O'REILLY®

Übersetzung der 2. englischen Auflage

Entwurfs-muster

von Kopf bis Fuß

Mit Design Patterns flexible objekt-orientierte Software erstellen

Eric Freeman & Elisabeth Robson

mit Kathy Sierra & Bert Bates

Übersetzung von Jørgen W. Lang

Ein gehirnfreundliches Buch

Entwurfsmuster von Kopf bis Fuß

Eric Freeman
Elisabeth Robson

mit
Kathy Sierra
Bert Bates

Deutsche Übersetzung von
Jørgen W. Lang

O'REILLY®

Eric Freeman, Elisabeth Robson, Kathy Sierra und Bert Bates

Lektorat: Alexandra Follenius
Übersetzung: Jørgen W. Lang
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: Ulrich Borstelmann, *www.borstelmann.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Ellie Volckhausen, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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

ISBN:
Print 978-3-96009-162-2
PDF 978-3-96010-503-9
ePub 978-3-96010-504-6
mobi 978-3-96010-505-3

3. Auflage 2022, Übersetzung der 2. englischen Auflage

Wieblinger Weg 17
69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Bei der Erstellung dieses Buchs wurden keinerlei Enten oder Truthähne verletzt.

Die Mitglieder der »Gang of Four« haben uns die Erlaubnis erteilt, ihre Fotos in diesem Buch abzudrucken. Doch, sie sehen wirklich so gut aus.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt.
Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC/04-31-0810 www.pefc.de

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Für die Gang of Four, deren Fachwissen und deren Erfahrung beim Beschreiben und Erklären von Entwurfsmustern die Welt des Softwaredesigns für immer verändert und das Leben von Entwicklern weltweit verbessert haben.

Aber mal im Ernst – *wann* kommt die zweite Auflage eures Buchs? Schließlich ist es erst ~~*zehn*~~ *Jahre* her!

fünfundzwanzig

Die Autor:innen von >>Entwurfsmuster von Kopf bis Fuß<<

Eric Freeman

Elisabeth Robson

Eric wurde von seiner Mitstreiterin bei der Entwicklung der Von-Kopf-bis-Fuß-Reihe, Kathy Sierra, bezeichnet als »eines dieser seltenen Individuen, die fließend die Sprache, Praxis und Kultur verschiedener Lebenswelten wie die der Hipster-Hacker, Unternehmensdirektoren, Ingenieure und Think-Tanks beherrschen«.

Von Haus aus ist Eric Informatiker, seinen Doktortitel erlangte er an der Yale University. Beruflich war er als CTO von Disney Online & Disney.com bei der Walt Disney Company tätig.

Aktuell ist er Kodirektor der Head-First-Reihe, die in Deutschland als Von-Kopf-bis-Fuß-Reihe bekannt ist. Bei der von ihm mitgegründeten Online-Lernplattform WickedlySmart widmet er seine Zeit der Erstellung von Print- und Video-Inhalten, die über die führenden Bildungskanäle vertrieben werden.

Erics Bücher in der Von-Kopf-bis-Fuß-Reihe Bücher umfassen die Themen Entwurfsmuster, HTML & CSS, JavaScript- und HTML5-Programmierung sowie das Programmierenlernen.

Eric lebt in Austin, Texas.

Elisabeth ist Softwareentwicklerin, Autorin und IT-Trainerin. Seit ihren Studententagen an der Yale University, wo sie ihren Master in Informatik erworben hat, ist sie mit Leib und Seele Informatikerin.

Aktuell ist sie Mitbegründerin von WickedlySmart, wo sie Bücher, Artikel, Videos und mehr erstellt. Zuvor hat sie als Director of Special Projects bei O'Reilly Media Präsenz-Workshops und Onlinekurse zu einer Vielzahl von technischen Themen gestaltet und eine Leidenschaft für das Entwerfen von Lehrsystemen entwickelt, die Menschen helfen, Technologien zu verstehen.

Wenn sie nicht vor ihrem Computer sitzt, geht Elisabeth, stets die Kamera in Griffweite, in die freie Natur, um zu wandern oder Rad oder Kajak zu fahren, oder sie widmet sich der Gartenarbeit.

Die Köpfe hinter dieser Reihe
(und Mitverschwörer:innen bei diesem Buch)

Kathy Sierra

Bert Bates

Kathy interessiert sich für Lerntheorie seit ihrer Zeit als Spieleentwicklerin für Virgin, MGM und Amblin' und als Dozentin für New Media Authoring an der UCLA. Sie war Master-Java-Trainerin für Sun Microsystems, und sie gründete JavaRanch.com (jetzt CodeRanch.com), das in den Jahren 2003 und 2004 den Jolt Cola Productivity Award gewann.

Im Jahr 2015 erhielt sie den Electronic Frontier Foundation's Pioneer Award für ihre Arbeit zur Schaffung kompetenter Nutzer und den Aufbau nachhaltiger Gemeinschaften.

In jüngster Zeit konzentriert sich Kathy auf modernste Bewegungswissenschaften und das Coaching zum Erwerb von Fähigkeiten, bekannt als Ecological Dynamics oder »Eco-D«. Ihre Arbeit, bei der sie Eco-D für das Training von Pferden einsetzt, führt zu einem weitaus humaneren Ansatz in der Reitkunst, was die einen erfreut (und die anderen, traurigerweise, verwirrt). Die Pferde, die das Glück haben, dass ihre Besitzer Kathys Ansatz anwenden, sind zufriedener, autonomer, gesünder und sportlicher als ihre traditionell trainierten Artgenossen.

Sie können Kathy Sierra auf Instagram folgen: *@pantherflows*.

Bevor **Bert** Autor wurde, war er Entwickler, spezialisiert auf KI der alten Schule (hauptsächlich Expertensysteme), Echtzeit-Betriebssysteme und komplexe Planungssysteme.

Im Jahr 2003 schrieben Bert und Kathy Head First Java und starteten die Head-First-Reihe. Seitdem hat er weitere Java-Bücher geschrieben und Sun Microsystems und Oracle bei vielen ihrer Java-Zertifizierungen beraten. Außerdem hat er Hunderte von Autor:innen und Lektor:innen darin geschult, Bücher zu schreiben, die gute Lerninhalte bieten.

Bert ist Go-Spieler und hat 2016 mit Entsetzen und Faszination zugesehen, wie AlphaGo Lee Sedol besiegt hat. In letzter Zeit nutzt er Eco-D (Ecological Dynamics), um sein Golfspiel zu verbessern und seinen Papagei Bokeh zu trainieren.

Bert und Kathy haben das Privileg, Beth und Eric seit 16 Jahren zu kennen, und die Head-First-Reihe hat das große Glück, sie zu den wichtigsten Mitwirkenden zu zählen.

Sie können Bert unter *CodeRanch.com* eine Nachricht schicken.

Über den Übersetzer dieses Buchs

Jørgen W. Lang lebt und arbeitet als freier Autor (»CSS Kochbuch«) und Übersetzer in Oldenburg/Niedersachsen. Mitte der Neunzigerjahre des vergangenen Jahrhunderts begann er, sich mit dem damals noch jungen World Wide Web und seinen Möglichkeiten zu beschäftigen. Pünktlich zum Jahrtausendwendejahr erschien seine erste Übersetzung für den O'Reilly Verlag. Mittlerweile ist der Umfang auf mehr als 10.000 Seiten angewachsen.

Mit großer Energie und Ausdauer bringt Jørgen seit fast schon zwei Jahrzehnten Webseiten bei, das zu tun, was von ihnen erwartet wird – unabhängig davon, auf welchem Gerät sie betrachtet werden (elektrische Zahnbürsten ausgenommen).

Das zweite Standbein von Jørgen Lang ist die Musik. Außerhalb der Welt der semantischen Elemente, Selektoren und Objekte hat er sich einen Namen als hervorragender Gitarrist, Sänger, Komponist und Arrangeur gemacht und kann auf eine Vielzahl veröffentlichter Alben und mehrere Hundert Konzerte in aller Welt (z. B. für die UNESCO in Seoul) zurückblicken.

Zur deutschen Übersetzung

Eine Herausforderung bei der Übersetzung von »Head First Design Patterns« war, möglichst alle verwendeten Bilder und Metaphern ins Deutsche zu übertragen, bei den Fachausdrücken aber auf in der Praxis ungebräuchliche Übersetzungen zu verzichten. So werden wir im Buch zwar Fabriken bauen, aber unser Entwurfsmuster heißt Factory, wir werden Beobachter entsenden, das Entwurfsmuster aber Observer nennen. Wir hoffen, dass unsere deutschen Leserinnen und Leser die Entwurfsmuster so leichter vor Augen haben, aber dennoch wissen, wie der Profi diese Muster in der Praxis bezeichnet.

Der Inhalt (im Überblick)

	Einführung	xxiii
1	Willkommen bei den Entwurfsmustern: *Einführung in Entwurfsmuster*	1
2	Ihre Objekte auf dem Laufenden halten: *Das Observer-Muster*	37
3	Objekte dekorieren: *Das Decorator-Muster*	79
4	In der OO-Bäckerei …: *Das Factory-Muster*	109
5	Einmalige Objekte: *Das Singleton-Muster*	169
6	Aufrufe verkapseln: *Das Command-Muster*	191
7	Anpassungsfähigkeit beweisen: *Die Adapter- und Facade-Muster*	237
8	Algorithmen verkapseln: *Das Template Method-Muster*	277
9	Erfolgreiche Collections: *Die Iterator- und Composite-Muster*	317
10	Der (Zu-)Stand der Dinge: *Das State-Muster*	381
11	Objektzugriff kontrollieren: *Das Proxy-Muster*	425
12	Muster von Mustern: *Zusammengesetzte Muster*	493
13	Muster in der wahren Welt: *Schöner leben mit Mustern*	563
14	Anhang: *Übrig gebliebene Muster*	597

Inhalt (jetzt ausführlich)

Einführung

Ihr mustergültiges Gehirn. Sie versuchen, etwas zu lernen, und Ihr Hirn tut sein Bestes, damit das Gelernte nicht hängen bleibt. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z. B. für das Wissen, welche Tiere einem gefährlich werden könnten, oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, wie schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, etwas über Entwurfsmuster zu wissen?

