnen“.

Die Eingabefelder werden als Datenfelder gespeichert. Wenn der Anwender den Befehl zum Speichern der Daten gibt, überträgt der Editor die Werte aus den Eingabefeldern und aus dem JSpinner in die Datenfelder der äußeren Klasse. Wenn Sie das Projekt weiter ausbauen, könnten Sie beispielsweise vorsehen, dass die Klasse serialisiert wird. Die Methode getEditor() gibt das im Konstruktor aufgebaute Panel zurück.

```
private class MyEditor implements EintragsPanel {
    private final JPanel pnlEingabe = new JPanel();
    private final JTextField edtVorname = new JTextField("<Vorname>");
```

```

private final JTextField edtNachname = new JTextField("<Nachname>");
private final JSpinner spnGeburtstag = new JSpinner();

MyEditor() {
    // ... gekürzt
}

@Override
public void speichern() {
    vorname = edtVorname.getText();
    nachname = edtNachname.getText();
    geburtstag = (Date) spnGeburtstag.getValue();
}

@Override
public JPanel getEditor() {
    return pnlEingabe;
}
}
}

```

Wie geht der Client mit diesen Klassen um?

16.4.4 Die Klasse FactoryMethod als Client

In dieser Klasse gibt es eine Liste, in der alle Einträge gespeichert werden. Jeder neue Eintrag bekommt einen eigenen Reiter in einer JTabbedPane auf der GUI. Wenn der Anwender im Menü auf Speichern klickt, vergleicht ein Objekt vom Typ Action jeden Eintrag in die Liste mit jedem Eintrag mit den Komponenten der JTabbedPane. Von der gerade selektierten Komponente wird der Editor ermittelt und darauf die Methode speichern() aufgerufen. Der Konstruktor setzt die Steuerelemente zusammen und zeigt die GUI auf dem Bildschirm an.

```

public class FactoryMethod {
    // ... gekürzt
    private final JMenuItem mnSpeichern = new JMenuItem();
    private final Action speichernAction = new AbstractAction("Speichern") {
        @Override
        public void actionPerformed(ActionEvent e) {
            listeEintraege.forEach((tempEintrag) -> {
                Object selectedPanel = tbbMain.getSelectedComponent();
                if (tempEintrag.getEintragsPanel().getEditor() == selectedPanel)

```

```
        {
            tempEintrag.getEintragsPanel().speichern();
            var index = tbbMain.getSelectedIndex();
            tbbMain.setTitleAt(index, tempEintrag.toString());
        }
    });
}
};

}
```

Die meisten kniffligen Punkte in dieser Klasse liegen im Bereich der Swing-Programmierung. Für das Factory Method Pattern wichtig ist, dass die Kopplung des Clients an die Einträge lose ist, was es Ihnen erlaubt, sehr einfach weitere Einträge zu entwickeln.

Sie sehen an diesem Beispiel die Stärke der Factory Method: Der Client muss keine Kenntnis darüber haben, von welcher Klasse das Objekt vom Typ `EintragsPanel` instanziert wurde. Es genügt, dass er auf dem Objekt vom Typ `Eintrag` die Fabrikmethode `getEintragsPanel()` aufruft. Da Interfaces Klassen mit ausschließlich abstrakten Methoden sind, können Sie sagen, dass der Eintrag (Kontakt oder Vererbung) die Verantwortung für die Erzeugung des `EintragsPanels` an eine Unterklasse delegiert hat.

16.5 Unterschied zur Abstract Factory

Lassen Sie uns abschließend die Unterschiede zur Abstract Factory ansehen. Auf den ersten Blick scheint der Unterschied verhältnismäßig klar zu sein: Die abstrakte Fabrik erstellt eine Produktfamilie; denken Sie an ein einheitliches Look and Feel oder einen bestimmten Garten. Mit der Fabrikmethode wird ein einzelnes Produkt erstellt. Da aber nirgendwo steht, dass eine Familie zwingend aus zwei oder mehr Produkten bestehen muss, darf der Unterschied nicht allein daran festgemacht werden.

Der meines Erachtens wichtigere Unterschied ist der, dass die Abstract Factory objektbasiert ist, die Factory Method klassenbasiert. Objektbasierte Muster beruhen auf dem Zusammenspiel von Objekten, die vom Client ausgewählt werden. Klassenbasierte Muster beziehen Vererbung mit ein; die Oberklasse ruft die „richtige“ Methode auf.

Background Information

Ich habe Ihnen in Abschn. 1.2 gesagt, dass die Patterns in drei Kategorien unterteilt sind: Erzeugungsmuster, Verhaltensmuster und Strukturmuster. Die GoF hat innerhalb dieser drei Kategorien aber noch zwei weitere Unterschiede getroffen: nach dem Gültigkeitsbereich.

Klassenbasierte Muster sind das Template Method und der Interpreter. Beim Adapter gibt es zwei Ausprägungen – einen objektbasierten und einen klassenbasierten. Alle anderen Patterns sind objektbasiert. Die GoF folgt also ihrem eigenen Postulat, dass Komposition der Vererbung vorzuziehen ist. Einen Überblick über die Musterkategorien finden Sie in Tabelle Tab. 1.1.

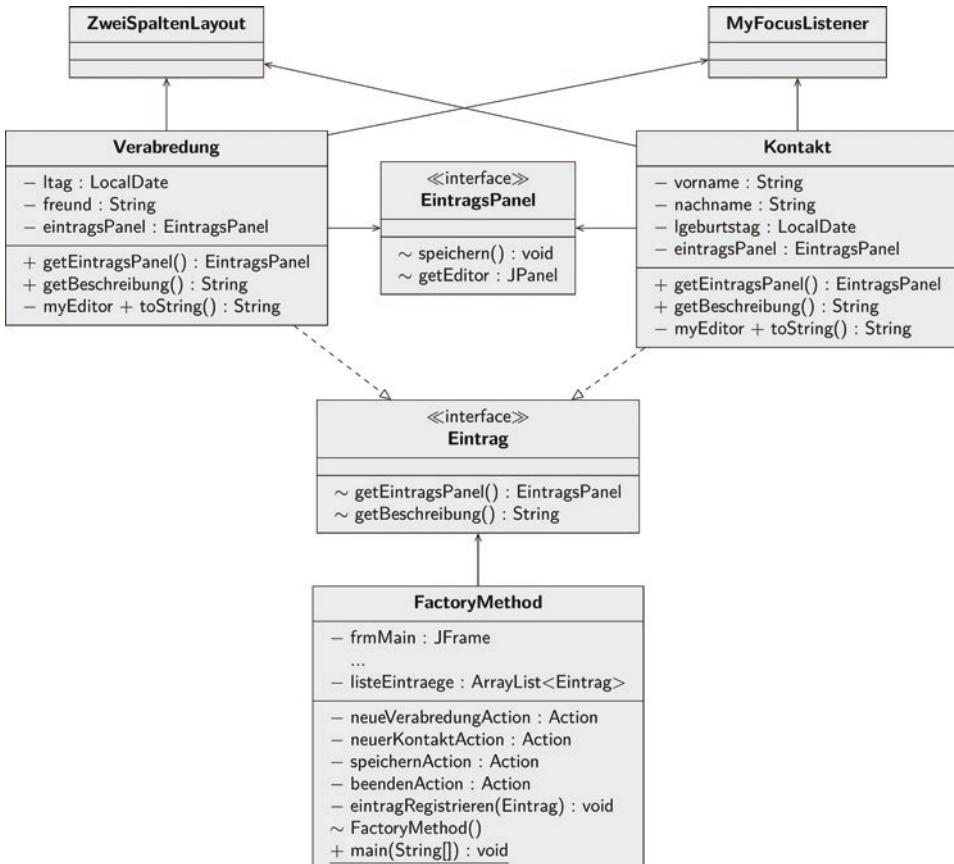


Abb. 16.4 UML-Diagramm des Factory Method Pattern (Beispielprojekt Adressbuch)

16.6 Factory Method – Das UML-Diagramm

Aus dem Beispielprojekt Adressbuch finden Sie das UML-Diagramm in Abb. 16.4.

16.7 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Die Factory Method unterstützt SRP: Objekterstellung und Objektgebrauch werden getrennt.
- Die Objekterstellung wird in eine eigene Klasse ausgelagert.
- Der Erzeuger delegiert die Objekterstellung an eine UnterkLASSE.

- Er ruft dazu die namensgebende Factory Method auf: z. B. `createCar()`.
- Die Unterklasse muss diese Methode auf ihre eigene für sie typische Art und Weise definieren.
- Die Unterklasse entscheidet, welches konkrete Produkt/Objekt erstellt wird.
- Erzeuger und Produkte stützen sich auf Abstraktionen und sind daher für den Client austauschbar.
- Drei Variationen sind denkbar:
 - Der Erzeuger kann neben der Factory Method weitere Methoden definieren, die auf alle erzeugten Produkte angewendet werden, bevor diese zurückgegeben werden,
 - Der Erzeuger kann in der Factory Method eine Default-Implementierung vorsehen,
 - Die Fabrikmethode kann parametrisiert sein.

Zweckbeschreibung

Die Gang of Four beschreibt den Zweck des Patterns „Factory Method“ wie folgt:

„Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“



Das Prototype Pattern gehört zur Kategorie der Erzeugungsmuster. Sie haben eine bestimmte Anzahl von Produkten, die sich nicht wesentlich unterscheidet. Sie erzeugen von einem Produkt einen Prototyp. Wenn Variationen angefordert werden, klonen Sie den Prototyp und variieren ihn. Beispiel: Sie bieten verschiedene Arten Pizzen an. Um mit der Menge an Bestellungen fertig zu werden, fertigen Sie von einer Pizza Margherita einen Prototyp an. Wenn eine Pizza Hawaii bestellt wird, klonen Sie den Prototyp, belegen ihn mit Schinken und Ananas und servieren die fertige Pizza. Damit ist das Pattern eigentlich auch schon erklärt. In diesem Kapitel stelle ich Ihnen weitere Details vor und gehe auf die Frage ein, wie man Objekte in Java klonen.

17.1 Objekte klonen

Das Prototype Pattern ist nicht wirklich schwer zu verstehen – das Problem liegt eher in der Frage, wie ein Objekt geklont werden kann. Schauen Sie sich das Beispielprojekt Autofabrik an. Dort gibt es die Klasse `Motor` mit den Attributen `kennung`, `ps`, `hubraum` und den jeweiligen Zugriffsmethoden. Die Klasse `Auto` hält eine Referenz auf eine Motor-Instanz, Auto speichert die Anzahl der Sitze und schließlich die Sonderedition des Autos. Die Klasse implementiert ferner das Interface `Cloneable` und überschreibt die Methode `clone()` von `Object`.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_17

```
public class Auto implements Cloneable {  
    private Motor motor;  
    private Edition edition;  
    private final int anzahlSitze;  
  
    Auto(Motor motor, Edition edition, int anzahlSitze) {  
        this.motor = motor;  
        this.edition = edition;  
        this.anzahlSitze = anzahlSitze;  
    }  
  
    // ... gekürzt  
  
    @Override  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

Die main-Methode der Test-Klasse führt die Verwendung des Projekts vor. Zuerst wird ein Motor erzeugt und in das als Prototyp konzipierte Auto eingebaut. Der Prototyp soll in der Edition TINY erstellt werden.

```
var motor = new Motor("General Motors", 100, 1.6);  
var prototyp = new Auto(motor, Edition.TINY, 4);  
System.out.println("Prototyp: " + prototyp);  
var neuesAuto = (Auto) prototyp.clone();  
System.out.println("Neues Auto: " + neuesAuto + "\n");
```

Auf der Konsole wird folgender Text ausgegeben:

```
Prototyp: 4-Sitzer PKW, Edition TINY, Motorisierung: General Motors mit  
100 PS und 1.6 Liter Hubraum  
Neues Auto: 4-Sitzer PKW, Edition TINY, Motorisierung: General Motors  
mit 100 PS und 1.6 Liter Hubraum
```

Sie haben also einen Klon vom Prototyp erzeugt.

17.1.1 Kritik an der Realisierung

Die Lösung funktioniert zwar, aber unproblematisch ist sie nicht. Betrachten Sie den ersten Ansatz kritisch.

17.1.1.1 Das Interface Cloneable

Die Methode `clone()` der Klasse `Object` hat die Sichtbarkeit `protected`. Sie muss also von einer Subklasse erst `public` gemacht werden. Welche Rolle spielt das Interface `Cloneable`? Bitte kommentieren Sie testweise im Projekt das Interface aus:

```
public class Auto // implements Cloneable {  
    // ... gekürzt  
  
    @Override  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

Das Projekt kompiliert weiterhin fehlerfrei. Rufen Sie jetzt die Testmethode erneut auf. Sie werden sehen, dass eine `CloneNotSupportedException` geworfen wird. Die Default-Implementierung der Methode `clone()` prüft zunächst, ob das zu klonende Objekt vom Typ `Cloneable` ist. Das Interface ist ein reines Marker-Interface – es schreibt keine Methode vor. Wenn das zu klonende Objekt nicht von diesem Typ ist, wirft die Standardimplementierung die Exception. Wenn das zu klonende Objekt vom Typ `Cloneable` ist, wird eine bitweise Kopie hiervon erstellt und zurückgegeben. Der Typ des Klons ist gleich dem Typ des Originals.

Die Dokumentation zur Methode `Object.clone()` empfiehlt, dass das ursprüngliche Objekt und der Klon folgende Beziehungen haben:

- Die Objekte haben unterschiedliche Referenzen `prototyp != klon`
- Die Klassen sind identisch: `prototyp.getClass() == klon.getClass()`
- Die Objekte sind gleich im Sinne von `klon.equals(prototyp) == true`
Die main-Methode der Testklasse prüft diese Bedingungen.

```
System.out.println("Teste auf Referenzidentitaet (==): " +  
                    (neuesAuto == prototyp));  
System.out.println("Klasse Prototyp: " + prototyp.getClass());  
System.out.println("Klasse neues Auto: " + neuesAuto.getClass());  
System.out.println("Teste auf Gleichheit (equals): " +  
                    neuesAuto.equals(prototyp));
```

Auf der Konsole wird ausgegeben:

```
Teste auf Referenzidentitaet (==): false  
Klasse Prototyp: class Auto  
Klasse neues Auto: class Auto  
Teste auf Gleichheit (equals): true
```

Die Standardimplementierung der equals-Methode in der Klasse Object prüft die Referenzgleichheit der zu vergleichenden Objekte (`prototyp == klon`). Da Prototyp und Klon jedoch unterschiedliche Referenzen haben, werden Sie, wenn Sie `clone()` überschreiben, auch `equals()` überschreiben. Wenn Sie `equals()` überschreiben, sollten Sie auch `hashCode()` überschreiben. Die Dokumentation von `Object.equals()` sagt dazu:

„The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (`x == y` has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.“

17.1.1.2 Das Problem gleicher Referenzen

Im letzten Code-Abschnitt lässt sich die main-Methode den Motor des Prototypen zurückgeben und setzt dort eine andere Kennung. Erneut lassen Sie sich die Daten der Fahrzeuge auf der Konsole ausgeben.

```
motor = prototyp.getMotor();
motor.setKennung("Marshall Motors");
System.out.println("Prototyp: " + prototyp);
System.out.println("Neues Auto: " + neuesAuto);
```

Auf der Konsole wird nun (gekürzt) ausgegeben:

```
Prototyp: (...) Motorisierung: Marshall Motors mit 100 PS und ...
Neues Auto: (...) Motorisierung: Marshall Motors mit 100 PS und ...
```

Eine Änderung, die Sie am Prototyp vorgenommen haben, schlägt bis zum Klon durch. Sie müssen kritisch hinterfragen, ob das so gewollt ist. Alle Arbeitnehmer einer Firma – die Prototypen und die Klone – haben den gleichen Arbeitgeber, und wenn er heute „Dr. Z“ und morgen „Dr. Q“ heißt, betrifft das alle Arbeitnehmer. Hier ist eine Änderung an einem referenzierten Objekt eines Prototyps sicher für alle Klone wichtig. Es ist aber auch der umgekehrte Fall denkbar: Dolly¹ und das Prototyp-Schaf dürfen die gleiche DNA haben. Es darf aber keinesfalls passieren, dass Sie Dolly scheren und das Ursprungsschaf dann ebenfalls kahlgeschoren auf der Wiese steht. Änderungen am Klon oder am Prototyp dürfen in diesem Fall auf das jeweils andere Exemplar keinen Einfluss haben. Aber lassen Sie uns zunächst betrachten, warum der Motor bei beiden Autos derselbe ist, obwohl Sie nur an einem die Änderung vorgenommen haben.

¹ [https://de.wikipedia.org/wiki/Dolly_\(Schaf\)](https://de.wikipedia.org/wiki/Dolly_(Schaf)).

17.1.1.3 Was passiert beim Klonen

Das Klonen habe ich im oberen Beispiel an die Standardimplementierung der Methode `clone()` delegiert:

```
public Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}
```

Diese Standardimplementierung kopiert das zu klonende Objekt bitweise. Dabei werden die Inhalte der Variablen unverändert mitkopiert. Was passiert dann bei einer Variablen mit primitivem Datentyp? An der Stelle, wo Sie zum Beispiel eine `int`-Variable anlegen, wird nach einer Zuweisung der Wert der Zahl abgelegt. Die Variable erleichtert Ihnen lediglich den Zugriff auf die Speicherstelle innerhalb des Objekts. Wenn Sie der Variablen einen neuen Wert zuweisen, wird die Speicherstelle innerhalb des Objekts überschrieben. Auf den Klon, der seinen eigenen Speicherbereich hat, hat diese Zuweisung keinen Einfluss. Wenn Sie eine Variable mit einem komplexen Datentyp haben, wird an der Stelle der Variablen eine Referenz auf das zugewiesene Objekt hinterlegt. Das Objekt wird auf dem Heap gespeichert. Und diese Referenz wird genauso kopiert wie der Wert der primitiven Variablen. Also halten zwei Objekte unabhängig voneinander eine Referenz auf das gleiche Objekt. Unkritisch ist das, wenn Sie immutable Objekte referenzieren. Wenn Klon und Prototyp auf das gleiche `String`-Objekt verweisen und Sie diesen `String` entweder auf dem Klon oder auf dem Prototyp ändern, wird dort ein neues `String`-Objekt angelegt und referenziert. Das Vorgehen der bitweisen Kopie wird *shallow copy* oder *flache Kopie* genannt. Wenn Sie Arrays oder Collections klonen, erhalten Sie immer eine flache Kopie. Das Beispielprojekt `Test_Sammlungen` demonstriert Ihnen die flache Kopie anhand mehrerer Beispiele.

Das kann wie im Beispiel des gemeinsamen Arbeitgebers gewünscht sein. Es kann aber auch sein, dass Sie keine zwei geschorenen Schafe auf der Wiese stehen haben möchten. Einen Motor können Sie nur in ein einziges Auto einbauen. Es bietet sich also an, die Tiefe der Kopie zu ändern und eine *deep copy*, eine tiefe Kopie zu erstellen. Dazu verfolgen Sie rekursiv alle referenzierten Objekte und erstellen Kopien von ihnen.

Die Klasse `Object` definiert die protected Methode `clone()` mit. Unterklassen machen diese Methode public, sie überschreiben die Methode mit `super.clone()`. Die Standardimplementierung der Methode in `Object` fordert Speicherplatz für den Laufzeityp des Objekts an. Anschließend wird das Objekt bitweise kopiert. Wenn das zu klonende Objekt nicht vom Typ `Cloneable` ist, wirft die Standardimplementierung eine Exception. Sie als Programmierer sind dafür verantwortlich, die Methode `clone()` sinnvoll zu überschreiben. Das bedeutet insbesondere auch, dass Sie entscheiden müssen, ob Sie tief oder flache Kopien erstellen. Der Aufrufer der Ihnen überschriebenen `clone`-Methode kann nicht erkennen, ob Sie tief oder flach kopieren. Sie sollten also Ihr Vorgehen in jedem Fall in der JavaDoc dokumentieren, damit keine Unsicherheiten bzgl. des Kopierverhaltens beim Verwender auftreten.

17.1.2 In Vererbungshierarchien klonen

Es ist einzig und allein Ihnen überlassen, wie Sie die clone-Methode implementieren. Denkbar sind daher auch die folgenden dargestellten Lösungen. Die clone-Methode der Klasse Auto könnte mit dem new-Operator ein neues Auto-Objekt erzeugen. Die Anzahl Sitze und die Referenz auf die Enum werden kopiert; ein neuer Motor (hier kein Klon!) wird angelegt.

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    var neuerMotor = new Motor("Bishop Motors", 80, 1.4);  
    var auto = new Auto(neuerMotor, Edition.TINY, 4);  
    return auto;  
}
```

Einen ähnlichen Effekt erreichen Sie mit einem CopyConstructor. Auch hierbei wird eine neue Auto-Instanz erzeugt; der Prototyp übergibt sich beim Klonen dann selbst als Argument an den privaten Konstruktor. Im Beispielprojekt Autofabrik_CopyConstructor finden Sie diesen Code:

```
public class Auto implements Cloneable {  
    private Motor motor;  
    private Edition edition;  
    private final int anzahlSitze;  
  
    Auto(Motor motor, Edition edition, int anzahlSitze) {  
        this.motor = motor;  
        this.edition = edition;  
        this.anzahlSitze = anzahlSitze;  
    }  
  
    private Auto(Auto auto) {  
        this.motor = new Motor("Bishop Motors", 80, 1.4);  
        this.anzahlSitze = auto.anzahlSitze;  
        this.edition = auto.edition;  
    }  
  
    @Override  
    public Object clone() throws CloneNotSupportedException {  
        return new Auto(this);  
    }  
  
    // ... gekürzt  
}
```

Bitte betrachten Sie diese beiden Ansätze sehr skeptisch sie sind nicht unproblematisch. Gehen Sie davon aus, dass Sie irgendwann mit Vererbung zu tun haben – Auto soll z. B. Superklasse für Roadster sein. Wie werden die Sub-Klassen geklont? Der Roadster könnte versuchen, mit `super.clone()` einen Klon anzufordern – so, wie er es in der vorigen Version auch gemacht hat. Die `clone`-Methode von Auto würde ein Objekt vom dynamischen Typ Auto zurückliefern, und das kann nicht in einen Subtyp, beispielsweise Roadster, gecastet werden; ein entsprechender Cast würde eine `ClassCastException` auslösen.

Denkbar wäre es, den dynamischen Typ abzufragen und den entsprechenden Konstruktor aufzurufen:

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    if(auto.getClass == Roadster.class)  
        return new Roadster(this);  
    else  
        return new Auto(this);  
}
```

Jede neue Subklasse von Auto würde einen neuen `if`-Zweig bekommen – das ist Horror! Es muss aber sichergestellt sein, dass der Klon dem Typ der Subklasse entspricht, da ein Klon die Bedingung `prototyp.getClass() == klon.getClass()` erfüllen soll. Wenn eine Klasse möglicherweise Subklassen hat, sie also nicht final ist, muss sie die `clone`-Methode also anders implementieren. Da eine Superklasse niemals alle Subklassen kennen kann, darf es nicht ihre Aufgabe sein, den Laufzeittyp des zu klonenden Objektes zu ermitteln. Den Laufzeittyp kann nur die Standardimplementierung von `Object.clone()` berücksichtigen. Eine Klasse, die nicht final ist, sollte also das zurückzugebende Objekt mit `super.clone()` anfordern.

Möglich ist natürlich, dass eine Superklasse private Datenfelder ändern muss, bevor der Klon zurückgegeben wird. Im Beispielprojekt `Autofabrik_Superklasse` erbt die Klasse `Roadster` von `Auto`. Zusätzlich wollen Sie erreichen, dass der `Roadster` beim Klonen einen neuen eigenen Motor bekommt. Wie erreichen Sie dieses Ziel? In dieser Projektversion habe ich die `setMotor`-Methode entfernt. Also kann ein neues `Auto` nur auf drei Wegen einen neuen Motor bekommen: Entweder der Konstruktor wird entsprechend parametriert, ein anderes `Auto`-Objekt ändert den Motor oder das Objekt ändert seinen Motor selbst. Einen Konstruktor werden Sie nicht aufrufen. Also muss es eine andere Lösung geben. Schauen Sie jetzt in das Beispielprojekt `Autofabrik_Superklasse`.

Der Client ruft die `clone`-Methode der Klasse `Roadster` auf, die ihrerseits die `clone`-Methode der Superklasse aufruft. In der Klasse `Roadster` finden Sie:

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}
```

Und in der Klasse Auto dann:

```
@Override
public Object clone() throws CloneNotSupportedException {
    var klon = super.clone();
    var autoKlon = (Auto) klon;
    var neuerMotor = new Motor("Roadster Star", 80, 1.4);
    autoKlon.motor = neuerMotor;
    return autoKlon;
}
```

Die `clone`-Methode von `Auto` ruft zuerst die `clone`-Methode von `Object` auf. Sie erhält ein Objekt vom dynamischen Typ `Roadster` zurück – Sie erinnern sich: `Object.clone()` liefert den Laufzeittyp zurück. Dieses Objekt wird auf den Typ `Auto` gecastet; das ist nach dem Liskov'schen Substitutionsprinzip erlaubt – eine Subklasse soll ihre Basisklasse ersetzen können. Jetzt hat die `Auto`-Instanz Zugriff auf das private Datenfeld `motor` und weist einen neuen Motor zu. Danach wird das geklonte Objekt zurückgegeben. Der Client erhält ein Objekt vom dynamischen Typ `Roadster` mit einem nagelneuen Motor.

Wie wird das in Praxis gemacht? In der Praxis ist es sinnvoll, wenn Klassen, von denen andere Klassen abgeleitet werden, die `clone`-Methode nur `protected` überschreiben und das Interface `Cloneable` nicht implementieren. Unterklassen sind dann frei, Klonen zu unterstützen oder nicht.

Bisher war die `clone`-Methode entsprechend der Signatur der Klasse `Object` implementiert: Der Rückgabewert ist vom Typ `Object`; außerdem wird die `CloneNotSupportedException` propagiert. Lassen Sie uns diese beiden Punkte noch genauer betrachten.

Wenn eine Methode ein Objekt von einem bestimmten Typ zurückgibt, muss der Client sie zuerst auf den erwarteten Typ casten. Seit Java 5 sind kovariante Rückgabetypen erlaubt; die `clone`-Methode darf also ein `Auto`-Objekt zurückgeben, wie der folgende Code-Schnipsel zeigt.

```
@Override
public Auto clone() throws CloneNotSupportedException {
    var klon = super.clone();
    var autoKlon = (Auto) klon;
    var neuerMotor = new Motor("Roadster Star", 80, 1.4);
    autoKlon.motor = neuerMotor;
    return autoKlon;
}
```

Wenn Sie diese Änderung in der Klasse `Auto` durchführen, müssen Sie das aber auch in der Unterklasse `Roadster` tun. Auch dort muss der Rückgabewert vom Typ `Auto` sein, nicht vom Typ `Roadster`.

Über das andere Thema, die CloneNotSupportedException wird in den Foren leidenschaftlich gestritten; die herrschende Meinung in der Literatur ist über diese Exception nicht glücklich. Auf der einen Seite gibt es eine Methode `clone()`, die von der Klasse überschrieben wird; auf der anderen Seite sagt genau diese Methode, dass sie Klonen eventuell gar nicht unterstützt. Geworfen wird die CloneNotSupportedException, wenn die Standardimplementierung der Methode `clone()` feststellt, dass das zu klonende Objekt nicht vom Typ `Cloneable` ist. Relevant ist dieser Fall jedoch nur, wenn in einer Vererbungshierarchie keine der Superklassen das Interface `Cloneable` implementiert. Dieser Fehler ist jedoch zur Compile-Zeit zu erkennen und heilbar; es ist daher unverhältnismäßig, eine checked Exception zu propagieren. Für die Praxis bietet es sich daher an, die Exception innerhalb der `clone`-Methode zu fangen und stattdessen einen AssertionError zu werfen:

```
public Auto clone() {  
    try {  
        var klon = super.clone();  
        var autoKlon = (Auto) klon;  
        var neuerMotor = new Motor("Roadster Star", 80, 1.4);  
        autoKlon.motor = neuerMotor;  
        return autoKlon;  
    } catch (CloneNotSupportedException exc) {  
        throw new AssertionError();  
    }  
}
```

Im folgenden Abschnitt entwerfen Sie ein größeres Projekt, das auf Prototypen aufbaut.

17.2 Ein größeres Projekt

Sie zeichnen verschiedenfarbige Kreise und verbinden diese mit Linien. Sowohl Kreise als auch Linien sind Grafiken. Sie können sowohl Kreise als auch Linien löschen. Linien, die nach dem Löschen keine zwei Grafiken mehr verbinden, werden ebenfalls gelöscht. Auf Abb. 17.1 sehen Sie das Programm in Aktion.

17.2.1 Besprechung der ersten Version

Mit einem rechten Mausklick öffnen Sie das Kontextmenü und wählen, ob Sie entweder eine Linie zeichnen oder einen Kreis hinzufügen oder eine Grafik auswählen möchten. Ihnen stehen verschiedene Kreise zur Verfügung: rote, blaue, grüne und so weiter. Wenn Sie auswählen, dass Sie eine Linie zeichnen möchten, klicken Sie entweder auf einen Kreis oder eine Linie und ziehen eine Linie zur Zielgrafik, die auch entweder eine Linie

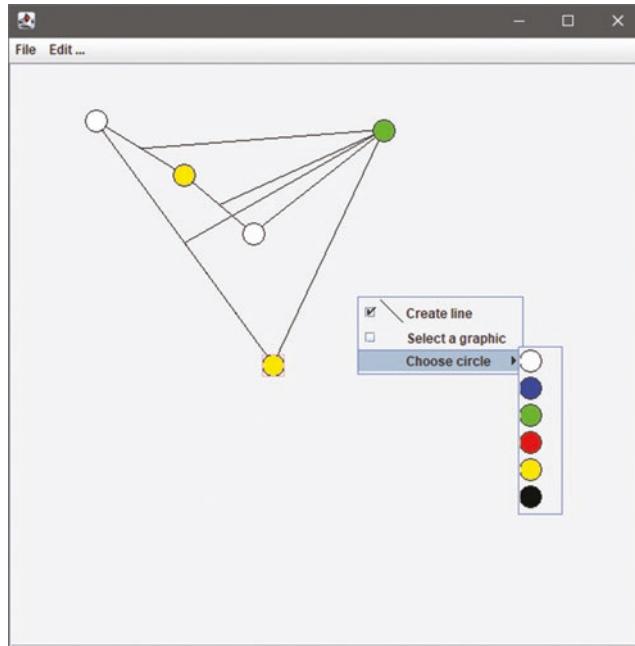


Abb. 17.1 Beispielprogramm GraphEditor

oder ein Kreis sein kann. Wenn Sie die Option Select a graphic auswählen, können Sie entweder einen Kreis oder eine Linie auswählen. Wenn Sie eine Auswahl getroffen haben, wählen Sie im Hauptmenü Edit und Delete selected. Die gewählte Grafik wird dann gelöscht, wobei alle betroffenen Verbindungen – Linien – ebenfalls gelöscht werden. Wenn Sie einen Kreis ausgewählt haben, können Sie diesen auch verschieben.

Sie können mit dieser Version des Grafikeditors das Autobahnnetz von Deutschland, den Metroplan von Paris oder was auch immer modellieren. In weiteren Ausbaustufen könnten Sie neben Kreisen auch Kästen hinzufügen und das Framework zu einem kleinen UML-Editor umbauen. Bevor ich Ihnen die wesentlichen Klassen des Beispielprojekts GraphEditor_1 zeige, öffnen Sie es am besten mit NetBeans und machen sich mit der Handhabung vertraut.

Das Programm wird mit der main-Methode in der Klasse App1Start gestartet. Diese Methode erzeugt ein JFrame und fügt dort das Hauptmenü und die Zeichenfläche, eine Instanz der Klasse PanelCanvas, hinzu. Die Klasse PrototypeManager erzeugt die Prototypen der zu zeichnende Kreise. In der Praxis kann diese Klasse beliebig groß werden; umfangreiche Initialisierungsroutinen zum Erstellen der Prototypen werden dorthin ausgelagert. Die Klasse Diagram ist das Datenmodell des Editors. Hier werden die Grafiken in einer List gespeichert und verwaltet. Das Interface GraphicIF definiert Methoden, die von allen Grafiken (Kreise und Linien) implementiert werden müssen. Das hiervon abgeleitete Interface RelationIF beschreibt Methoden, die nur Linien imple-

mentieren müssen. Die Klassen `Circle` und `Relation` implementieren diese Schnittstellen und definieren die vorgeschriebenen Methoden. Jede Grafik muss beispielsweise das sie umschließende Rechteck beschreiben können und eine bestimmte Farbe und ihre Position speichern. Am wichtigsten ist vielleicht, dass eine Grafik sich selbst zeichnen und vor allem sich klonen können muss. Linien verbinden immer zwei andere Grafiken; sie müssen also zusätzlich die Anfangs- und die Endgrafik speichern und zurückgeben können.

Auf der `PanelCanvas` wird das Diagramm gezeichnet. Der `MouseAdapter` überschreibt einige `EventHandler`. Das `PanelCanvas` definiert außerdem einige Aktionen: Die `Create_Line_Action` setzt ein Flag, dass eine Linie gezeichnet werden soll. Die `Select_Action` legt fest, dass eine Grafik ausgewählt werden kann. Die Aktionen werden an `JCheckBox-MenuItem`-Instanzen übergeben und in das Kontextmenü des Panels eingehängt. Die Aktionen werden in eine `ButtonGroup` zusammengefasst, so dass immer nur eine ausgewählt werden kann.

```
private final ButtonGroup group = new ButtonGroup();  
...  
group.add(mnCreateLine);  
group.add(mnSelect);
```

Im Konstruktor werden außerdem die Prototypkreise als Menüpunkte angelegt. Zuerst werden vom Prototypenverwalter alle Prototypen abgefragt. Dann wird iterativ für jeden Prototyp ein eigener Menüpunkt angelegt; beim Aufruf wird der Prototyp an das Datenfeld `nextGraphic` übergeben.

```
public PanelCanvas(Diagram diagram) {  
    // ... gekürzt  
  
    for (final GraphicIF tempGraphic : PrototypeManager.getPrototypes())  
        addPrototype(tempGraphic);  
    // ... gekürzt  
}  
  
//... gekürzt  
  
private void addPrototype(final GraphicIF prototype)  
{  
    final var drawAction = new AbstractAction() {  
        @Override  
        public void actionPerformed(ActionEvent event) {  
            createLine = false;  
            nextGraphic = prototype;  
        }  
    };
```

```

var mnNewGraphic = new JCheckBoxMenuItem(drawAction);
mnCirclePrototypes.add(mnNewGraphic);
group.add(mnNewGraphic);
var icon = prototype.getIcon();
mnNewGraphic.setIconTextGap(0);
mnNewGraphic.setIcon(icon);
}

```

Wenn Sie einen Kreis ausgewählt haben und auf die Zeichenfläche klicken, wird der Eventhandler `mouseClicked` aufgerufen, der vom Datenfeld `nextGraphic` den Prototyp erfragt, ihn klonen und den Klon im Diagramm speichert.