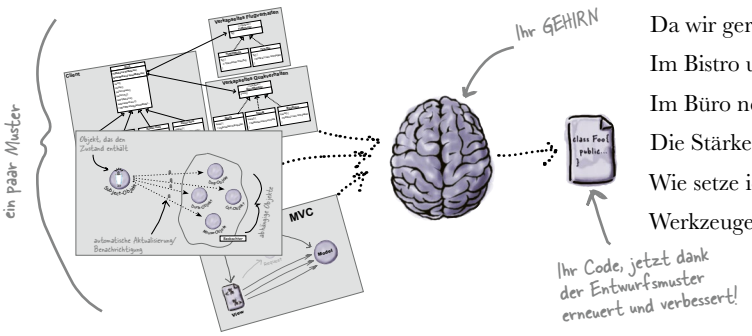
Für wen ist dieses Buch?	xxiv
Wir wissen, was Sie gerade denken	xxv
Und wir wissen, was Ihr Gehirn gerade denkt	xxv
Metakognition: Nachdenken übers Denken	xxvii
Machen Sie sich Ihr Hirn untertan	xxix
Fachgutachter	xxxii
Danksagungen	xxxiv

Einführung in Entwurfsmuster

1 Willkommen bei den Entwurfsmustern

Irgendjemand hat Ihr Problem schon gelöst. In diesem Kapitel lernen Sie, warum (und wie) Sie die Weisheit und die Lehren anderer Entwickler nutzen können, die die gleichen Designprobleme bereits hatten und die Reise überlebt haben. Bevor dieses Kapitel zu Ende ist, kümmern wir uns um die Verwendung und die Vorteile der Entwurfsmuster, sehen uns ein paar grundsätzliche objektorientierte (OO-)Designprinzipien an und gehen mit Ihnen zusammen ein Beispiel für die Funktionsweise von Entwurfsmustern durch. Die beste Möglichkeit, die Muster zu verwenden, ist, sie *in Ihr Gehirn zu laden* und dann die Stellen in Ihren Designs und bestehenden Programmen zu *erkennen*, an denen der Einsatz sinnvoll ist. Im Gegensatz zur Codewiederverwendung können Sie mit Entwurfsmustern die *Erfahrung* anderer Menschen wiederverwenden.

Es begann mit einer einfachen SimUDuck-App	2
Aber jetzt sollen die Enten FLIEGEN können	3
Aber irgendetwas ging furchtbar schief ...	4
Joe denkt über Vererbung nach …	5
Wie wäre es mit einem Interface?	6
Was würden Sie an Joes Stelle tun?	7
Die einzige Konstante in der Softwareentwicklung	8
Das Problem eingrenzen	9
Veränderliches und Unveränderliches voneinander trennen	10
Das Entenverhalten entwerfen	11
Das Entenverhalten implementieren	13
Das Entenverhalten integrieren	15
Den Entencode testen	18
Verhalten dynamisch festlegen	20
Das große Ganze: Verkapseltes Verhalten	22
HAT-EIN ist besser als IST-EIN	23
Da wir gerade von Entwurfsmustern sprechen ...	24
Im Bistro um die Ecke aufgeschnappt ...	26
Im Büro nebenan aufgeschnappt …	27
Die Stärke eines gemeinsamen Mustervokabulars	28
Wie setze ich Entwurfsmuster ein?	29
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	32

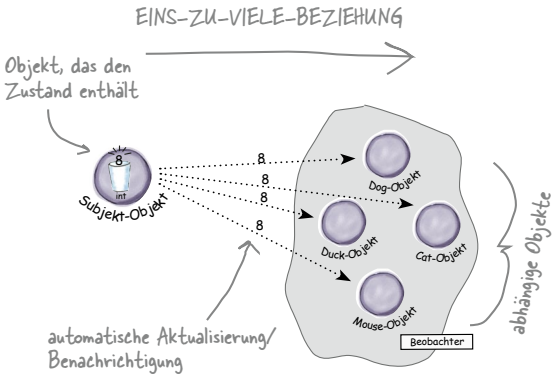


Das Observer-Muster

2 Ihre Objekte auf dem Laufenden halten

Sie wollen interessante Ereignisse doch nicht verpassen, oder? Es gibt ein Muster, das unsere Objekte auf dem Laufenden hält, wenn etwas für sie Wichtiges passiert, und zwar eins der am häufigsten verwendeten und nützlichsten Entwurfsmuster überhaupt: das Observer-Muster. In diesem Kapitel sehen wir uns alle möglichen interessanten Eigenschaften dieses Musters an, wie *Eins-zu-viele-Beziehungen* und *lose Kopplungen*. Mit dem Observer-Muster im Gepäck sind Sie der Star jeder Muster-Party!

Die Wetterstation im Überblick	39
Willkommen zum Observer-Muster	44
Herausgeber + Abonnenten = Observer-Muster	45
Die Observer-Muster-Definition	51
Die Macht der losen Kopplung	54
Die Wetterstation entwerfen	57
Die Wetterstation implementieren	58
Die Wetterstation hochfahren	61
Das Observer-Muster in freier Wildbahn	65
Die lebensverändernde Applikation programmieren	66
Inzwischen bei Weather-O-Rama	69
Probefahrt für den neuen Code	71
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	72
Die Entwurfsprinzipien-Challenge	73



Das Decorator-Muster

3 Objekte dekorieren

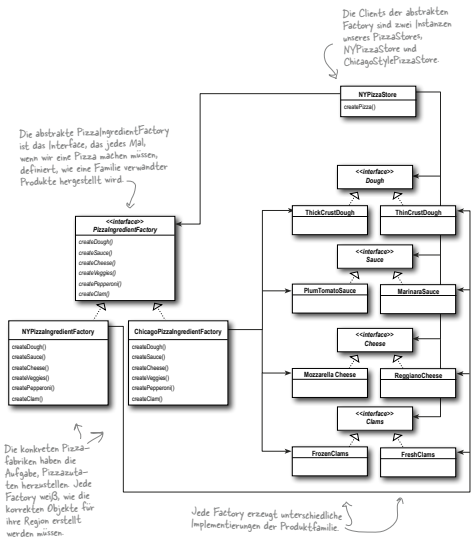
Nennen wir dieses Kapitel einfach »Vererbst du noch, oder designst du schon?«. Hier werfen wir einen weiteren Blick auf das typische Überstrapazieren von Vererbung. Sie werden lernen, wie Sie Ihre Klassen zur Laufzeit mit einer Form der Objektkomposition dekorieren können. Warum? Sobald Sie die Dekoration beherrschen, können Sie Ihren Objekten (oder denen anderer Leute) neue Verantwortung geben, *ohne hierfür den Code der zugrunde liegenden Klassen ändern zu müssen*.

Willkommen bei Starbuzz Coffee	80
Das Offen/Geschlossen-Prinzip	86
Wir stellen vor: das Decorator-Muster	88
Eine Getränkebestellung mit Dekoratoren aufbauen	89
Die Definition des Decorator-Musters	91
Unsere Getränke dekorieren	92
Den Starbuzz-Code schreiben	95
Getränke programmieren	96
Zutaten programmieren	97
Den Kaffee servieren	98
Dekoratoren in freier Wildbahn: Java I/O	100
Die java.io-Klassen dekorieren	101
Einen eigenen Java-I/O-Dekorator schreiben	102
Unseren neuen Java-I/O-Dekorator testen	103
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	105

Das Factory-Muster

4 In der OO-Bäckerei ...

Machen Sie sich bereit, ein paar lose gekoppelte OO-Entwürfe zu backen. Zur Erstellung von Objekten gehört mehr, als einfach den **new**-Operator einzusetzen. Sie werden lernen, dass Instanziierung nicht in der Öffentlichkeit durchgeführt werden sollte und oft zu *Kopplungsproblemen* führen kann. Und *das* wollen wir nun wirklich nicht, oder? Finden Sie heraus, wie das Factory-Muster Sie vor peinlichen Abhängigkeiten bewahren können.



Das Veränderliche finden 112
Die Objekterstellung verkapseln 114
Eine einfache Pizzafabrik erstellen 115
Die einfache Fabrik definieren 117
Ein Framework für die Pizzeria 120
Den Subklassen die Entscheidung überlassen 121
Eine Fabrikmethode deklarieren 125
Jetzt ist es endlich Zeit, das Factory Method-Muster kennenzulernen 131
Ein paralleler Blick auf Hersteller und Produkte 132
Die Definition des Factory Method-Musters 134
Ein Blick auf Objektabhängigkeiten 138
Das Prinzip der Umkehrung der Abhängigkeiten 139
Das Prinzip anwenden 140
Zutatenfamilien 145
Die Zutatenfabriken bauen 146
Die Pizzas überarbeiten ... 149
Unsere Pizzerien überarbeiten 152
Was haben wir getan? 153
Die Definition des Abstract Factory-Musters 156
Factory Method und Abstract Factory im Vergleich 160
Werkzeuge für Ihren Entwurfs-Werkzeugkasten 162

Das Singleton-Muster

5 Einmalige Objekte

Unser nächster Halt ist das Singleton-Muster, unsere Fahrkarte für die Erstellung einmaliger Objekte, von denen es immer nur eine Instanz gibt. Es wird Sie freuen, zu erfahren, dass das Singleton-Muster, bezogen auf sein Klassendiagramm, das einfachste Muster von allen ist. Tatsächlich enthält es nur eine einzige Klasse! Aber machen Sie es sich nicht zu bequem. Trotz des einfachen Klassendesigns müssen wir für seine Implementierung einige tiefgehende objektorientierte Überlegungen anstellen. Also, setzen Sie Ihre Denkmütze auf, und los geht's.

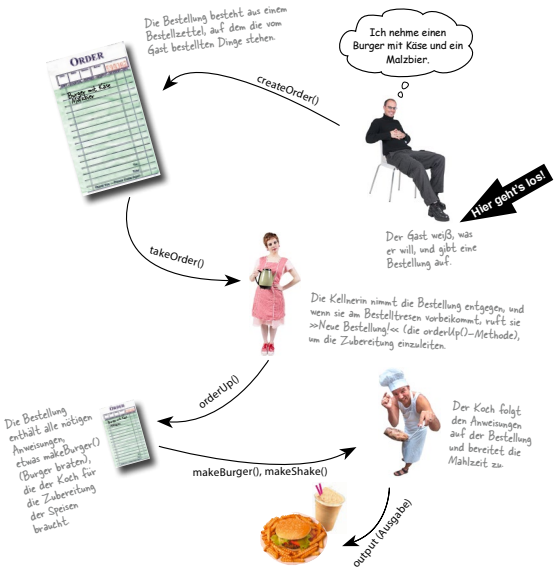
Die Implementierung des klassischen Singleton-Musters im Detail 173
Die Schokoladenfabrik 175
Definition des Singleton-Musters 177
Brüssel ~~Houston~~, wir haben ein Problem ... 178
Mit Multithreading umgehen 180
Können wir das Multithreading verbessern? 181
Inzwischen in der Schokoladenfabrik ... 183
Werkzeuge für Ihren Entwurfs-Werkzeugkasten 186

Das Command-Muster

6 Aufrufe verkapseln

In diesem Kapitel heben wir die Verkapselung auf ein ganz neues Niveau: Wir werden Methodenaufrufe verkapseln.

Ja, richtig gehört: Methodenaufrufe. Damit können wir Teile von Berechnungen »einfrieren«, wodurch sich das aufrufende Objekt nicht um die Details der Berechnung kümmern muss. Es nutzt einfach die eingefrorene Methode für die Erfüllung seiner Aufgabe. Mit diesen verkapselten Methodenaufrufen sind aber noch ganz andere schlaue Dinge möglich. Wir können sie beispielsweise zur Protokollierung nutzen oder sie wiederverwenden, um eine »Rückgängig«-Funktionalität zu implementieren.

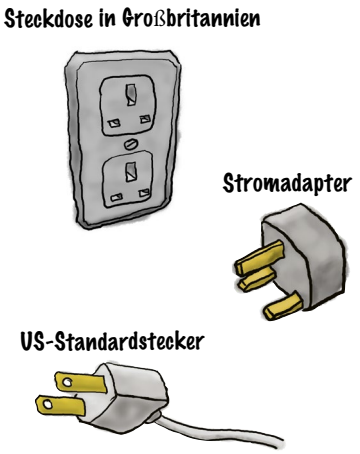


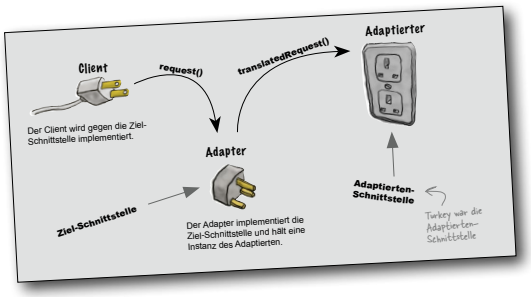
Ein Blick auf die Drittherstellerklassen	194
Inzwischen im Bistro ...	197
Vom Bistro zum Command-Muster	201
Unser erstes Command-Objekt	203
Das Command-Objekt verwenden	204
Befehle den Plätzen zuweisen	209
Die Fernsteuerung implementieren	210
Die Befehle implementieren	211
Die Fernsteuerung auf Herz und Nieren testen	212
Zeit für die Dokumentation ...	215
Was machen wir hier?	217
Zeit, den Rückgängig-Knopf auf seine Qualität zu testen!	220
Zustände für die Implementierung der »Rückgängig«-Funktion verwenden	221
Die Deckenventilator-Befehle mit einer »Rückgängig«-Funktion versehen	222
Jede Fernsteuerung braucht einen Partymodus!	225
Einen Makro-Befehl benutzen	226
Viele Verwendungsmöglichkeiten für das Command-Muster: Warteschlangen für Befehle	229
Weitere Anwendungen des Command-Musters: Aufträge protokollieren	230
Das Command-Muster in der wahren Welt	231
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	233

Die Adapter- und Facade-Muster

7 Anpassungsfähigkeit beweisen

In diesem Kapitel versuchen wir so unmögliche Dinge wie die Quadratur des Kreises. Klingt ausgeschlossen? Aber nicht mit Entwurfsmustern. Erinnern Sie sich noch an das Decorator-Muster? Wir haben Objekte verpackt, um sie mit Verantwortlichkeiten zu versehen. Diesmal **verpacken wir Objekte**, damit ihre Schnittstellen wie etwas aussehen, das sie nicht sind. So können wir Designs, die bestimmte Schnittstellen erwarten, an Klassen anpassen, die eine andere Schnittstelle implementieren. Und das ist noch nicht alles. Wenn wir schon dabei sind, sehen wir uns gleich noch ein anderes Muster an, das Objekte verpackt, um ihre Schnittstelle zu vereinfachen.



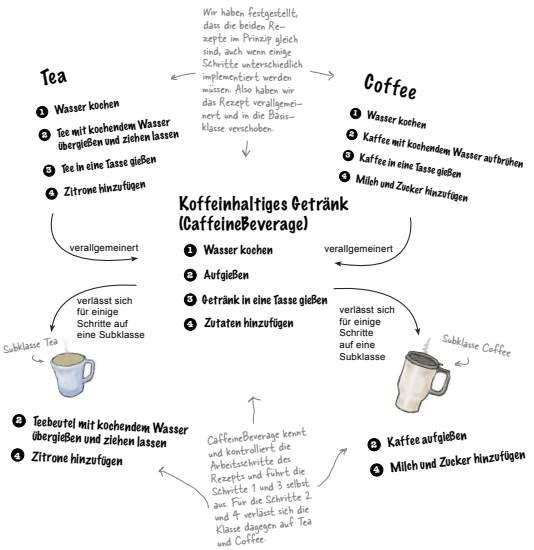


Überall Adapter	238
Objektorientierte Adapter	239
Probelauf für den Adapter	242
Das Adapter-Muster erklärt	243
Die Definition des Adapter-Musters	245
Objekt- und Klassen-Adapter	246
Adapter im echten Leben	250
Einen Enumerator an einen Iterator anpassen	251
Unser eigenes Heimkino	257
Einen Film ansehen (auf die harte Tour)	258
Licht, Kamera, Facade!	260
Die Heimkino-Facade konstruieren	263
Die vereinfachte Schnittstelle implementieren	264
Einen Film anschauen (auf die sanfte Tour)	265
Die Definition des Facade-Musters	266
Das Prinzip der Verschwiegenheit	267
Wie man KEINE Freunde gewinnt und KEINE Objekte beeinflusst	268
Das Facade-Muster und das Prinzip der Verschwiegenheit	271
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	272

Das Template Method-Muster

8 Algorithmen verkapseln

Wir sind auf dem Verkapselungstrip. Was haben wir schon alles verkapselt? Objekterstellung, Methodenaufrufe, komplexe Schnittstellen, Enten, Pizzas – was kommt wohl als Nächstes? In diesem Kapitel gehen wir der Verkapselung von Algorithmenteilen auf den Grund, damit Subklassen sich bei Bedarf jederzeit direkt in eine Berechnung einklinken können. Außerdem lernen wir ein Entwurfsprinzip kennen, das von Hollywood inspiriert ist. Na dann mal los ...

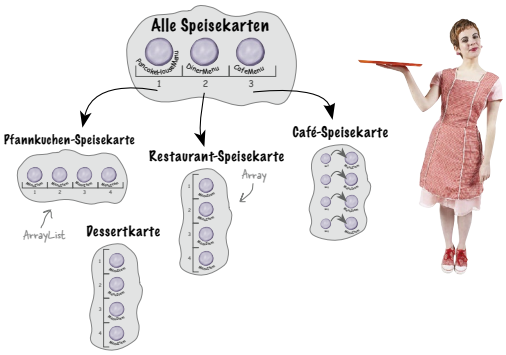


Zeit für mehr Koffein	278
Ein paar Kaffee- und Tee-Klassen zusammenrühren (in Java)	279
Kaffee und Tee abstrahieren	282
Das Design verbessern	283
prepareRecipe() abstrahieren	284
Was haben wir getan?	287
Willkommen beim Template Method-Muster	288
Was hat uns das Template Method-Muster gebracht?	290
Die Definition des Template Method-Musters	291
Eingehängt in die Template Method ...	294
Den Hook verwenden	295
Das Hollywood-Prinzip und das Template Method-Muster	299
Template-Methoden in freier Wildbahn	301
Mit dem Template Method-Muster sortieren	302
Was ist compareTo()?	303
Enten mit Enten vergleichen	304
Ein paar Enten sortieren	305
Das Making-of der Entensortiermaschine	306
Swinging mit Frames	308
Eigene Listen mit AbstractList	309
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	313

Die Iterator- und Composite-Muster

9 Erfolgreiche Collections

Es gibt viele Möglichkeiten, Objekte in einer Collection zu speichern. Zum Beispiel in einem Array, einem Stack, einer Liste oder einer HashMap – Sie haben die Wahl. Dabei hat jede Form ihre Vor- und Nachteile. Irgendwann werden Ihre Clients allerdings über diese Objekte iterieren wollen. Und wollen Sie ihnen dann Ihre Implementierung offenbaren? Hoffentlich nicht! Das wäre einfach nicht professionell. Aber keine Sorge. Ihre Karriere ist nicht gefährdet. In diesem Kapitel werden Sie sehen, wie Ihre Clients über Ihre Objekte iterieren können, ohne zu wissen, wie sie gespeichert sind. Außerdem lernen Sie, wie man Super Collections von Objekten erstellt, die mit einem einzigen Satz einige eindrucksvolle Datenstrukturen überspringen können. Und als wäre das noch nicht genug, werden Sie auch noch das eine oder andere über Objektverantwortlichkeit lernen.