```

@Override
public void mouseClicked(MouseEvent event) {
    mousePosition = event.getPoint();
    if (nextGraphic != null)
        try {
            var newGraphic = (GraphicIF) nextGraphic.clone();
            selected = newGraphic;
            diagram.add(newGraphic, mousePosition);
        } catch (CloneNotSupportedException ex) {
            new ErrorDialog(ex);
        }
    else
        // Bei der Maus-Position ein Grafikobjekt suchen und
        // auswählen
        selected = diagram.findGraphic(mousePosition);
    repaint();
}

```

Das sind die wesentlichen Punkte des Programms. Alles Weitere sind Spielereien, die die Handhabung des Programms erleichtern. Bitte analysieren Sie selbstständig den Code des Programms.

17.2.2 Die zweite Version – deep copy

In der zweiten Version (Beispielprojekt GraphEditor_2) soll es möglich sein, das gesamte Diagramm zu klonen. Auf dem JFrame werden zwei Canvas-Instanzen platziert. Wenn Sie im Menüpunkt File auf Clone Diagram klicken, wird das Diagramm von der linken Seite auf die rechte kopiert. Sie können nun beide Diagramme unabhängig voneinander verändern. Abb. 17.2 zeigt Ihnen die Oberfläche der neuen Version.

Der Menüpunkt wird in der Klasse `AppStart` definiert und im Menü eingehängt. Er fragt bei der linken Zeichenfläche das Diagramm ab und setzt es auf der rechten Seite ein.

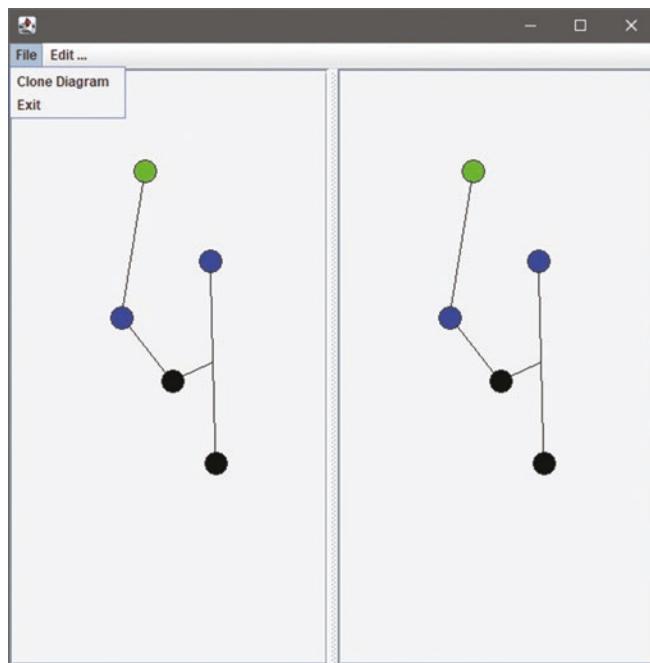


Abb. 17.2 Geklontes Diagramm

Die Klasse `PanelCanvas` definiert in dieser Version die Methode `getDiagramAsClone()`. Zuerst wird das `ByteArrayOutputStream` `baos` erzeugt und dem `ObjectOutputStream` `oos` übergeben. In das `baos` wird das Diagramm serialisiert. Anschließend erzeugen Sie den `ByteArrayInputStream` `bais`. Diesem werden die Daten des `baos` übergeben. Der `bais` wird dem `ObjectInputStream` `ois` übergeben, der das serialisierte Diagramm deserialisiert. Das so erzeugte Objekt wird auf ein Diagramm gecastet und zurückgegeben. Serialisierung und Deserialisierung bewirken, dass die Objekte inklusive der von ihnen referenzierten Objekte unabhängig voneinander sind.

```
public Diagram getDiagramAsClone() {  
    Diagram clone = null;  
    try {  
        this.nextGraphic = null;  
        this.selected = null;  
        this.createLine = false;  
        ObjectOutputStream oos;  
        ByteArrayInputStream bais;  
        ObjectInputStream ois;  
        try ( var baos = new ByteArrayOutputStream())  
        {  
            oos = new ObjectOutputStream(baos);  
            // serialisieren  
        }  
        ois = new ObjectInputStream(bais);  
        // deserialisieren  
        clone = (Diagram) ois.readObject();  
    } catch (IOException | ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
    return clone;  
}
```

```
        oos.writeObject(diagram);
        bais = new ByteArrayInputStream(baos.toByteArray());
        ois = new ObjectInputStream(bais);
        clone = (Diagram) ois.readObject();
    }
    oos.close();
    bais.close();
    ois.close();
} catch (IOException ex) {
    new ErrorDialog(ex);
} finally {
    return clone;
}
}
```

In der letzten Version des Programms werden Sie eigene Prototypen entwerfen.

17.2.3 Eigene Prototype definieren

Eine Besonderheit des Prototype Patterns ist, dass Sie eigene Prototypen zur Laufzeit entwickeln und hinzufügen können. Das Beispielprojekt GraphEditor_3 beruht auf der ersten Projektversion. Sie kennen also den größten Teil der Funktionalität bereits. Es gibt im Menü Edit den neuen Menüpunkt New Prototype. Wenn Sie diesen Menüpunkt auswählen, erscheint ein Farbauswahldialog. Wählen Sie eine Farbe aus und klicken Sie auf Ok. Ein Kreis mit der gewünschten Farbe steht Ihnen jetzt als Prototyp im Kontextmenü zur Verfügung. In der Klasse ApplStart wird die Aktion newPrototypeAction definiert, die auf der Zeichenfläche die Methode createPrototype() aufruft. Innerhalb der Methode wird ein JColorChooser aufgerufen. Mit dem Rückgabewert wird ein neuer Kreis erzeugt und als Prototyp sowohl beim Kontextmenü als auch im Prototypen-Manager eingehängt.

```
public void createPrototype() {
    var newColor = JColorChooser.
        showDialog(PanelCanvas.this, "New Circle", Color.cyan);
    if (newColor != null) {
        GraphicIF newPrototype = new Circle(newColor);
        addPrototype(newPrototype);
        PrototypeManager.add(newPrototype);
    }
}
```

17.3 Prototype – Das UML-Diagramm

Aus dem Beispielprojekt GraphEditor_3 sehen Sie in Abb. 17.3 zunächst das UML-Diagramm der Pakete `graphics` und `prototype`, und in Abb. 17.4 dann das UML-Diagramm des Pakets `delegate` und der eigentlichen Anwendung.

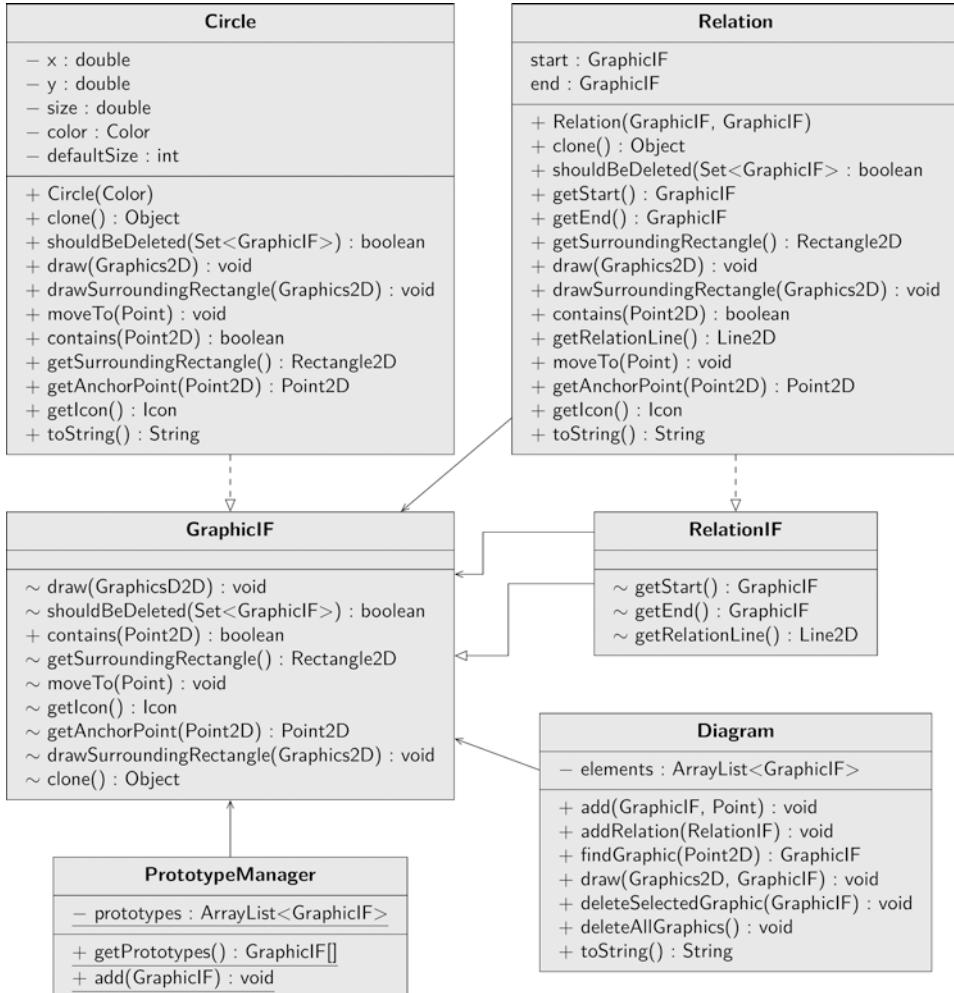


Abb. 17.3 UML-Diagramm des Prototype Patterns (Beispielprojekt GraphEditor_3, Pakete `graphics` und `prototype`)

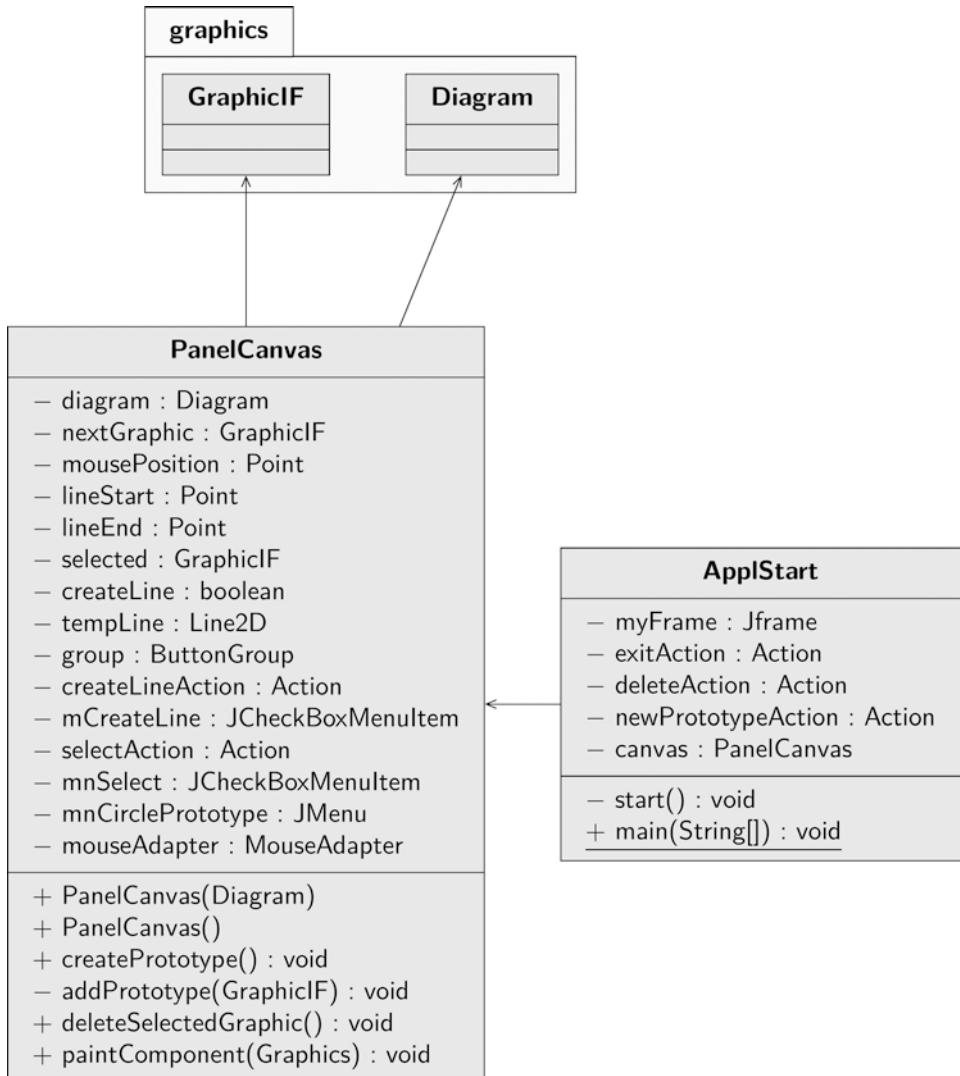


Abb. 17.4 UML-Diagramm des Prototype Pattern (Beispielprojekt GraphEditor_3, Paket delegate und Anwendung)

17.4 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Das Prototype Pattern versteckt die Objekterzeugung vor dem Client.
- Sie möchten zur Laufzeit unterschiedliche Objekte (Produkte) haben.

- Von jedem Produkt wird ein Prototyp erstellt, der geklont wird, wenn ein neues Objekt angefordert wird.
- Die Prototypen werden vom Prototypen-Manager verwaltet.
- Jeder Prototyp muss die Methode `clone()` überschreiben, wenn das Cloneable-Interface verwendet wird.
- Die Methode `clone()` wird in `Object` mit der Sichtbarkeit `protected` definiert.
- Die Standardimplementierung dieser Methode führt zu einer flachen Kopie des Objekts; referenzierte Objekte werden nicht geklont.
- Sammlungen klonen standardmäßig ebenfalls als flache Kopie.
- Superklassen sollten `clone()` zumindest `protected` überschreiben, um Subklassen eine sinnvolle Implementierung zu ermöglichen.

Zweckbeschreibung

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

„Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototyps.“



Aus der Reihe der Erzeugungsmuster kennen Sie das Singleton Pattern, die zwei Fabriken und das Prototype Pattern. Das Builder Pattern ist das letzte Muster, mit dem Sie Objekte erzeugen, ohne den new-Operator zu verwenden. Beim Builder Pattern haben Sie ein Objekt, das komplex oder kompliziert zu konstruieren ist. Dieses Objekt kann nicht in einem Durchgang erzeugt werden, sondern durchläuft einen aufwendigen Konstruktionsprozess.

18.1 Ein Objekt erzeugt andere Objekte

Um in die Materie hineinzufinden, fangen wir mit einem einfachen Beispiel an. Wenn Sie online eine Reise buchen wollen, können Sie eine Vielzahl von Such-Parametern vorgeben. Am wichtigsten dürfte wohl sein, in welchem Zeitraum und wie viele Tage Sie verreisen möchten. Außerdem ist interessant, mit wie vielen Personen Sie verreisen und wie viele Kinder (unter 14 Jahren) Sie mitnehmen. Sicher werden Sie auch angeben wollen, wie viele Sterne Ihre Unterkunft haben soll. Und wichtig sind die Bewertung der Unterkunft und die Weiterempfehlungsrate. Vielleicht wollen Sie auf Nummer Sicher gehen und möchten nur Hotels angezeigt bekommen, die bereits von mindestens x Gästen bewertet wurden. Alle diese Suchparameter speichern Sie in den Datenfeldern eines Reise-Objekts.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_18

18.1.1 Telescoping Constructor Pattern

Schauen Sie sich das Beispielprojekt Telescoping_Constructor_Pattern an. Die Suchparameter übergeben Sie dem Konstruktor.

```
public class Reise {  
    public final LocalDate vonDatum;  
    public final LocalDate bisDatum;  
    public final int dauer;  
    public final int anzahlPersonen;  
    public final int anzahlKinder;  
    public final int anzahlSterneMindestens;  
    public final int weiterempfehlungMindestens;  
    public final int bewertung;  
    public final int anzahlBewertungenMindestens;  
  
    Reise(LocalDate vonDatum, LocalDate bisDatum, int dauer,  
          int anzahlPersonen, int anzahlKinder, int anzahlSterne,  
          int weiterempfehlung, int bewertung, int anzahlBewertungen)  
    {  
        this.vonDatum = vonDatum;  
        this.bisDatum = bisDatum;  
        this.dauer = dauer;  
        this.anzahlPersonen = anzahlPersonen;  
        this.anzahlKinder = anzahlKinder;  
        this.anzahlSterneMindestens = anzahlSterne;  
        this.weiterempfehlungMindestens = weiterempfehlung;  
        this.bewertung = bewertung;  
        this.anzahlBewertungenMindestens = anzahlBewertungen;  
    }  
}
```

Der Aufruf dieses Konstruktors ist blander Horror! Die main-Methode der Klasse AppStart demonstriert dies:

```
public static void main(String[] args) {  
    var reise = new Reise(LocalDate.now(), LocalDate.now(), 7, 2, 3, 3,  
                        80, 5, 30);  
    System.out.println(reise);  
}
```

Das Problem ist nicht nur, dass Sie sehr viele Parameter mitgeben müssen – diese sind darüber hinaus vom gleichen Typ. Sie werden ohne Dokumentation niemals herausfinden, wofür welcher Parameter steht. Fehler werden sich bei so einem Aufruf sehr zuverlässig

einschleichen. Eine Lösung könnte sein, dass Sie den Konstruktor überladen. Wenn der Client keine Bewertung und auch keine Mindestanzahl von Bewertungen voraussetzt, soll er diese weglassen dürfen. Der neue Konstruktor ruft den eben vorgestellten mit `this()` auf und übergibt Standardwerte:

```
Reise(LocalDate vonDatum, LocalDate bisDatum, int dauer,
       int anzahlPersonen, int anzahlKinder, int anzahlSterne,
       int weiterempfehlung) {
    this(vonDatum, bisDatum, dauer, anzahlPersonen, anzahlKinder,
         anzahlSterne, weiterempfehlung, 0, 0);
}
```

Der Client erzeugt ein Reise-Objekt jetzt mit einer etwas kürzeren Parameterliste:

```
public static void main(String[] args) {
    var reise = new Reise(LocalDate.now(), LocalDate.now(), 7, 2,
3, 3, 80);
    System.out.println(reise);
}
```

Ist dieser Aufruf ansprechender oder eindeutiger? Nein! Diese Lösung, die Telescoping Constructor Pattern heißt, hat Einzug in die Liste der Antipatterns gehalten. Es ist nie eine gute Idee, zu viele Parameter an eine Methode übergeben zu müssen – umso mehr, als sie vom gleichen Typ sind. Lassen Sie uns eine andere Lösung betrachten.

18.1.2 JavaBeans Pattern

In Java-Einführungen finden Sie den Rat, dass Datenfelder gekapselt werden müssen und nur über Getter und Setter ausgelesen und verändert werden dürfen – und daran orientiert sich die Version im Beispielprojekt JavaBeans_Pattern. Informationen, auf die bei der Konstruktion nicht verzichtet werden kann, also Zeitrahmen, Dauer und Anzahl Reisender, werden in finalen Datenfeldern gespeichert, die vom Konstruktor initialisiert werden. Die restlichen Informationen werden mit Standardwerten vorbelegt und können mit Settern optional überschrieben werden.

```
public class Reise {
    public final LocalDate vonDatum;
    public final LocalDate bisDatum;
    public final int dauer;
    public final int anzahlPersonen;
    private int anzahlKinder = 0;
    private int anzahlSterneMindestens = 0;
```

```

private int weiterempfehlungMindestens = 0;
private int bewertung = 0;
private int anzahlBewertungenMindestens = 0;

Reise(LocalDate vonDatum, LocalDate bisDatum, int dauer,
       int anzahlPersonen) {
    this.vonDatum = vonDatum;
    this.bisDatum = bisDatum;
    this.dauer = dauer;
    this.anzahlPersonen = anzahlPersonen;
}

public void setAnzahlKinder(int anzahlKinder) {
    this.anzahlKinder = anzahlKinder;
}

// ... gekürzt
}

```

Der Client kann nun sehr viel übersichtlicher eine Instanz der Klasse Reise erzeugen.

```

public static void main(String[] args) {
    var reise = new Reise(LocalDate.now(), LocalDate.now(), 14, 2);
    reise.setAnzahlSterneMindestens(3);
    reise.setWeiterempfehlungMindestens(80);
    System.out.println(reise);
}

```

Der Code ist zwar jetzt sehr viel sprechender. Allerdings sind die anderen Datenfelder nicht final und können es auch nicht sein. Eine nachträgliche Veränderung der Werte ist somit erlaubt, auch wenn sie vielleicht nicht gewünscht sein sollte. Die Vorgehensweise entspricht nicht ganz der JavaBeans-Spezifikation, sondern übernimmt nur das wesentliche Prinzip. Die JavaBeans-Spezifikation sieht vor, dass Sie immer einen Default-Konstruktor definieren. Übrigens werden Sie in der Literatur auch gelegentlich auf Meinungen treffen, die die JavaBeans-Spezifikation in das Reich der Antipatterns verschiebt – aus den eben genannten Gründen.

18.1.3 Builder Pattern

Da der Werkzeugkasten der OOP keine befriedigende Lösung erlaubt, wird es Zeit für ein Pattern. Das Builder Pattern setzt ein Objekt voraus, dessen Aufgabe sich darauf beschränkt, ein anderes Objekt zu konstruieren. In der Version im Beispielprojekt Builder_Pattern realisieren Sie jetzt die Reise-Klasse, indem Sie sämtliche Informationen in finalen

Datenfeldern speichern. Der Konstruktor ist wie bei Singleton private, so dass ein Objekt nur innerhalb der Klasse erzeugt werden kann. Als Parameter erwartet der Konstruktor ein Objekt vom Typ Builder.

```
public class Reise {  
    // ... gekürzt  
  
    private final LocalDate vonDatum;  
    private final LocalDate bisDatum;  
    public final String von;  
    public final String bis;  
    public final int dauer;  
    public final int anzahlPersonen;  
    public final int anzahlKinder;  
    public final int anzahlSterneMindestens;  
    public final int weiterempfehlungMindestens;  
    public final int bewertung;  
    public final int anzahlBewertungenMindestens;  
  
    private Reise(Builder builder) {  
        this.vonDatum = builder.vonDatum;  
        this.bisDatum = builder.bisDatum;  
        var formatter = DateTimeFormatter.ofPattern("dd.MM.YYYY");  
        VON = dateFormat.format(vonDatum);  
        BIS = dateFormat.format(bisDatum);  
        this.dauer = builder.dauer;  
        // ... gekürzt  
    }  
}
```

Der Typ `Builder` wird als statische innere Klasse der `Reise` definiert. Die Datenfelder der `Reise` werden hier ebenfalls deklariert. Die wichtigen Datenfelder werden im Konstruktor initialisiert, die übrigen bekommen Standardwerte. Für jedes Datenfeld gibt es einen Setter, der entgegen der Sprachkonvention nur den Namen des Felds wiederholt. Innerhalb des Setters wird das entsprechende Datenfeld mit einem Wert belegt; schließlich gibt der Setter die `Builder`-Instanz zurück.

```
public static class Builder {  
    private LocalDate vonDatum;  
    private LocalDate bisDatum;  
    private int dauer;  
    private int anzahlPersonen;  
    private int anzahlKinder = 0;  
    private int anzahlSterneMindestens = 0;
```

```
private int weiterempfehlungMindestens = 0;
private int bewertung = 0;
private int anzahlBewertungenMindestens = 0;

public Builder(LocalDate vonDatum, LocalDate bisDatum, int dauer,
               int anzahlPersonen) {
    this.vonDatum = vonDatum;

    // ... gekürzt

}

public Builder anzahlKinder(int anzahlKinder) {
    this.anzahlKinder = anzahlKinder;
    return this;
}

public Builder anzahlSterneMindestens(int anzahlSterne) {
    this.anzahlSterneMindestens = anzahlSterne;
    return this;
}

// ... gekürzt
}
```

Die Methode `build()` erzeugt dann endlich das Reise-Objekt und übergibt dafür das Builder-Objekt an dessen Konstruktor.

```
public Reise build() {
    return new Reise(this);
}
```

Der Client erzeugt im ersten Schritt einen Builder, versorgt ihn – soweit gewünscht – mit den relevanten Daten und lässt sich erst dann ein Reise-Objekt zurückgeben.

```
public static void main(String[] args) {
    var builder =
        new Reise.Builder(LocalDate.now(), LocalDate.now(), 15, 2);
    Reise reise = builder
        .anzahlSterneMindestens(3)
        .bewertung(5)
        .anzahlKinder(0)
        .build();
    System.out.println(reise);
}
```

Diese erste Realisierung des Builder Patterns soll Ihnen ein Gefühl dafür vermitteln, dass es sinnvoll sein kann, Objekte zu haben, deren Funktionalität sich darauf beschränkt, andere Objekte zu erzeugen. Bitte wundern Sie sich nicht, dass ich aus Vereinfachungsgründen im Beispiel für das vonDatum wie für das bisDatum jeweils LocalDate.now() vorgegeben habe; in der Praxis ist es natürlich undurchführbar, in der Zeit „von heute bis heute“ 15 Tage Urlaub zu machen.

Etwas unschön ist noch das new Reise.Builder in der main-Methode. Sie können noch in der Klasse Reise mit einer statischen Methode einen Builder erzeugen:

```
public class Reise {  
    // ... gekürzt  
    public static Reise.Builder builder(LocalDate vonDatum, LocalDate  
                                         bisDatum, int dauer, int anzahlPersonen) {  
        return new Reise.Builder(vonDatum, bisDatum, dauer,  
                               anzahlPersonen);  
    }  
}
```

Wenn Sie dann in der main-Methode auf eine separate Builder-Variable verzichten wollen, funktioniert auch der folgende Code (den Sie auch im Beispiel finden):

```
public static void main(String[] args) {  
    var reise = Reise.builder(LocalDate.now(),  
                             LocalDate.now().plusDays(15), 15, 2)  
        .anzahlSterneMindestens(3)  
        .bewertung(5)  
        .anzahlKinder(0)  
        .build();  
    System.out.println(reise);  
}
```

18.2 Ein komplexerer Konstruktionsprozess

Sinn und Zweck des Builder Patterns ist es, den Konstruktionsprozess in eine eigene Klasse auszulagern. Sinn hat das immer dann, wenn die Konstruktion eines Objekts aufwendig ist oder ausgetauscht werden können soll. Ich möchte im folgenden Beispiel das Haushaltsbuch aus dem letzten Kapitel aufgreifen. Öffnen Sie dazu das Beispielprojekt HaushaltsBuilder. In der Praxis möchten Sie die verschiedenen Positionen und Kategorien sicher nicht im Quelltext hinterlegen, so wie ich es in Kap. 12 beim Composite Pattern gezeigt habe. Sie wollen die Daten vielleicht aus einer XML-Datei lesen und in einem JTree anzeigen. Das Builder Pattern ist hierfür optimal geeignet, da ein Knoten im JTree

nicht in einem Durchgang erzeugt werden kann; erst, wenn er keine weiteren Unterknoten hat, kann die Objekterstellung abgeschlossen werden.

Die XML-Datei sieht im Beispiel gekürzt so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Haushaltsbuch
[

    <!ELEMENT Haushaltsbuch (Position*, Monat+)>
    <!ELEMENT Monat (Kategorie*)>
    <!ELEMENT Kategorie (Position| Kategorie)*>
    <!ELEMENT Position EMPTY>

    <!ATTLIST Haushaltsbuch beschreibung          CDATA #REQUIRED>
    <!ATTLIST Monat beschreibung                 CDATA #REQUIRED>
    <!ATTLIST Kategorie beschreibung            CDATA #REQUIRED>
    <!ATTLIST Position beschreibung
        betrag          CDATA #REQUIRED
        erforderlich (ja|nein)      CDATA #REQUIRED
                                    "ja" >

]>
<Haushaltsbuch beschreibung = "Haushaltsbuch 2021">
    <Position beschreibung="Lebensversicherung" betrag = "-1600" erforderlich = "ja"/>
        <Monat beschreibung = "Januar">
            <Kategorie beschreibung = "Einnahmen">
                <Position beschreibung = "Hauptjob" betrag = "2000" erforderlich = "ja"/>
                    <Position beschreibung = "Vorträge" betrag = "5000" erforderlich = "ja"/>
                </Kategorie>
                <Kategorie beschreibung = "Ausgaben">
                    <Kategorie beschreibung = "Miete">
                        <Position beschreibung = "Wohnung" betrag = "-700"/>
                        <Position beschreibung = "Hobbyraum" betrag = "-150" erforderlich = "nein"/>
                    </Kategorie>
                    <Kategorie beschreibung = "Versicherungen">
                        <Position beschreibung = "KFZ" betrag = "-34.50"/>
                    </Kategorie>
                </Kategorie>
            </Monat>
        ...
    ... gekürzt

</Haushaltsbuch>
```

Ihre Aufgabe wird es nun sein, aus dem XML-Dokument ein TreeModel zu machen und dieses in einem JTree anzuzeigen.

18.2.1 XML-Dateien in ein TreeModel konvertieren

Ich habe mir zunächst die letzte Projektversion des Kapitels über das Composite Pattern genommen. Hierin habe ich ein neues Package builder angelegt, in dem Sie die abstrakte Klasse Builder finden. Builder erbt von org.xml.sax.helpers.DefaultHandler und zwingt ihre Subklassen, die Methode startElement() und endElement() zu definieren. Wenn ein Element des XML-Dokuments geöffnet bzw. geschlossen wird, wird eine dieser Methoden aufgerufen. Wenn die XML-Daten nicht valide sind, geben die Methoden warning(), error() oder fatalError() entsprechende Fehlermeldungen aus. Der Client lässt sich von einer Instanz vom Typ Builder zuerst ein Objekt erstellen und holt sich das fertige Produkt dann von dieser Instanz ab; jede Unterklasse muss also auch die Methode getProduct() anbieten.

```
public abstract class Builder extends DefaultHandler {  
    public abstract Object getProduct();  
  
    @Override  
    public abstract void startElement(String uri, String localName,  
                                      String name, Attributes attributes);  
  
    @Override  
    public abstract void endElement(String uri, String localName,  
                                    String name);  
  
    @Override  
    public void warning(SAXParseException exception) {  
        System.err.println("-> warning: " + exception.getMessage());  
    }  
  
    @Override  
    public void fatalError(SAXParseException exception) {  
        System.err.println("-> FATAL ERROR: " + exception.getMessage());  
    }  
  
    @Override  
    public void error(SAXParseException exception) {  
        System.err.println("-> error: " + exception.getMessage());  
    }  
}
```

Der erste konkrete Builder, den Sie erstellen werden, ist der TreeModelBuilder. Er hat zwei Datenfelder: den Wurzel-Knoten und eine Liste, in der alle Knoten temporär gespeichert werden. Die Methoden `startElement()` und `endElement()` sind für den eigentlichen Konstruktionsprozess zuständig. Wenn ein Element geöffnet wird, prüft die Methode `startElement()` zunächst, ob das Element „Position“ heißt. In diesem Fall werden die Attribute abgefragt und ein neues Blatt erstellt; anderenfalls wird ein neues Kompositum erzeugt. Wenn die Wurzel gleich null ist, also nur beim allerersten Element, wird das aktuell erstellte Element an das Datenfeld `wurzel` übergeben. Wenn das Datenfeld `wurzel` ungleich null ist, also ab dem zweiten Element, wird das neue Element an das letzte in der Liste gespeicherte Kompositum als Unterknoten angehängt. Anschließend wird der Knoten in der Liste gespeichert.

```
public class TreeModelBuilder extends Builder {
    private Kompositum wurzel;
    private final LinkedList<Knoten> stack = new LinkedList<>();

    @Override
    public void startElement(String uri, String localName, String name,
        Attributes attributes) {
        Knoten knoten;
        if (name.equalsIgnoreCase("Position")) {
            var tempBeschreibung = attributes.getValue("beschreibung");
            var tempBetrag = Double.parseDouble(attributes.getValue("betrag"));
            var tempErforderlich = attributes.getValue("erforderlich").
                equalsIgnoreCase("ja");
            knoten = new Blatt(tempBeschreibung, tempBetrag,
                tempErforderlich);
        } else {
            var tempBeschreibung = attributes.getValue("beschreibung");
            knoten = new Kompositum(tempBeschreibung);
        }
        if (wurzel == null)
            wurzel = (Kompositum) knoten;
        else {
            var tempKnoten = (Kompositum) stack.peekLast();
            tempKnoten.add(knoten);
        }
        stack.add(knoten);
    }
    // ... gekürzt
}
```

Die Methode `endElement()` wird aufgerufen, wenn das Element geschlossen wird. Sie kann sich darauf beschränken, das zuletzt in der Liste gespeicherte Element dort zu entfernen. Die Methode `getProduct()` erzeugt mit dem Datenfeld `wurzel` ein `MyTreeModel` und gibt dieses zurück.

```
@Override  
public void endElement(String uri, String localName, String name) {  
    stack.pollLast();  
}  
  
@Override  
public TreeModel getProduct() {  
    return new MyTreeModel(wurzel);  
}
```

Der Client erzeugt zuerst eine Builder-Instanz, in diesem Projekt eben einen `TreeModelBuilder`. Er übergibt diesen Builder an die Methode `build()`, die zuerst die XML-Datei liest, sie in einen String umwandelt und ihn mit dem Builder als Handler parst.

```
Haushalt() {  
    var builder = new TreeModelBuilder();  
    build(builder);  
    var treeModel = (TreeModel) builder.getProduct();  
    var frmMain = new JFrame("Builder Pattern Demo");  
    frmMain.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    var trvHaushaltsbuch = new JTree(treeModel);  
    trvHaushaltsbuch.setCellRenderer(new MyTreeCellRenderer());  
    trvHaushaltsbuch.setEditable(true);  
    trvHaushaltsbuch.setCellEditor(new MyTreeCellEditor());  
    var scrTree = new JScrollPane(trvHaushaltsbuch);  
    frmMain.add(scrTree);  
    frmMain.setSize(500, 500);  
    frmMain.setVisible(true);  
}  
  
public void build(Builder builder) {  
    var filePath = Paths.get(datei);  
    try {  
        var content = Files.readString(filePath);  
  
        var factory = SAXParserFactory.newInstance();  
        factory.setValidating(true);  
        var saxParser = factory.newSAXParser();  
  
        saxParser.parse(new InputSource(new StringReader(content)),
```

```

        builder);
    } catch (IOException | ParserConfigurationException | SAXException ex)
    {
        ex.printStackTrace();
    }
}

```

Background Information

Wenn Sie das Beispiel kritisch durchsehen, fällt Ihnen ein Nachteil des Patterns auf: Der Builder ist stark an das zu erzeugende Objekt gebunden. Da abgefragt wird, ob ein Element „Position“ heißt, ist dieser Builder an die gegebene DTD gebunden. Ebenso muss der Builder genaue Kenntnis von der Konstruktion von Blatt- und Kompositum-Klassen haben.