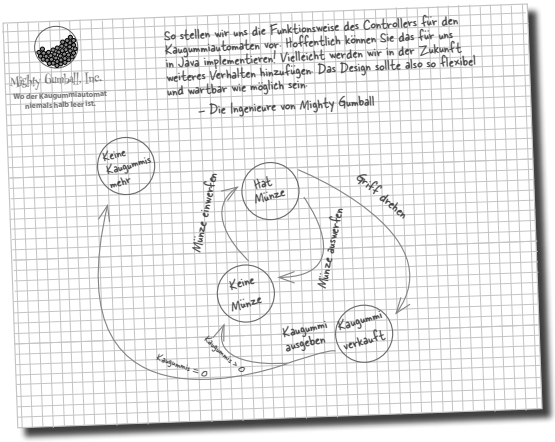
Große Neuigkeiten: Das Restaurant und das Pfannkuchenhaus von Objectville fusionieren	318
Sehen wir uns die Gerichte an	319
Die Spezifikation implementieren: unser erster Versuch	323
Können wir die Iteration verkapseln?	325
Willkommen zum Iterator-Muster	327
DinerMenu mit einem Iterator versehen	328
Die Restaurant-Speisekarte mit einem Iterator überarbeiten	329
Den Kellnerin-Code aufmöbeln	330
Den Code testen	331
Unser aktueller Entwurf auf dem Prüfstand ...	333
Aufräumen mit java.util.Iterator	335
Die Definition des Iterator-Musters	338
Die Struktur des Iterator-Musters	339
Das Prinzip der einzelnen Verantwortlichkeit	340
Willkommen zu Javas Iterable-Interface	343
Javas erweiterte for-Schleife	344
Ein Blick auf die Speisekarte des Cafés	347
Iteratoren und Collections	353
Die Definition des Composite-Musters	360
Speisekarten mit dem Composite-Muster entwerfen	363
MenuComponent implementieren	364
Das Gericht (MenuItem) implementieren	365
Die Komposita-Speisekarte implementieren	366
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	376

Das State-Muster

10 Der (Zu-)Stand der Dinge

Eine kaum bekannte Tatsache: Die Strategy- und das State-Muster sind Zwillinge, die bei der Geburt getrennt wurden.

Vielleicht denken Sie, dass beide ein ähnliches Leben führen. In Wirklichkeit hat Strategy jedoch ein unglaublich erfolgreiches Unternehmen rund um austauschbare Algorithmen aufgebaut, während State den vermutlich edleren Weg gewählt hat. Es hilft anderen Objekten, ihr Verhalten zu kontrollieren, indem es ihren inneren Zustand verändert. So unterschiedlich die Wege der beiden auch scheinen – hinter den Kulissen ist ihr Design fast identisch. Wie das sein kann und worum es beim State-Muster wirklich geht, werden wir herausfinden. Am Ende des Kapitels sehen wir uns dann an, welche Beziehung beide Muster tatsächlich zueinander haben.



Ein echter Java-Plombenzieher	382
Kurze Einführung in Zustandsautomaten	384
Den Code schreiben	386
Interner Testlauf	388
Sie haben es geahnt – ein Änderungswunsch!	390
ZUSTÄNDE wie bei Hempels unterm Sofa ...	392
Der neue Entwurf	394
Das Interface State und die Klassen definieren	395
Umbau des Kaugummiautomaten	398
Ein Blick auf die komplette Klasse GumballMachine ...	399
Weitere Zustände implementieren	400
Die Definition des State-Musters	406
Wir müssen uns wieder dem 1-von-10-Kaugummispiel widmen	409
Das Spiel fertigstellen	410
Demo für den CEO von Mighty Gumball, Inc.	411
Stimmt alles?	413
Das haben wir fast vergessen!	416
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	419

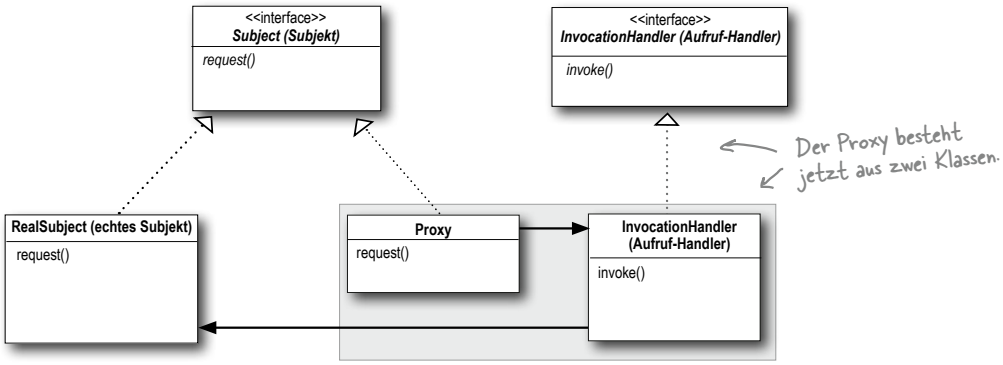
Das Proxy-Muster

11 Objektzugriff kontrollieren

Haben Sie schon einmal »guter Bulle, böser Bulle« gespielt?

Sie sind der gute Polizist und stellen alle Ihre Dienste auf eine nette und freundliche Weise bereit. Wenn Sie aber nicht wollen, dass jeder ungefragt Ihre Dienste nutzt, übernimmt der böse Polizist die Zugangskontrolle für Sie. Denn genau das tun Proxies (»Stellvertreter«): Sie kontrollieren und verwalten den Zugriff. Wie Sie sehen werden, gibt es viele Möglichkeiten, Proxies als Vertreter für andere Objekte zu nutzen. Proxies sind dafür bekannt, dass sie für die von ihnen vertretenen Objekte komplette Methodenaufrufe über das Internet abwickeln. Außerdem nehmen sie bekanntermaßen den Platz einiger ziemlich fauler Objekte ein.

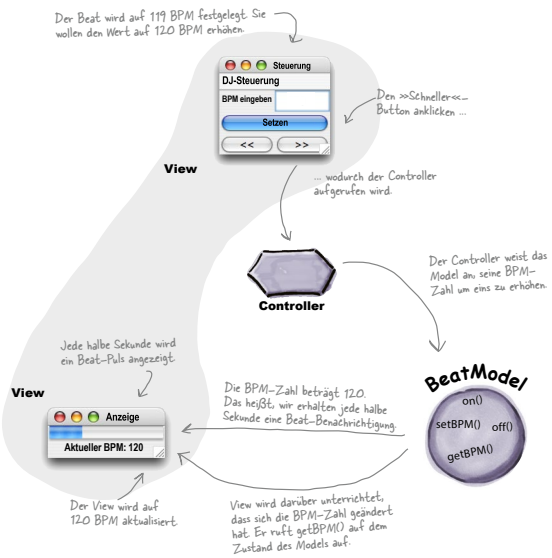
Den Überwachungscode schreiben 427
Den Überwachungscode testen 428
Einführung in entfernte Methodenaufrufe 433
Den Kaugummiautomaten (GumballMachine) als entfernten Dienst einrichten 446
Bei der RMI-Registry anmelden ... 448
Die Definition des Proxy-Musters 455
Bereitmachen für den virtuellen Proxy 457
Den virtuellen Proxy für die Albencover entwerfen 459
Den Bild-Proxy schreiben 460
Partnervermittlung für Geeks in Objectville 470
Die Person implementieren 471
Fünf-Minuten-Drama: Subjekte schützen 473
Das große Ganze: Einen dynamischen Proxy für Person erstellen 474
Der Proxy-Zoo 482
Werkzeuge für Ihren Entwurfs-Werkzeugkasten 485
Der Code für den Albumcover-Viewer 489



Zusammengesetzte Muster

12 Muster von Mustern

Wer hätte gedacht, dass Muster auch zusammenarbeiten können? Sie haben bereits die erbitterten Kamingespräche miterlebt (und dabei haben wir Ihnen die Seiten mit dem »Pattern Death Match« noch gar nicht gezeigt, die wir auf Druck des Verlegers wieder entfernen mussten). Wer hätte gedacht, dass Muster eigentlich sogar recht gut miteinander auskommen können? Ob Sie's glauben oder nicht – einige der mächtigsten OO-Entwürfe verwenden Kombinationen mehrerer Muster. Machen Sie sich bereit für die nächste Stufe Ihrer Entwurfsmuster-Fähigkeiten: zusammengesetzte Muster (»Compound-Muster«)

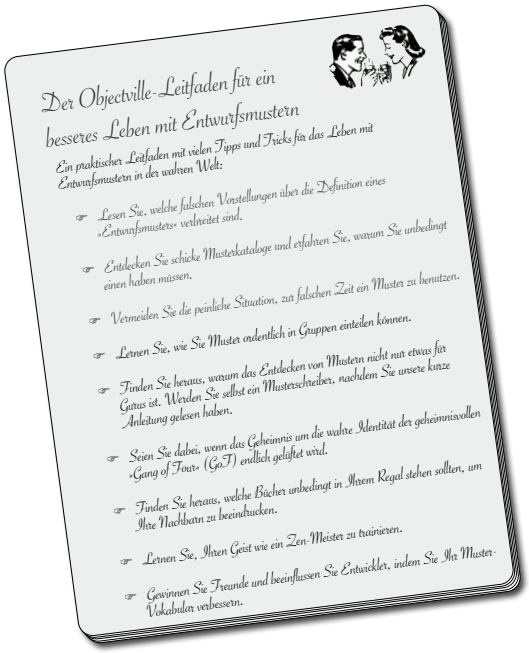


Mustergültige Zusammenarbeit	494
Ein Wiedersehen mit den Enten	495
Was haben wir getan?	517
Ein Blick aus der ~~Vogel~~Entenperspektive: das Klassendiagramm	518
Der König der zusammengesetzten Muster	520
Willkommen zu Model-View-Controller	523
Genauer hingesehen ...	524
MVC als eine Reihe von Mustern verstehen	526
MVC, um den Beat zu steuern	528
Erstellung der Einzelteile	531
Ein Blick auf die konkrete Klasse BeatModel	532
Der View	533
Den View implementieren	534
Und damit zum Controller	536
Die Einzelteile zusammensetzen ...	538
Strategy erforschen	539
Das Model adaptieren	540
Noch ein Probelauf ...	542
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	545

Schöner leben mit Mustern

13 Muster in der wahren Welt

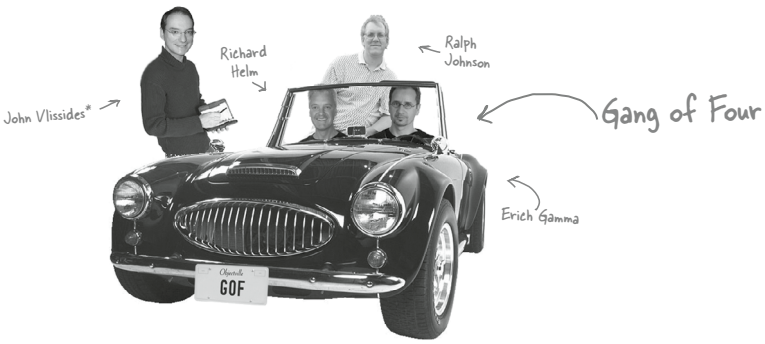
Und damit sind Sie bereit für eine strahlende neue Welt voller Entwurfsmuster. Aber bevor Sie all die neuen Chancen nutzen, müssen wir uns noch um ein paar Details kümmern, die Ihnen draußen in der wahren Welt begegnen können, wo die Dinge etwas komplexer sind als hier in Objectville. Auf den folgenden Seiten haben wir für Sie einen kleinen Reiseführer (oder Leitfaden) vorbereitet, der Sie beim Übergang begleiten wird ...


Der Objectville-Leitfaden für ein besseres Leben mit Entwurfsmustern

Ein praktischer Leitfaden mit vielen Tipps und Tricks für das Leben mit Entwurfsmustern in der wahren Welt:

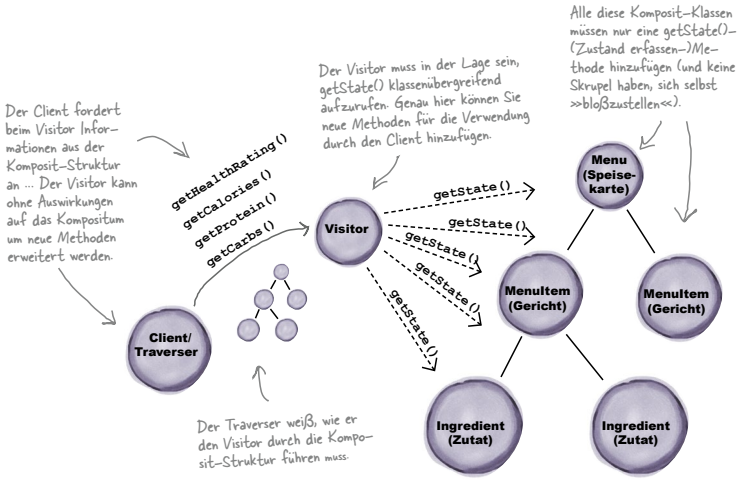
- Lesen Sie, welche falschen Vorstellungen über die Definition eines »Entwurfsmusters« verbreitet sind.
- Entdecken Sie schicke Musterkataloge und erfahren Sie, warum Sie unbedingt einen haben müssen.
- Vermeiden Sie die peinliche Situation, zur falschen Zeit ein Muster zu benutzen.
- Lernen Sie, wie Sie Muster ordentlich in Gruppen einteilen können.
- Finden Sie heraus, warum das Entdecken von Mustern nicht nur etwas für Gurus ist. Werden Sie selbst ein Musterschreiber, nachdem Sie unsere kurze Anleitung gelesen haben.
- Seien Sie dabei, wenn das Geheimnis um die wahre Identität der geheimnisvollen »Gang of Four« (GoF) endlich gelüftet wird.
- Finden Sie heraus, welche Bücher unbedingt in Ihrem Regal stehen sollten, um Ihre Nachbarn zu beeindrucken.
- Lernen Sie, Ihren Geist wie ein Zen-Meister zu trainieren.
- Gewinnen Sie Freunde und beeinflussen Sie Entwickler, indem Sie Ihr Muster-Vokabular verbessern.

Definition von Entwurfsmustern	565
Ein genauerer Blick auf die Entwurfsmusterdefinition	567
Möge die Macht mit Ihnen sein	568
Sie wollen also eigene Entwurfsmuster schreiben	573
Entwurfsmuster ordnen	575
In Mustern denken	580
Denk-Muster	583
Vergessen Sie nicht die Macht des gemeinsamen Vokabulars	585
Spritztour durch Objectville mit der Gang of Four	587
Ihre Reise hat gerade erst begonnen!	588
Der Muster-Zoo	590
Mit Anti-Mustern das Böse auslöschen	592
Werkzeuge für Ihren Entwurfs-Werkzeugkasten	594
Abschied von Objectville ...	595



14 Anhang: Übrig gebliebene Muster

Nicht jeder kann der Beliebteste sein. Seit der Erstveröffentlichung von *Design Patterns: Elements of Reusable Object-Oriented Software* hat sich viel verändert. Entwickler haben diese Muster tausendfach verwendet. Die in diesem Anhang vorgestellten Muster sind Vollmitglieder der offiziellen GoF-Musterfamilie, werden aber nicht so oft genutzt wie die bisher gezeigten. Trotzdem sind sie auf ihre eigene Art großartig, und wenn die Situation es erfordert, können Sie sie einsetzen, ohne sich dafür zu schämen. In diesem Anhang wollen wir Ihnen einen Überblick darüber geben, worum es bei diesen Mustern geht.



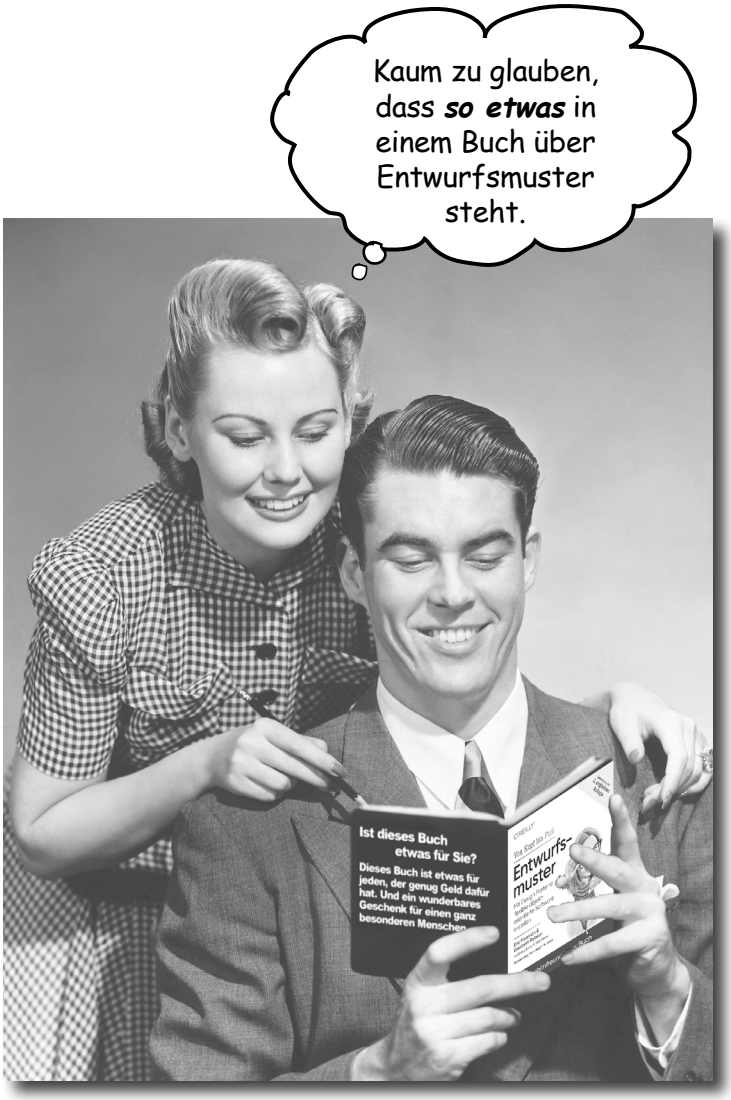
Bridge	598
Builder	600
Chain of Responsibility	602
Flyweight	604
Interpreter	606
Mediator	608
Memento	610
Prototype	612
Visitor	614

i Index

617

Wie man dieses Buch benutzt

Einführung



In diesem Abschnitt beantworten wir die brennende Frage:
>>Warum STEHT so was in einem Buch über Entwurfsmuster?<<

Für wen ist dieses Buch?

Wenn Sie alle diese Fragen mit »Ja« beantworten können …

1. **Können Sie Java? (Sie müssen kein Guru sein.)**

2. Möchten Sie Entwurfsmuster **lernen, verstehen, behalten und *anwenden,*** einschließlich der OO-Entwurfsprinzipien, auf denen Entwurfsmuster basieren?