Auf der anderen Seite sehen Sie aber auch den großen Vorteil: Sie können jede beliebige XML-Datei, die der DTD entspricht, an den Builder übergeben – Sie werden immer ein passendes Tree-Model daraus generieren können.

Ihr Programm ist jetzt sehr viel stärker modularisiert. In der Folge kann die Routine, die das XML-Dokument liest, durch eine Routine ersetzt werden, die das XML-Dokument aus einer anderen Quelle, beispielsweise dem Netzwerk, bezieht. Auf den Builder hat diese Änderung keinen Einfluss.

Im nächsten Abschnitt werden Sie einen neuen Builder definieren, der die Daten ins HTMLFormat bringt.

18.2.2 XML-Dateien als HTML darstellen

Das folgende Beispiel erweitert das Projekt um einen neuen Builder. Es soll möglich sein, die XML-Datei als HTML-Dokument zu erzeugen und anzuzeigen. Verfolgen Sie die notwendigen Schritte im Beispielprojekt HaushaltsBuilder_Ext. Im ersten Schritt legen Sie einen neuen Builder an, den `HTMLBuilder`, der von der Klasse `Builder` erbt. An den Methoden `startElement()` und `endElement()` ändert sich nichts. Die Methode `getProduct()` erzeugt aus den Informationen der Wurzel einen HTML-String und gibt ihn zurück.

```

@Override
public String getProduct() {
    html.append("<html><body><h1 align=\"center\">");
    html.append(wurzel.getBeschreibung());
    html.append("</h1>");
    html.append("<b>" + "Jährlich anfallende Positionen:</b><br/>");
    for (var i = 0; i < wurzel.getAnzahlKindKnoten(); i++) {
        var tempKnoten = wurzel.getIndex(i);
        if (tempKnoten.getClass() == Blatt.class) {
            html.append("&#09;");
        }
    }
}

```

```

        var position = (Blatt) tempKnoten;
        formatBlatt(position);
        html.append("<br/>");
    } else {
        html.append("<p>");
        appendElements(tempKnoten, 0);
        html.append("</p>");
    }
}

html.append("</body></html>");
return html.toString();
}
}

```

In dieser Methode wird die Methode `appendElements()` aufgerufen, die rekursiv alle Knoten durchläuft und den HTML-String erweitert.

```

private void appendElements(Knoten knoten, int tab) {
    html.append("<br/>");
    for (var i = 0; i < tab; i++)
        html.append("\t");
    if (knoten.getClass() == Blatt.class)
        formatBlatt((Blatt) knoten);
    else {
        if (tab == 0)
            html.append("<b>");
        html.append(knoten);
        if (tab == 0)
            html.append("</b>");
    }
    for (var j = 0; j < knoten.getAnzahlKindKnoten(); j++) {
        var kindKnoten = knoten.getIndex(j);
        appendElements(kindKnoten, tab + 1);
    }
}
}

```

Beide Methoden rufen die Methode `formatBlatt()` auf, die eine Ausgabenposition darauf überprüft, ob sie erforderlich war. Wenn nein, wird der Anzeigetext rot eingefärbt.

```

private void formatBlatt(Blatt position) {
    if (!position.ausgabeIstErforderlich())
        html.append("<font color=\"#FF0000\">");
    double betrag = position.getValue();
    html.append(position
        .getBeschreibung())
        .append(": ")

```

```
        .append(NumberFormat.getCurrencyInstance().format(betrag));
    if (!position.ausgabeIstErforderlich())
        html.append("</font>");
}
```

Background Information

Um einen String in mehreren Schritten zu erzeugen, könnten Sie Zeichenketten konkatenieren: `String x += stringY;`. Da Strings immutable sind, wird bei jeder Konkatenation ein neues String-Objekt angelegt und x zugewiesen. Die Kosten sind entsprechend hoch. Alternativ greifen Sie auf einen `StringBuilder` oder einen threadsicheren `StringBuffer` zu und hängen dem bisher gespeicherten Text neue Zeichenketten an. Erst, wenn die Konstruktion abgeschlossen ist, geben Sie mit `StringBuilder.toString()`; den fertigen String zurück. Hier sehen Sie das Builder Pattern noch einmal im praktischen Einsatz.

Der Client legt einen Builder an und bezieht von dort den HTML-String. Dieser wird in einer `JEditorPane` angezeigt:

```
Haushalt() {
    var builder = new HTMLBuilder();
    build(builder);
    var html = builder.getProduct();

    var frmMain = new JFrame("Builder Pattern Demo");
    frmMain.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    System.out.println(html);
    var editorPane = new JEditorPane();
    editorPane.setContentType("text/html");
    editorPane.setText(html);
    var scrTree = new JScrollPane(editorPane);
    frmMain.add(scrTree);
    frmMain.setSize(500, 500);
    frmMain.setVisible(true);
}
```

Übrigens können Sie Builder auch „staffeln“, wenn Sie z. B. eine ganz bestimmte Reihenfolge im Erstellungsprozess des fertigen Objekts benötigen. Sie müssen dann aber die entsprechenden Builder individuell benennen. Das könnten dann für ein SQL-Statement zum Beispiel ein `selectBuilder`, ein `fromBuilder` und ein `whereBuilder` sein, die jeweils erst in ihrem letzten `build`-Schritt dann den Nachfolge-Builder zurückgeben können. Das führt dann zu verketteten Buildern, sogenannten „`NestedBuilder`“ und funktioniert z. B. auch für den Aufbau von `Composite`-Objekten.

18.3 Builder – Das UML-Diagramm

Sie finden in Abb. 18.1 das UML-Diagramm aus dem Beispielprojekt HaushaltsBuilder_Ext.

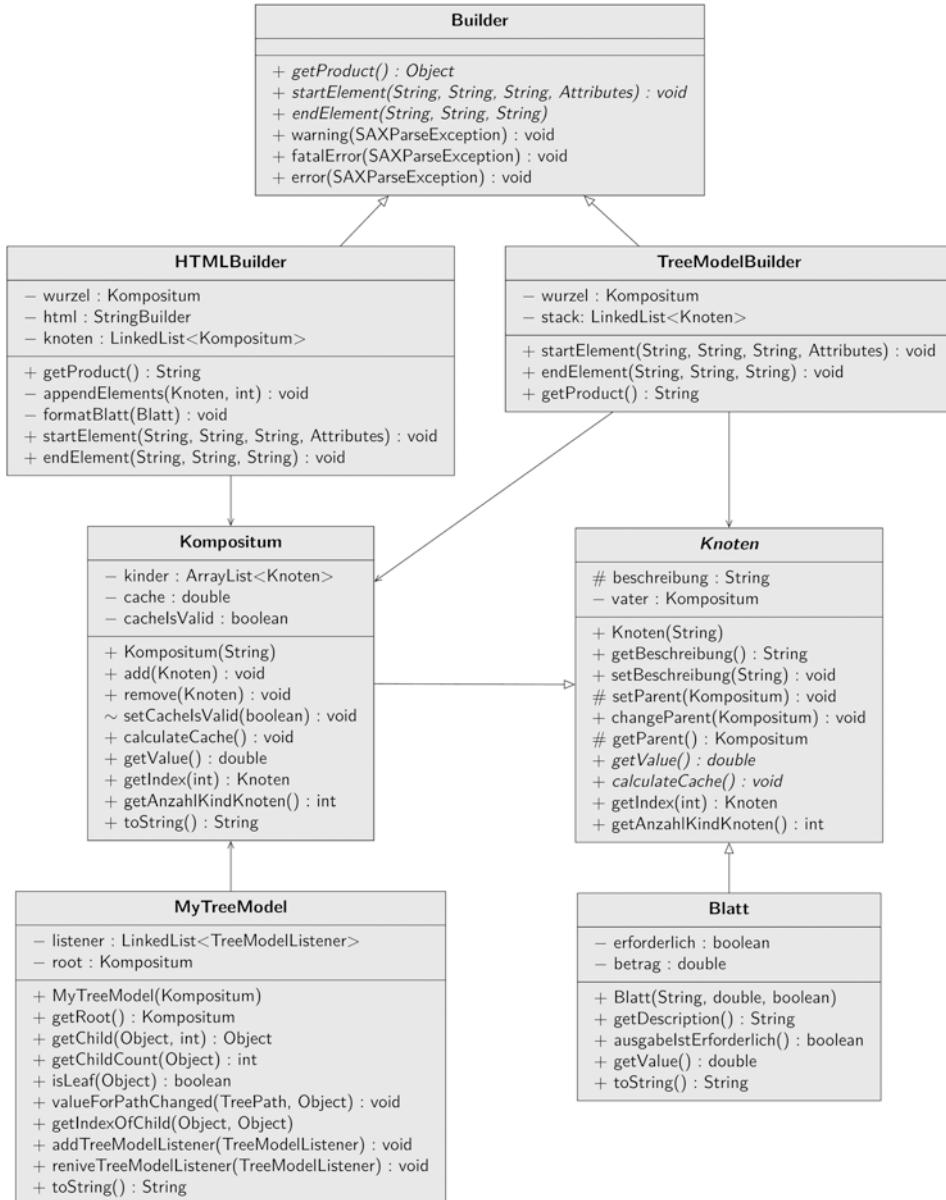


Abb. 18.1 UML-Diagramm des Builder Pattern (Beispielprojekt HaushaltsBuilder_Ext)

18.4 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Ein Builder ist ein Objekt, dessen Aufgabe darin besteht, andere Objekte zu bauen-
- Der Konstruktionsprozess wird im Builder isoliert.
- In der Regel ist der Konstruktionsprozess der zubauenden Objekte komplex oder kompliziert; oft sind mehrere Schritte erforderlich.
- Builder lassen sich auch untereinander verketten und damit voneinander abhängig machen.
- Da die Daten unabhängig von der Objekterzeugung sind, können viele gleichartige Objekte erzeugt werden.
- Typischerweise werden Builder benutzt, um ein Composite zu bauen.

Zweckbeschreibung

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

„Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass der selbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.“



Wenn Sie mit dem Visitor, einem weiteren Verhaltensmuster, arbeiten, haben Sie es immer mit einer Sammlung von Objekten mit unterschiedlichen Schnittstellen zu tun. Auf allen Objekten sollen verwandte Operationen ausgeführt werden, die in einer eigenen Klasse, dem Visitor, zusammengefasst werden.

19.1 Ein einfaches Beispiel

Ein Auto ist ein Aggregat; es besteht aus Rädern, einem Motor und dem Chassis. Das Auto sowie seine Bauteile werden unter der Schnittstelle Element zusammengefasst.

19.1.1 Das Aggregat

Bitte analysieren Sie die Klassen im Package `visitordemo.auto` des Beispielprojekts Visitor. Sie werden dabei finden, dass jedes Element seine eigenen Datenfelder und Methoden hat. Daneben werden die Methoden `getBeschreibung()` und `accept()` in der Schnittstelle vorgeschrieben. Die eine Methode wird eine kurze Beschreibung zurückgeben, die andere stellt eine Schnittstelle für ein Visitor-Objekt zur Verfügung. Nehmen Sie als Beispiel die Klasse Rad. Das Rad kann seine Position speichern – beispielsweise vorne links. Es kann außerdem den Luftdruck speichern und zurückgeben. Die

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_19

Methode `accept()` ist so überschrieben, dass auf dem übergebenen Visitor-Objekt die Methode `visit()` aufgerufen und hierbei die eigene Instanz – `this` – übergeben wird.

```
public class Rad implements Element {
    private final Position position;
    private final double luftdruck = 2.0;

    public double getLuftdruck() {
        return luftdruck;
    }

    public Rad(Position position) {
        this.position = position;
    }

    @Override
    public void accept(VisitorIF visitor) {
        visitor.visit(this);
    }

    @Override
    public String getBeschreibung() {
        return "- Rad " + position;
    }
}
```

Am Chassis werden die Räder montiert. Wenn die `accept`-Methode auf ihr aufgerufen wird, ruft sie im ersten Schritt auf dem Visitor die `visit`-Methode auf und übergibt sich selbst als Parameter. Im zweiten Schritt wird für jedes Rad die `visit`-Methode aufgerufen.

```
public class Chassis implements Element {
    private final Rad[] raeder;

    public Rad[] getRaeder() {
        return raeder;
    }

    Chassis(Rad[] raeder) {
        this.raeder = raeder;
    }

    @Override
    public void accept(VisitorIF visitor) {
        visitor.visit(this);
        for (var rad : raeder)
```

```
    rad.accept(visitor);  
}  
  
@Override  
public String getBeschreibung() {  
    return "- Chassis";  
}  
}
```

Das Auto soll auch noch in der gebotenen Kürze beschrieben werden. Es speichert die Informationen, ob der Tank voll ist, ob genügend Öl vorhanden ist und ob Blinkerwasser (dieses innovative Gebräu steht hier stellvertretend für jede andere mögliche Verbrauchsstoffflüssigkeit) nachgefüllt wurde. Zu diesen Attributen gibt es entsprechende Zugriffsmethoden. Der Konstruktor baut einen neuen Motor ein, erstellt das Chassis und hängt die Räder daran.

```
public class Auto implements Element {  
    private final List<Element> bauteile = new ArrayList<>();  
  
    // ... gekürzt  
  
    public Auto() {  
        bauteile.add(new Motor());  
        bauteile.add(new Chassis(new Rad[] {  
            new Rad(Position.VL), new Rad(Position.VR),  
            new Rad(Position.HL), new Rad(Position.HR)  
        }));  
    }  
}
```

Die Methode `accept()` ruft auf jedem Bauteil dessen Visitor-Methode auf und über gibt ihnen eine Referenz auf den Visitor. Anschließend wird die `visit()`-Methode auf dem Visitor aufgerufen und das Auto übergibt sich selbst dort als Parameter. Auf den Abdruck der Klasse `Motor` verzichte ich – sie folgt derselben Logik wie die vorigen Klassen.

19.1.2 Der Visitor

Die Visitor-Klasse definiere ich in einem eigenen Package, in `visitordemo.visitor`. Ein Visitor wird durch das Interface `VisitorIF` beschrieben. Jede Klasse, die als Visitor agieren können soll, muss die darin deklarierten Methoden überschreiben; es geht dabei um die Methode `visit()`, die vierfach überladen ist, für jedes Element einmal.

- Die Vererbungshierarchie des Visitor ist eng an die Element-Klassen gebunden. Sie muss, wie Sie gleich sehen werden, genaue Kenntnis von den Schnittstellen der zu besuchenden Objekte haben. Umgekehrt müssen die Element-Klassen nur das Visitor-Interface kennen.

```
public interface VisitorIF {  
    void visit(Rad rad);  
    void visit(Motor motor);  
    void visit(Chassis chassis);  
    void visit(Auto auto);  
}
```

Ein konkreter Visitor ist der `BauteileVisitor`. Er stellt eine Liste der Beschreibungen aller Teile zusammen. Dazu überschreibt er die Methode `visit()` so, dass auf dem übergebenen Parameter, also einem Rad, einem Motor, einem Chassis oder einem Auto, die Methode `getBeschreibung()` aufgerufen wird. Die Beschreibungen aller Einzelteile werden durch einen `StringBuilder` zusammengefasst. Die Methode `getBauteile()` gibt einen String mit der Liste aller Bauteile zurück.

```
public class BauteileVisitor implements VisitorIF {  
    private final StringBuilder builder = new StringBuilder();  
  
    public String getBauteileListe() {  
        return "Bauteile: \n" + builder.toString();  
    }  
  
    @Override  
    public void visit(Rad rad) {  
        builder.append(rad.getBeschreibung()).append("\n");  
    }  
  
    @Override  
    public void visit(Motor motor) {  
        builder.append(motor.getBeschreibung()).append("\n");  
    }  
  
    // ... gekürzt  
}
```

Wie kann jetzt der Client diese Konstruktion nutzen?

19.1.3 Der Client

Der Client legt eine Instanz der Klasse Auto und einen Bauteile-Visitor an. Danach ruft er die Methode `accept()` mit dem Visitor als Parameter auf. Der resultierende String wird auf der Konsole ausgegeben.

```
public class ApplStart {  
    public static void main(String[] args) {  
        var auto = new Auto();  
        var visitor = new BauteileVisitor();  
        auto.accept(visitor);  
        var bauteile = visitor.getBauteileListe();  
        System.out.println(bauteile);  
        // ... gekürzt  
    }  
}
```

Auf der Konsole wird die vom Visitor zusammengetragene Liste der Bauteile ausgegeben:

```
Bauteile:  
- Motor  
- Chassis  
- Rad vorne links  
- Rad vorne rechts  
- Rad hinten links  
- Rad hinten rechts  
- Auto (übrige Bestandteile)
```

19.1.4 Ein weiterer Visitor

Wenn Sie wegen einer Unregelmäßigkeit an Ihrem Fahrzeug in die Werkstatt fahren, hängt dort ein Mensch ein Diagnosegerät an eine definierte Schnittstelle im Auto und kann kurz danach präzise sagen, ob das Auto einen Defekt hat, und gegebenenfalls in welchem Element. Diese Situation wird durch den DiagnoseVisitor abgebildet. Er funktioniert ganz ähnlich wie der BauteileVisitor. Er fragt auf jedem Element bestimmte Informationen ab und wertet sie aus. Es gibt hierbei zwei unterschiedliche Vorgehensweisen. Das Auto kann selbstständig messen, ob der Tank voll ist und ob der Ölstand stimmt; es gibt daher mit den entsprechenden Zugriffsmethoden einen booleschen Wert zurück. Ich unterstelle, dass die

wenigsten Autos den Luftdruck in ihren Reifen selbstständig messen, obwohl das technisch möglich ist. Hier lässt der Visitor sich den aktuellen Wert zurückgeben und wertet diesen aus. Am Schluss kann er Auskunft darüber geben, ob das Auto fahrbereit ist. Bitte analysieren Sie den DiagnoseVisitor selbstständig – die Klasse ist nicht kompliziert.

19.1.5 Kritik am Projekt

Es wird deutlich, was ich eingangs mit verwandten Operationen gemeint habe: Da gibt es eine Vielzahl von Klassen, auf denen jeweils eine Diagnose durchgeführt werden soll. Anstatt die Diagnosemethoden in den jeweiligen Klassen zu definieren, werden sie in einer eigenen Klasse zusammengefasst. Daraus ergeben sich zwei Vorteile: Zunächst werden die Klassen nicht durch Code „verschmutzt“, der nicht zu ihrem eigentlichen Kerngeschäft gehört. Darüber hinaus können die neu definierten Operationen sehr viel einfacher gewartet werden; sie befinden sich in einer einzigen Klasse und müssen nicht in zig Klassen separat gepflegt werden.

Die beteiligten Klassen müssen keine gemeinsame Schnittstelle haben. Im Projekt „Visitor“ waren sowohl das Auto als auch die Bauteile Subklassen der Schnittstelle Element. Das ist jedoch im Sinne des Visitor Patterns nicht zwingend vorgeschrieben.

Ein Nachteil des Visitor Patterns ist, dass das besuchte Objekt eine ausreichend umfangreiche Schnittstelle haben muss, um alle relevanten Daten abfragen zu können. Die Klasse Rad muss beispielsweise Auskunft über den Luftdruck geben können, der Motor muss seinen Zustand benennen können usw. Möglicherweise werden Sie mit dem Visitor Pattern sogar den Zugriff auf Daten ermöglichen müssen, die Sie eigentlich kapseln wollten.

Ein weiterer Nachteil ist, dass Sie umfangreiche Wartungsarbeiten an allen konkreten Visitor-Klassen haben werden, wenn ein neues Element eingefügt wird. Stellen Sie sich vor, Sie definieren zusätzlich die Klasse Bremse – die Schnittstelle Visitor muss eine entsprechende visit-Methode deklarieren, die von allen konkreten Visitor-Klassen überschrieben werden muss. Das Visitor Pattern lässt sich immer dann optimal einsetzen, wenn eher neue Operationen benötigt werden, als dass sich häufiger die Objektstruktur ändert.

Das Visitor Pattern ist ein ausgezeichnetes Beispiel für das Open-Closed-Principle. Sie haben eine Reihe von Klassen, die geschlossen sind gegen Veränderung. Sie lassen die Tür aber einen Spalt offen für Erweiterungen, indem sie eine Schnittstelle anbieten, die es anderen Objekten ermöglicht, neues Verhalten zu definieren.

Ein letzter Aspekt soll angesprochen werden: die Art der Iteration. In dem Projekt, das Sie gerade bearbeitet haben, finden Sie einen internen Treiber – Sie rufen auf dem Auto-Objekt die Methode `accept()` auf, die dafür sorgt, dass über alle Bauteile iteriert wird. Denkbar wäre auch, dass Sie einen externen Iterator definieren. Die Lösung mit einem internen Iterator muss nicht die sein, die Sie als praktikabel empfinden.

19.2 Visitor – Das UML-Diagramm

Sie sehen in Abb. 19.1 das UML-Diagramm aus dem Beispielprojekt Visitor.

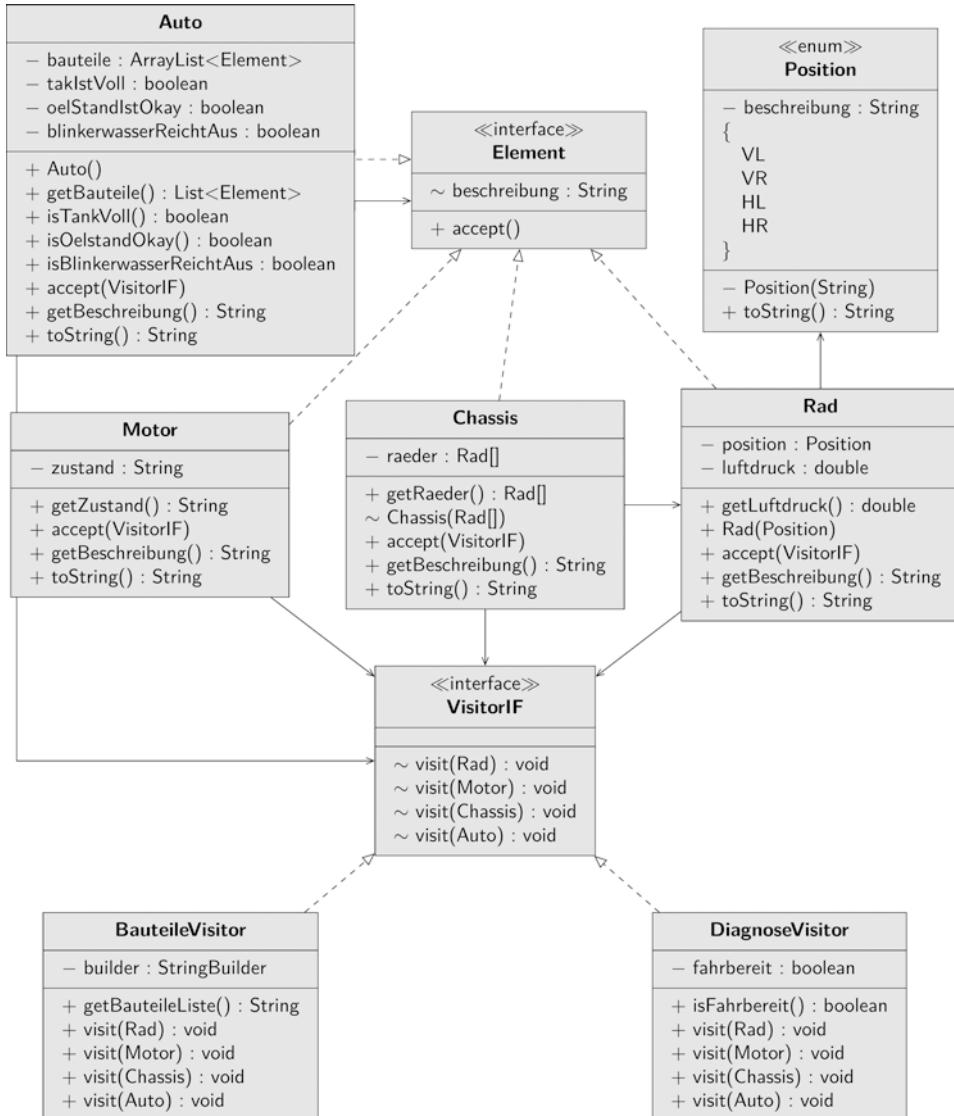


Abb. 19.1 UML-Diagramm des Visitor Pattern (Beispielprojekt Visitor)

19.3 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sie haben es mit Objekten mit verschiedenen Schnittstellen zu tun.
- Auf allen Objekten sollen verwandte Operationen – eine Diagnose – ausgeführt werden.
- Die Operationen werden in einer einzigen Klasse, dem Visitor, zusammengefasst.
- Jedes Objekt bietet eine Schnittstelle, die der Visitor aufrufen kann.
- Das Objekt ruft auf dem Visitor dessen überladene visit-Methode auf und übergibt sich selbst als Parameter.
- Der Visitor definiert für jedes Objekt in der Sammlung eine visit-Methode.
- Nach den Regeln der Polymorphie wird die für das Objekt passende visit-Methode aufgerufen.

Zweckbeschreibung

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

„Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es Ihnen, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.“



Kapselung ist ein zentrales Thema der objektorientierten Programmierung. Jedes Objekt hat Attribute, die es nicht nach außen bekanntgeben darf. Stattdessen bietet es eine möglichst schmale Schnittstelle an. Im letzten Kapitel, beim Visitor Pattern, habe ich diesen Konflikt angesprochen – dort mussten eigens für den Visitor Zugriffsmethoden erstellt werden. Auch für das Memento Pattern ist das Thema Kapselung ein Problem.

20.1 Aufgabe des Memento Patterns

Ausgangspunkt soll ein beliebiges Objekt A sein, dessen Zustand Sie extern speichern möchten, um ihn später wiederherstellen zu können. Wie kann der Zustand ermittelt und gespeichert werden? Zunächst liegt es auf der Hand, ein Objekt B zu nehmen, das die Daten von A liest und sie speichert. Aber wie könnte das speichernde Objekt B an die Daten des zu speichernden Objekts A herankommen?

20.1.1 Öffentliche Datenfelder

Ein erster Ansatz könnte sein, die Datenfelder öffentlich zugänglich zu machen.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_20

```
public class Memento {
    public int antwort = 42;
    public String passwort = "Ken sent me";
}
```

Jetzt könnte ein anderes Objekt den Zustand auslesen und speichern. Aber warum werden Sie diesen Ansatz nicht weiterverfolgen? Die Datenfelder sind öffentlich einsehbar und änderbar, was dem Grundsatz der Datenkapselung, dem Information Hiding, zuwidert läuft.

20.1.2 Das JavaBeans Pattern

Die meisten Java-Einführungen setzen den Grundsatz der Datenkapselung dadurch um, dass es Methoden gibt, die lesend und schreibend auf die Datenfelder zugreifen können, was mehr oder weniger der JavaBeans-Spezifikation entspricht.

```
public class Memento {
    private int antwort = 42;
    private String passwort = "Ken sent me";

    public int getAntwort() {
        return antwort;
    }

    public void setAntwort(int antwort) {
        this.antwort = antwort;
    }

    // ... gekürzt
}
```

Jetzt sind die Daten zwar gekapselt, aber gegenüber dem Ansatz mit den öffentlichen Feldern hat sich nichts geändert – immer noch kann jeder lesend und schreibend auf die Daten zugreifen, wenn auch über den Umweg der Zugriffsmethoden.

20.1.3 Default-Sichtbarkeit

Denkbar wäre, dass Sie die Felder – oder die Zugriffsmethoden – auf default-Sichtbarkeit (package-private) setzen; dann könnten nur noch Klassen im gleichen Paket auf die Daten zugreifen. Ein Objekt, das den Zustand eines anderen Objekts speichern soll, muss folglich im gleichen Paket definiert werden. Aber auch dieser Ansatz ist nicht optimal. Packa-

ges sollen helfen, Klassen zu sinnvollen Einheiten zu gruppieren. Beispielsweise kommen alle Klassen, die Daten am Bildschirm anzeigen, in ein Paket, während Daten, die das Datenmodell beschreiben, in einem anderen Paket definiert werden. Meines Erachtens wäre es überzogen, wenn ein Paket nur dafür angelegt wird, um eine Klasse und ihren Datenspeicher darin abzulegen.

Die bisherigen Vorschläge konnten nicht überzeugen – es scheint eine schlechte Idee zu sein, die Daten lesen zu wollen. Also gehen Sie einen anderen Weg: Das Objekt, dessen Zustand gespeichert werden soll, wird selbst dafür verantwortlich gemacht, seinen Datenspeicher zu erstellen.

20.2 Eine mögliche Realisierung

Bevor ich Ihnen das Projekt vorstelle, möchte ich ein paar Begriffe definieren. Das Objekt, dessen Zustand gespeichert werden soll, ist der *Originator*; in der deutschen Übersetzung des GoF-Buchs wird er auch Urheber genannt. Das Objekt, das den Zustand eines anderen speichert, ist das *Memento*. Der *Caretaker* verwaltet eine Liste von Mementos. Das Beispielprojekt *Memento_Simple* realisiert das Pattern auf eine sehr übersichtliche Weise. Der *Caretaker* ist die Klasse *Stack*. Sie speichert die verschiedenen Memento-Objekte. Als *Stack* habe ich nicht die Java-eigene Implementierung genommen, sondern eine eigene Klasse entwickelt. Das *Memento* ist als leeres Marker-Interface definiert.

```
public interface Memento {  
}
```

Sie möchten den Zustand eines Auto-Objekts speichern. Es hat die Attribute *geschwindigkeit* und *aktuellerSpritVerbrauch*. Die Geschwindigkeit kann beeinflusst werden, hierfür gibt es entsprechende Zugriffsmethoden. Auf den Spritverbrauch kann nur indirekt über die Geschwindigkeit Einfluss genommen werden.

```
public class Auto  
{  
    // ... gekürzt  
  
    private int geschwindigkeit = 0;  
    private int aktuellerSpritverbrauch = 0;  
  
    public void fahreSchneller() {  
        geschwindigkeit++;  
        berechneSpritverbrauch();  
    }  
  
    public void fahreLangsamer() {
```

```

        if (geschwindigkeit > 0) {
            geschwindigkeit--;
            berechneSpritverbrauch();
        }
    }

    private void berechneSpritverbrauch() {
        // ... gekürzt
    }
}

```

Jetzt suche ich einen Weg, um alle Datenfelder speichern zu können. Hierzu lege ich eine innere Klasse AutoMemento an, die geeignet ist, die Datenfelder zu kopieren. Jedes Mal, wenn Sie eine Instanz der Klasse AutoMemento erzeugen, werden die Datenfelder des AutoObjekts kopiert.

```

public class Auto {
    private class AutoMemento implements Memento {
        private final int tempo = geschwindigkeit;
        private final int durst = aktuellerSpritverbrauch;
    }

    private int geschwindigkeit = 0;
    private int aktuellerSpritverbrauch = 0;

    // ... gekürzt
}

```

Die Verantwortung, ein Memento zu erzeugen, geben Sie an die Auto-Klasse ab. Dort fordern Sie ein Memento an; außerdem ist das Auto auch dafür zuständig, einen alten Zustand aus einem gegebenen Memento wiederherzustellen.

```

public class Auto {
    // ... gekürzt

    public Memento createMemento() {
        return new AutoMemento();
    }

    public void setMemento(Memento memento) {
        AutoMemento myMemento = (AutoMemento) memento;
        this.aktuellerSpritverbrauch = myMemento.durst;
        this.geschwindigkeit = myMemento.tempo;
    }
}

```

Der Client legt einen Stack, den Caretaker, und ein Auto-Objekt an. Dann führt er einige Operationen durch, die die Geschwindigkeit und damit den Spritverbrauch beeinflussen, und speichert den Zustand nach jeder Änderung in einem Memento. Das Memento wird auf den Stack gelegt; jede Änderung kann jetzt rückgängig gemacht werden. Die main-Methode der Klasse `App1Start` demonstriert dieses Vorgehen. Bitte analysieren Sie den Code selbstständig.

Wie bewerten Sie diese Lösung? Auf der einen Seite können die Attribute weder des Originators noch des Mementos gelesen werden – sie sind also sehr konsequent gekapselt. Die Kapselung erkaufen Sie sich aber um den Preis, dass die Klasse selbst für das Erstellen und Wiederherstellen des Mementos zuständig ist; der Code wird insofern „verschmutzt“. Falls Ihnen diese Lösung zusagt, beachten Sie bitte, dass das Beispiel ausgesprochen simpel ist. Die Attribute sind vom primitiven Typ `int`. Wenn Sie `mutable` Objekte haben, müssen diese geklont werden – die Problematik der kopierten Referenzen, die Sie im Prototype-Kapitel beim Klonen kennengelernt haben, begegnet Ihnen hier wieder.

Im nächsten Beispiel greifen wir den GrafikEditor des Prototype Patterns wieder auf.