3. **Ziehen Sie eine anregende Unterhaltung beim Abendessen einer trockenen, langweiligen Vorlesung vor?**

Alle unsere Beispiele sind in Java geschrieben. Sie sollten die Grundkonzepte dieses Buchs aber auch verstehen, wenn Sie eine andere objektorientierte Sprache beherrschen.

… ist dieses Buch etwas für Sie.

Wer sollte eher Abstand von diesem Buch nehmen?

Wenn Sie eine dieser Fragen mit »Ja« beantworten müssen …

1. Ist die **objektorientierte Programmierung** komplettes Neuland für Sie?

2. Sind Sie ein Top-OO-Entwickler, der ein ***Buch zum Nachschlagen*** sucht?

3. Sind Sie ein Softwarearchitekt auf der Suche nach ***Enterprise-***Entwurfsmustern?

4. Haben Sie **Angst, etwas Neues auszuprobieren?** Ist Ihnen eine Wurzelkanalbehandlung lieber, als Streifen kombiniert mit Karos zu tragen? Glauben Sie, dass ein Technikfachbuch, in dem Java-Komponenten vermenschlicht werden, nicht seriös sein kann?

… ist dieses Buch ***nicht*** das richtige für Sie.

[Anmerkung aus dem Marketing: Dieses Buch ist etwas für jeden, der eine Kreditkarte besitzt. Auch Barzahlung ist möglich.]

Wir wissen, was Sie gerade denken

»Kann *das* wirklich ein seriöses Programmierlehrbuch sein?«

»Was sollen all die Abbildungen?«

»Kann ich das auf diese Weise wirklich *lernen*?«

Und wir wissen, was Ihr *Gehirn* gerade denkt.

Ihr Gehirn lechzt nach Neuem. Es ist ständig dabei, Ihre Umgebung abzusuchen, und es *wartet* auf etwas Ungewöhnliches. So ist es nun einmal gebaut, und es hilft Ihnen zu überleben.

Also, was macht Ihr Gehirn mit all den gewöhnlichen, normalen Routinesachen, denen Sie begegnen? Es tut alles in seiner Macht Stehende, damit es dadurch nicht bei seiner *eigentlichen* Arbeit gestört wird: Dinge zu erfassen, die wirklich *wichtig* sind. Es gibt sich nicht damit ab, Langweiliges zu speichern, sondern lässt dieses gar nicht erst durch den »Dies-ist-offensichtlich-nicht-wichtig«-Filter.

Woher *weiß* Ihr Gehirn denn, was wichtig ist? Nehmen Sie an, Sie machen einen Tagesausflug und ein Tiger springt vor Ihnen aus dem Gebüsch: Was passiert dabei in Ihrem Kopf und Ihrem Körper?

Neuronen feuern. Gefühle werden angekurbelt. *Chemische Substanzen durchfluten Sie.*

Und so weiß Ihr Gehirn:

Dies muss wichtig sein! Vergiss es nicht!

Aber nun stellen Sie sich vor, Sie sind zu Hause oder in einer Bibliothek. In einer sicheren, warmen, tigerfreien Zone. Sie lernen. Bereiten sich auf eine Prüfung vor. Oder Sie versuchen, sich in irgendein schwieriges Thema einzuarbeiten, von dem Ihr Chef glaubt, Sie bräuchten dafür eine Woche oder höchstens zehn Tage.

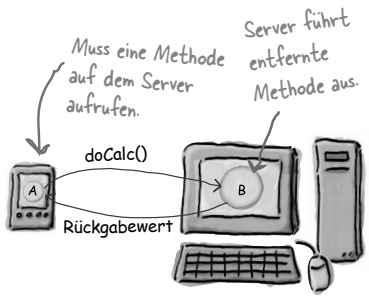
Da ist nur ein Problem: Ihr Gehirn versucht, Ihnen einen großen Gefallen zu tun. Es versucht, dafür zu sorgen, dass diese *offensichtlich* unwichtigen Inhalte nicht knappe Ressourcen verstopfen. Ressourcen, die besser dafür verwendet würden, die wirklich *wichtigen* Dinge zu speichern. Wie Tiger. Wie die Gefahren des Feuers. Wie die Notwendigkeit, schnell das Browserfenster mit dem YouTube-Video zu einer Alien-Entführung zu verbergen, wenn Ihr Chef die Nase ins Büro steckt.

Und es gibt keine einfache Möglichkeit, Ihrem Gehirn zu sagen: »Hey, Gehirn, vielen Dank, aber egal, wie langweilig dieses Buch auch ist und wie klein der Ausschlag auf meiner emotionalen Richterskala gerade ist, ich *will* wirklich, dass du diesen Kram behältst.«

Wir stellen uns unsere Leser:innen als aktiv Lernende vor.

Also, was ist nötig, damit Sie etwas lernen? Erst einmal müssen Sie es aufnehmen und dann dafür sorgen, dass Sie es nicht wieder vergessen. Es geht nicht darum, Fakten in Ihren Kopf zu schieben. Nach den neuesten Forschungsergebnissen der Kognitionswissenschaft, der Neurobiologie und der Lernpsychologie gehört zum Lernen viel mehr als nur Text auf einer Seite. Wir wissen, was Ihr Gehirn anmacht.

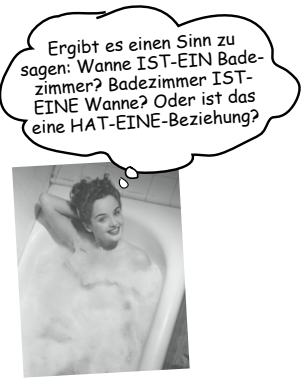
Einige der Lernprinzipien dieser Buchreihe:



Bilder einsetzen. An Bilder kann man sich viel besser erinnern als an Worte allein und lernt so viel effektiver (bis zu 89 % Verbesserung bei Abrufbarkeits- und Lerntransferstudien). Außerdem werden die Dinge dadurch verständlicher. **Text in oder neben die Grafiken setzen,** auf die sie sich beziehen, anstatt darunter oder auf eine andere Seite. Die Leserinnen und Leser werden auf den Bildinhalt bezogene Probleme dann mit *doppelt* so hoher Wahrscheinlichkeit lösen können.

Verwenden Sie einen gesprächsorientierten Stil mit persönlicher Ansprache. Nach neueren Untersuchungen haben Studierende nach dem Lernen bei Tests um bis zu 40 % besser abgeschnitten, wenn der Inhalt Leser:innen direkt in der ersten Person und im lockeren Stil angesprochen hat statt in einem formalen Ton. Halten Sie keinen Vortrag, sondern erzählen Sie Geschichten. Benutzen Sie eine zwanglose Sprache. Nehmen Sie sich selbst nicht zu ernst. Würden *Sie* einer anregenden Unterhaltung beim Abendessen mehr Aufmerksamkeit schenken oder einem Vortrag?

Bringen Sie die Lernenden dazu, intensiver nachzudenken. Mit anderen Worten: Falls Sie nicht aktiv Ihre Neuronen strapazieren, passiert in Ihrem Gehirn nicht viel. Ein Leser oder eine Leserin muss motiviert, begeistert und neugierig sein und angeregt werden, Probleme zu lösen, Schlüsse zu ziehen und sich neues Wissen anzueignen. Und dafür brauchen Sie Herausforderungen, Übungen, zum Nachdenken anregende Fragen und Tätigkeiten, die beide Seiten des Gehirns und mehrere Sinne einbeziehen.



Ziehen Sie die Aufmerksamkeit der Lernenden auf sich – und behalten Sie sie. Wir alle haben schon Erfahrungen dieser Art gemacht: »Ich will das wirklich lernen, aber ich kann einfach nicht über Seite 1 hinaus wach bleiben.« Ihr Gehirn passt auf, wenn Dinge ungewöhnlich, interessant, merkwürdig, auffällig, unerwartet sind. Ein neues, schwieriges, technisches Thema zu lernen, muss nicht langweilig sein. Wenn es das nicht ist, lernt Ihr Gehirn viel schneller.

Sprechen Sie Gefühle an. Wir wissen, dass Ihre Fähigkeit, sich an etwas zu erinnern, wesentlich von dessen emotionalem Gehalt abhängt. Sie erinnern sich an das, was Sie *bewegt*. Sie erinnern sich, wenn Sie etwas *fühlen*. Nein, wir erzählen keine herzzerreißenden Geschichten über einen Jungen und seinen Hund. Was wir erzählen, ruft Überraschungs-, Neugier-, Spaß- und Was-soll-das?-Emotionen hervor und dieses Hochgefühl, das Sie beim Lösen eines Puzzles empfinden oder wenn Sie etwas lernen, das alle anderen schwierig finden. Oder wenn Sie merken, dass Sie etwas können, das dieser »Ich-bin-ein-besserer-Techniker-als-du«-Typ aus der Technikabteilung *nicht kann*.

Metakognition: Nachdenken übers Denken

Wenn Sie wirklich lernen möchten, und zwar schneller und nachhaltiger, dann schenken Sie Ihrer Aufmerksamkeit Aufmerksamkeit. Denken Sie darüber nach, wie Sie denken. Lernen Sie, wie Sie lernen.

Die meisten von uns haben in ihrer Jugend keine Kurse in Metakognition oder Lerntheorie gehabt. Es wurde von uns *erwartet*, dass wir lernen, aber nur selten wurde uns auch *beigebracht*, wie man lernt.

Wir nehmen aber an, dass Sie wirklich etwas über Entwurfsmuster lernen möchten, wenn Sie dieses Buch in den Händen halten. Und wahrscheinlich möchten Sie nicht viel Zeit aufwenden. Und Sie wollen sich an das *erinnern*, was Sie lesen, und es anwenden können. Und deshalb müssen Sie es *verstehen*. Wenn Sie so viel wie möglich von diesem Buch profitieren wollen oder von irgendeinem anderen Buch oder einer anderen Lernerfahrung, übernehmen Sie Verantwortung für Ihr Gehirn. Ihr Gehirn im Zusammenhang mit diesem Lernstoff.