20.3 Ein größeres Projekt

In der zweiten Ausbaustufe des GrafikEditors im Kapitel über das Prototype Pattern haben Sie ein Diagramm geklont, indem Sie es serialisiert und deserialisiert haben. Genau dieses Vorgehen kann auch für das Memento interessant sein. Für das folgende Beispielprojekt Memento habe ich den `GrafikEditor_2` als Vorlage genommen und abgewandelt. Der Vorteil jetzt ist, dass in der Klasse `PanelCanvas` die Methode `getMemento()` definiert ist – sie erzeugt und returniert ein geklontes Diagramm als tiefe Kopie, das Memento.

```
public Diagram getMemento() {  
    Diagram clone = null;  
    try {  
        ObjectOutputStream oos;  
        ByteArrayInputStream bais;  
        ObjectInputStream ois;  
        try ( var baos = new ByteArrayOutputStream() ) {  
            oos = new ObjectOutputStream(baos);  
            oos.writeObject(diagram);  
            bais = new ByteArrayInputStream(baos.  
                toByteArray());  
            ois = new ObjectInputStream(bais);  
            clone = (Diagram) ois.readObject();  
        }  
        oos.close();  
        bais.close();  
        ois.close();  
    }  
}
```

```

        } catch (IOException ex) {
            new ErrorDialog(ex);
        } finally {
            return clone;
        }
    }
}

```

Die Methode `setDiagram()` nimmt ein Diagramm-Objekt entgegen und füllt die Zeichenfläche mit dessen Daten.

```

public void setDiagram(Diagram clone) {
    selected = null;
    this.diagram = clone;
    repaint();
}

```

In der main-Methode der Klasse `ApplStart` wird die Aktion `undoAction` definiert, die vom Caretaker das letzte Diagramm anfordert und an die Zeichenfläche übergibt. Diese Aktion wird an einen MenuItem übergeben. Der Caretaker ist eine Instanz der Klasse `Stack`, die Sie aus dem letzten Beispiel kennen. Dieselbe Instanz übergeben Sie an die Zeichenfläche.

```

private final PanelCanvas canvas = new PanelCanvas(caretaker);

// ... gekürzt

private final Action undoAction = new AbstractAction("Undo") {
    @Override
    public void actionPerformed(ActionEvent e) {
        myFrame.setTitle("Memento Demo");
        if (!caretaker.empty()) {
            Diagram diagram = caretaker.pop();
            canvas.setDiagram(diagram);
        }
        else
            myFrame.
                setTitle("Undo kann nicht durchgeführt werden - Stack
ist leer");
    }
};

```

Die meisten Aktionen konnten Sie über Mausklicks auslösen: Verschieben von Kreisen, Erzeugen von Linien oder Kreisen und Markieren von Grafikelementen. Das Löschen von Grafikelementen haben Sie über das Menü ausgeführt. Alle Methoden, die Sie so aufgerufen haben, waren in der Klasse PanelCanvas definiert – entweder als eigenständige Methode oder als EventListener. Diese Methoden werden ergänzt, so dass gleichzeitig zu der definierten Maßnahme ein Snapshot erzeugt und gespeichert wird.

```
public void deleteSelectedGraphic() {
    if (selected != null) {
        createMemento();
        diagram.deleteSelectedGraphic(selected);
        selected = null;
        repaint();
    }
}

private void createMemento() {
    var tempDiagram = getMemento();
    caretaker.push(tempDiagram);
}
```

Sie speichern bei dieser Realisierung ein Objekt vom Typ `Diagram` im Caretaker. Alternativ hätten Sie auch das serialisierte Objekt im Caretaker speichern können.

20.4 Memento – Das UML-Diagramm

Aus dem Beispielprojekt Memento zeige ich Ihnen in Abb. 20.1 nur einen Teil der beteiligten Klassen. Die Pakete `graphics` und `prototype` habe ich hier nicht noch mal abgebildet. Sie finden deren Darstellung im UML-Diagramm zum Prototype in Abschn. 17.3.

20.5 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Das Ziel ist es, den Zustand eines Objekts extern zu speichern.
- Dabei muss auf die Attribute des Objekts zugegriffen werden.
- Die Kapselung soll möglichst nicht geschwächt werden.
- Um das Pattern zu realisieren, lassen Sie den Originator ein Memento erstellen.
- Der Caretaker speichert die Mementos.

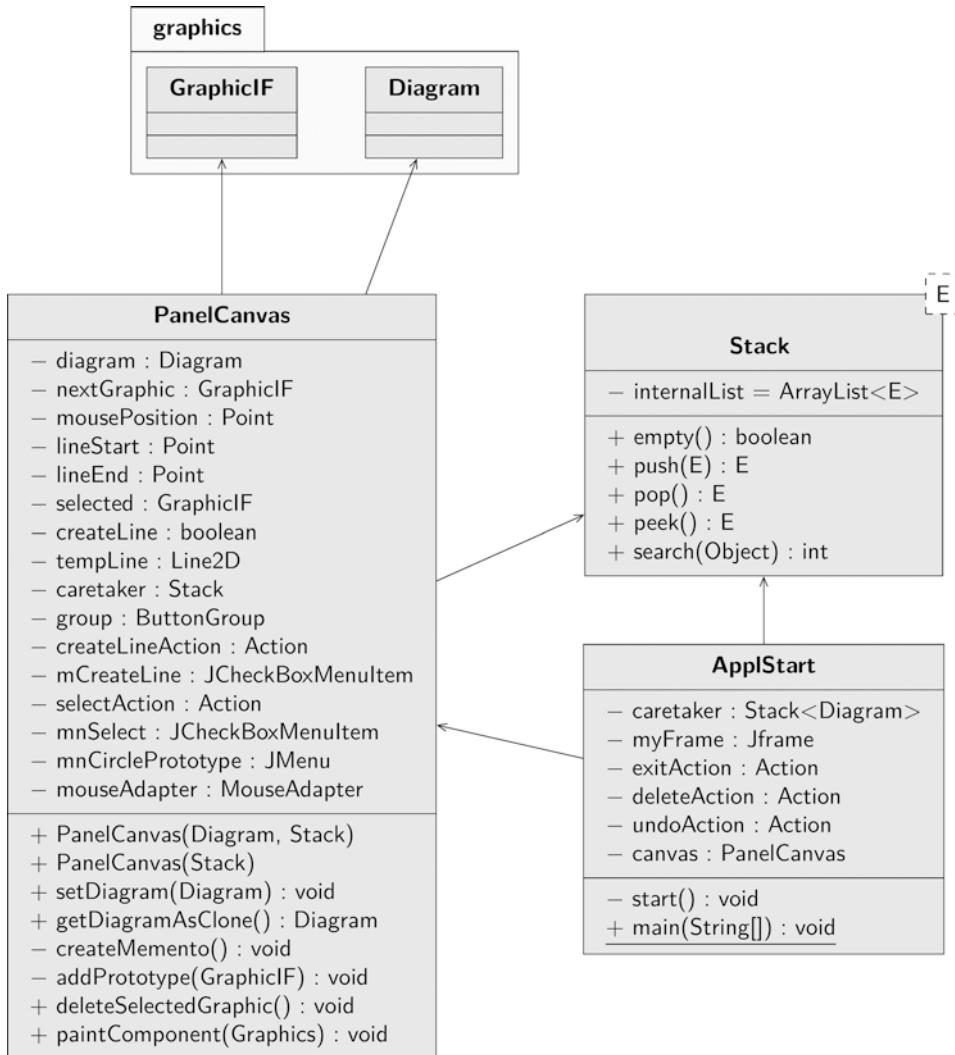


Abb. 20.1 UML-Diagramm des Memento Pattern (Beispielprojekt Memento)

Zweckbeschreibung

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

„Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.“



Die Fassade gehört zu den Strukturmustern; sie beschreibt, wie Sie unkompliziert auf ein kompliziertes oder komplexes Subsystem zugreifen können.

21.1 Ein Beispiel außerhalb der IT

Stellen Sie sich vor, dass Sie ein komplexes oder kompliziertes System haben – beispielsweise eine Spiegelreflex-Kamera. Wenn Sie ein Porträt machen, soll die Schärfentiefe nur einen geringen Bereich umfassen. Sie machen die Blende möglichst weit auf, wählen also eine kleine Blendenzahl. Dadurch fällt mehr Licht auf den Film bzw. den Prozessor. Damit das Bild nicht zu hell wird, müssen Sie die Belichtungszeit verringern.

Vielleicht möchten Sie später kein Porträt, sondern eine Landschaft fotografieren. Dort soll der Schärfentiefebereich möglichst groß sein; Sie wählen eine große Blendenzahl und machen dadurch die Blende zu. Jetzt kommt weniger Licht auf den Prozessor, also muss die Belichtungszeit verlängert werden. Die Belichtungszeit kann aber nicht unbegrenzt verlängert werden; erfahrungsgemäß kann ein Bild nur dann frei aus der Hand aufgenommen werden, wenn die Belichtungszeit kürzer ist als 1/Brennweite. Ist die Belichtungszeit länger, besteht die Gefahr, dass Sie das Bild verwackeln. Sie müssen also die Filmempfindlichkeit, die ISO-Zahl, erhöhen.

Klingt das kompliziert? Ich denke schon! Und das sehen die meisten Kamerahersteller genauso. Moderne Kompaktkameras, aber auch (digitale) Spiegelreflex-Kameras, bringen Motivprogramme mit. Sie müssen der Kamera nur noch sagen: „Ich möchte ein Porträt

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_21

machen!“ oder: „Ich möchte Landschaften fotografieren!“. Die Kamera stellt automatisch Blende und Belichtungszeit so ein, dass das Ergebnis optimal ist.

Damit ist das Prinzip der Fassade eigentlich auch schon erklärt: Sie haben ein kompliziertes oder komplexes System. Um Ihnen den Zugriff auf das System zu erleichtern, wird eine Fassade erstellt. Als Anwender bzw. Fotograf müssen Sie sich nicht mehr mit den Details beschäftigen. Es soll genügen, dass Sie der Fassade – dem Motivprogramm „Portrait“ – sagen, was Sie haben möchten. Das „Wie“ wird von der Fassade realisiert. Der Zugriff auf die einzelnen Komponenten des Systems wird Ihnen weiterhin ermöglicht: Sie müssen die Fassade nicht nutzen, Sie können weiterhin von Hand Blende und Belichtungszeit einstellen.

Im folgenden Abschnitt werden Sie nach diesem kleinen Ausflug wieder ein Beispiel aus der IT kennenlernen.

21.2 Die Fassade in einem Java-Beispiel

Schauen Sie sich das Beispielprojekt Fassade an. Es besteht aus den Packages demo1 bis demo4 und dem Package reise. Sie werden in diesem Beispiel eine Reise erstellen. Für eine Reise müssen Sie zunächst mit dem Zug zum Flughafen fahren. Von dort fliegen Sie in Ihr Urlaubsgebiet. Nachdem Sie am fernen Flughafen angekommen sind, werden Sie vom Transfer-Service zum Hotel gebracht. Im Hotel werden Sie entweder all-inclusive oder mit Halbpension oder nur mit Frühstück versorgt. Optional können Sie sich einen Mietwagen nehmen, der versichert und betankt werden muss. Im Package reise sind alle erforderlichen Klassen zu finden. Wie dieses Projekt in der Entwicklungsumgebung aussieht, sehen Sie in Abbildung Abb. 21.1.

Wenn ein Client eine Reise anlegen möchte, muss er alle Komponenten anlegen. Schauen Sie auf den Client-Code im package demo1.

```
public class Client {
    public static void main(String[] args)
    {
        System.out.println("Ein Client:");
        var zugZumFlug = new ZugZumFlug();
        zugZumFlug.einsteigen();
        var flug = new Flug();
        flug.gepaekAufgeben();
        flug.persoVorzeigen();
        var transfer = new Transfer();
        transfer.gepaekEinladen();
        var hotel = new Hotel();
        hotel.safeMieten();
        var halbpension = new Halbpension();
        halbpension.bierBestellen();
    }
}
```

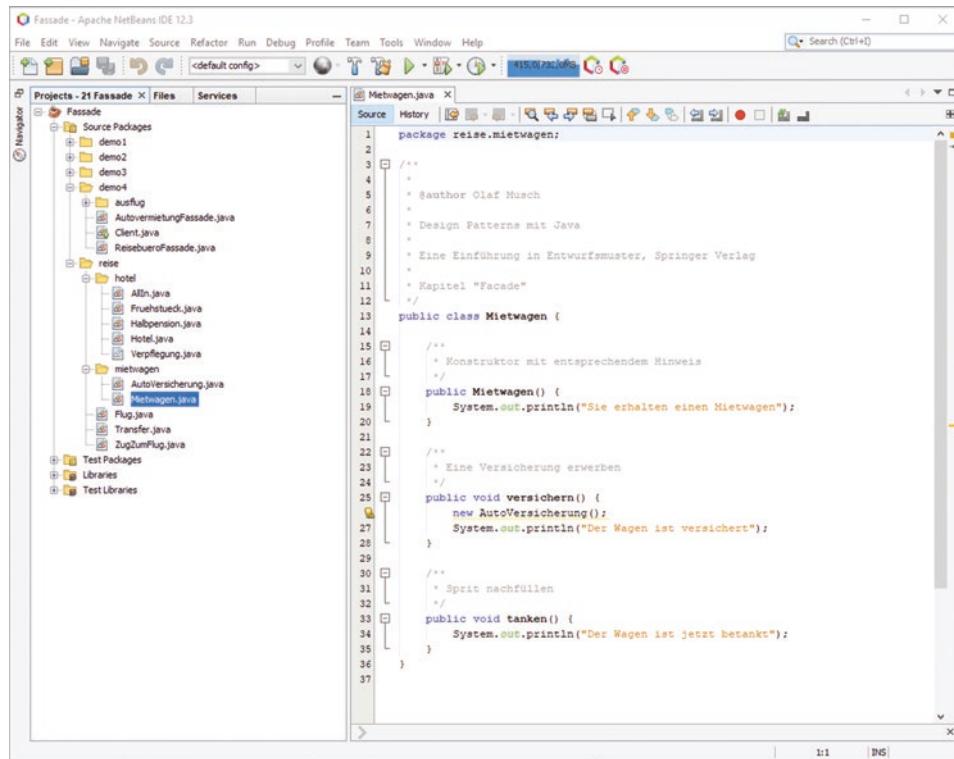


Abb. 21.1 Das Projekt „Fassade“ in der Entwicklungsumgebung NetBeans

Wenn der Kunde außerdem noch einen Mietwagen benötigt, muss er zusätzlich zuerst ein Mietwagen-Objekt erzeugen und dann darauf die Methoden `versichern()` und `tanken()` aufrufen. Sie haben also auf der einen Seite viele Komponenten, die sie bedienen müssen. Auf der anderen Seite haben Sie unterschiedliche Konfigurationsmöglichkeiten – im einen Fall genügt es, Objekte zu erzeugen, im anderen müssen Sie darüber hinaus auf den Objekten Methoden aufrufen. Sieht das für Sie nach einem komplexen System aus?

```

public class Client {
    public static void main(String[] args) {
        // ... gekürzt
        System.out.println("\nEin anderer Client:");
        zugZumFlug = new ZugZumFlug();
        transfer = new Transfer();
        hotel = new Hotel();
        halbpension = new Halbpension();
        var mietwagen = new Mietwagen();
        mietwagen.versichern();
        mietwagen.tanken();
    }
}

```

Ihnen ist sicher aufgefallen, dass im oberen Listing vergessen wurde, den Flug zu buchen. Aber vielleicht kennen Sie das Problem aus der Praxis: Fehler in umfangreichen Konfigurationen findet man erst nach einem noch umfangreicheren Testverfahren. Der Code von oben ist also fehleranfällig – im nächsten Abschnitt werden Sie eine Lösung für dieses Dilemma kennenlernen.

21.2.1 Einführung einer Fassade

Sie führen eine Fassade ein, deren einzige Aufgabe darin besteht, auf der einen Seite eine vereinfachte Handhabung für den Client zu ermöglichen, auf der anderen aber die komplexen Zusammenhänge des Systems zu beherrschen. Im Package demo2 finden Sie die Klasse ReisebueroFassade, die die Methode reiseBuchen() anbietet. Dieser Methode geben Sie einen Wahrheitswert als Parameter mit, der angibt, ob ein Auto gemietet werden soll.

```
public final class ReisebueroFassade {
    public void reisebuchen(boolean mitMietwagen) {
        var zugZumFlug = new ZugZumFlug();
        var flug = new Flug();
        var transfer = new Transfer();
        var hotel = new Hotel();
        var halbpension = new Halbpension();
        if (mitMietwagen) {
            var mietwagen = new Mietwagen();
            mietwagen.versichern();
            mietwagen.tanken();
        }
    }
}
```

Die Client-Klasse im Package demo2 kann nun sehr einfach die Methode aufrufen.

```
public class Client {
    public static void main(String[] args) {
        System.out.println("Ein Client:");
        new ReisebueroFassade().reisebuchen(false);

        System.out.println("\nEin anderer Client:");
        new ReisebueroFassade().reisebuchen(true);
    }
}
```

Für den Client ist es nun sehr viel einfacher, eine Reise zu buchen. Lassen Sie mich auf eines hinweisen: In der Praxis werden Sie die Klasse nur `Reisebuero` nennen. Den Zusatz `ReisebueroFassade` habe ich lediglich zur Verdeutlichung angehängt.

21.2.2 Der Begriff „System“ genauer betrachtet

Der Begriff „System“ ist wichtig für die Fassade: Sie stellen einen vereinfachten Zugriff auf ein Subsystem zur Verfügung. Der Begriff System bzw. Subsystem ist dabei weit auszulegen. Es kann sich hierbei um eine beliebig große Einheit handeln; es kann aber auch der Zugriff auf eine einzelne Klasse gemeint sein. Im Package `demo3` habe ich die Autovermietung als eigenes System betrachtet und in ein separates Package ausgelagert. Außerdem habe ich die Fassade Autovermietung definiert, die eine einzelne statische Methode hat, mit der ein Auto gemietet werden kann: Ein Zugriff auf dieses (Sub)system ist nur noch über die Fassade möglich.

```
public class AutovermietungFassade {  
    public static void autoMieten() {  
        var mietwagen = new Mietwagen();  
        mietwagen.versichern();  
        mietwagen.tanken();  
    }  
}
```

Das Reisebüro hat jetzt einen vereinfachten Zugriff auf dieses Subsystem.

```
public class ReisebueroFassade {  
    public void reisebuchen(boolean mitMietwagen) {  
        var zugZumFlug = new ZugZumFlug();  
        var flug = new Flug();  
        var transfer = new Transfer();  
        var hotel = new Hotel();  
        var halbpension = new Halbpension();  
        if (mitMietwagen)  
            AutovermietungFassade.autoMieten();  
    }  
}
```

Der Client kann entweder im Reisebüro ein Auto mieten oder direkt bei der Autovermietung. In der Implementierung, die ich Ihnen hier vorgestellt habe, ist es übrigens kein Problem, dass der Client weiterhin auf die einzelnen Elemente des Subsystems zugreift.

Die Fassade schränkt nicht ein, sie führt auch keine neue Logik ein – sie vereinfacht lediglich den Zugriff auf ein Subsystem.

Ein alternatives Vorgehen demonstriert das Package `demo4`. Hier gibt es das Subsystem (Subpackage) `ausflug`. Dieses System wird vereinfacht durch die Klasse `Ausflug` dargestellt. Die Klasse und deren Methoden sind package-private, so dass ein Zugriff nur über die Fassade möglich ist. Bitte analysieren Sie unter diesem Aspekt `demo4`. Denkbar ist auch, dass die Fassade nicht nur zwischen Client und Subsystem vermittelt, sondern dass sie eigene Logik definiert.

- ▶ Der Vorteil der Fassade ist offensichtlich: Der Zugriff wird vereinfacht, die Abhängigkeit von Client und Subsystem wird gelockert. Dadurch wird vermieden, dass Client-Code bricht, wenn das Subsystem geändert oder ersetzt wird. Es ist sogar möglich, ein komplettes Subsystem auszutauschen.

Lassen Sie uns im nächsten Abschnitt ansehen, wo die Fassade sich in der Java-Klassenbibliothek nachweisen lässt.

21.3 Die Fassade in der Klassenbibliothek

Ich möchte Ihnen ein Beispiel aus der Java-Klassenbibliothek zeigen. Sie haben eine grafische Oberfläche erstellt und möchten dem Anwender eine Nachricht anzeigen. Sie könnten nun einen `JDialog` entwerfen, der ein `BorderLayout` hat. Auf `BorderLayout.WEST` legen Sie ein Icon, das ein warnendes Ausrufezeichen anzeigt. Auf `BorderLayout.CENTER` zeigen Sie die Nachricht an. `BorderLayout.SOUTH` enthält einen `JButton`, der mit „Ok“ beschriftet ist. Wenn der Anwender daraufklickt, wird der `Event Listener` veranlassen, dass der `JDialog` geschlossen wird.

Alternativ könnten Sie sich mit der Fassade der Klasse `JOptionPane` behelfen, welche verschiedene statische Methoden mitbringt, denen Sie unterschiedliche Parameter mitgeben können, um den Dialog zu konfigurieren. Betrachten Sie die folgenden Codezeilen.

```
String frage = "Mögen Sie Design Patterns?";  
String titel = "Eine Gewissensfrage";  
JOptionPane.showConfirmDialog(null, frage, titel,  
    JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
```

Wenn Sie diesen Code ausführen, wird Ihnen eine Nachricht angezeigt, wie in Abbildung Abb. 21.2 zu sehen ist.

Die Fassade benötigt eine Vielzahl von Klassen, die sie aufruft. Um Ihnen einen Eindruck von der großen Anzahl beteiligter Klassen zu geben, drucke ich Ihnen in Abbildung

Abb. 21.2 Von „ JOptionPane“ erzeugte Nachricht

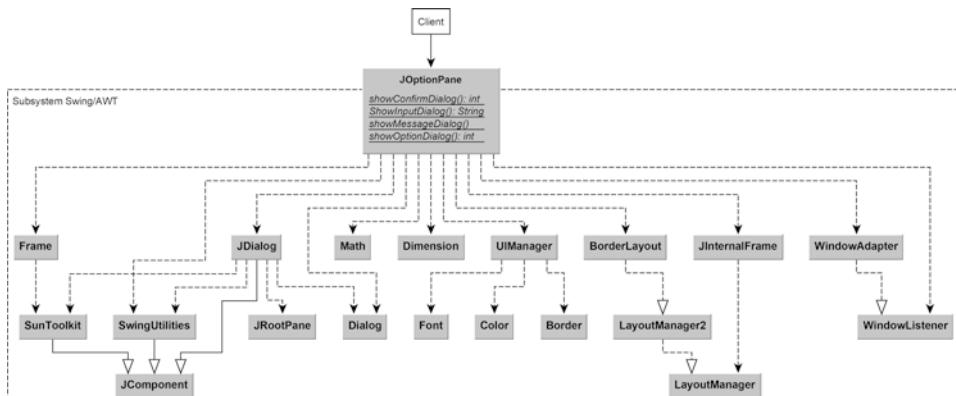


Abb. 21.3 Extrakt aus der API-Doku der Klasse JOptionPane

Abb. 21.3 eine Übersicht ab, die ich bei Philipp Hauer¹ entlehnt (und im Layout etwas angepasst) habe.

Ich denke, an diesem Beispiel wird die Bedeutung der Formulierung „vereinfachter Zugriff auf ein Subsystem“ ziemlich deutlich.

21.4 Das „Gesetz von Demeter“

In Kap. 2 habe ich Ihnen einige Entwurfsprinzipien gezeigt. Jetzt werde ich ein weiteres Entwurfsprinzip ansprechen. Es geht um das „Gesetz von Demeter“,² welches – verkürzt – besagt, dass Objekte ihre Kommunikation auf enge Freunde beschränken sollen. Die Fassade zeigt Ihnen eine Möglichkeit, wie Sie dies möglichst einfach realisieren. Und was sagt das Gesetz von Demeter nun genau? Objekte sollen nur mit engen Freunden kommunizieren. Enge Freunde sind:

¹ <https://www.philippauer.de/study/se/design-pattern/facade.php>.

² Demeter ist in der griechischen Mythologie die Mutter von Persephone und die unfreiwillige Schwiegermutter von Hades. Sie sorgt für Jahreszeiten, Fruchtbarkeit und Wachstum.

- Attribute und Methoden des eigenen Objekts, also alles, was `this` heißt,
- Methoden von Objekten, die als Parameter übergeben wurden,
- Methoden von Objekten, die die Methode selbst anlegt,
- globale Objekte.

Wenn Sie sich das Reise-Beispiel von oben ansehen, erkennen Sie, dass der Client nur auf einen Freund – die Fassade – zugreifen muss. Die Fassade gibt dem Client die Möglichkeit, sich nicht mit Fremden, also weiteren Klassen, abgeben zu müssen. Die Fassade ist ein anschauliches Beispiel für die Realisierung des Prinzips. Auf dem Blog von Matthias Kleine habe ich eine ausführliche Beschreibung des LoD gefunden. Wenn Sie sich weiter mit objektorientierten Entwurfsprinzipien und Prinzipien der Softwaretechnik beschäftigen möchten, ist diese Seite eine gute Ausgangsbasis für Ihre Recherchen:

<http://prinzipien-der-softwaretechnik.blogspot.com/2013/06/das-gesetz-von-demeter.html>

Sie finden den Text auch als „LoD.pdf“ im Verzeichnis der Beispielprojekte zu diesem Kapitel.

Im Kapitel über objektorientierte Entwurfsmuster war es mir wichtig zu sagen, dass nicht Sie als Programmierer dafür da sind, die Gesetze zu erfüllen. Vielmehr sind die Gesetze für Sie da – sie sollen Ihnen die Arbeit erleichtern. Daher gilt auch beim Gesetz von Demeter: *nulla regula sine exceptione* oder auch *Keine Regel ohne Ausnahme*. Sie dürfen gegen das Gesetz verstößen, wenn Sie sich darüber im Klaren sind und den Verstoß begründen können. Wenn Sie einen Text auf der Standardausgabe ausgeben möchten, codieren Sie weiterhin `System.out.println()`. Und wenn Sie den Namen einer Klasse wissen möchten, ist die Anweisung `myObject.getClass().getName()` weiterhin zulässig. Vorsicht ist jedoch immer dann geboten, wenn Sie auf Objekte zugreifen, die nicht Teil der Standard-Klassenbibliothek von Java sind.

Ich möchte Ihnen mit meinem Buch eine Einführung in Design Patterns anbieten. Darauf beschränke ich mich auf wenige Entwurfsprinzipien. Ich hoffe aber, dass ich Sie motivieren kann, sich in eigener Recherche mit dem Thema zu beschäftigen.

21.5 Fassade – Das UML-Diagramm

In Abb. 21.4 sehen Sie das UML-Diagramm zum Beispielprojekt Fassade, Paket demo4.

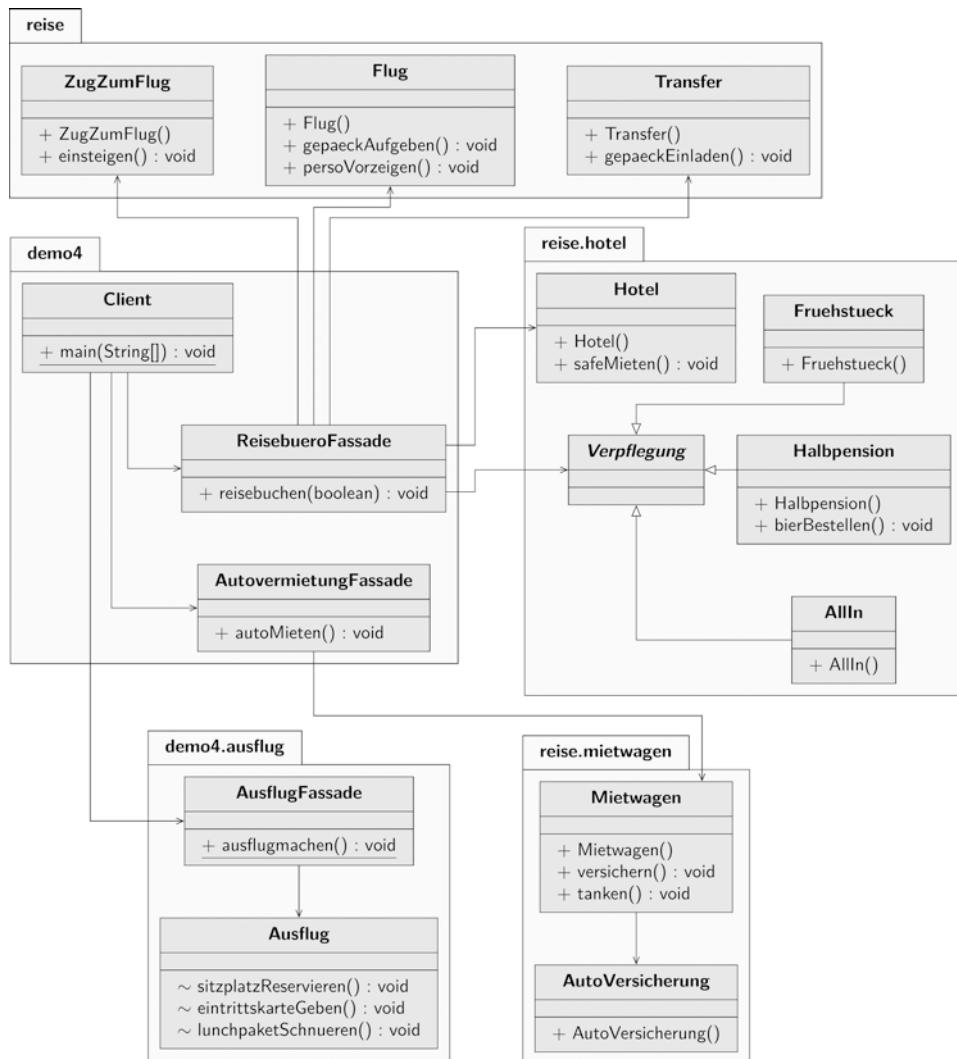


Abb. 21.4 UML-Diagramm des Facade Pattern (Beispielprojekt Fassade, Paket demo_4)

21.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Die Ausgangslage ist ein kompliziertes System, mit dem Sie arbeiten wollen.
- Der Zugriff auf das System wird durch eine Fassade vereinfacht.
- Der Client greift auf die Fassade zu und kennt nicht notwendigerweise die Einzelheiten des Systems.
- Die Fassade kann entweder Geschäftslogik enthalten oder Anfragen einfach an das zuständige System-Objekt weiterleiten.
- Das System wird austauschbar.
- Abhängigkeiten zwischen Systemen werden vereinfacht bzw. aufgelöst.
- Optional kann ein Client dennoch auf das System zugreifen.
- Das Gesetz von Demeter wird umgesetzt.

Zweckbeschreibung

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

„Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht.“



Sie fahren nach London und wollen Ihren Fön in die Steckdose stecken – da der deutsche Stecker nicht in die britische Steckdose passt, brauchen Sie einen Adapter. Ich stelle Ihnen das Muster auf der Ebene einzelner Klassen vor. Tatsächlich lässt es sich skalieren – das Prinzip ist das gleiche, wenn Sie ganze Subsysteme adaptieren.

22.1 Ein einleitendes Beispiel

Lassen Sie uns das Beispiel mit dem Stecker verallgemeinern. Sie haben zwei Systeme – den Fön und die Steckdose – die miteinander arbeiten müssen. Das „müssen“ bedeutet, dass Sie in London gar keine andere Möglichkeit haben, als an die britische Steckdose anzudocken. In der IT kann „müssen“ bedeuten, dass Sie ein System oder eine Klasse nutzen möchten, die etwas Einzigartiges oder Hochkomplexes darstellt, beispielsweise eine Beitrags- oder Fristenberechnung in der Versicherungswirtschaft. Ein komplexer Adapter ist beispielsweise auch JDBC. Das Zusammenspiel der Systeme, die zusammenarbeiten „müssen“, wird lediglich dadurch gestört, dass der Client eine andere Schnittstelle erwartet, als das zu nutzende System deklariert. Der Adapter hat die Aufgabe, zwischen beiden Systemen zu vermitteln. Die GoF beschreibt den Adapter in zwei Ausprägungen: objekt- und klassenbasiert. Beide Entwürfe kommen in diesem Kapitel zur Sprache.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_22

22.2 Ein klassenbasierter Entwurf

Welche Aufgabe gilt es in diesem Kapitel zu lösen? Schauen Sie in das Beispielprojekt Adapter_Klassenbasiert. Sie haben eine Fremdbibliothek eingekauft, die mit einem rasend schnellen Algorithmus Zahlen sortiert.

```
public class SorterFremdprodukt {
    List<Integer> sortiere(List<Integer> zahlenListe) {
        // ... black box
    }
}
```

Sie wissen von dieser Bibliothek nur, dass Sie eine Integer-Liste als Parameter mitgeben müssen und eine sortierte Integer-Liste zurückbekommen. Ihre Software ist jedoch so programmiert, dass sie ein Sorter-Objekt mit folgender Schnittstelle erwartet.

```
public interface Sorter {
    int[] sort(int... zahlen);
}
```

Der Quelltext der Fremdbibliothek liegt Ihnen nicht vor, so dass Sie daran keine Änderungen vornehmen können. Ihre eigene Applikation ist ein etabliertes getestetes Verfahren, das schon an viele Kunden ausgeliefert wurde – es darf keinesfalls mehr geändert werden. Sie müssen also einen Adapter entwerfen, der zwischen Fremdbibliothek und Ihrer Anwendung vermittelt. Wie sieht so ein Adapter aus? Der klassenbasierte Ansatz sieht vor, dass Sie den Adapter von der Fremdbibliothek erben lassen. Gleichzeitig nimmt der Adapter die Rolle eines Sorter-Objekts ein, implementiert also das erwartete Interface.

```
public class SorterAdapter extends SorterFremdprodukt implements Sorter {
    @Override
    public int[] sort(int[] zahlen) {
        // ... behandeln wir später
    }
}
```

Der Client kann nun so seine Nachricht senden, wie er es erwartet.

```
public class Client {
    public static void main(String[] args) {
        final int[] zahlen = { 9, 4, 7, 3, 5 };
        var sorter = new SorterAdapter();
        final var sortiert = sorter.sort(zahlen);
```

```
        for (var zahl : sortiert)
            System.out.println(zahl);
    }
}
```

Die sort-Methode des Adapters setzt das Zahlen-Array in eine Liste um und ruft die richtige Methode auf der Fremdbibliothek auf. Der Adapter tut also so, als wäre er ein Sorter, ist aber in Wirklichkeit eine Subklasse der Fremdbibliothek. Hier nun der gesamte Code des Adapters:

```
public class SorterAdapter extends SorterFremdprodukt implements Sorter {
    @Override
    public int[] sort(int[] zahlen) {
        int z = new int[zahlen.length];
        List<Integer> zahlenListe = new ArrayList<>();
        for (var zahl : zahlen)
            zahlenListe.add(zahl);
        List<Integer> sortierteListe = sortiere(zahlenListe);
        for (var i = 0; i < sortierteListe.size(); i++)
            z[i] = sortierteListe.get(i);
        return z;
    }
}
```

Die GoF beschreibt den klassenbasierten Ansatz mittels Mehrfachvererbung (in C++). Da Java keine Mehrfachvererbung kennt, müssen Sie den Kompromiss eingehen, dass Sie die erwartete Schnittstelle durch ein Interface deklarieren.

22.3 Ein objektbasierter Entwurf

Wenn mit dem klassenbasierten Entwurf das Problem schon gelöst ist, könnten wir doch eigentlich das Buch zu machen, oder? Zwei Gründe sprechen dagegen, zu schnell zufrieden zu sein. Der erste Grund ist, dass ein Entwurfsprinzip lautet: Ziehe Komposition der Vererbung vor. Der zweite Grund ist ganz trivial – der Anbieter der Fremdbibliothek hat in einem Update die Klasse als final markiert.

```
public final class SorterFremdprodukt {
    // ... gekürzt
}
```

Der Adapter der vorigen Lösung kann nun nicht mehr verwendet werden. Ihnen bleibt nur, einen objektbasierten Adapter zu programmieren. Wie geht das? Sie legen ein Attribut an, das eine Referenz auf ein Objekt der Fremdbibliothek hält. Die sort-Methode ruft auf diesem Attribut die Methode `sortiere()` auf. Im Beispielprojekt Adapter_Objektbasiert finden Sie diese Variante.

```
public class SorterAdapter implements Sorter {
    private final SorterFremdprodukt fremdprodukt =
        new SorterFremdprodukt();

    @Override
    public int[] sort(int[] zahlen) {
        List<Integer> zahlenListe = new ArrayList<>();
        for (var zahl : zahlen)
            zahlenListe.add(zahl);
        var sortierteListe = fremdprodukt.sortiere(zahlenListe);
        for (var i = 0; i < sortierteListe.size(); i++)
            zahlen[i] = sortierteListe.get(i);
        return zahlen;
    }
}
```

Sie kennen nun beide Ansätze – objekt- und klassenbasiert. Im folgenden Abschnitt betrachten Sie die Vor- und Nachteile beider Ansätze.

22.4 Kritik am Adapter Pattern

Betrachten Sie bitte noch mal die Clients der beiden Ansätze. Sie sehen, dass der Client die Zielschnittstelle und den Adapter kennt. Alles, was jenseits des Adapters liegt, ist für den Client nicht ersichtlich. Daher kann hinter dem Client eine andere Klasse oder ein ganzes System liegen. Da der Client das System hinter dem Adapter nicht kennt, kann dieses problemlos ausgetauscht werden. Wichtig ist nur, dass der Adapter richtig implementiert ist. Im Abschn. 21.4 haben Sie sich mit dem Law of Demeter beschäftigt. Auch das Adapter-Muster hilft Ihnen, Systeme zu entwickeln, bei denen Klassen nur mit engsten Freunden kommunizieren.

Lassen Sie mich in diesem Zusammenhang den Adapter von der Fassade abgrenzen. In beiden Fällen haben Sie ein System, auf das ein Client zugreifen möchte. Er greift nicht direkt auf dieses System zu, sondern über eine weitere Abstraktion. Der wesentliche Unterschied ist die Zielrichtung. Bei der Fassade musste der Zugriff auf ein komplexes oder kompliziertes System erleichtert werden. Der Adapter hat die Aufgabe, zwei Systeme

überhaupt in die Lage zu versetzen, miteinander zu kommunizieren. Damit liegt ein weiterer Unterschied auf der Hand: Wer erstellt die Abstraktion? Bei der Fassade war eindeutig, dass der Anbieter des Systems eine Fassade erstellen muss; er stellt eine vereinfachte Handhabung zur Verfügung. Beim Adapter sieht es in der Regel anders aus. Der Hersteller eines Systems investiert viel Zeit und Arbeit in die Erstellung seiner Software. Er entwickelt eine Schnittstelle, an die der Client seine Anfragen richten kann. Damit ist sein Job getan. Wenn der Client eine andere Schnittstelle braucht, muss er sie sich erstellen und einen Adapter definieren.

Ein Adapter ist in der Regel eine Klasse, die für einen einzigen Anwendungsfall geschrieben und optimiert wurde. Der Adapter, den ich im oberen Beispiel entwickelt habe, lässt sich nur schwer wiederverwenden, was in der Literatur gelegentlich als Nachteil beschrieben wird. Sie werden in solchen Fällen zweifelsohne überlegen müssen, ob es besser ist, eine der Schnittstellen zu überarbeiten. Die Schnittstelle können Sie aber nicht überarbeiten, wenn der Adapter Ihnen den Zugriff aufs Internet oder auf eine Datenbank (JDBC) ermöglicht.

Wenn Sie den klassenbasierten Ansatz wählen, kann der Adapter problemlos Methoden der zu adaptierenden Klassen überschreiben. Er ist aber dann an eine Klasse gebunden, weil er selbst Unterklasse der zu adaptierenden Klasse geworden ist. Für jede Unterklasse der zu adaptierenden Klassen muss ein neuer Adapter entwickelt werden. Wie ist das beim objektbasierten Ansatz? Im Abschn. 2.4 habe ich das Liskov'sche Substitutionsprinzip angesprochen, das besagt, dass eine Unterklasse sich genauso verhalten soll wie ihre Basisklasse; anders ausgedrückt: Eine Unterklasse muss ihre Basisklasse vertreten können. Wenn eine Vererbungshierarchie dieses Prinzip berücksichtigt, ist es möglich, dass ein objektbasierter Adapter nicht nur für eine Klasse, sondern auch für deren Unterklassen verwendet werden kann.

Background Information

Wussten Sie eigentlich, dass das Adapter Pattern sich schon bei den Gebrüdern Grimm nachweisen lässt? Sicher kennen Sie das Märchen Rotkäppchen. Der Wolf verkleidet sich als Großmutter; Rotkäppchen sieht ein Objekt im Bett liegen, das der Schnittstelle Grossmutter entspricht, und schöpft zunächst keinen Verdacht. Später erkennt es jedoch, dass hinter der Schnittstelle ein klassenbasierter WolfAdapter verborgen ist – sonst wäre es niemals zu dem legendären Zitat gekommen: „Großmutter, warum hast Du so ein entsetzlich großes Maul?“ Der Adapter ist also offensichtlich schlecht implementiert und löst eine WolfException aus.

Das Märchen kann uns lehren, dass es immer eine schlechte Idee ist, wenn die Leute sich selbst irgendwelche Lösungen zusammenstricken, ohne vorher mal in der IT nachgefragt zu haben.

22.5 Ein Etikettenschwindel

Sie finden in der Klassenbibliothek eine Vielzahl von Klassen, die „Adapter“ im Namen haben. In Swing gibt es beispielsweise MouseAdapter, FocusAdapter usw. Die Klasse MouseAdapter implementiert die Schnittstellen MouseListener, MouseMotionListener und MouseWheelListener. Sämtliche Methoden werden leer überschrieben, so dass Sie einen MouseListener definieren können, ohne selbst alle Methoden überschreiben zu müssen. Wenn Sie eine Methode aus der Schnittstelle MouseMotionListener überschreiben wollen, erweitern Sie den MouseAdapter und überschreiben gezielt nur diese eine Methode.

Handelt es sich dabei um einen Adapter im Sinne des Adapter Patterns? Nein – die Klasse MouseAdapter bringt Ihnen keinen Mehrwert, sie vermittelt auch nicht zwischen zwei Systemen. Sie erleichtert Ihnen jedoch den Umgang mit den genannten Schnittstellen. Ein vereinfachter Zugang zu einem Subsystem war das Kennzeichen der Fassade. Die Klasse MouseAdapter müsste also MouseFacade heißen. Nicht überall, wo Adapter draufsteht, ist also auch Adapter drin.

22.6 Adapter – Das UML-Diagramm

In den Abbildungen Abb. 22.1 und 22.2 sehen Sie im Vergleich die UML-Diagramme aus den beiden Beispielprojekten Adapter_Klassenbasiert und Adapter_Objektbasiert.

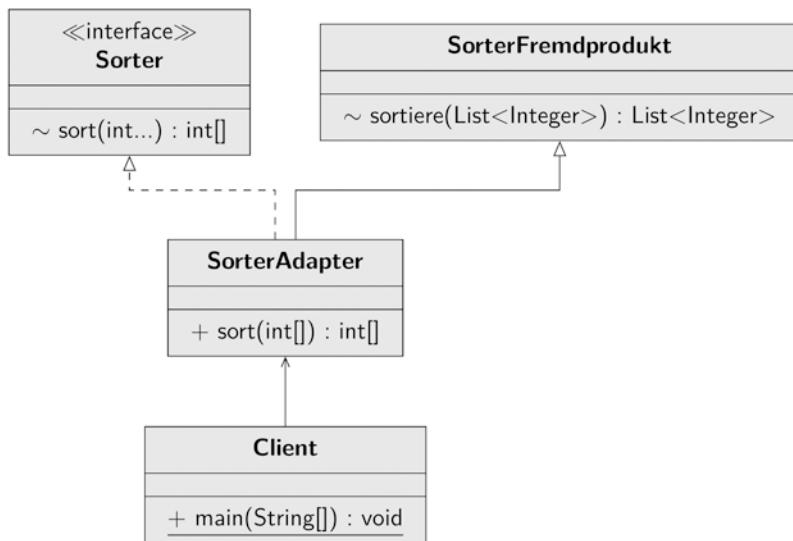


Abb. 22.1 UML-Diagramm des Adapter Pattern (Beispielprojekt Adapter_Klassenbasiert)

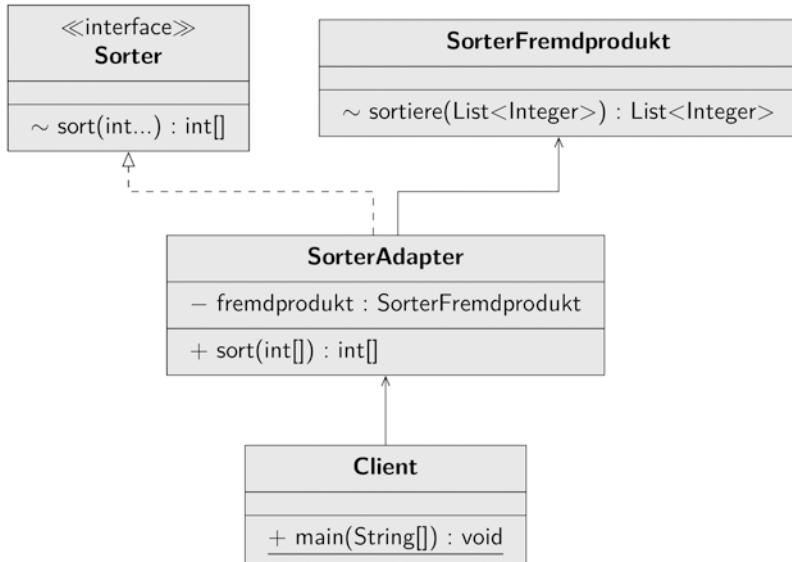


Abb. 22.2 UML-Diagramm des Adapter Pattern (Beispielprojekt Adapter_Objektbasiert)

22.7 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Es gibt zwei Systeme, deren Schnittstellen inkompatibel sind.
- Ein Adapter vermittelt zwischen beiden Systemen.
- Er konvertiert die Schnittstelle des zu adaptierenden Systems in die vom Client erwartete.
- Die ursprünglichen Klassen werden dabei nicht verändert.
- Ein Adapter wird für genau einen Anwendungsfall geschrieben.
- Client und adaptiertes System sind lose gekoppelt.
- Beide Systeme können unabhängig voneinander ausgetauscht werden, solange der Adapter passt.
- Den Adapter gibt es in zwei Ausprägungen: Objekt- und klassenbasiert.

Zweckbeschreibung

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

„Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatisabler Schnittstellen ansonsten dazu nicht in der Lage wären.“



Das Proxy Pattern, das ich Ihnen in diesem Kapitel zeigen werde, ist ein Strukturmuster und weist Ähnlichkeiten mit der Fassade und dem Adapter auf. Bei allen drei Mustern greifen Sie nicht direkt auf ein Objekt zu, sondern über ein drittes. Erinnern Sie sich – die Fassade hat Ihnen den Umgang mit einer komplizierten Schnittstelle ermöglicht. Beim Adapter hat das dritte Objekt das Zusammenspiel von zwei anderen erst ermöglicht. Und wie wird das beim Proxy sein? Beim Proxy kapseln Sie ebenfalls den Zugriff auf ein Objekt. Aber warum sollten Sie das machen wollen? Vier Gründe können sein:

1. Virtual Proxy: Sie brauchen für eine bestimmte Dauer einen Platzhalter, um ein großes Objekt zu erzeugen.
2. Security Proxy: Sie möchten den Zugang zum Objekt sperren oder zumindest kontrollieren.
3. Smart Reference: Sie möchten das Objekt um eine Funktionalität erweitern; beispielsweise soll vor dem eigentlichen Zugriff geprüft werden, ob das Objekt gelockt ist.
4. Remote Proxy: Das gewünschte Objekt liegt nicht im gleichen Adressraum.

Diese vier Anwendungsbereiche sind die häufigsten; sie werden von der GoF besprochen. In der Literatur finden Sie jedoch weitere Proxys: den Firewall-Proxy, den Synchronisation-Proxy und viele mehr. Dieses Kapitel gehört zu den längeren in diesem Buch. Machen Sie es sich also gemütlich, dann legen wir los.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_23

23.1 Virtual Proxy

Das Virtual Proxy soll hier nur der Vollständigkeit halber theoretisch angesprochen werden. Stellen Sie sich vor, Sie erstellen eine Software, mit der Fotos angezeigt werden können. In einem bestimmten Ordner sollen beispielsweise 100 Bilder liegen. Bei der gewählten Einstellung passen 25 Bilder auf eine Bildschirmseite. Wenn Sie den Ordner öffnen, müssen zuerst alle Bilder registriert, deren Größe ermittelt, die Vorschauen erstellt und gezeichnet werden. Wenn die Fotos ziemlich groß sind, kann dieser Vorgang durchaus eine Weile dauern. Es wäre nun für den Anwender sehr unbefriedigend, wenn Ihr Programm einfrieren würde. Sehr viel komfortabler ist es, wenn nur von den Bildern die Vorschau erzeugt wird, die gerade am Bildschirm angezeigt werden. Der Bildschirm zeigt zunächst 25 Rahmen als Platzhalter für die späteren Vorschauen. Nach und nach werden diese Platzhalter, die Proxys, durch die eigentlichen Vorschaubilder ersetzt. Mit den folgenden Bildern verfahren Sie entsprechend. Sie erzeugen Rahmen als Platzhalter und wenn der Anwender die Rahmen in den sichtbaren Bereich des Bildschirms scrollt, werden die Vorschauen erzeugt.

23.2 Security Proxy

Es kann sinnvoll sein, den schreibenden Zugriff auf ein Objekt zu verhindern. Nehmen Sie als Beispiel die Methode `unmodifiableList()` der Klasse `Collections` in der Java-Klassenbibliothek. Sie übergeben der Methode ein List-Objekt und erhalten eine „read-only“-Liste zurück. Die Methode packt Ihre Liste in ein Objekt der inneren Klasse `UnmodifiableList` ein. Die `UnmodifiableList` erweitert die ebenfalls innere Klasse `UnmodifiableCollection`. Wenn Sie nun auf die eigentliche Datenbasis zugreifen wollen, greifen Sie nicht auf die ursprüngliche Liste zu, sondern auf die sie umgebende `UnmodifiableList`. Die `UnmodifiableList` leitet Anfragen an die ursprüngliche Datenbasis weiter – Schreibzugriffe ausgenommen. Schauen Sie in das Beispielprojekt `Unmodifiable`; dort habe ich die relevanten Code-Anteile der Klasse `Collections` aus der Klassenbibliothek extrahiert. Diese Datei ist in diesem Ausschnitt nicht vollständig und wird daher nicht übersetzt werden können. Aber dafür ist sie auch nicht vorgesehen, sondern nur zum Nachschlagen für Sie. Sie können natürlich alternativ direkt im Quelltext der Klassenbibliothek nachschauen, wenn sie ihn ebenfalls heruntergeladen oder verlinkt haben.

```
class UnmodifiableList<E> extends UnmodifiableCollection<E>
    implements List<E> {
    final List<? extends E> list;

    UnmodifiableList(List<? extends E> list) {
        super(list);
        this.list = list;
```

```
}

@Override
public E get(int index) {
    return list.get(index);
}

@Override
public E set(int index, E element) {
    throw new UnsupportedOperationException();
}
// ... gekürzt
}
```

Für den Test erzeugt ein Client eine ArrayList mit verschiedenen Strings. Dann lässt er sich von der Klasse Collections eine read-only-Liste zurückgeben. Diese Liste ist dann keine Instanz der Klasse ArrayList mehr. Sie ist vielmehr ein Objekt der Klasse UnmodifiableRandomAccessList. Beim Versuch, einen weiteren String hinzuzufügen, wird erwartungsgemäß eine UnsupportedOperationException geworfen.

```
List<String> text = new ArrayList<>();
System.out.println(text.getClass().getName());
text.add("alpha");
text.add("bravo");
text.add("charly");
text.add("delta");
text.add("echo");
text = Collections.unmodifiableList(text);
System.out.println(text.getClass().getName());
Thread.sleep(2);
text.add("foxtrott");
```

Auf der Konsole wird ausgegeben:

```
java.util.ArrayList
java.util.Collections$UnmodifiableRandomAccessList
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.base/java.util.Collections$UnmodifiableCollection.add(
Collections.java:1060)
    at Test.main(Test.java:36)
C:\...\build-impl.xml:111: The following error occurred while executing
this line:
C:\...\build-impl.xml:68: Java returned: 1
```

Die UnmodifiableRandomAccessList ist eine Unterklasse der UnmodifiableList, die verwendet wird, um die Performance zu steigern. Die Dokumentation in der Collections-Bibliothek sagt dazu: „Many of the List algorithms have two implementations, one of which is appropriate for RandomAccess lists, the other for „sequential.“ Often, the random access variant yields better performance on small sequential access lists.“ Am Prinzip ändert sich jedoch nichts: Der Client greift nur noch über den Proxy auf das eigentliche Ziel-Objekt zu.

23.3 Smart Reference

Die Smart Reference fügt der vertretenen Klasse weitere Funktionen hinzu, um sie beispielsweise zu kontrollieren. Im ersten Schritt zeige ich Ihnen eine Version, die mit dem Proxy (noch) nichts zu tun hat. Sie soll Ihnen nur das Projekt vorstellen.

23.3.1 Die Grundversion

Das Beispiel in diesem Abschnitt geht auf das Beispiel des Adapter Patterns zurück. Sie finden es im Beispielprojekt Proxy_1: Es gibt eine Fremdbibliothek, die einen Algorithmus anbietet, der ziemlich einzigartig ist. Der superstrengeheime Algorithmus sortiert eine Liste auf hocheffiziente Weise.

```
public final class SorterFremdprodukt {
    List<Integer> sortiere(List<Integer> zahlenListe) {
        // ... gekürzt
    }
}
```

Ihr Client arbeitet intern jedoch mit einem int-Array. Um dem Client die Arbeit mit der Fremdbibliothek zu ermöglichen, bietet ein zwischengeschaltetes Objekt eine Methode, um ein int-Array in eine Integer-List zu konvertieren. Die so veränderte Datenbasis kann zum Sortieren an die Fremdbibliothek weitergereicht werden. Die Fremdbibliothek bietet schließlich eine Methode an, um die sortierte Liste in ein Array zurück umzuwandeln. Die Schnittstelle Sorter deklariert diese Methoden.

```
public interface Sorter {
    List<Integer> convertToList(int[] zahlen);
    int[] convertToArray(List<Integer> zahlenListe);
    List<Integer> sort(List<Integer> zahlenListe);
}
```

Eine Helferklasse implementiert diese drei Methoden.

```
public class SorterHelper implements Sorter {  
    private final SorterFremdprodukt fremdprodukt =  
        new SorterFremd-  
produkt();  
  
    @Override  
    public List<Integer> convertToList(int... zahlen) {  
        // ... gekürzt  
    }  
  
    @Override  
    public int[] convertToArray(List<Integer> zahlenListe) {  
        // ... gekürzt  
    }  
  
    @Override  
    public List<Integer> sort(List<Integer> zahlenListe) {  
        // ... gekürzt  
    }  
}
```

Der Client definiert mit der Methode `createArray()` eine bestimmte Anzahl unsortierter Zufallszahlen. Danach legt er eine Instanz der Klasse `SorterHelper` an, lässt von dieser das Array mit der Datenbasis in eine Liste konvertieren, sortiert diese Liste, konvertiert sie zurück in ein Array und verarbeitet das Array weiter.

```
public static void main(String[] args) {  
    var anzahl = 100;  
    var zahlen = new int[anzahl];  
    zahlen = createArray(zahlen);  
    var sorter = new SorterHelper();  
    var convertedList = sorter.convertToList(zahlen);  
    var sortedList = sorter.sort(convertedList);  
    var sortedArray = sorter.convertToArray(sortedList);  
  
    for (var zahl : sortedArray)  
        System.out.println(zahl);  
}
```

Der Anwender fragt nach, ob es sich lohnt, die Client-Software so umzustellen, dass sie intern mit einer Liste arbeitet. Die Konvertierung würde damit entfallen. Sie möchten herauszufinden, ob die Zeit, die benötigt wird, um die Daten zu konvertieren, so erheblich ist, dass sich ein Refactoring lohnt. Hierfür messen Sie Zeit für die Konversionen.

23.3.2 Einführung eines Proxys

Um eine Zeitmessung vorzunehmen, entwickeln Sie eine Klasse, die dieselben Methoden implementiert wie die ursprüngliche Klasse SorterHelper. Der Proxy kann ein Objekt vom Typ Sorter referenzieren. Am Beispiel der Methode sort() wird das Vorgehen beschrieben. Die Methode speichert zunächst den aktuellen Zeitstempel, führt auf der Sorter-Instanz die eigentliche Methode aus und misst anschließend wieder die Zeit. Die Differenz der zwei Zeitstempel wird ermittelt und formatiert auf der Konsole ausgegeben. Diesen Code finden Sie im Beispielprojekt Proxy_2.

```
public class SorterTimeProxy implements Sorter {  
    private final Sorter sorter;  
  
    private final SimpleDateFormat dateFormat = new  
        SimpleDateFormat("HH:mm:ss:SSS");  
  
    SorterTimeProxy(Sorter sorter) {  
        this.sorter = sorter;  
    }  
  
    @Override  
    public List<Integer> convertToList(int... zahlen) {  
        // ... gekürzt  
    }  
  
    @Override  
    public int[] convertToArray(List<Integer> zahlenListe) {  
        // ... gekürzt  
    }  
  
    @Override  
    public List<Integer> sort(List<Integer> zahlenListe) {  
        var start = Instant.now();  
  
        zahlenListe = sorter.sort(zahlenListe);  
  
        var ende = Instant.now();  
        var dauer = Duration.between(start, ende);  
        var zeit = dauer.toMillis();  
        System.out.println("sort: " + dateFormat.format(zeit));  
  
        return zahlenListe;  
    }  
}
```

Die sort-Methode lässt sich in vier Schritte unterteilen:

1. Code vor dem Aufruf – Pre-Invoke
2. Aufruf der eigentlichen Methode – Invoke
3. Code nach dem Aufruf – Post-Invoke
4. Rückgabe des Ergebnisses

Der Client legt wie im vorigen Projekt ein Array mit unsortierten Zufallszahlen an. Anschließend erzeugt er eine Instanz der Klassen SorterHelper und SorterTimeProxy. Dem Konstruktor der Klasse SorterProxy übergeben Sie die Instanz der Klasse SorterHelper. Die Befehle zum Konvertieren und Sortieren der Datenbasis übergeben Sie dem Proxy.

```
public static void main(String[] args) {  
    var anzahl = 5000000;  
    var zahlen = new int[anzahl];  
    zahlen = createArray(zahlen);  
    var sorter = new SorterHelper();  
    var sorterTimeProxy = new SorterTimeProxy(sorter);  
    var convertedList = sorterTimeProxy.convertToList(zahlen);  
    var sortedList = sorterTimeProxy.sort(convertedList);  
    var sortedArray = sorterTimeProxy.convertToArray(sortedList);  
}
```

Jede Methode, die aufgerufen wird, führt einerseits ihr eigenes Verhalten aus, löst andererseits aber auch das gewünschte Verhalten des Ziel-Objekts aus. Durch den vorgesetztenen Proxy messen Sie die Zeit und erhalten die sortierte Datenbasis. Auf der Konsole wird der folgende Text ausgegeben.

```
convertToList: 00:00:00:149  
sort: 00:00:02:239  
convertToArray: 00:00:00:104
```

Im nächsten Abschnitt erweitern wir diesen Ansatz noch.

23.3.3 Einen zweiten Proxy einführen

Ihr Auftraggeber wünscht sich nun, den Algorithmus loggen zu können. Sie müssen also einen zweiten Proxy schreiben. Und jetzt wird auch klar, warum es sinnvoll ist, den Proxy das gleiche Interface implementieren zu lassen wie den SorterHelper?

Sie ahnen, dass der neue Proxy, den Sie im Beispielprojekt Proxy_3 finden, ebenfalls die Rolle Sorter einnehmen muss. Und genau wie der alte Proxy hält er eine Referenz auf

ein anderes Sorter-Objekt. Am Beispiel der Methode `sort()` soll gezeigt werden, dass der Proxy erst seine eigene Aufgabe ausführt – das Logging – und anschließend die gewünschte Methode des Ziel-Objekts aufruft.

```
public class SorterLogProxy implements Sorter {  
    private final Sorter sorter;  
  
    SorterLogProxy(Sorter sorter) {  
        this.sorter = sorter;  
    }  
  
    @Override  
    public List<Integer> convertToList(int... zahlen) {  
        // ... gekürzt  
    }  
  
    @Override  
    public int[] convertToArray(List<Integer> zahlenListe) {  
        // ... gekürzt  
    }  
  
    @Override  
    public List<Integer> sort(List<Integer> zahlenListe) {  
        // Pre-Invoke  
        var strAnzahl = Long.toString(zahlenListe.size());  
        print("Sortiert wird eine Liste mit " + strAnzahl + " Zahlen");  
  
        // Invoke und Rückgabe  
        return sorter.sort(zahlenListe);  
    }  
  
    private void print(String message) {  
        System.out.println("\tLog-Level INFO: " + message);  
    }  
}
```

Der Client bestimmt, welche Klasse an den Konstruktor übergeben wird – das kann entweder eine Instanz der Klasse `SorterHelper` oder eine Instanz der Klasse `SorterTimeProxy` sein. Der Client bestimmt also, ob er einen, keinen oder mehrere Proxys braucht.

```
public static void main(String[] args) {  
    var anzahl = 1000000;  
    var zahlen = new int[anzahl];  
    zahlen = createArray(zahlen);
```

```
var sorter = new SorterHelper();
var sorterLogProxy = new SorterLogProxy(sorter);
var sorterTimeProxy = new SorterTimeProxy(sorterLogProxy);

var convertedList = sorterTimeProxy.convertToList(zahlen);
var sortedList = sorterTimeProxy.sort(convertedList);
var sortedArray = sorterTimeProxy.convertToArray(sortedList);
}
```

Wenn Sie den Code ausführen, wird der folgende Text auf der Konsole ausgegeben:

```
Log-Level INFO: Konvertiert wird ein Array mit 1000000 Zahlen
convertToList: 00:00:00:033
Log-Level INFO: Sortiert wird eine Liste mit 1000000 Zahlen
sort: 00:00:00:368
Log-Level INFO: Konvertiert wird eine Liste mit 1000000 Zahlen
convertToArray: 00:00:00:024
```

Der Client kann den Proxy vor den eigentlichen Aufruf des Ziel-Objekts schalten und so den Zugriff kontrollieren oder steuern.

23.3.4 Dynamic Proxy

Wenn Sie den Code der Proxy-Klassen analysieren, stellen Sie fest, dass das Pre-Invoke und das Post-Invoke in jeder Methode gleich oder sehr ähnlich sind. Code-Duplikate sind meist ein Zeichen für ein schlechtes Design. Ein weiterer Nachteil kommt dazu: Die Proxy-Klassen implementieren die Schnittstellen der Zielklassen vollständig. Wenn Sie im Interface Sorter eine neue Methode einfügen, müssen sämtliche Proxy-Klassen angepasst werden und diese neue Methode implementieren.

Das Ziel dieses Abschnitts ist es, mithilfe von Reflection gemeinsame Codeanteile zu isolieren und die Proxy-Klassen automatisch erzeugen zu lassen. Anders ausdrückt: Das Verhalten wird extrahiert und die Proxy-Klassen lassen Sie sich zur Laufzeit generieren. Klingt irre? Ist es auch! Gehen wir schrittweise vor.

23.3.4.1 Das Reflection API

Ich kann an dieser Stelle keine umfangreiche Einführung ins Reflection API geben. Meine Notizen hierzu beschränken sich auf das, was für Dynamic Proxy benötigt wird.

Als Programmierer hat man oft ein Interesse daran, zur Laufzeit Objekte und Klassen zu untersuchen. Zum Sprachumfang von Java gehört der Operator `instanceof`, der prüft, ob ein Objekt (auch) vom Typ der bezeichneten Klasse ist. Der Aufruf `new Student() instanceof Mensch` liefert `true` zurück, wenn die Klasse `Student` von `Mensch` erbt.

Die übrigen Werkzeuge des Reflection API finden Sie in den Packages `java.lang` und `java.lang.reflect` und dort in erster Linie in den Klassen `Class`, `Method` und `Field`. Mit dem Aufruf `Class.forName()` laden Sie zur Laufzeit die Klasse, deren voll qualifizierten Klassennamen Sie der Methode als Parameter mitgeben. Ein `Class`-Objekt erhalten Sie, wenn Sie auf einem Objekt die Methode `getClass()` aufrufen. Das `Class`-Objekt beschreibt die Klasse, auf die es sich bezieht, aus einer Meta-Sicht. Sie können hierauf beispielsweise die deklarierten Methoden erfragen. In der main-Methode des Clients im Beispielprojekt `ProxyDynamisch` habe ich auszugsweise folgenden Code hinterlegt:

```
var sorter = new SorterHelper();
var sorterClass = sorter.getClass();
var methods = sorterClass.getMethods();
for (var tempMethod : methods)
    System.out.println(tempMethod.getName());
```

Vom `Class`-Objekt werden alle Methoden abgefragt und deren Namen auf der Konsole ausgegeben:

```
sort
convertToArray
convertToList
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
```

Die Klasse `SorterHelper` hat keine Superklasse, erbt also automatisch von der Klasse `Object`, in der die letzten neuen Methoden mit jeweils unterschiedlichen Parametern `public` deklariert sind. Die ersten drei Methoden werden im Interface `Sorter` vorgeschrieben und von `SorterHelper` implementiert. Der Aufruf `sorterClass.getClassLoader()` gibt ein Objekt vom Typ `ClassLoader` zurück. Der Klassenlader ist dafür zuständig, eine Klasse in den Speicher zu laden.

23.3.4.2 Der InvocationHandler

Für das Dynamic Proxy brauchen Sie zwei Komponenten: den `InvocationHandler` und den `Proxy`. Der `InvocationHandler` beschreibt das Verhalten aller Methoden; die `Proxy`-Klasse wird dynamisch erzeugt und hat die gleiche Aufgabe wie der `Proxy` aus dem vorigen Projekt. In diesem Abschnitt wird der `InvocationHandler` besprochen; er beschreibt in allge-

meiner Form, wie die Methoden sich zu verhalten haben. Um den InvocationHandler zu generieren, muss eine Klasse das Interface InvocationHandler implementieren und die Methode `invoke()` definieren. An diese Methode werden das Proxy-Objekt, die aufzurufende Methode und ein Array mit den Parametern an die aufzurufende Methode übergeben. Neben `invoke()` kann der InvocationHandler eigenes Verhalten und eigene Datenfelder definieren. Der InvocationHandler im Beispielprojekt ProxyDynamisch hält beispielsweise eine Referenz auf das Sorter-Objekt, dessen Methoden aufgerufen werden. Außerdem speichert der InvocationHandler, wie oft die Fremdbibliothek aufgerufen wurde. Der wohl wichtigste Aufruf innerhalb der Methode ist der Aufruf der Methode `invoke()` auf dem Parameter `Method method`. Dieser Methode übergeben Sie das Ziel-Objekt, auf dem die Methode ausgeführt werden soll. Außerdem übergeben Sie die Parameter an die Methode. Im Beispiel können die Parameter einfach weitergereicht werden. Den Rückgabewert der Methode des Ziel-Objekts reichen Sie ebenfalls einfach weiter.

```
public class TimeHandler implements InvocationHandler {  
    // ... gekürzt  
  
    private final Object object;  
    private static int aufrufe = 0;  
  
    // ... gekürzt  
  
    public static int getAufrufe() {  
        return aufrufe;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {  
        var start = Instant.now();  
  
        var result = method.invoke(object, args);  
  
        var ende = Instant.now();  
        var dauer = Duration.between(start, ende);  
        var millis = dauer.toMillis();  
        var methodName = method.getName();  
        System.out.println(methodName + " : " + dateFormat.format(-  
millis));  
        if (methodName.equals("sort"))  
            aufrufe++;  
  
        return result;    }  
}
```

Was jetzt noch fehlt, ist die eigentliche Proxy-Klasse – und die wird erst zur Laufzeit generiert, was ich im folgenden Abschnitt zeigen werde.

23.3.4.3 Die Proxy-Klasse

Wichtig ist, dass beim Dynamic Proxy die Proxy-Klasse nicht nur zur Laufzeit dynamisch **geladen** wird, sondern dass sie sogar erst zur Laufzeit **definiert** wird. Im vorigen Abschnitt haben Sie das Verhalten der Methoden beschrieben. Nun soll die eigentliche Proxy-Klasse erzeugt werden. Der Code für die Proxy-Klasse wird dynamisch zur Laufzeit definiert. Dazu rufen Sie die statische Methode `newProxyInstance()` der Klasse `Proxy` auf. Übergeben Sie ihr das Klassenlader-Objekt der Ziel-Schnittstelle, ein Array mit den Ziel-Schnittstellen und den InvocationHandler. Den Rückgabewert dieser Methode casten Sie auf die Schnittstelle `Sorter` und übergeben ihn einer Variablen vom Typ `Sorter`. Diese Variable referenziert nun eine Instanz der zur Laufzeit erzeugten Proxy-Klasse.

```
var sorter = new SorterHelper();
var timeHandler = new TimeHandler(sorter);

var zielKlassen = new Class[] {Sorter.class};

var sorterProxy = (Sorter) Proxy.newProxyInstance(
                    Sorter.class.getClassLoader(), zielKlassen, time-
Handler);
```

Das war's! Wenn Sie die main-Methode des Clients ausführen, verhält Ihr Programm sich genauso, wie wenn Sie den Proxy selbst generiert hätten.

Wenn Sie noch nie mit dem Reflection API gearbeitet haben, ist der Dynamic Proxy vielleicht etwas verwirrend. Vielleicht versteht man das Prinzip am ehesten dann, wenn man sich klarmacht, dass der InvocationHandler in der Methode `invoke()` das Verhalten aller Methoden im Ziel-Interface beschreibt. Der Proxy, also die Instanz der Klasse, die zur Laufzeit erst erzeugt wird, ermittelt aus dem Interface alle Methoden. Jede Methode wird mit dem gleichen Verhalten ausgestattet, nämlich einer Weiterleitung der Anfrage an den InvocationHandler: `handler.invoke()`; Analysieren Sie das Beispielprojekt. Lesen Sie unbedingt auch die API-Dokumentation der Klasse `Proxy` und des Interface `InvocationHandler`. Sie finden dort wichtige Hinweise zur Arbeit mit Dynamic Proxy.