Der Trick besteht darin, Ihr Gehirn dazu zu bringen, neuen Lernstoff als etwas wirklich Wichtiges anzusehen. Als entscheidend für Ihr Wohlbefinden. So wichtig wie einen Tiger. Andernfalls stecken Sie in einem dauerhaften Kampf, in dem Ihr Gehirn sein Bestes gibt, um die neuen Inhalte davon abzuhalten, hängen zu bleiben.

Wie bringen Sie also Ihr Gehirn dazu, Entwurfsmuster für so wichtig zu halten wie einen Tiger?

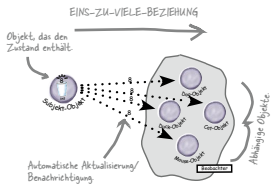
Da gibt es den langsamen, ermüdenden Weg oder den schnelleren, effektiveren Weg. Der langsame Weg geht über bloße Wiederholung. Natürlich ist Ihnen klar, dass Sie lernen und sich sogar an die langweiligsten Themen erinnern *können*, wenn Sie sich dieselbe Sache immer wieder einhämmern. Wenn Sie nur oft genug wiederholen, sagt Ihr Gehirn: »Er hat zwar nicht das *Gefühl*, dass das wichtig ist, aber er sieht sich dieselbe Sache *immer und immer wieder* an – dann muss sie wohl wichtig sein.«

Der schnellere Weg besteht darin, **alles zu tun, was die Gehirnaktivität erhöht**, vor allem verschiedene Arten von Gehirnaktivität. Eine wichtige Rolle dabei spielen die auf der vorhergehenden Seite erwähnten Dinge – alles Dinge, die nachweislich dabei helfen, dass Ihr Gehirn *für* Sie arbeitet. So hat sich z. B. in Untersuchungen gezeigt: Wenn Wörter *in* den Abbildungen stehen, die sie beschreiben (und nicht irgendwo anders auf der Seite, z. B. in einer Bildunterschrift oder im Text), versucht Ihr Gehirn, herauszufinden, wie die Wörter und das Bild zusammenhängen, und dadurch feuern mehr Neuronen. Und je mehr Neuronen feuern, umso größer ist die Chance, dass Ihr Gehirn mitbekommt: Bei dieser Sache lohnt es sich, aufzupassen, und vielleicht auch, sich daran zu erinnern.

Ein lockerer Sprachstil hilft, denn Menschen tendieren zu höherer Aufmerksamkeit, wenn ihnen bewusst ist, dass sie ein Gespräch führen – man erwartet dann ja von ihnen, dass sie dem Gespräch folgen und sich beteiligen. Das Erstaunliche daran ist: Es ist Ihrem Gehirn ziemlich egal, dass die »Unterhaltung« zwischen Ihnen und einem Buch stattfindet! Wenn der Schreibstil dagegen formal und trocken ist, hat Ihr Gehirn den gleichen Eindruck wie bei einem Vortrag, bei dem in einem Raum passive Zuhörer sitzen. Nicht nötig, wach zu bleiben.

Aber Abbildungen und ein lockerer Sprachstil sind erst der Anfang.

Das haben WIR getan:



Wir haben ***Bilder*** verwendet, weil Ihr Gehirn auf visuelle Eindrücke eingestellt ist, nicht auf Text. Soweit es Ihr Gehirn betrifft, sagt ein Bild *wirklich* mehr als 1.024 Worte. Und dort, wo Text und Abbildungen zusammenwirken, haben wir den Text *in* die Bilder eingebettet, denn Ihr Gehirn arbeitet besser, wenn der Text *innerhalb* der Sache steht, auf die er sich bezieht, und nicht in einer Bildunterschrift oder irgendwo vergraben im Text.

Wir haben ***Redundanz*** eingesetzt, d. h. dasselbe auf *unterschiedliche* Art und mit verschiedenen Medientypen ausgedrückt, damit Sie es über *mehrere Sinne* aufnehmen. Das erhöht die Chance, dass die Inhalte an mehr als nur einer Stelle in Ihrem Gehirn verankert werden.

Die Meisterin der Muster

Wir haben Konzepte und Bilder in ***unerwarteter*** Weise eingesetzt, weil Ihr Gehirn auf Neuigkeiten programmiert ist. Und wir haben Bilder und Ideen mit zumindest *etwas* ***emotionalem*** *Charakter* verwendet, weil Ihr Gehirn darauf eingestellt ist, auf die Biochemie von Gefühlen zu achten. An alles, was ein *Gefühl* in Ihnen auslöst, können Sie sich mit höherer Wahrscheinlichkeit erinnern, selbst wenn dieses Gefühl nicht mehr ist als ein bisschen ***Belustigung, Überraschung oder Interesse.***

Wir haben einen ***umgangssprachlichen Stil*** mit direkter Anrede benutzt, denn Ihr Gehirn ist von Natur aus aufmerksamer, wenn es Sie in einer Unterhaltung wähnt, als wenn es davon ausgeht, dass Sie passiv einer Präsentation zuhören – sogar dann, wenn Sie *lesen.*

Wir haben mehr als 100 ***Aktivitäten*** für Sie vorgesehen, denn Ihr Gehirn lernt und behält von Natur aus besser, wenn Sie Dinge ***tun,*** als wenn Sie nur darüber *lesen.* Und wir haben die Übungen zwar anspruchsvoll, aber doch lösbar gemacht, denn so ist es den meisten Lesern am liebsten.

Punkt für Punkt

Wir haben ***mehrere unterschiedliche Lernstile*** eingesetzt, denn vielleicht bevorzugen *Sie* ein Schritt-für-Schritt-Vorgehen, während ein anderer erst einmal den groben Zusammenhang verstehen und ein Dritter einfach nur ein Codebeispiel sehen möchte. Aber ganz abgesehen von den jeweiligen Lernvorlieben profitiert *jeder* davon, wenn er die gleichen Inhalte in unterschiedlicher Form präsentiert bekommt.

Puzzles

Wir liefern Inhalte für ***beide Seiten Ihres Gehirns,*** denn je mehr Sie von Ihrem Gehirn einsetzen, umso wahrscheinlicher werden Sie lernen und behalten, und umso länger bleiben Sie konzentriert. Wenn Sie mit einer Seite des Gehirns arbeiten, bedeutet das häufig, dass sich die andere Seite des Gehirns ausruhen kann; so können Sie über einen längeren Zeitraum produktiver lernen.

Und wir haben ***Geschichten*** und Übungen aufgenommen, die ***mehr als einen Blickwinkel repräsentieren,*** denn Ihr Gehirn lernt von Natur aus intensiver, wenn es gezwungen ist, selbst zu analysieren und zu beurteilen.

Wir haben ***Herausforderungen*** eingefügt: in Form von Übungen und indem wir ***Fragen*** stellen, auf die es nicht immer eine eindeutige Antwort gibt, denn Ihr Gehirn ist darauf eingestellt, zu lernen und sich zu erinnern, wenn es an etwas *arbeiten* muss. Überlegen Sie: Ihren *Körper* bekommen Sie ja auch nicht in Form, wenn Sie die Menschen auf dem Sportplatz nur *beobachten.* Aber wir haben unser Bestes getan, um dafür zu sorgen, dass Sie – wenn Sie schon hart arbeiten – an den *richtigen* Dingen arbeiten. Dass Sie ***nicht einen einzigen Dendriten darauf verschwenden,*** ein schwer verständliches Beispiel zu verarbeiten oder einen schwierigen, mit Fachbegriffen gespickten oder übermäßig gedrängten Text zu analysieren.

Wir haben ***Menschen*** eingesetzt. In Geschichten, Beispielen, Bildern usw. – denn *Sie sind* ein Mensch. Und Ihr Gehirn schenkt *Menschen* mehr Aufmerksamkeit als *Dingen.*

Schneiden Sie dies aus und heften Sie es an Ihren Kühlschrank.

Und das können SIE tun, um sich Ihr Gehirn untertan zu machen

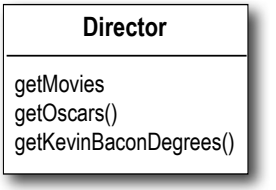
So, wir haben unseren Teil der Arbeit geleistet. Der Rest liegt bei Ihnen. Diese Tipps sind ein Anfang; hören Sie auf Ihr Gehirn und finden Sie heraus, was bei Ihnen funktioniert und was nicht. Probieren Sie neue Wege aus.