Sie finden auch das Beispielprojekt `ProxyDynamischTable`. In diesem Projekt wird das Ergebnis-Array in einer Tabelle angezeigt. Ich habe hierzu in der main-Methode der Klasse Client unter anderem die hier abgedruckten Zeilen eingefügt:

```
var tblModel = new TableModel() {
    @Override
    public int getRowCount() {
        // Methoden des Interface TableModel
    }
};
```

```
var handler = new TimeHandler(tblModel);
var schnittstelle = new Class[] {
    TableModel.class
};
var tableModelProxy = (TableModel) Proxy.newProxyInstance(
    TableModel.class.getClassLoader(), schnittstelle,
    handler);

tblZahlen.setModel(tableModelProxy);
```

Auf der Konsole werden jetzt alle Zugriffe auf die Methoden des Interface TableModel protokolliert und ausgewertet. Um messbare Ergebnisse zu erzeugen, wird jeder Methodenaufruf zufallsgesteuert gebremst. Was ich Ihnen damit zeigen möchte, ist, dass der InvocationHandler wiederverwendet werden kann. Er wird erst durch die dynamisch erzeugte Proxy-Klasse an eine Schnittstelle gebunden.

23.4 Remote Proxy

Jedes Java-Programm wird in einer eigenen Virtuellen Maschine ausgeführt. In dieser VM werden Instanzen aller benötigten Klassen erzeugt. Diese Objekte schwimmen in der VM rum wie die Fische im Aquarium. Wenn ein Objekt Informationen von einem anderen Objekt braucht oder dessen Dienste in Anspruch nehmen will, sendet es diesem Objekt eine Nachricht. Genauso wie ein Fisch in einem Aquarium nur mit einem Fisch aus dem gleichen Aquarium kommunizieren kann, kennen Objekte nur andere Objekte auf der gleichen VM. Mit RMI haben Sie eine Möglichkeit, über den Aquariumrand hinauszuschauen.

23.4.1 Aufbau von RMI grundsätzlich

In dem Projekt, das ich Ihnen in diesem Abschnitt vorstellen möchte, werden Sie einen Client programmieren, der wissen möchte, welchen Wert Pi hat. Es gibt einen Server, der diesen Wert kennt und auf Anfrage bekanntgibt. Dieses Projekt ist ein wenig vereinfacht und deckt nicht alle Aspekte der dahinterstehenden Technologie, RMI, ab. Es zeigt aber den Bezug zum Proxy Pattern.

Ein Client kommuniziert nicht direkt mit dem Server, sondern über einen Stellvertreter, den Proxy, der auf der Client-Seite Stub genannt wird. Für den Client sieht es so aus, als wäre der Proxy das Server-Objekt. Tatsächlich aber tut der Proxy nur so, als wäre er das Server-Objekt; er verpackt die Anfrage des Clients in einen Datenstrom und schickt ihn an den Server. Der Server entpackt die Anfrage und schickt sie an das eigentliche Objekt. Umgekehrt funktioniert der Weg genauso. Das Server-Objekt schickt seine Antwort an einen Proxy, der auf Server-Seite Skeleton genannt wird. Der Skeleton verpackt die Ant-

wort in einen Datenstrom und schickt ihn über das Netzwerk zurück an den Client. Der Datenstrom beruht auf Serialisierung; daher müssen alle Parameter und Rückgabewerte serialisierbar oder ein primitiver Datentyp sein.

Das folgende Projekt baut auf kleinem Niveau diesen Proxy-Zugriff nach. Sie finden den Code im Unterverzeichnis RMI_Test, und diesmal ausnahmsweise nicht als NetBeans-Projekt. Wie man das Ganze dann zum Laufen bringt, erkläre ich weiter unten.

23.4.2 Der RMI-Server

Lassen Sie uns zunächst einen RMI-Server programmieren. Der Server soll aus zwei Klassen und einem Interface bestehen. Die Klasse `PiImpl` stellt drei Methoden zur Verfügung. Die Methode `getState()` gibt zurück, dass die Instanz lebt. Der Rückgabewert der anderen Methode – die Methode `getPi()` – ist die Zahl Pi. Die Methode `getZaehler()` schließlich gibt Auskunft darüber, wie oft die Methode `getPi()` aufgerufen wurde. Die Schnittstelle `PiIF` deklariert die Methoden `getPi()` und `getZaehler()`, die von `PiImpl` implementiert werden. Die Klasse `ServerStart` erzeugt ein Objekt der Klasse `PiImpl`, gibt diese als RMIServer bekannt und fragt in einer Endlosschleife den Zustand des Servers ab.

23.4.2.1 Die Schnittstelle PiIF

Fangen wir am besten mit der Betrachtung der Schnittstelle `PiIF` an. Die Schnittstelle legt fest, welche Methoden von einem RMI-Client aufgerufen werden können; sie erweitert die Schnittstelle `Remote`. `Remote` schreibt selbst keine Methoden vor, sondern signalisiert, dass alle von ihr abgeleiteten Schnittstellen remote aufgerufen werden können; `Remote` ist also ein reines Marker-Interface. Sofern die Kommunikation zwischen Client und Server nicht funktioniert, wird eine `RemoteException` geworfen. Da dies in jeder Methode passieren kann, muss jede Methode eine `RemoteException` deklarieren.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface PiIF extends Remote {
    public int getZaehler() throws RemoteException;
    public double getPi() throws RemoteException;
}
```

Die Klasse `PiImpl` implementiert das Interface `PiIF`.

23.4.2.2 Die Serverklasse PiImpl

Die Serverklasse PiImpl implementiert PiIF und definiert die vorgeschriebenen Methoden `getZaehler()` und `getPi()`. Darüber hinaus wird die Methode `getState()` definiert.

```
public class PiImpl implements PiIF {
    private int zaehler = 0;

    @Override
    public int getZaehler() {
        return zaehler;
    }

    @Override
    public double getPi() {
        zaehler++;
        return 3.1415;
    }

    public String getState() {
        return "Server lebt";
    }
}
```

Bitte beachten Sie, dass nur Objekte innerhalb der gleichen VM die Methode `getState()` aufrufen können. Die Methoden `getZaehler()` und `getPi()` können auch von Objekten außerhalb der eigenen VM aufgerufen werden – das sichert das Interface Remote zu.

23.4.2.3 Die Klasse ServerStart startet den Server

Die main-Methode der Klasse ServerStart erzeugt zunächst eine Instanz der Klasse PiImpl. Anschließend wird die statische Methode `UnicastRemoteObject.exportObject(pi)` aufgerufen, damit auf das Objekt überhaupt remote zugegriffen werden kann. Diese Instanz wird anschließend bei der RMI-Registry angemeldet – auch: „gebunden“. Die Registry ist das Register der Server-VM, in dem der Client den Stub erfragen kann. Im Beispiel wird die Instanz unter dem Namen „Rechner“ eingetragen. Statt `rebind()` hätten Sie auch `bind()` codieren können – die Instanz hätte dann nur ein einziges Mal eingetragen werden können.

```
public static void main(String[] args) {
    var pi = new PiImpl();
    try {
        var remote = UnicastRemoteObject.exportObject(pi, 8077);
        Naming.rebind("PiRechner", remote);
    }
```

```

    } catch (RemoteException | MalformedURLException ex) {
        ex.printStackTrace();
    }

    // ... gekürzt
}

```

Zum Schluss tritt die main-Methode in eine Endlosschleife ein, in der alle fünf Sekunden abgefragt wird, ob der Server noch aktiv ist.

```

public static void main(String[] args) {
    // ... gekürzt

    while (true) {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ex) {
            // ... gekürzt
        }
        var state = pi.getState();
        System.out.println(state);
    }
}

```

Damit sind die Server-Klassen auch schon beschrieben. Lassen Sie uns folgend betrachten, wie der Client aussieht.

23.4.3 Der RMI-Client

Die Startklasse erzeugt eine Instanz der Klasse Rechenmaschine und fragt dort die Zahl Pi ab; der zurückgegebene Wert wird auf der Konsole ausgegeben. Danach wird der Rechner angewiesen, auf der Konsole auszugeben, wie oft Pi bereits abgefragt wurde.

```

public class ClientStart {
    public static void main(String[] args) {
        var rechner = new Rechenmaschine();
        System.out.println("Pi ist: " + rechner.tellMePi());
        rechner.druckeZaehler();
    }
}

```

Aus Sicht der RMI ist der Konstruktor der Rechenmaschine interessant. Er ruft die Methode `lookup()` auf, die von der Registry eine Referenz auf das Stub-Objekt zurück-

liefert. Naming.lookup() ist so etwas wie der Blick in das Register des Servers. Zurückgeliefert wird der Stub des Service, der auf die richtige Schnittstelle gecastet wird.

```
public class Rechenmaschine {  
    private final PiIF pi;  
  
    Rechenmaschine() {  
        pi = lookup();  
    }  
  
    private PiIF lookup() {  
        PiIF result = null;  
        var serverName = "localhost";  
        var serviceName = "PiRechner";  
        try {  
            result =  
                (PiIF) Naming.lookup("rmi://" + serverName + "/" + serviceName);  
        } catch (NotBoundException |  
                 MalformedURLException | RemoteException ex) {  
            ex.printStackTrace();  
        } finally {  
            return result;  
        }  
    }  
  
    // ... gekürzt  
}
```

Auf dem Stub kann der Client nun die vorgesehenen Methoden aufrufen, als lägen diese lokal vor.

```
public class Rechenmaschine {  
    // ... gekürzt  
  
    public double tellMePi() {  
        double result = 0;  
        try {  
            result = pi.getPi();  
        } catch (RemoteException ex) {  
            ex.printStackTrace();  
        } finally {  
            return result;  
        }  
    }  
}
```

Sie kennen jetzt sämtliche Codes – lassen Sie uns im folgenden Absatz schauen, wie Sie das Projekt zum Laufen bringen.

23.4.4 Das Projekt zum Laufen bringen

Sie finden alle java-Dateien wie gesagt im Unterverzeichnis RMI_Test. Kopieren Sie diese lokal auf Ihren Rechner. Öffnen Sie ein Konsolenfenster und wechseln Sie in dieses Verzeichnis. Übersetzen Sie mit `javac *.java` die Quelldateien. Rufen Sie nun die RMI-Registry auf: `rmiregistry`. Warten Sie bitte auch ein paar Sekunden und bestätigen Sie ggf. noch eine Windows-Freigabe-Anfrage für den Netzwerkzugriff. Jetzt können Sie in einer neuen Eingabeaufforderung mit `java ServerStart` den Server starten. Warten Sie auch hier wieder (und bestätigen ggf. ebenfalls eine weitere Netzwerk-Freigabe-Anfrage), bis Sie die Meldung „Server lebt“ in der Ausgabe sehen, die alle paar Sekunden wiederholt ausgegeben wird. Wenn Sie nun in einer dritten Konsole mit `java ClientStart` den Client aufrufen, werden der Wert von Pi sowie die Zahl der seit dem letzten Serverstart beantworteten Anfragen nach Pi auf der Konsole ausgegeben.

Sollten Sie den Server zu schnell nach dem Aufruf von `rmiregistry` starten, erhalten Sie höchstwahrscheinlich zunächst eine Exception, die aber dann nach dem Aufbau der Kommunikation von den „Server lebt“-Meldungen gefolgt werden wird. Wenn Sie den Client zu früh starten, wird er dagegen mit einer Exception und der Werteangabe 0.0 für Pi abbrechen. Sie müssen ihn dann erneut starten und sollten dann die richtigen Angaben vom Server erhalten.

Sowohl der Client als auch das Server-Objekt kommunizieren mit ihren Proxys – Stub und Skeleton – ohne es zu wissen. Die Proxy-Objekte sind dafür verantwortlich, dass die Datenströme übers Netzwerk in die jeweils andere Virtuelle Maschine transportiert werden.

Den Ablauf der Kommunikation sehen Sie im Sequenzdiagramm in Abb. 23.1.

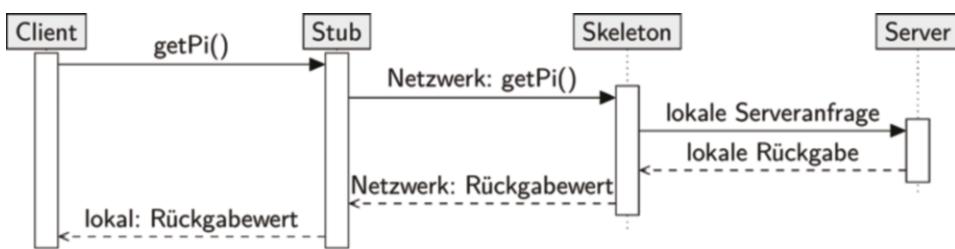


Abb. 23.1 RMI – Kommunikation über vorgeschalteten Proxy

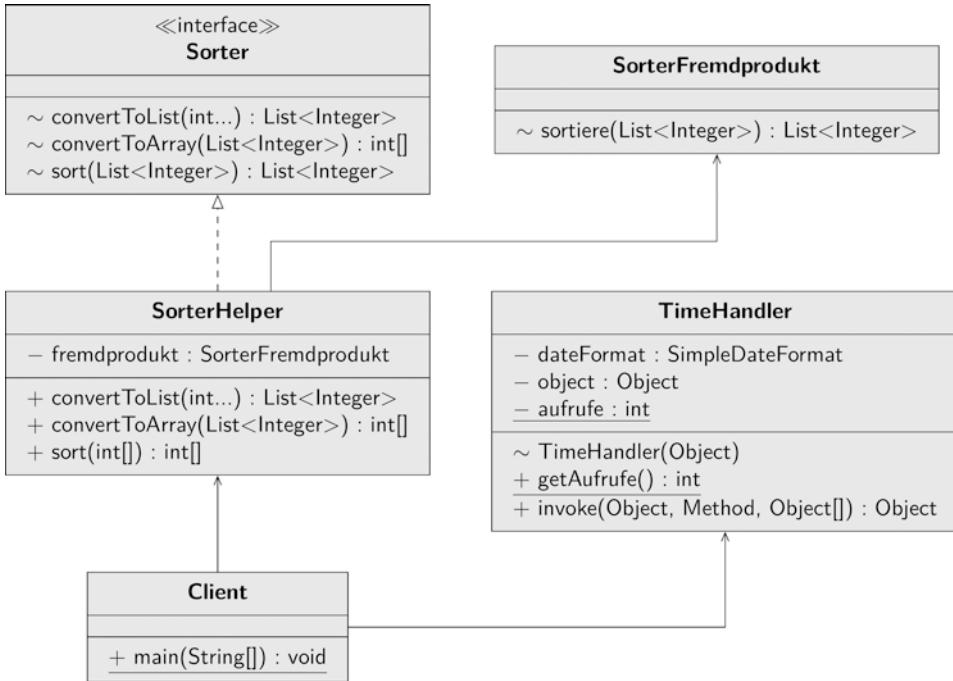


Abb. 23.2 UML-Diagramm des Proxy Pattern (Beispielprojekt ProxyDynamischTable)

23.5 Proxy – Das UML-Diagramm

In Abb. 23.2 sehen Sie das UML-Diagramm aus dem Beispielprojekt ProxyDynamischTable.

23.6 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Beim Proxy greifen Sie nicht direkt auf ein Objekt zu, sondern über dessen Stellvertreter.
- Die wesentlichen Anwendungsbereiche sind Remote Proxy, Security Proxy, Smart Reference und Virtual Proxy; es gibt aber noch weitere Bereiche.
- Virtual Proxy: Sie laden nicht alle Bilder in einem Album, sondern nur die, die gerade am Bildschirm angezeigt werden sollen.
- Security Proxy: Das eigentliche Ziel-Objekt wird in einem anderen Objekt gekapselt, das – wie im Beispiel UnmodifiableList – Schreibzugriffe verhindert.

- Smart Reference: Die Funktion eines Objekts wird mit der Motivation erweitert, Zugriffe zu protokollieren oder zu erweitern.
 - Der Proxy implementiert die Schnittstelle des Ziel-Objekts.
 - Er wird dem eigentlichen Objekt vorgelagert.
 - Der Client ruft die gewünschte Methode auf.
 - Der Aufruf wird zunächst vom Proxy bearbeitet.
 - Der Aufruf wird anschließend an das eigentliche Ziel-Objekt weitergeleitet.
 - Das Vorgehen ist unflexibel, wenn die Schnittstelle erweitert wird.
- Dynamic Proxy ist seit Java 5 eine Erleichterung für den Programmierer:
 - Der InvocationHandler definiert auf allgemeine Weise das Verhalten aller Methoden einer Schnittstelle.
 - Die Proxy-Klasse wird zur Laufzeit dynamisch erzeugt.
- Remote Proxy: ermöglicht Zugriffe auf Objekte außerhalb des eigenen Adressraums
 - Ein Client möchte auf ein Server-Objekt außerhalb des eigenen Adressraums zugreifen.
 - Der Client bekommt vom Server einen Stub, an den er seine Anfrage sendet.
 - Der Stub – der Proxy – leitet die Anfrage über das Netzwerk an den Server weiter.
 - Der Server-Proxy – der Skeleton – übergibt die Anfrage an das Server-Objekt.
 - Das Server-Objekt sendet den Rückgabewert an den Skeleton.
 - Der Skeleton schickt die Antwort an den Stub, der die Antwort dem anfragenden lokalen Objekt sendet.
 - Weder Client- noch Server-Objekt wissen, dass sie ihre Nachrichten an Proxies senden.

Zweckbeschreibung

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

„Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.“



Das Decorator Pattern – ein weiteres Strukturmuster – erweitert Objekte um Zuständigkeiten. Im letzten Kapitel haben Sie gesehen, wie der Proxy Objekte um Zuständigkeiten erweitert. Erinnern Sie sich – das eigentlich anzusprechende Objekt und die Proxys haben das gleiche Interface implementiert; ein Proxy referenziert ein anderes Objekt, ohne zu wissen, ob es sich dabei um einen weiteren Proxy oder um das gewünschte Objekt handelt. Ähnlich werden Sie beim Decorator vorgehen. Wenn ich Ihnen gleich ein Beispiel vorstelle, werden Sie den Unterschied von Proxy und Decorator vielleicht gar nicht eindeutig finden – darauf werde ich weiter unten aber noch eingehen.

24.1 Autos bauen

Werfen Sie einen Blick in das Angebot eines Autohändlers. Da gibt es wenige Grundmodelle mit gefühlten tausend Ausstattungsvarianten. Modell A ist die rollende Einkaufstasche, Modell B der bodenständige Mittelklassewagen und Modell C die gehobene Variante. Daneben gibt es für jedes Modell optional ein Navigationssystem, eine Klimaanlage, Lederausstattung usw. Lassen Sie uns überlegen, wie Sie diese Situation codieren könnten.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_24

24.1.1 Ein Attribut für jede Sonderausstattung

Man könnte zunächst auf die Idee kommen, für jedes Ausstattungsmerkmal ein Attribut vorzusehen. Wenn das Attribut null ist, wird es nicht gewünscht; ansonsten wurde es dazu bestellt.

```
public class ModelA {
    private Klimaanlage klimaanlage = null;
    private Navigation navigation = null;
    // ... gekürzt

    public void setKlimaanlage(Klimaanlage klimaanlage) {
        System.out.println("Der Kunde kauft eine Klimaanlage");
        this.klimaanlage = klimaanlage;
    }

    // ... gekürzt
}
```

Der Nachteil liegt offen auf der Hand: Jedes Objekt schleppt eine Menge unnötiger Daten mit sich herum, die gar nichts mit seinem eigentlichen Kerngeschäft zu tun haben. Es gibt zu viel doppelten Code, was das Projekt fehleranfällig und wenig wartungsfreundlich macht. Wenn Änderungen anstehen, muss bestehender Code aufgebohrt werden. Das **OpenClosed-Principle** wurde offensichtlich verletzt.

24.1.2 Mit Vererbung erweitern

Ein anderer Ansatz könnte sein, dass Sie spezialisieren, denn ein Auto mit Klimaanlage ist ein Auto. Beispielsweise könnte ModelA_Klima wie folgt definiert sein:

```
public class ModelA {
    // Eigenschaften von Model A
}

public class ModelA_Klima extends ModelA {
    // zusätzliche Eigenschaften einer Klimaanlage
}
```

Was wäre die Folge? Die Anzahl von Subklassen würde explodieren, weil Sie alle erkennbaren Kombinationen abdecken müssen. Sie brauchen beispielsweise folgende Klassen: ModelA, ModelA_Mit_Klima, ModelA_Mit_Navi, ModelA_Mit_Klima_Und_Navi, ModelB, ModelB_Mit_Klima usw. Bei drei Grundmodellen mit drei Sonderausstattungen sind das $3 * 2^3 = 24$ Subklassen. Weiter vorne war die Rede davon, dass Sie Komposition der Vererbung vorziehen – und auch hier zeigt sich, dass Vererbung nicht das Mittel zum Zweck ist. In diesem zugegeben konstruierten Bei-

spiel ist die Schwäche der Vererbung offensichtlich. Ich möchte jedoch davon ausgehen, dass es in der Praxis Fälle gibt, in denen Vererbung weniger evident zu einem starren unflexiblen System führt.

24.1.3 Dekorieren nach dem Matroschka-Prinzip

Kennen Sie das Matroschka-Prinzip? Sie haben eine schön bemalte Holzpuppe, die eine andere Holzpuppe umschließt, die ihrerseits eine weitere Holzpuppe umschließt, die selbst eine Holzpuppe usw. So in etwa funktioniert der Decorator. Jedes neue Ausstattungsmerkmal umschließt („wrappt“) das bisher zusammengesteckte Produkt. Klingt kompliziert? Ist es überhaupt nicht.

Aber beachten Sie bitte, dass wir hier im Gegensatz zum Composite-Pattern aus Kap. 12 keine verzweigten Strukturen zusammenbauen (das würde dann in ähnlich vielen Varianten ausarten, wie bei dem Ansatz mit der Vererbung), sondern die Komponenten „umeinander“ schließen. Dieses Vorgehen gibt uns die Flexibilität, in beliebiger Reihenfolge und Häufigkeit Sonderausstattungen zum Grundmodell hinzuzufügen.

24.1.3.1 Definieren der Grundmodelle

Grundmodelle und Sonderausstattungen implementieren das gleiche Interface: Komponente. In diesem Interface ist beschrieben, was jedes Element können muss, nämlich seinen Preis benennen und sich selbst beschreiben. Öffnen Sie das Beispielprojekt AutoKatalog. Sie finden dort im Package commons das Interface Komponente.

```
public interface Komponente {  
    public int getPreis();  
  
    public String getBeschreibung();  
}
```

Im Package grundmodelle finden Sie verschiedene Modelle, beispielsweise ModelC.

```
public class ModelC implements Komponente {  
    @Override  
    public int getPreis() {  
        return 50000;  
    }  
  
    @Override  
    public String getBeschreibung() {  
        return "Ein Wagen der oberen Mittelklasse";  
    }  
}
```

```
}
```

Im folgenden Absatz werden Sie die Sonderausstattungen definieren.

24.1.3.2 Definieren der Sonderausstattungen

Die Sonderausstattungen sollen von der gemeinsamen Oberklasse Sonderausstattung erben, die ihrerseits auch vom Typ Komponente ist und eine Referenz auf eine andere Komponente hält.

```
public abstract class Sonderausstattung implements Komponente {  
    protected final Komponente basisKomponente;  
  
    protected Sonderausstattung(Komponente komponente) {  
        this.basisKomponente = komponente;  
    }  
}
```

Alle Sonderausstattungen definieren die Methoden des Interface Komponente. Sie greifen dabei auf die Daten der referenzierten Komponente zu und addieren bzw. konkatenieren ihren eigenen Wert.

```
public class Klimaanlage extends Sonderausstattung {  
    public Klimaanlage(Komponente komponente) {  
        super(komponente);  
    }  
  
    @Override  
    public int getPreis() {  
        return basisKomponente.getPreis() + 500;  
    }  
  
    @Override  
    public String getBeschreibung() {  
        return basisKomponente.getBeschreibung() + " und Klimaanlage";  
    }  
}
```

Der folgende Absatz zeigt Ihnen, wie die einzelnen Komponenten zusammengesteckt werden.

24.1.3.3 Der Client steckt die Komponenten zusammen

Die main-Methode der Klasse App1Start legt zunächst das Grundmodell an und wrapppt dieses mit einem Navigationssystem. Anschließend wird das Navigationssystem mit einer Klimaanlage gewrappt. Da der Kunde nun keine weiteren Wünsche mehr hat, können Sie

auf der Klimaanlage als der äußersten Hülle `getBeschreibung()` und `getPreis()` aufrufen.

```
public class ApplStart {  
    public static void main(String[] args) {  
        Komponente grundmodell = new ModelC();  
        Komponente navigation = new Navigationssystem(grundmodell);  
        Komponente kaelte = new Klimaanlage(navigation);  
  
        System.out.println("Kundenwunsch: \n\t" + kaelte.getBeschreibung()  
                           + "\nPreis: \n\t" + kaelte.getPreis());  
    }  
}
```

Auf der Konsole wird ausgegeben:

```
Kundenwunsch:  
    Ein Wagen der oberen Mittelklasse und ein Navigtionssystem und  
    Klimaanlage  
Preis:  
    51380
```

Sie können die Grundmodelle und die Ausstattungsvarianten beliebig und vor allem flexibel zusammenstecken. Auch die Anzahl von Komponenten spielt keine Rolle mehr. Theoretisch könnten Sie zwei Klimaanlagen und drei Navigationsgeräte einbauen – sofern der Kunde das wünscht. Der Code ist robust und wartbar, da jede Klasse eine einzige Funktion hat. Hohe Kohäsion ist ein Hinweis auf ein angemessenes Design. Wenn der Preis für das Navigationssystem steigt, müssen Sie den Code nur an einer einzigen Stelle ändern. Und schließlich möchte ich noch die Terminologie des Decorator einführen: Die Schnittstelle Komponente ist eine Komponente. Jedes Grundmodell ist eine konkrete Komponente. Die Schnittstelle der Sonderausstattungen ist ein Dekorierer; die Sonderausstattungen selbst sind die konkreten Dekorierer.

Die Realisierung des Patterns ähnelt der Realisierung des Proxy Patterns. Beim Proxy habe ich Ihnen das Beispiel der `UnmodifiableList` angeboten. Im Internet und in der Literatur finden Sie dieses Beispiel teilweise auch beim Decorator. Stimmt da was nicht? Wo gehört diese Klasse denn nun hin? Um zu einer Antwort zu kommen, muss ich mir Gedanken über die Aufgabe der `UnmodifiableList` machen. Außerdem muss ich das Ziel der Patterns im Auge haben. Die `UnmodifiableList` kontrolliert den Zugriff auf eine Liste. Wenn ich sie beim Proxy einsortiere, betone ich den kontrollierenden Charakter; ein Proxy kontrolliert den Zugang zum eigentlichen System, schränkt ihn vielleicht sogar ein. Der Decorator hingegen erweitert ein System um Funktionalität, ohne es selbst zu verändern. Daher ist die `UnmodifiableList` für mich eindeutig ein Proxy.

Im nächsten Absatz werden Sie erfahren, wo Sie den Decorator in der Praxis schon angetroffen haben.

24.2 Praxisbeispiele

Mit dem Decorator Pattern erweitern Sie Swing-Komponenten um Scroll-Balken und schicken Daten über Streams.

24.2.1 Die Klasse „JScrollPane“

Sie können beliebige Komponenten mit Scroll-Balken versehen: ein JTree, ein JPanel, eine JTable usw. Dabei stecken Sie die Komponente wie die Matroschka zusammen und fügen sie einem Container hinzu.

```
JFrame frmAnzeige = new JFrame();
JTable tblDaten = new JTable();
JScrollPane scrTabelle = new JScrollPane(tblDaten);
frmAnzeige.add(scrTabelle);
```

Auf dem JFrame wird eine Tabelle mit Scrollbalken angezeigt. Sie können die Komponenten ineinanderstecken, weil alle vom Typ `JComponent` sind und der Konstruktor der `JScrollPane` ein Objekt vom Supertyp `Component` erwartet. Das Zusammenspiel finden Sie in Abb. 24.1.

24.2.2 Streams in Java

Wenn Sie sich mit Streams beschäftigen, sehen Sie den Nachteil des Decorator Patterns. Sie haben eine Vielzahl von Klassen, die auf den ersten Blick alle das Gleiche machen. Ein Blick in das Paket `java.io` bereitet keinem Programmierer so richtig Freude. Mit dem Wissen über das Decorator Pattern können Sie Ordnung in das Sammelsurium von Klassen bringen.

24.2.2.1 Streams als Decorator

Streams verbinden Datenquellen und Ziel. Zu übertragende Daten können entweder Bytes oder Characters sein. Die Vererbungshierarchien sind jeweils unterschiedlich: Byte-Streams werden durch die abstrakten Oberklassen `InputStream` und `OutputStream` verarbeitet, CharacterStreams durch Reader und Writer.

Die Klasse `FilterInputStream` ist die Oberklasse für die Decorator-Klassen. Ein Objekt dieser Instanz referenziert einen anderen `InputStream` und überschreibt alle Methoden so, dass jede Anfrage an das referenzierte Objekt weitergeleitet wird. In der API-Dokumentation des `FilterInputStream` heißt es:

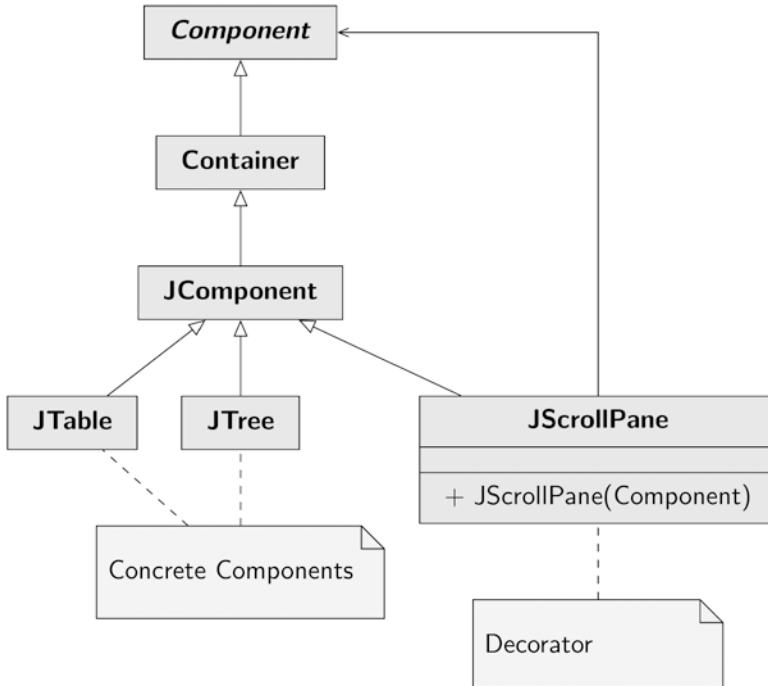


Abb. 24.1 Klassenhierarchie der Swing-Komponenten um JComponent (vereinfacht)

„A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream. Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.“

Wenn Sie das Zitat durchlesen, finden Sie genau die Zweckbeschreibung des Decorator Patterns wieder.

Es geht darum, die Funktionalität von Objekten zu erweitern, ohne Subklassen zu bilden. Eine Unterklasse von FilterInputStream ist beispielsweise BufferedInputStream, die die Daten in einen Puffer schreibt, was die Performance erhöht. Die API-Dokumentation beschreibt die Aufgabe so:

„A BufferedInputStream adds functionality to another input stream—namely, the ability to buffer the input ...“

Die Klassen FileInputStream und ByteArrayInputStream sind die zu dekorierenden Komponenten; sie erben beide von InputStream. Die Klasse FileInputStream stellt die Verbindung zu einer Datei her und ByteArrayInputStream erwartet als Datenquelle ein Array aus Bytes. Beide Klassen können byteweise von der Datenquelle lesen und mit einem BufferedInputStream dekoriert werden. Eine gezippte

Datei könnten Sie mit einem weiteren Decorator, dem `ZipInputStream`, versehen. Dazu stecken Sie die Objekte wie die Matroschkas zusammen.

```
InputStream in =
    new ZipInputStream(
        new BufferedInputStream(
            new FileInputStream( /* Dateiname */ )));
```

Die Vererbungshierarchie nach der Klasse `OutputStream` ist entsprechend aufgebaut. Sie haben dort beispielsweise einen `FileOutputStream`, der die Verbindung zu einer Datei herstellt, in die Sie Daten schreiben. Die Schnittstelle der Decorator wird durch die Klasse `FilterOutputStream` dargestellt. Und von dieser Klasse wird `BufferedOutputStream` abgeleitet:

„By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.“

Mit dem, was Sie in diesem Absatz gelesen haben, können Sie eine Datei byteweise lesen und schreiben. Den Code solch eines kleinen Kopierprogramms finden Sie im Beispielprojekt Streams_1.

```
public static void main(String[] args) {
    var chooser = new JFileChooser();
    var rueckgabeWert = chooser.showOpenDialog(null);
    if (rueckgabeWert == JFileChooser.APPROVE_OPTION) {
        var eingang = chooser.getSelectedFile();
        rueckgabeWert = chooser.showSaveDialog(null);
        if (rueckgabeWert == JFileChooser.APPROVE_OPTION) {
            var ausgang = chooser.getSelectedFile();
            try ( var in = new FileInputStream(eingang);
                  var out = new FileOutputStream(ausgang)) {
                int i;
                while ((i = in.read()) != -1)
                    out.write(i);
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

So weit ist alles gut, das Decorator Pattern können Sie in der API nachweisen.

24.2.2.2 Streams als Non-Decorator

Sie könnten auch einen Schritt weitergehen und den `FileOutputStream` in einen `DataOutputStream` einpacken, um primitive Datentypen ausgeben zu können. Sie erweitern da-

mit – ganz im Sinne des Decorator Patterns – die Funktionalität des FileOutputStream, ohne ihn zu ändern. Im Beispielprojekt Streams_2 finden Sie ein entsprechendes Beispiel.

```
private static void schreiben() {
    final var file = new File("person.txt");
    try ( var fos = new FileOutputStream(file);
          var dos = new DataOutputStream(fos);) {
        var id = 4711;
        var gehalt = 2466.77;
        var istChef = true;
        dos.writeInt(id);
        dos.writeDouble(gehalt);
        dos.writeBoolean(istChef);
    } catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Und entsprechend lesen Sie die Datei wieder aus.

```
private static void lesen() {
    final var file = new File("person.txt");
    try ( var fis = new FileInputStream(file);
          var dis = new DataInputStream(fis);) {
        var id = dis.readInt();
        var gehalt = dis.readDouble();
        var istChef = dis.readBoolean();
        System.out.println(
            "Der Mitarbeiter mit der Personalnummer " + id
            + " verdient " + gehalt + " Euro");
        if (istChef)
            System.out.println("Er ist der Chef");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Wenn eine Methode dann schreiben() und lesen() aufruft, wird auf der Konsole ausgegeben:

```
Der Mitarbeiter mit der Personalnummer 4711 verdient 2466.77 Euro
Er ist der Chef
```

Warum ist dieses Konstrukt kein Decorator? Das Decorator Pattern erlaubt es zwar, dass ein anderes Objekt die Schnittstelle erweitert. Der Client-Code darf dadurch jedoch

nicht tangiert sein, Dekoration muss für den Client transparent sein. In diesem Beispiel wäre jedoch genau das der Fall – die spezialisierten Methoden können nicht auf einem Objekt vom Typ `InputStream`, bzw. `OutputStream` aufgerufen werden. Wenn der Programmierer davon Gebrauch machen möchte, muss er gegen die Schnittstelle des `DataInput/OutputStream` programmieren.

Streams werden im Kontext Decorator Pattern in nahezu jedem Patterns-Buch – auch bei der GoF – erwähnt. Das ist so lange richtig, wie Sie die Schnittstelle nicht ändern müssen. Ich habe wiederholt gelesen (sinngemäß zitiert): „Alles was IO ist, ist auch Decorator.“ Ich denke nicht, dass eine solch absolut formulierte Aussage haltbar ist.

24.3 Decorator – Das UML-Diagramm

Sie sehen diesmal aus dem ersten Beispielprojekt AutoKatalog das UML-Diagramm in Abb. 24.2.

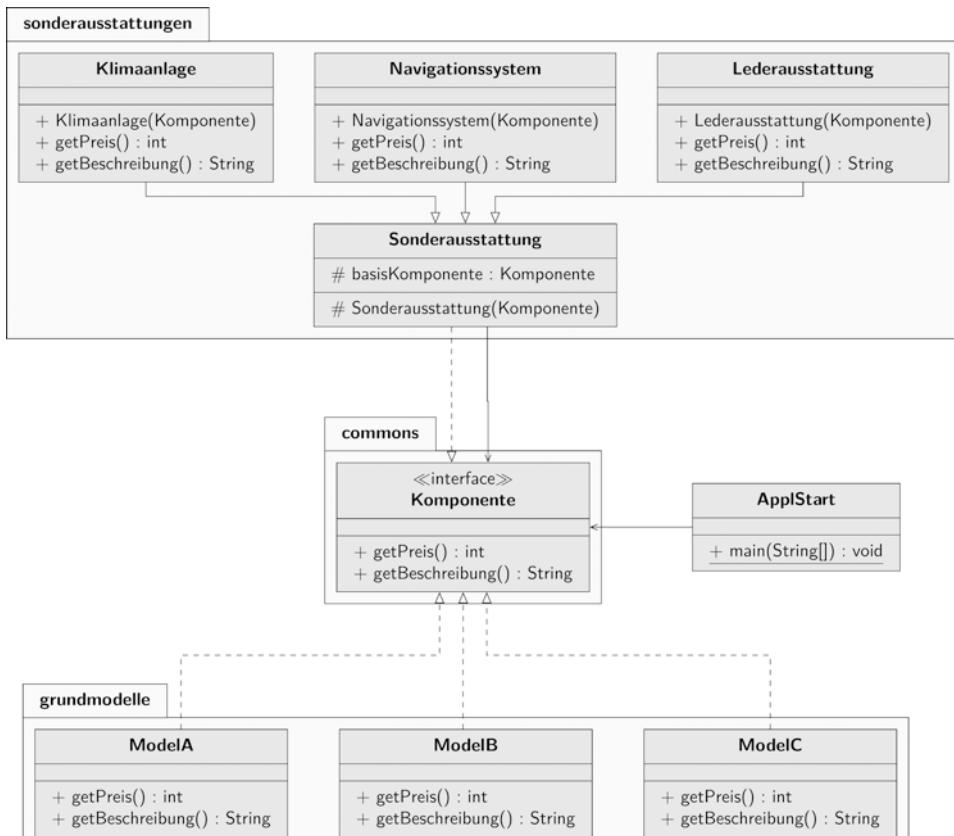


Abb. 24.2 UML-Diagramm des Decorator Pattern (Beispielprojekt AutoKatalog)

24.4 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sie müssen Objekte um Zuständigkeiten erweitern.
- Vererbung führt Sie in ein starres, unflexibles und nicht wartbares System.
- Die Erweiterung soll das zu erweiternde Objekt nicht verändern.
- Das Decorator Pattern bietet eine Alternative zur Unterklassenbildung.
- Beim Decorator gibt es Komponenten, die dekoriert werden.
- Die Decorator-Klassen implementieren die gleiche Schnittstelle wie die Komponenten.
- Die Decorator referenzieren das zu dekrierende Objekt.
- Decorator können eigene Methoden haben, müssen aber die Komponentenschnittstelle einhalten.
- Der Vorteil ist ein stabiles, flexibles und kohäsives System.
- Der Nachteil ist, dass viele ähnliche Klassen gebildet werden.
- Der Unterschied zum Proxy ist, dass der Proxy das Objekt kontrolliert, der Decorator das Objekt aber um Funktionalität erweitert.
- Im Gegensatz zum Composite gibt der Decorator keine feste Abhängigkeit und Häufigkeit vor.
- Im Gegensatz zum Adapter ist beim Decorator die Schnittelle in beide Richtungen identisch – damit auch weitere Decorator angeschlossen werden können.

Zweckbeschreibung

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

„Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.“



Das Bridge Pattern überbrückt im wahrsten Sinne des Wortes die Kluft zwischen der Implementierung einer Funktionalität und ihrer Anwendung. Es trennt die Abstraktion von der Implementierung. Das ermöglicht beiden Seiten, unabhängig voneinander weiterentwickelt zu werden. Gleichzeitig bleibt die Implementierung dem Client verborgen. Auch in der Java Klassenbibliothek finden sich dafür diverse Beispiele.

25.1 Zwei Definitionen

Ich möchte zwei Begriffe definieren, bevor ich auf das Bridge Pattern zu sprechen komme. Wenn Sie diesen Abschnitt lesen, werden Sie meine Ausführung vielleicht belanglos finden. Eine genaue Definition der Begriffe Abstraktion und Implementierung ist für das Verständnis des Bridge Patterns jedoch wichtig.

25.1.1 Was ist eine Abstraktion

Der erste Begriff, den ich ansprechen möchte, ist die Abstraktion. Wenn Sie eine Klasse entwickeln, müssen Sie sich immer entscheiden, welchen Teil der realen Welt Sie mit welcher Genauigkeit abbilden wollen. Wenn Sie eine Software schreiben, mit der eine Hochschule ihre Studenten verwaltet, werden Sie ganz bestimmte relevante Merkmale eines Studenten herauspicken und diese in Ihren Klassen berücksichtigen. Für die Hoch-

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_25

schule sind beispielsweise das Geburtsdatum, die Adresse und die bestandenen Prüfungen relevant. Irrelevant dürften die Haarfarbe, das Gewicht, die Anzahl der Geschwister und viele weitere Merkmale sein. Sie **reduzieren** in Ihrer Hochschul-Software den Studenten auf die relevanten Merkmale; Sie erstellen ein **Modell**, Sie **abstrahieren**. Eine andere Abstraktion ist beispielsweise ein Taschenrechner, der gleich für das Beispielprojekt relevant sein wird. Wenn Sie den Taschenrechner auf Ihrem Schreibtisch betrachten, dann hat er bestimmte Merkmale: ein spezielles Gewicht, eine Farbe, verschiedene Rechenoperationen usw. Wenn Sie die Klasse Taschenrechner entwickeln, ignorieren Sie zahlreiche Eigenschaften und Fähigkeiten, Sie reduzieren das reale Objekt auf die Merkmale und Fähigkeiten, die für den gegebenen Kontext relevant sind.

Das nächste Listing aus dem Beispielprojekt Taschenrechner zeigt eine solche Abstraktion. Ich drucke hier nur die Schnittstelle ab, da sie für den Client wichtig ist. Die dahinterstehende Implementierung lasse ich außen vor.

```
public class Taschenrechner {
    public double addieren(double summand_1, double summand_2) {
        // Implementierung
    }

    public double subtrahieren(double minuend, double subtrahend) {
        // Implementierung
    }

    public double multiplizieren(double faktor_1, double faktor_2) {
        // Implementierung
    }

    public double dividieren(double dividend, double divisor) {
        // Implementierung
    }
}
```

Wenn ein Client ein Objekt dieser Klasse erzeugt, ruft er die Methoden auf, die die Schnittstelle anbietet. Die Abstraktion legt fest, was das Objekt leisten kann.

25.1.2 Was ist eine Implementierung

Wenn der Client sich auf die Abstraktion stützt, muss er sich über die dahinterstehende Implementierung keine Gedanken machen. Die Details der Implementierung bleiben ihm vielleicht sogar verborgen. Es hat mich nie interessiert, wie der Süßigkeiten Automat im Büro aus einem Euro-Stück einen Schokoriegel macht. Und es interessiert mich auch nicht, wie der Taschenrechner zwei Zahlen addiert. Für mich als Anwender ist die Implementierung, das Wie einer Methode, nicht von Bedeutung.

Lassen Sie uns im folgenden Schritt die Implementierung ansehen. Im ersten Ansatz könnten Sie einfach die Schulmethoden realisieren.

```
public class Taschenrechner {  
    public double addieren(double summand_1, double summand_2) {  
        return summand_1 + summand_2;  
    }  
  
    public double subtrahieren(double minuend, double subtrahend) {  
        return minuend - subtrahend;  
    }  
  
    public double multiplizieren(double faktor_1, double faktor_2) {  
        return faktor_1 * faktor_2;  
    }  
  
    public double dividieren(double dividend, double divisor) {  
        return dividend / divisor;  
    }  
}
```

Sie möchten jetzt die Implementierung ändern. Es gibt alternative Multiplikationsverfahren, die – speziell bei Langzahlen – wesentlich performanter sind als der Multiplikationsalgorithmus, der in der Schule gelehrt wird. Wenn Sie die Implementierung ändern wollen, können Sie einfach eine Unterklasse erstellen, die die Methode multipliziere() überschreibt. Wenn Sie zwei Zahlen mit der Fast Fourier Transformation (FFT) multiplizieren wollen, verwenden Sie die folgende Klasse.

```
public class FFT extends Taschenrechner {  
    @Override  
    public double multiplizieren(double faktor_1, double faktor_2) {  
        // nicht abgedruckt – FFT-Multiplikation  
    }  
}
```

Ein Client kann sich nun aussuchen, ob er den einfachen Multiplikationsalgorithmus verwenden will oder die FFT. Dementsprechend kann er eine Instanz des Taschenrechners entweder mit

```
Taschenrechner rechner = new Taschenrechner();
```

Erzeugen, oder mit

```
Taschenrechner rechner = new FFT();
```

Übrigens finde ich die verschiedenen Multiplikationsverfahren ziemlich spannend. Wenn Sie sich mit der FFT-Multiplikation beschäftigen möchten, empfehle ich Ihnen folgende Seite: <http://www.inf.fh-flensburg.de/lang/algorithmen/fft/fft.htm>. Den dort hinterlegten Artikel von Herrn Prof. Dr. Lang finden Sie auch als „Transformationen.pdf“ im Verzeichnis der Beispielprojekte zu diesem Kapitel.

Es gibt viele weitere Multiplikationsverfahren wie beispielsweise die russische Bauernmultiplikation. Ein paar dieser Algorithmen sind hoch performant, andere bestechen durch Eleganz.

25.1.3 Ein Problem beginnt zu reifen

Gleich nachdem Sie Ihren Rechner veröffentlicht haben, kommen Erweiterungswünsche. Der Taschenrechner soll auch noch die Quadratur unterstützen. Da Sie dieses Feature gesondert vertreiben möchten, entwickeln Sie die Abstraktion, die Schnittstelle, weiter. Da quadrieren bedeutet, dass Sie eine Zahl mit sich selbst multiplizieren, können Sie die bestehenden Methoden verwenden.

```
public class TaschenrechnerDeluxe extends Taschenrechner {
    public double quadrieren(double zahl) {
        return multiplizieren(zahl, zahl);
    }
}
```

Im Ergebnis haben Sie das Klassendiagramm in Abb. 25.1.

Diese Lösung hat einen Haken. Sie hatten weiter vorne die Implementierung weiterentwickelt und hoch performante Multiplikationsalgorithmen implementiert. Doch wenn ein Kunde den TaschenrechnerDeluxe kauft, bekommt er nur die standardmäßige Implementierung des Multiplikationsverfahrens. Die FFT-Multiplikation kann er genauso wenig einsetzen wie die russische Bauernmultiplikation. Vielleicht ist die Kuh ja vom Eis, wenn der TaschenrechnerDeluxe nicht von Taschenrechner, sondern von FFT erbt.

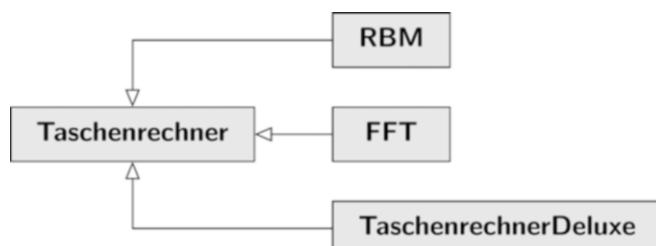


Abb. 25.1 Unvollständiges Klassendiagramm des Projekts Taschenrechner

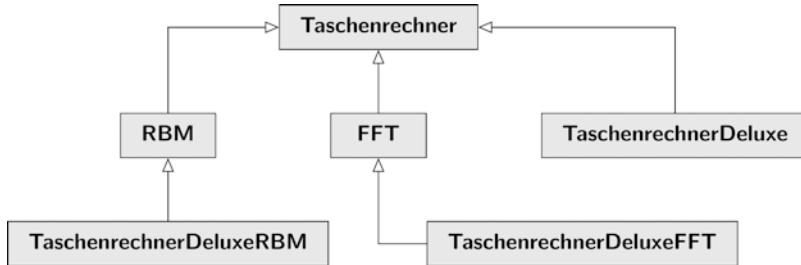


Abb. 25.2 „Weiterentwickeltes“ Klassendiagramm des Projekts Taschenrechner

```

public class TaschenrechnerDeluxeFFT extends FFT {
    public double quadrieren(double zahl) {
        // Implementierung
    }
}
  
```

Bei dieser Lösung bleiben jedoch die russische Bauernmultiplikation und die standardmäßig implementierte Schulmultiplikation außen vor. Sie kommen nicht umhin, die Vererbungshierarchie nach diesen beiden Algorithmen entsprechend zu erweitern (Abb. 25.2).

Es wird deutlich, dass die Vererbungshierarchie zu breit und zu unflexibel ist. Wenn entweder die Abstraktion erweitert wird oder die Implementierung um einen Algorithmus erweitert wird, wächst die Anzahl Klassen rasant an. Gehen Sie davon aus, dass die Division ersetzt werden soll. Der Algorithmus, den Sie aus der Schule kennen, fragt, wie oft der Divisor in den Dividenden passt. Ein alternatives Vorgehen könnte sein, dass Sie mit dem Kehrwert des Divisors multiplizieren. Dem Kehrwert einer Zahl nähern Sie sich mit einer Formel von Newton iterativ an. Würden Sie versuchen, den Divisionsalgorithmus durch Vererbung zu realisieren, müssten Sie berücksichtigen, dass Sie zwei Abstraktionen haben – den Taschenrechner und den TaschenrechnerDeluxe. Beide Abstraktionen möchten wahlweise mit FFT, der RBM oder der Schulmethode multiplizieren und wahlweise mit der Schulmethode oder dem Newtonschen Verfahren dividieren.

Fazit: Das Problem ist, dass Abstraktion und Implementierung zu stark voneinander abhängen und sich gegenseitig beeinflussen. Das Bridge Pattern wird dieses Problem lösen.

25.2 Das Bridge Pattern im Einsatz

Das Bridge Pattern trennt die Abstraktion von der Implementierung. Wie geht das? Vielleicht erinnern Sie sich an das State Pattern und das Strategy Pattern! Sie haben dort das Verhalten eines Objekts in eigenen Klassen definiert. Sie haben in Ihrer Abstraktion eine Schnittstelle vorgegeben, der die Implementierung der Klassen, die das Verhalten definieren, entsprechen muss.

25.2.1 Erster Schritt

Ähnlich geht das Bridge Pattern vor. Sie definieren die Abstraktion mit allen Methoden, die sie anbieten soll. Die Ausführung wird an ein Objekt vom Typ Implementor delegiert. Es ist erlaubt, dass die Methoden der Abstraktion und des Implementors unterschiedliche Bezeichner haben. Schauen Sie in das Beispielprojekt TaschenrechnerBridge.

```
public class Taschenrechner {  
    private Implementor implementor;  
  
    public Taschenrechner(Implementor implementor) {  
        this.implementor = implementor;  
    }  
  
    public void setImplementor(Implementor implementor) {  
        this.implementor = implementor;  
    }  
  
    public double addieren(double summand_1, double summand_2) {  
        return implementor.plus(summand_1, summand_2);  
    }  
  
    public double subtrahieren(double minuend, double subtrahend) {  
        return implementor.minus(minuend, subtrahend);  
    }  
  
    public double multiplizieren(double faktor_1, double faktor_2) {  
        return implementor.mal(faktor_1, faktor_2);  
    }  
  
    public double dividieren(double dividend, double divisor) {  
        return implementor.durch(dividend, divisor);  
    }  
}
```

Der Implementor kann entweder ein Interface oder eine (abstrakte) Klasse sein. Im aktuellen Projekt habe ich den Implementor mit den Rechenverfahren ausgestattet, die ich in der Schule gelernt habe.

```
public class Implementor {  
    double plus(double summand_1, double summand_2) {  
        return summand_1 + summand_2;  
    }
```

```

        double minus(double minuend, double subtrahend) {
            return minuend - subtrahend;
        }

        double mal(double faktor_1, double faktor_2) {
            return faktor_1 * faktor_2;
        }

        double durch(double dividend, double divisor) {
            return dividend / divisor;
        }
    }
}

```

Der Client erzeugt eine Instanz der Implementierung und parametrisiert die Abstraktion damit:

```

Implementor implementor = new Implementor();
Taschenrechner taschenrechner = new Taschenrechner(implementor);
System.out.println("30 * 12 = " + taschenrechner.multiplizieren(30, 12));

```

Das Klassendiagramm sieht bei diesem Entwicklungsfortschritt wie folgt aus (Abb. 25.3).

Der Implementor und die Abstraktion sind über eine Aggregation miteinander verbunden – dadurch wird die Brücke zwischen beiden geschlagen.

25.2.2 Zweiter Schritt

Bis hierhin sieht das alles noch harmlos aus. Aber lassen Sie sich davon überzeugen, dass Sie bei dieser Lösung ein unglaublich mächtiges Werkzeug in der Hand halten.

25.2.2.1 Die Abstraktion erweitern

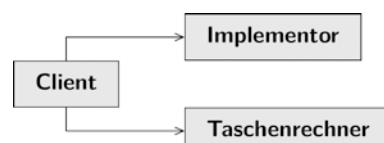
Lassen Sie uns zunächst die Abstraktion erweitern. Wie im ersten Beispiel möchte der Kunde die Möglichkeit haben, eine Zahl quadrieren zu können. Da das Quadrat einer Zahl die Zahl mal sich selbst ist, bleibt die Implementierung davon gänzlich unberührt.

```

public class TaschenrechnerDeluxe extends Taschenrechner {
    public TaschenrechnerDeluxe(Implementor implementor) {

```

Abb. 25.3 Klassendiagramm des Projekts TaschenrechnerBridge



```

        super(implementor);
    }

    public double quadrieren(double zahl) {
        return multiplizieren(zahl, zahl);
    }
}

```

Weitere Abstraktionen wären denkbar. Wenn Sie die Wurzel einer Zahl berechnen möchten, können Sie auf die bereits definierte Implementierung zurückgreifen. Die Quadratwurzel aus y ist die Multiplikation von y mit dem Kehrwert der Quadratwurzel von y. Um den Kehrwert der Quadratwurzel einer Zahl zu berechnen, gibt es ein iteratives Verfahren, das auf Newton zurückgeht. Ausgehend von einem angenäherten Wert wird der jeweilige Folgewert berechnet – der Folgewert wird allein durch Addition, Multiplikation, Differenz und Division berechnet.

```

public class TaschenrechnerNewton extends TaschenrechnerDeluxe {
    public TaschenrechnerNewton(Implementor implementor) {
        super(implementor);
    }

    public double radizieren(double zahl) {
        // iteratives Schema nach Newton
    }
}

```

Der Client kann sich jetzt die Abstraktion aussuchen, die er haben möchte, und sie mit dem Implementor parametrisieren. Wichtig ist, dass sich nur die Abstraktion ändert, bzw. erweitert wird.

25.2.2.2 Die Implementierung erweitern

Unabhängig von der Abstraktion kann die Vererbungshierarchie nach der Klasse Implementor erweitert werden. Der Implementor selbst bietet nur die einfache Implementierung der Grundrechenarten, die dann im Taschenrechner genutzt werden. Wie im vorigen Projekt habe ich die Klassen FFT und RBM definiert, die den Multiplikationsalgorithmus überschreiben – also keine grundsätzlich neuen Funktionalitäten hinzufügen. Die jeweiligen Verfahren habe ich nicht implementiert. Das dürfen Sie aber gerne selbst versuchen.

```

public class RBM extends Implementor {
    @Override
    double mal(double faktor_1, double faktor_2) {
        System.out.println("\tRussische Bauernmultiplikation");
        return faktor_1 * faktor_2;
    }
}

```

Der Client kann sich nun nach Belieben seine Abstraktion und eine passende Implementierung aussuchen.

```
implementor = new RBM();
taschenrechner = new Taschenrechner(implementor);
System.out.println("30 * 12 = " + taschenrechner.multiplizieren(30, 12));
```

Zur Laufzeit kann der Implementor erweitert werden.

```
taschenrechner.setImplementor(new FFT());
System.out.println("30 * 12 = " + taschenrechner.multiplizieren(30, 12));
```

Wenn Sie den Code ausführen, wird auf der Konsole ausgegeben:

```
Russische Bauernmultiplikation
30 * 12 = 360.0
FFT-Multiplikation
30 * 12 = 360.0
```

Wichtig ist – das soll das Klassendiagramm in Abb. 25.4 noch einmal verdeutlichen – dass die Implementierung sich nun unabhängig von der Abstraktion fortentwickeln kann.

Einige Aufgaben lassen sich aber mit den bereits definierten Grundrechenarten nicht mehr bewältigen. Wenn der Kunde beispielsweise einen Zufallsgenerator benötigt, müssen Sie sowohl die Schnittstelle der Abstraktion als auch den Implementor erweitern.

25.3 Diskussion des Bridge Patterns

In diesem Abschnitt werde ich abschließend zeigen, wo Sie die Bridge in der Klassenbibliothek finden. Außerdem werde ich die Bridge von anderen Patterns abgrenzen.

25.3.1 Die Bridge in freier Wildbahn

Wenn Sie sich mit AWT und Peer-Klassen beschäftigen, graben Sie tief im Urschleim der Evolution von Java. Eine AWT-Komponente wird nicht von Java selbst gezeichnet, sondern vom Betriebssystem. Der Entwickler hat beispielsweise eine Klasse `Button`, die von `Component` erbt. `Component` ist die Spitze der Vererbungshierarchie der Abstraktion. Auf der anderen Seite der Brücke steht das Interface `ComponentPeer`, von der die Implementierung, beispielsweise die Klasse `ButtonPeer`, abgeleitet wird. Auf diese Implementierung haben Sie als Programmierer (zumindest theoretisch) keinen Zugriff.

Das Bridge Pattern finden Sie auch in einem ganz anderen Zusammenhang wieder – bei der Datenbankprogrammierung. Schauen Sie sich an, wie Sie mit JDBC arbeiten. Sie la-

den mit `Class.forName(<Treibername>)` einen Datenbanktreiber. Dann lassen Sie sich die Verbindung zur Datenbank geben:

```
Connection connection =
    DriverManager.getConnection(<url>, <user>, <password>)
```

Von dieser Verbindung holen Sie ein Objekt vom Typ `Statement` ab:

```
Statement state = connection.createStatement();
```

Auf diesem `Statement`-Objekt setzen Sie Ihren SQL-Befehl ab:

```
ResultSet result = state.executeQuery( ... Befehl ... );
```

Sie iterieren über das zurückgegebene `ResultSet` und erhalten so die gewünschten Daten. Wenn Sie die Abstraktion – Ihre Applikation – mit einem bestimmten Treiber initialisieren, können Sie wahlweise auf eine Access-Datenbank oder eine Oracle-Datenbank oder jede beliebige andere Datenbank zugreifen. Sie arbeiten immer mit den gegebenen Komponenten `Connection`, `Statement` und `ResultSet`. Die Frage, wie Ihre Abfrage verarbeitet wird, ist für Sie nicht relevant, sprich: Auf die im Hintergrund laufende Implementierung haben Sie keinen Zugriff.

Abstraktion und Implementierung entwickeln sich unabhängig voneinander. Beide können sich unabhängig voneinander weiterentwickeln und wiederverwendet werden.

25.3.2 Abgrenzung zu anderen Patterns

Fassade, Adapter, Proxy, Decorator und Bridge sind sich sehr ähnlich. Sie nutzen Komposition, um ein anderes Objekt zu wrappen. Ein Methodenaufruf wird an das gewrapped Objekt weiter delegiert.

Die Aufgabe des Adapters ist es, zwei verschiedenartige Schnittstellen so in Beziehung zu setzen, dass sie zusammenarbeiten können. Da mehrere Objekte adaptiert werden können, wird dadurch fehlende Mehrfachvererbung in Java ausgeglichen. Adapter sind typischerweise sehr schlank, weil ihre einzige Aufgabe darin besteht, zwei Systeme in Verbindung zu bringen; eigene Intelligenz könnte beispielsweise vorsehen, dass Daten konvertiert werden.

Decorator bringen eigene Intelligenz mit – sie sind vom gleichen Typ wie das gewrapped Objekt und erweitern dessen Verhalten. Typisches Lehrbuchbeispiel ist der `FileInputStream`, der in einen `BufferedInputStream` gewrapped wird. Beide Klassen sind vom Typ `InputStream`.

Der Proxy ist in erster Linie ein Stellvertreter für ein anderes Objekt. Die Implementierung eines Proxy kann dem Decorator sehr ähnlich sein. Das Ziel eines Proxy kann sein, den Zugriff auf das vertretene Objekt zu kontrollieren.

Die Fassade wird am ehesten mit dem Adapter verwechselt. Ihre Aufgabe ist es, den Zugriff auf ein (Sub-)System zu vereinfachen. Denken Sie an die Buchung einer Urlaubsreise, die aus vielen Einzelschritten besteht.

Eine Implementierung variieren Sie bei vielen Mustern: Das Strategy Pattern lässt einen Algorithmus variieren. Beim State Pattern wird kontextabhängig unterschiedliches Verhalten definiert. Beim Adapter greifen Sie auf eine andere Bibliothek zu. Das Bridge Pattern unterscheidet sich dadurch von diesen Mustern, dass neben der Implementierung auch die Abstraktion weiterentwickelt werden kann. Während Sie jederzeit einen Adapter oder eine Fassade implementieren können, muss die Entscheidung für eine Bridge möglichst frühzeitig getroffen werden.

Die Aufgabe der Bridge ist es schließlich, eine Implementierung – also die Rechenoperationen – von der Abstraktion – dem Taschenrechner, auf dem der Anwender seine Rechnung eintippt – zu trennen. Ziel ist es, Verhalten und Abstraktion unabhängig voneinander weiterentwickeln zu können.

25.4 Bridge – Das UML-Diagramm

Das UML-Diagramm aus dem Beispielprojekt TaschenrechnerBridge finden Sie in Abb. 25.4.

25.5 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Sie modellieren in einer Klasse einen Realweltausschnitt.
- Die Klasse ist eine Abstraktion der Realität.
- Die Abstraktion stellt die Schnittstelle für den Client dar.
- Die Schnittstelle ist der Vertrag zwischen Client und Klasse.
- Der Client darf sich darauf verlassen, dass die angebotenen Methoden das definierte Verhalten ausführen.
- Das Verhalten wird durch die dahinterstehende Implementierung definiert.
- Abstraktionen und Implementierung können sich ändern.
- Ein möglicher Weg, neues Verhalten oder neue Abstraktionen zu definieren, ist Vererbung.
- Wenn Abstraktion und Implementierung sich beide ändern, führt Vererbung in ein rasch unüberschaubares System.
- Das Bridge Pattern trennt die Abstraktion von der Implementierung.

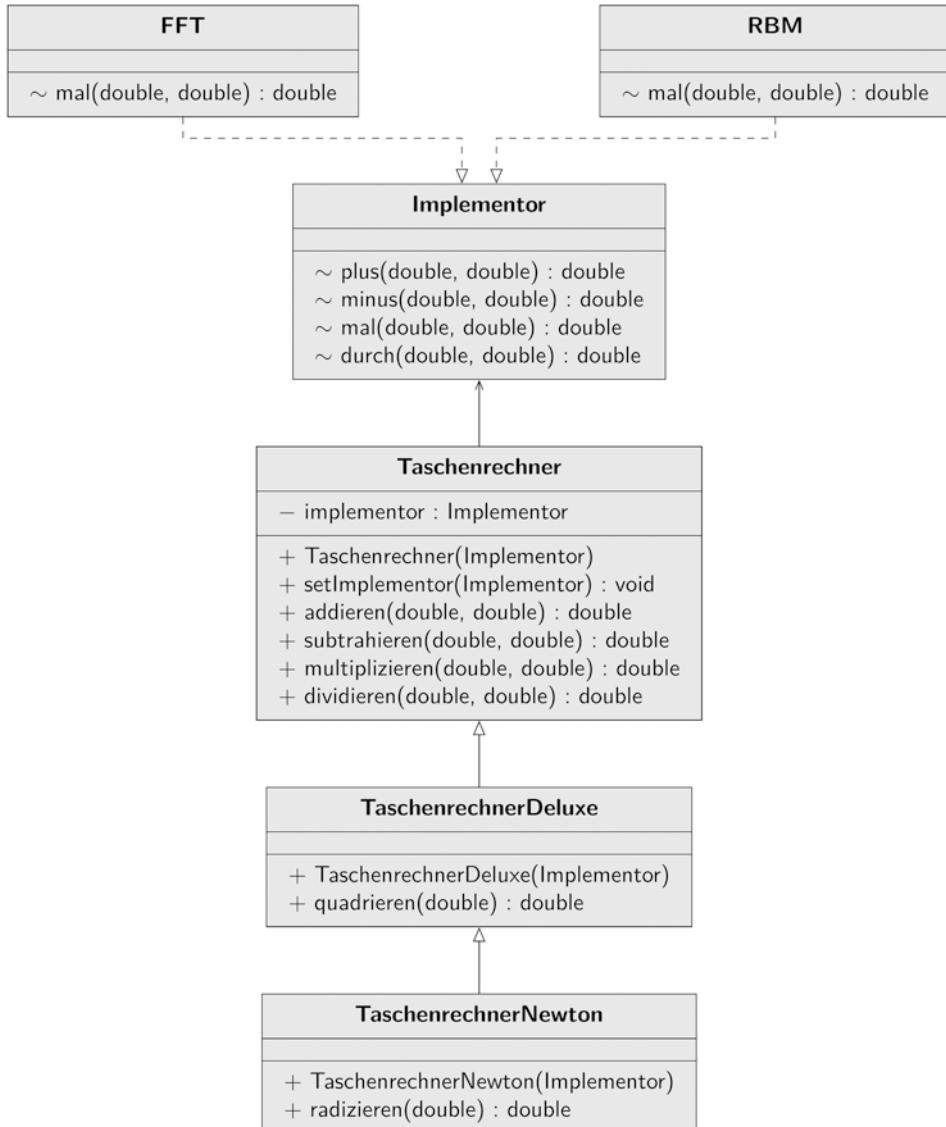


Abb. 25.4 UML-Diagramm des Bridge Pattern (Beispielprojekt TaschenrechnerBridge)

- Die Implementierung wird nicht in der Klasse der Abstraktion definiert, sondern in einer eigenen.
- Abstraktion und Implementierung sind über eine Aggregation miteinander verbunden.
- Abstraktion und Implementierung bilden voneinander unabhängige Vererbungshierarchien.
- Details der Implementierung bleiben dem Client verborgen.

Zweckbeschreibung

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

„Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.“



In den letzten 23 Kapiteln habe ich die einzelnen Muster erläutert. Jetzt befassen wir uns mit einem Beispiel, für das wir mehrere Muster gleichzeitig verwenden wollen. Anhand eines konkreten Beispiels kombinieren wir 5 verschiedene Entwurfsmuster in einer Anwendung. Das geschieht unter anderem mittels Kombination mehrerer Interfaces oder Komposition von Klassen. Worauf dabei zu achten sein wird, soll Thema dieses Kapitels sein. Und ein paar Ansätze für eigene Erweiterungen und Verbesserungen gebe ich Ihnen auch noch an die Hand.

26.1 Das Beispiel

Zunächst eine Warnung vorab: Ich hatte im einleitenden Kapitel darauf hingewiesen, dass ich es für kritisch halte, wenn Programmierer zu exzessiv Gebrauch von Patterns machen, s. Abschn. 1.1.2.4. Dieses Beispiel soll nicht motivieren, Patterns „um jeden Preis“ zu implementieren. Es soll lediglich zeigen, wie Patterns kombiniert werden können, und was Sie dabei beachten sollten.

Wir wollen am Beispiel einer Alarmanlage eines rudimentären „Smart Homes“ mehrere Design Pattern kombinieren. Die Grundanforderungen an unser System sind:

Es soll sowohl Rauchmelder als auch Bewegungssensoren geben, die aber von verschiedenen Herstellern stammen können. Für den Test sollen alle Sensoren an eine „Zentrale“ angebunden sein, von der aus sie auch ausgelöst werden können. Bei einer Auslösung versenden die Sensoren SMS-Nachrichten mit ggf. unterschiedlichem Inhalt. Bei

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, das berechtigten Benutzern zur Verfügung steht. https://doi.org/10.1007/978-3-658-35492-3_26

Raucherkennung soll auch eine Sprinkler-Anlage aktiviert werden, bei Bewegungserkennung soll eine rotierende Warnleuchte eingeschaltet werden. Sowohl Sprinkler-Anlage als auch Warnleuchte sollen per Fernbedienung aber auch wieder ausgeschaltet werden können. Und alle Ereignisse werden natürlich an einer zentralen Stelle protokolliert.

In den folgenden Unterkapiteln schauen wir uns diese Dinge mal im Einzelnen an. Sie finden das zugehörige Beispielprojekt unter dem Namen SmartHome.

Ich habe die Anregung zu diesem Beispiel im Internet gefunden (<https://ruysal.com/post/2019-01-23-Combining-Multiple-Design-Patterns-in-Java/>) und freundlicherweise die Genehmigung des Autors Ramazan Uysal zur Verwendung seiner Idee bekommen. Den Code habe ich überarbeitet und für dieses Buch angepasst.

Schon vorab sei verraten, dass Sie unter Abb. 26.1 ein Klassendiagramm finden, an dem Sie sich beim Durcharbeiten dieses Kapitels orientieren können.

26.2 Singleton: Protokollierung

Das „einfachste“ Muster für unser Beispiel ist das Singleton, das wir einsetzen, um einen zentralen Protokoll-Dienst bereit zu stellen.

Jedes nutzende Objekt holt sich zunächst die Instanz und setzt dann dort seine Nachricht ab. Die Log-Ausgabe könnte dann beliebig kompliziert ausfallen, wir geben hier aber nur ergänzten Text auf der Konsole aus.

```
public class LogAusgabe {  
    private static final LogAusgabe INSTANZ = new LogAusgabe();  
  
    private LogAusgabe() {  
        System.out.println("Protokoll-Dienst bereit");  
    }  
  
    public static LogAusgabe holeInstanz() {  
        return INSTANZ;  
    }  
  
    public void zeigeNachricht(String s) {  
        System.out.println("logFile: " + s + " erkannt");  
    }  
}
```

Sie sehen, dass im (privaten) Konstruktur bereits eine Ausgabe erfolgt, dass der Protokoll-Dienst jetzt zur Verfügung steht. Die Methode `holeInstanz` stellt diesen Dienst zentral bereit, und der Dienst selbst besteht nur aus der Methode `zeigeNachricht`, die einen eingerahmten Text ausgibt.

26.3 Abstrakte Fabrik: Bau der Sensoren

Zwei verschiedene Arten von Sensoren, die von verschiedenen Produzenten kommen können. Das klingt doch verdächtig nach einer abstrakten Fabrik, die wir schon in Kap. 15 besprochen haben. Und der Ansatz passt auch hier ziemlich gut: Als erstens definieren wir das Interface einer Sensorenfabrik mit den Methoden, einen Bewegungssensor oder einen Rauchmelder zu bauen.

```
public interface SensorenFabrik {  
    Bewegungssensor baueBewegungssensor();  
    Rauchmelder baueRauchsensor();  
}
```

Auch für die Sensoren brauchen wir Interfaces. Die Version für den Bewegungssensor drücke ich hier ab, beim Rauchmelder sieht das aber sehr ähnlich aus.

```
public interface Bewegungssensor {  
    String holeBeschreibung();  
  
    void setzeLog(LogAusgabe l);  
    void setzeWarnleuchte(Warnleuchte w);  
}
```

Eine Klasse für einen Sensor sieht dann – hier am Beispiel des Bewegungssensors vom Hersteller A – in etwa so aus:

```
public class BewegungssensorA implements Bewegungssensor, Sensor-  
Listener {  
    static final String BESCHREIBUNG = "... Bewegungssensor ... A";  
  
    LogAusgabe logFile;  
    Warnleuchte warnleuchte;  
  
    @Override  
    public String holeBeschreibung() {  
        return BESCHREIBUNG;  
    }  
  
    @Override  
    public void setzeLog(LogAusgabe l) {  
        this.logFile = l;  
    }  
  
    @Override
```

```

public void setzeWarnleuchte(Warnleuchte w) {
    this.warnleuchte = w;
}

@Override
public void detected() {
    // ... gekürzt
}
}

```

Für den Bewegungssensor des Herstellers B und die Rauchmelder beider Hersteller finden Sie jeweils sehr ähnlichen Code. Der sollte soweit keine Überraschungen bergen. Die Methoden `setzeLog` und `setzeWarnleuchte` ermöglichen den „Einbau“ eines Sensors in eine konkrete Anlage. Da diese Informationen bei der Herstellung nicht vorliegen, müssen sie über nachträgliche Setter-Methoden zugänglich sein und vom Kunden natürlich aufgerufen werden. Zusätzlich zum bereits gezeigten Interface `Bewegungssensor` implementieren wir hier auch ein weiteres Interface `SensorListener`. Zu dessen den Details kommen wir im nächsten Abschnitt über die Auslösung, aber letztlich findet hier die Integration der abstrakten Fabrik mit einem Observer-Muster statt, über den wir die Sensoren in diesem Testfall auslösen können.

Lassen Sie uns zunächst weiter bei der abstrakten Fabrik bleiben. Es fehlt jetzt nur noch der vergleichsweise einfache Code für einen Hersteller. Schauen wir uns diesmal Hersteller B an:

```

public class HerstellerB implements SensorenFabrik {
    @Override
    public Bewegungssensor baueBewegungssensor() {
        return new BewegungssensorB();
    }

    @Override
    public Rauchmelder baueRauchsensor() {
        return new RauchmelderB();
    }
}

```

Ein Klassendiagramm erspare ich mir an dieser Stelle. Wenn Sie sich im Beispielprojekt das package `smarthome`.`AbstrakteFabrik` anschauen finden Sie dort 3 Interfaces und 6 Klassen. Das Diagramm sieht also praktisch genauso aus, wie in Abschn. 15.4. Sie müssen lediglich die Namen austauschen.

Damit haben wir unser erstes Design Pattern implementiert und den Ansatz für die Verbindung zu einem zweiten Muster geschaffen, das wir uns im nächsten Abschnitt anschauen.

26.4 Observer: Die Auslösung

In unserem Beispiel wollen wir alle angeschlossenen Sensoren mit einem einzigen Befehl auslösen können. Das ist natürlich nur für Testzwecke sinnvoll. Im wahren Leben würde jeder Sensor natürlich auf ein entsprechendes Ereignis innerhalb seiner Reichweite reagieren. Hier verwenden wir aber für den Test das Observer-Muster aus Kap. 5. Wir registrieren also jeden verbauten Sensor bei einer zentralen Instanz, die wiederum eine einheitliche Nachricht an jeden Sensor verschickt, die dieser natürlich verarbeiten können muss.

Also brauchen wir zunächst das recht einfache Interface des SensorListener:

```
public interface SensorListener {  
    void detected();  
}
```

Sie sehen, dass hier lediglich die Auslöse-Methode spezifiziert wird, die von außen angesprochen werden soll.

Der Client, den wir später sehen werden, muss aber beim Einbau eines Sensors ins Haus diesen auch bei der zentralen Stelle, dem Sensor-System, registrieren:

```
public class SensorSystem {  
    private final List<SensorListener> listeners = new ArrayList<>();  
  
    public void register(SensorListener sensorListener) {  
        listeners.add(sensorListener);  
    }  
  
    public void ausloesen() {  
        listeners.forEach(SensorListener::detected);  
    }  
}
```

In der ArrayList werden alle Sensoren mittels der register-Methode aufgenommen und dann in der ausloesen-Methode angesprochen. Dort verwende ich diesmal die sogenannte Methoden-Referenz, um für jeden Sensor in der Liste dessen detected-Methode aufzurufen. Diese Art der Schreibweise ist seit Java 8 möglich, und zusammen mit der forEach-Methode auf der ArrayList ergibt deren Verwendung eine sehr kurze aber gut verständliche Form.

Wenn Sie sich das Klassenmodell aus dem Observer-Kapitel anschauen, finden Sie jetzt also das Interface (statt WohnungsObserver jetzt SensorListener) und die „Börse“ (statt WohnungsBoerse jetzt SensorSystem). Die Entsprechung zur Klasse Arbeitnehmer sind die Sensoren, die über das allen gemeinsame SensorListener-Interface registriert werden. Sie sind die Empfänger eines „Rundrufs“ aus dem Client. Eine Entsprechung zur

Klasse Wohnung brauchen wir hier in diesem Falle nicht, weil wir keine zusätzlichen Informationen über konkrete Objekte in einer weiteren Liste austauschen.

Sie werden übrigens feststellen, dass ich ein „deregister“ eines Sensors nicht implementiert habe. Ein einmal verbauter und registrierter Sensor würde also nicht mehr abgebaut werden können. Das sollte aber kein Problem für Sie darstellen, wenn Sie dieses Beispiel jetzt weiter ausbauen wollen.

Um die Methode `detected` werden wir uns noch etwas später im Detail kümmern, denn zu deren Verständnis fehlen noch ein paar Bausteine, die wir uns zunächst anschauen müssen.

26.5 Adapter: Benachrichtigungen verschicken

Es bleibt immer noch die Frage, was ein Sensor eigentlich macht, wenn er ausgelöst wird. Gemäß den Anforderungen soll er ja Nachrichten verschicken. Wenn man aber verschiedene Hersteller und verschiedene Sensoren im Spiel hat, muss man sich auf irgendein Format für Nachrichten einigen. Es geht aber auch etwas anders.

Nehmen wir an, die Hersteller hätten sich auf ein „Mindest“-Format für SMS-Nachrichten von Bewegungssensoren und eines von Rauchmeldern geeinigt. Dazu gäbe es zunächst einmal diese entsprechenden Klassen. Hier einmal die SMS eines Rauchmelders. Die des Bewegungssensors ist ähnlich einfach.

```
public class RauchSms {
    public void sendMessage(String s) {
        System.out.println("SMS vom Rauchmelder. ...: " + s);
    }
}
```

Jetzt könnte ein Sensor bei Auslösung direkt ein entsprechendes Objekt erzeugen und dessen `sendMessage`-Methode aufrufen. Allerdings wären die Kunden dann auf die Leistung dieser Methode vollständig angewiesen und hätten keine Möglichkeit, eigenständige Ergänzungen einzubringen.

Das umgehen die Hersteller mit einem Adapter – den wir aus Kap. 22 bereits kennen –, dessen Code sie dem Kunden zugänglich machen. An diesem Punkt kann der Kunde beliebige Anpassungen vornehmen (z. B. auch die originale `sendMessage`-Methode aufrufen). Zunächst einmal basiert alles auf einem einfachen SMS-Interface:

```
public interface Sms {
    void sendMessage(String s);
}
```

Das müssen jetzt die Adapter implementieren. Für den Rauchmelder schauen wir uns den einmal näher an:

```
public class RauchmelderSmsAdapter implements Sms {  
    private final RauchSms vomSensor;  
  
    public RauchmelderSmsAdapter(RauchSms rauchsms) {  
        this.vomSensor = rauchsms;  
    }  
  
    @Override  
    public void sendMessage(String s) {  
        vomSensor.sendMessage(s);  
        var log = LogAusgabe.getInstance();  
        log.zeigeNachricht("SMS-Versand des Rauchmelder-Adapters");  
    }  
}
```

Sie sehen hier, dass die Methode `sendMessage` zunächst die „mitgelieferte“ Methode aufruft. Vor dem Aufruf wären jetzt beliebige Anpassungen des Textes oder technisch für den SMS-Versand vielleicht andere notwendige Maßnahmen möglich. Auch weitere Benachrichtigungs-Systeme ließen sich hier anschließen, die den Hinweistext aus dem Sensor an andere Empfänger weiterreichen können. Anschließend wird dann noch ein Log-Eintrag generiert, der anzeigt, dass über den Adapter etwas passiert ist. Für den Bewegungsmelder sieht der Adapter im Prinzip gleich aus, ich habe aufgrund der größeren Flexibilität aber eine separate Version (anstatt eines einzigen „NachrichtenAdapters“) geschaffen. Auf die Meldung eines Bewegungssensors kann also anders reagiert werden, als auf eine Auslösung eines Rauchmelders.

Das Adapter-Muster nutzen wir hier also ein wenig anders als im Kapitel über den Adapter erläutert. Hier stellt der Hersteller sowohl das Interface `Sms` als auch eine Grundversion des Adapters zur Verfügung, ermöglicht aber dadurch dem Anwender des Sensors, ohne Eingriffe in den Code des Sensors in die Benachrichtigungsfunktion „einzugreifen“.

Die genaue Einbindung dieses Adapters aus einem Sensor heraus zeige ich Ihnen, wenn wir alle Komponenten zusammen haben. Eine fehlt noch.

26.6 Command: Aktoren und Fernbedienung

Ein Sensor, der Meldungen weitergeben kann, ist in einem Alarmsystem schon mal die berühmte halbe Miete, aber jetzt fehlen noch Geräte, um direkt auf den jeweiligen Alarm zu reagieren. Für Rauchmelder bietet sich eine Sprinkler-Anlage an, die jegliche aufkeimende Flammen löschen soll. Für Bewegungsmelder nutzen wir in unserem Beispiel Warnleuchten, die den „Eindringling“ verscheuchen oder „Wachen“ aufmerksam machen soll.

Diese Geräte müssen in unserem Beispiel einmal von den jeweiligen Sensoren eingeschaltet werden, aber von einer Fernbedienung auch wieder ausgeschaltet werden können. Dafür verwenden wir das Command-Pattern aus Kap. 9.

Auch hier sind die beiden Varianten für Rauchmelder und Bewegungssensoren nahezu identisch. Ich drucke also hier eine Variante ab, die andere finden Sie natürlich auch im Beispielcode.

Schauen wir uns also beispielhaft die Warnleuchte an:

```
public class Warnleuchte {
    public void einschalten() {
        System.out.println("Warnleuchte ist an");
    }

    public void ausschalten() {
        System.out.println("Warnleuchte ist aus");
    }
}
```

Das Befehls-Interface sieht jetzt bei uns so aus:

```
public interface Befehl {
    public void ausfuehren();
}
```

Und ein gekapseltes Kommando für die Warnleuchte hat dann dieses Aussehen:

```
public class WarnleuchteAusschalten implements Befehl {
    Warnleuchte warnleuchte;

    public WarnleuchteAusschalten(Warnleuchte warnleuchte) {
        super();
        this.warnleuchte = warnleuchte;
    }

    @Override
    public void ausfuehren() {
        var logFile = LogAusgabe.holeInstanz();
        warnleuchte.ausschalten();
        logFile.zeigeNachricht("Warnleuchte ausgeschaltet");
    }
}
```

Den Code für das Einschalten drucke ich hier nicht extra ab.

Diese Befehle lassen sich jetzt auch von einer Fernbedienung aus ansteuern:

```
public class SmartHomeFernbedienung {  
    Befehl kommando;  
  
    public void setzeBefehl(Befehl kommando) {  
        this.kommando = kommando;  
    }  
  
    public void knopfgedrueckt() {  
        kommando.ausfuehren();  
    }  
}
```

Das ist eine sehr einfache Version, die Sie aber natürlich beliebig ausweiten können. In der vorgestellten Fassung kann zu einem Zeitpunkt ein Befehl mit dem Knopf verbunden werden. Wird der Knopf gedrückt, wird der Befehl ausgeführt.

26.7 Ein Sensor löst aus

Jetzt haben wir alle Teile zusammen, um uns die Methode `detected` eines Sensors in Gänze anzuschauen. Hier ein Beispiel:

```
public class BewegungssensorA implements Bewegungssensor, Sensor-  
Listener {  
    // ... gekürzt  
  
    @Override  
    public void detected() {  
        var s = """  
            Bewegungssensor - Hersteller A  
            Bewegung erkannt  
        """;  
  
        if (logFile == null)  
            s = s.concat("Kein Protokoll-Dienst konfiguriert!\n");  
        else  
            logFile.zeigeNachricht("BEWEGUNG!");  
  
        if (warnleuchte == null)  
            s = s.concat("Kein Alarmmelder konfiguriert!\n");  
        else {  
            var einschalten = new WarnleuchteEinschalten(warnleuchte);  
            einschalten.ausfuehren();  
        }  
    }  
}
```

```

        Sms sms = new BewegungssensorSmsAdapter(new BewegungsSms());
        System.out.println("Bewegungssensor - Hersteller A");
        sms.sendMessage(s);
    }
}

```

Ich habe die Methode des Bewegungssensors von Hersteller A etwas umfangreicher gestaltet als die anderen. Hier bauen wir einen Text auf der abhängig von der Konfiguration eines Protokoll-Dienstes und einer Warnleuchte entsprechende Hinweise enthält oder aber eben die jeweiligen Aktionen durchführt. Am Schluss wird über den Adapter die Nachricht mit dem finalen Text verschickt. Hier verbinden wir also die Produkte der abstrakten Fabrik mit dem Adapter und auch mit dem Command Pattern.

Beachten Sie bitte zu Beginn der Methode den mehrzeiligen String. Diese Möglichkeit der Textdarstellung namens Text Block ist mit Java 15 (JEP 378) produktiv verfügbar geworden. Als Preview ist sie auch in Java 13 (JEP 355) und Java 14 (JEP 369) bereits verfügbar.

Im Wesentlichen können wir jetzt mit drei Anführungszeichen beginnend und endend mehrzeilige Texte auch ohne Steuerzeichen für den Zeilenumbruch im Quellcode nutzen. Nach wie vor handelt es sich aber um den Datentyp `String`. Wichtig ist vor allem die Einrückungstiefe der beginnenden Anführungszeichen, denn links davon stehende Whitespaces (Leerzeichen, Tabulator) werden nicht zum Bestandteil des erfassten Textes. Noch genauer: Alle Whitespaces in allen Zeilen eines Textblocks, die links vom am weitesten linksstehenden Zeichen irgendeiner Zeile stehen, werden nicht berücksichtigt.

Im Beispiel oben heißt das konkret, dass die Zeile „Bewegungssensor – Hersteller A“ linksbündig ausgegeben wird, obwohl sie im Quelltext 16 Zeichen eingerückt ist. Die darunter stehende Zeile „Bewegung erkannt“ wird aber dann um 4 Zeichen eingerückt ausgegeben.

Die Behandlung von Whitespaces in Textblöcken nimmt auch einen großen Teil in den oben genannten JEPs ein. Bitte schauen Sie sich diese unbedingt an, wenn Sie mit Textblöcken arbeiten wollen.

NetBeans markiert bei Textblöcken die tatsächlich zum `String` gehörenden Zeichen mit einem farbigen Hintergrund, so dass sie sehr einfach erkennen können, welche Leerzeichen oder Tabulatoren dazu gehören und welche nicht. Wenn Sie versuchsweise die Zeile mit „Bewegungssensor ...“ ein oder zwei Zeichen nach links schieben, indem Sie davorstehende Leerzeichen löschen, werden Sie an der geänderten Hintergrundmarkierung erkennen, was das für die anderen Zeilen zur Folge hat.

Die `detected`-Methoden der anderen Sensoren verwenden in diesem Beispiel keine Textblöcke und sind auch einfacher gestaltet. Daher zeige ich sie an dieser Stelle nicht.

26.8 Klassendiagramm

Jetzt haben wir insgesamt 5 Pattern „zusammengesteckt“. Das schauen wir uns mal im Klassendiagramm an, das Sie in Abb. 26.1 finden.

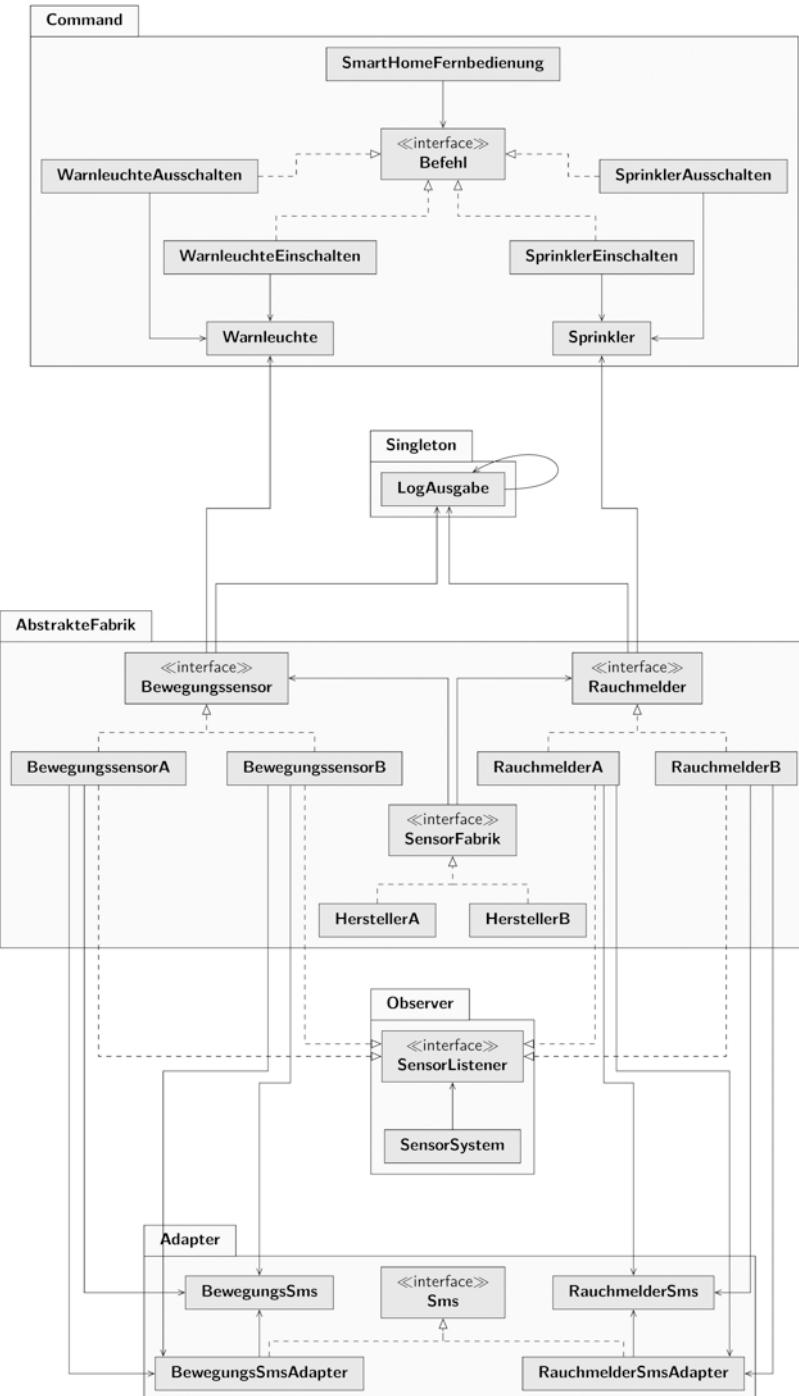


Abb. 26.1 Klassendiagramm des Beispielprojekts SmartHome

Wenn Sie dieses Diagramm mit den UML-Diagrammen der einzelnen Patterns aus den vorherigen Kapiteln vergleichen, sollten Sie die einzelnen Teile wiedererkennen, aber auch die neuen Verbindungen identifizieren können.

26.9 Der Client

Jetzt fehlt nur noch der Client, der die Sensoren der verschiedenen Hersteller in ein System zusammenbaut, testweise alle auslöst und dann die Sprinkler-Anlage und die Warnleuchte auch wieder ausschaltet.

Die main-Methode und der Konstruktor der Klasse SmartHome sind zunächst einfach:

```
public final class SmartHome {
    // ... gekürzt

    SmartHome() {
        init();
        test();
    }

    public static void main(String[] args) {
        var smartHome = new SmartHome();
    }
}
```

Die init-Methode verbindet jetzt vier Produkte der abstrakten Fabrik – je ein Bewegungssensor und ein Rauchmelder jeweils eines Herstellers – mit dem Singleton für die Log-Ausgabe, der Warnleuchte bzw. der Sprinkler-Anlage, die später die Kommandos zum Einschalten erhalten werden und der Zentrale, bei der die Sensoren als Observer fungieren. Zum Schluss wird noch eine Fernbedienung erzeugt, die für das Ausschalten der Anlagen sorgen kann.

```
public final class SmartHome {
    // ... gekürzt

    void init() {
        log = LogAusgabe.holeInstanz();

        sprinkler = new Sprinkler();
        warnleuchte = new Warnleuchte();

        SensorenFabrik fabrik;
```

```
fabrik = new HerstellerA();
rauchsensor1 = fabrik.baueRauchsensor();
bewegungssensor1 = fabrik.baueBewegungssensor();

fabrik = new HerstellerB();
rauchsensor2 = fabrik.baueRauchsensor();
bewegungssensor2 = fabrik.baueBewegungssensor();

rauchsensor1.setzeLog(log);
rauchsensor2.setzeLog(log);
bewegungssensor1.setzeLog(log);
bewegungssensor2.setzeLog(log);

rauchsensor1.setzeSprinkler(sprinkler);
rauchsensor2.setzeSprinkler(sprinkler);

bewegungssensor1.setzeWarnleuchte(warnleuchte);
bewegungssensor2.setzeWarnleuchte(warnleuchte);

sensorAnlage = new SensorSystem();
sensorAnlage.register((SensorListener) rauchsensor1);
sensorAnlage.register((SensorListener) rauchsensor2);
sensorAnlage.register((SensorListener) bewegungssensor1);
sensorAnlage.register((SensorListener) bewegungssensor2);

fernbedienung = new SmartHomeFernbedienung();
}

// .. gekürzt
}
```

Letztlich sind das alles nachvollziehbare Schritte, die beim Zusammenbauen einer Alarmanlage auch tatsächlich anstehen.

Und in der `test`-Methode probieren wir dann alles aus. Zunächst bekommen alle Sensoren den Befehl, ihren Alarm auszulösen. Im Anschluss werden über die Fernbedienung erst die Warnleuchte und dann die Sprinkler-Anlage wieder ausgeschaltet.

```
public final class SmartHome {
    // ... gekürzt

    void test() {
        System.out.println("Starte Tests:");

        sensorAnlage.ausloesen();
```

```
fernbedienung.setzeBefehl(new WarnleuchteAusschalten(warnleuchte));
fernbedienung.knopfgedrueckt();

fernbedienung.setzeBefehl(new SprinklerAusschalten(sprinkler));
fernbedienung.knopfgedrueckt();
}

// ... gekürzt
}
```

Wenn Sie das Programm aufrufen, sollten Sie folgende Ausgabe erhalten:

```
Protokoll-Dienst bereit
Starte Tests:
Rauchmelder - Hersteller A
SMS vom Rauchmelder. Die Nachricht lautet: RAUCH festgestellt!
logFile: SMS-Versand des Rauchmelder-Adapters erkannt
logFile: RAUCH! erkannt
Sprinkler ist an
logFile: Sprinkler-Anlage eingeschaltet erkannt
Rauchmelder - Hersteller B
SMS vom Rauchmelder. Die Nachricht lautet: RAUCH festgestellt!
logFile: SMS-Versand des Rauchmelder-Adapters erkannt
logFile: RAUCH! erkannt
Sprinkler ist an
logFile: Sprinkler-Anlage eingeschaltet erkannt
Bewegungssensor - Hersteller A
SMS vom Bewegungssensor. Die Nachricht lautet:
Bewegungssensor - Hersteller A
    Bewegung erkannt
Kein Protokoll-Dienst konfiguriert!
Kein Alarmmelder konfiguriert!

logFile: SMS-Versand des Bewegungssensor-Adapters erkannt
Bewegungssensor - Hersteller B
    SMS vom Bewegungssensor. Die Nachricht lautet: BEWEGUNG fest-
gestellt!
logFile: SMS-Versand des Bewegungssensor-Adapters erkannt
logFile: BEWEGUNG! erkannt
Warnleuchte ist an
logFile: Warnleuchte eingeschaltet erkannt
Warnleuchte ist aus
logFile: Warnleuchte ausgeschaltet erkannt
Sprinkler ist aus
logFile: Sprinkler-Anlage ausgeschaltet erkannt
```

Die erste Zeile stammt vom Singleton und wird bei dessen Initialisierung ausgegeben. Danach meldet sich die test-Methode und im Anschluss alle Sensoren nacheinander, erst die Rauchmelder, dann die Bewegungssensoren. Beim Bewegungssensor des Herstellers A sehen Sie in der Beispieldausgabe auch zwei Fehlerhinweise. Für diesen Durchlauf habe ich in der init-Methode die Zuordnungen der LogAusgabe und der Warnleuchte zum bewegungssensor1 auskommentiert, um dessen Fehlertoleranz zu demonstrieren. Wenn Sie das bei den anderen Sensoren versuchen, werden in der vorliegenden Version NullPointerExceptions geworfen.

26.10 Überlegungen

Das Beispiel soll Ihnen eine Methode zeigen, verschiedene Pattern sinnvoll zu kombinieren. Es gibt zwei recht offensichtliche Ansätze, dieses Beispiel auszubauen und zu verändern: Zunächst einmal sollten die `detect`-Methoden alle erweitert werden, um auf fehlende LogAusgaben oder Aktoren auch zu reagieren. Alternativ können Sie aber auch die `detect`-Methoden sehr rudimentär gestalten und sogar den Protokoll-Eintrag und das Auslösen der Warnleuchte bzw. der Sprinkler-Anlage in den jeweiligen Adapter verlegen. Experimentieren Sie gerne damit ein wenig rum.

Oder Sie gehen den anderen Weg, und entfernen mal eines der Patterns (gemäß meiner einleitenden Warnung in diesem Kapitel). Brauchen Sie den Adapter wirklich, oder geht es nicht auch ohne? Welche Alternativen hätte man dann? Nicht alles, was technisch möglich ist, muss an dieser Stelle auch unbedingt sinnvoll sein. Überlegen Sie für Ihre jeweiligen Aufgaben immer genau, was notwendig, was sinnvoll, und was vielleicht sogar störend sein kann.

Die Verbindungen der Patterns untereinander sollten Sie im Quellcode finden können:

- Das Singleton wird an verschiedenen Stellen genutzt, indem dort jeweils die Instanz angefragt und dann dessen `zeigeNachricht`-Methode aufgerufen wird.
- In den Sensoren, die in der abstrakten Fabrik erzeugt werden, wird zusätzlich das Interface `SensorListener` implementiert, das jeden Sensor als Observer ausweist.
- Ebenfalls in den Sensoren finden Sie in den `detected`-Methoden den Aufruf des jeweiligen Adapters.
- Und in den `detected`-Methoden sowie in der Fernbedienung befindet sich auch die Verwendung des Command Pattern.
- Die init-Methode des Clients enthält die Registrierung der Observer sowie die Erzeugung der Aktoren Warnleuchte und Sprinkler, die als Befehlsempfänger fungieren.
- In der test-Methode des Clients werden das Observer Muster und das Command Muster dann aktiv verwendet.

Versuchen Sie bitte selbst, dazu einmal ein detailliertes UML-Diagramm zu erstellen.

26.11 Zusammenfassung

Gehen Sie das Kapitel noch mal stichwortartig durch:

- Ein Singleton lässt sich vergleichsweise einfach in Programme einbinden, die Verbindung anderer Muster untereinander erfordert etwas mehr Aufwand.
- Die Implementierung mehrerer Interfaces kann eine Klasse zum Bestandteil mehrerer Muster machen.
- Die Komposition von Musterbestandteilen in Klassen anderer Muster ermöglicht ebenfalls die Verbindung mehrerer Muster.
- Je mehr Muster in einem Programm verbunden werden, desto komplexer kann die Struktur werden.
- Vorsicht ist geboten, um den Überblick und die jeweils gebotenen Funktionalitäten nicht zu verlieren. Die Strukturierung in packages kann dabei unterstützen.

Schlussbemerkungen

Ich hoffe, Ihnen hat das Lesen des Buchs Spaß gemacht, und Sie haben dabei auch vielleicht etwas gelernt. Wie bereits am Anfang erwähnt, sollten Sie es als Nachschlagewerk verwenden, wenn Sie sich nicht mehr ganz genau an einzelne Fragestellungen erinnern. Jetzt wäre auch ein geeigneter Zeitpunkt, noch mal in das Kapitel zu den Entwurfsprinzipien (Kap. 2) zu schauen und diese dann in den erwähnten Mustern noch einmal zu entdecken.

Sollten Sie aus Ihrer Sicht Ungereimtheiten oder Fehler entdeckt haben, wäre ich Ihnen für eine entsprechende Rückmeldung sehr dankbar. Sie erreichen mich dazu am besten unter designpatternsjava16@gmail.com.

Den Quellcode dieses Buchs dürfen Sie für Ihre privaten Zwecke frei verwenden und manipulieren – dafür stelle ich Ihnen den ja auch zur Verfügung.

Vielen Dank, dass Sie bis hierhin durchgehalten haben.

Olaf Musch

Stichwortverzeichnis

A

AbstractListModel_Beispiel 38
AbstractTableModel_Demo 56
Abstrakter_Garten 192
Action 109
Adapter_Klassenbasiert 286
Adapter_Objektbasiert 288
Adressbuch 219
Autofabrik 225
Autofabrik_CopyConstructor 230
Autofabrik_Superklasse 231
AutoKatalog 315

B

Builder_Pattern 246

C

CoR_1 62
CoR_2 65
CoR_3 67
CoR_4 68

F

Fassade 276

G

Geburten 161
Geburten_Flyweight 163
GraphEditor_1 234
GraphEditor_2 236
GraphEditor_3 238

H

Haus_1 199
Haus_2 205
Haus_3 207
Haus_4 209
Haushalt_1 144
Haushalt_2 147
Haushalt_3 149
Haushalt_4 152
Haushalt_5 155
Haushalt_6 156
Haushalt_7 158
HaushaltsBuilder 249
HaushaltsBuilder_Ext 254

I

Interpreter 171
Interpreter_1 180
Interpreter_2 182
Interpreter_3 182
Interpreter_4 184
Iterator 216

J

JavaBeans_Pattern 245
JTable_Demo 53

L

LayoutStrategy 128
ListenerDemo 50
ListModel_Beispiel 38

M

Mahlzeit 214
Mahlzeit_Variation 215
Memento 271
Memento_Simple 269
MVC_1 51
MVC_2 58

N

NaiverAnsatz 119

O

Observer_01 42
Observer_02 44
Observer_03 45
Observer_04 46
Observer_05 46
Observer_06 47
Observer_07 48
Observer_08 49

P

Pizza 164
PizzaFlyweight 166
POP3 95
Proxy_1 296
Proxy_2 298
Proxy_3 299
ProxyDynamisch 302
ProxyDynamischTable 304

R

RadioCommand 110
Reise_1 98
Reise_2 102
Reise_3 103
RMI_Test 306

S

Sammlungen_1 132
Sammlungen_2 136
Sammlungen_3 140
Schrebergarten 196
Singleton_1 25
Singleton_2 26
Singleton_3 27
Singleton_4 28
Singleton_5 29
SmartHome 341
Sortieren 121
StatePattern_1 89
StatePattern_2 93
StatePattern_3 93
Streams_1 320
Streams_2 321
Swing 112
SwingBeispiel 77

T

Taschenrechner 326
TaschenrechnerBridge 330
Telescoping_Constructor_Pattern 244
Template_1 33
Template_2 34
Template_3 36
Test_Sammlungen 229
Tormanager 84

U

Unmodifiable 294

V

Visitor 259

W

WeinSim 73