1. **Immer langsam. Je mehr Sie verstehen, umso weniger müssen Sie auswendig lernen.**
   Lesen Sie nicht nur. Halten Sie inne und denken Sie nach. Wenn das Buch Sie etwas fragt, springen Sie nicht einfach zur Antwort. Stellen Sie sich vor, dass Sie das wirklich jemand *fragt.* Je gründlicher Sie Ihr Gehirn zum Nachdenken zwingen, umso größer ist die Chance, dass Sie lernen und behalten.
2. **Bearbeiten Sie die Übungen. Machen Sie sich selbst Notizen.**
   Wir haben sie entworfen, aber wenn wir sie auch für Sie lösen würden, wäre dass, als würde ein anderer Ihr Training für Sie absolvieren. Und sehen Sie sich die Übungen *nicht einfach nur an.* **Benutzen Sie einen Bleistift.** Es deutet vieles darauf hin, dass körperliche Aktivität *beim* Lernen den Lernerfolg erhöhen kann.
3. **Lesen Sie die Abschnitte »Es gibt keine Dummen Fragen«.**
   Und zwar alle. Das sind keine Zusatzanmerkungen – ***sie gehören zum Kerninhalt!*** Überspringen Sie sie nicht.
4. **Lesen Sie dies als Letztes vor dem Schlafengehen. Oder lesen Sie danach zumindest nichts *Anspruchsvolles* mehr.**
   Ein Teil des Lernprozesses (vor allem die Übertragung in das Langzeitgedächtnis) findet erst statt, *nachdem* Sie das Buch zur Seite gelegt haben. Ihr Gehirn braucht Zeit für sich, um weitere Verarbeitung zu leisten. Wenn Sie in dieser Zeit etwas Neues aufnehmen, geht ein Teil dessen, was Sie gerade gelernt haben, verloren.
5. **Trinken Sie Wasser. Viel.**
   Ihr Gehirn arbeitet am besten in einem schönen Flüssigkeitsbad. Austrocknung (zu der es schon kommen kann, bevor Sie überhaupt Durst verspüren) beeinträchtigt die kognitive Funktion.
6. **Reden Sie drüber. Laut.**
   Sprechen aktiviert einen anderen Teil des Gehirns. Wenn Sie etwas verstehen wollen oder Ihre Chancen verbessern möchten, sich später daran zu erinnern, sagen Sie es laut. Noch besser: Versuchen Sie, es jemand anderem laut zu erklären. Sie lernen dann schneller und haben vielleicht Ideen, auf die Sie beim bloßen Lesen nie gekommen wären.
7. **Hören Sie auf Ihr Gehirn.**
   Achten Sie darauf, Ihr Gehirn nicht zu überladen. Wenn Sie merken, dass Sie etwas nur noch überfliegen oder dass Sie das gerade erst Gelesene vergessen haben, ist es Zeit für eine Pause. Ab einem bestimmten Punkt lernen Sie nicht mehr schneller, indem Sie mehr hineinzustopfen versuchen; das kann sogar den Lernprozess stören.
8. **Aber bitte mit *Gefühl!***
   Ihr Gehirn muss wissen, dass es *um etwas Wichtiges geht.* Lassen Sie sich in die Geschichten hineinziehen. Erfinden Sie eigene Bildunterschriften für die Fotos. Über einen schlechten Scherz zu stöhnen, ist *immer noch* besser, als gar nichts zu fühlen.
9. **Entwerfen Sie etwas!**
   Wenden Sie das hier Gelernte auf einen Entwurf an, an dem Sie gerade arbeiten, oder gestalten Sie ein älteres Projekt damit um. Tun Sie irgendetwas, um weitere Erfahrungen zu sammeln, die über die Übungen und Aktivitäten in diesem Buch hinausgehen. Sie brauchen dazu nur einen Bleistift und ein Problem, das es zu lösen gilt … ein Problem, das von einem oder mehreren Entwurfsmustern profitieren würde.

Lies mich

Dies ist ein Lehr- und Lernbuch, kein Nachschlagewerk. Wir haben absichtlich alles weggelassen, das dem Lernen im Weg stehen könnte, egal woran wir gerade im Buch arbeiten. Beim ersten Durchlesen müssen Sie am Anfang beginnen, weil das Buch an einigen Stellen davon ausgeht, dass sie bestimmte Dinge bereits gesehen und gelernt haben.

Wir benutzen eine einfachere, angepasste Version von UML.



Wir verwenden einfache Diagramme im UML-Stil.

Auch wenn Sie UML wahrscheinlich schon einmal begegnet sind, wird es in diesem Buch nicht behandelt und ist auch keine Voraussetzung. Keine Sorge, wenn Ihnen UML vollkommen neu ist. Wir geben Ihnen unterwegs die nötigen Hinweise. Das heißt, Sie müssen sich nicht gleichzeitig über Entwurfsmuster und UML den Kopf zerbrechen. Unsere Diagramme sind im »UML-*Stil*« gehalten. Das heißt, wir versuchen, möglichst nah am echten UML zu bleiben, werden die Regeln hier und da aber auch brechen – meistens aus rein egoistischen und künstlerischen Gründen.

Wir behandeln nicht alle je entwickelten Entwurfsmuster.

Inzwischen gibt es *eine Menge* Entwurfsmuster: Die ursprünglichen Grundmuster (auch als GoF-Muster – »Gang of Four« – bezeichnet), Enterprise-Java-Muster, architektonische Muster, Gamedesignmuster und viele mehr. Allerdings war es uns wichtig, dass das Buch weniger wiegt als die Person, die es liest. Daher konzentrieren wir uns hier auf die *wirklich wichtigen* objektorientierten Kernmuster und sorgen dafür, dass Sie wirklich, wahrhaftig und vollständig verstehen, wie und wann sie benutzt werden. Im Anhang dieses Buchs finden Sie einen Überblick über einige andere Muster (die Sie wahrscheinlich deutlich seltener nutzen werden – wenn überhaupt). Auf jeden Fall werden Sie nach der Lektüre von *Entwurfsmuster von Kopf bis Fuß* in der Lage sein, einen beliebigen Musterkatalog in die Hand zu nehmen und schnell loszulegen.

Die Übungen sind NICHT optional.

Die Übungen und Aktivitäten sind keine Extras, sondern gehören zum Kerninhalt dieses Buchs. Einige Übungen dienen der Erinnerung, andere dem Verständnis, und einige helfen Ihnen, das Gelernte anzuwenden. ***Überspringen Sie die Übungen nicht.*** Allein die Kreuzworträtsel sind *freiwillig*. Allerdings geben sie Ihrem Gehirn Gelegenheit, die Wörter in einem anderen Kontext wahrzunehmen.

Wir benutzen das Wort »Komposition« in der allgemeinen OO-Bedeutung, die flexibler ist als die striktere UML-Form.

Wenn wir sagen, dass »ein Objekt durch Komposition mit einem anderen Objekt zusammengefügt« wurde, meinen wir damit, dass eine HAT-EIN-Beziehung zwischen ihnen besteht. So wurde dieser Begriff traditionell und auch im GoF-Buch verwendet (was das ist, erfahren Sie später). In UML wurde dieser Begriff dann später in verschiedene Kompositionstypen aufgegliedert. Aber auch wenn Sie UML-Experte sind, werden Sie das Buch lesen und den Begriff »Komposition« jeweils leicht in den genaueren Fachbegriff übersetzen können.

Die Redundanz ist wichtig und beabsichtigt.

Ein wesentliches Anliegen eines *Von Kopf bis Fuß*-Buchs ist: Wir wollen, dass Sie es *wirklich* kapieren. Und wir wollen, dass Sie sich am Ende des Buchs an das Gelernte erinnern. Beständigkeit und Erinnern ist bei den meisten Referenzbüchern nicht das Ziel, aber in diesem Buch geht es ums *Lernen*. Deshalb werden Sie einige der hier gezeigten Konzepte mehr als einmal zu Gesicht bekommen.

Die Codebeispiele sind so kurz wie möglich.

Unsere Leserinnen und Leser berichten uns, dass sie es frustrierend finden, sich durch 200 Zeilen Code graben zu müssen, um die beiden Zeilen zu finden, die sie wirklich verstehen müssen. Die meisten Beispiele in diesem Buch werden mit so wenig Kontext wie möglich gezeigt, damit der Teil, den Sie lernen sollen, klar und einfach ist. Sie dürfen nicht erwarten, dass der Code robust oder gar vollständig ist. Die Beispiele in diesem Buch wurden speziell für Lehrzwecke geschrieben und sind nicht immer voll funktionsfähig (obwohl wir versucht haben, das so weit wie möglich sicherzustellen).

In manchen Fällen haben wir nicht alle nötigen Importanweisungen abgedruckt. Wir gehen davon aus, dass Sie als Java-Programmierer wissen, dass `ArrayList in` java.util zu finden ist. Wenn die Importe nicht Teil der Kern-JSE-API sind, erwähnen wir das. Außerdem haben wir den gesamten Quellcode zum Herunterladen bereitgestellt. Sie finden ihn unter:

`http://wickedlysmart.com/head-first-design-patterns`

Da wir uns auf den Lernaspekt des Codes konzentrieren wollen, haben wir unsere Klassen auch nicht in Pakete gesteckt (mit anderen Worten, sie liegen alle im Java-Standardverzeichnis). In der realen Welt empfehlen wir das allerdings nicht – wenn Sie sich die Codebeispiele herunterladen, werden Sie feststellen, dass dort alle Klassen in Paketen organisiert sind.

Zu den Kopfnuss-Übungen gibt es keine Lösungen.

Für manche dieser Übungen gibt es keine richtige Lösung, und bei anderen gehört es zum Lernprozess der Kopfnuss-Aktivitäten, dass Sie selbst überlegen, ob und wann Ihre Lösungen richtig sind. Bei einigen Kopfnuss-Übungen finden Sie Hinweise, die Sie in die richtige Richtung lenken.

Fachgutachter

Jef Cumps

Valentin Crettaz

Barney Marispini

Ike Van Atta

Furchtloser Leiter des Entwurfsmuster-Extreme-Review-Teams

Johannes deJong

Jason Menard

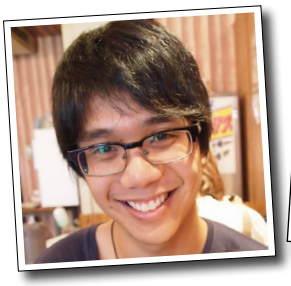
Im Gedenken an Philippe Maquet, 1960–2004. Dein erstaunliches Fachwissen, deine unermüdliche Begeisterung und dein intensives Bemühen um die Lernenden werden uns immer inspirieren.

Philippe Maquet

Mark Spritzler

Dirk Schreckmann

Fachgutachter, zweite Auflage



Julian Setiawan

George Heineman

2nd Edition Reviewer MVP!

David Powers

Trisha Gee

Danksagungen

Für die erste Auflage:

An O'Reilly:

Unser besonderer Dank geht an **Mike Loukides** bei O'Reilly, der alles ins Leben gerufen und geholfen hat, aus dem »Head First«-Konzept eine Buchreihe zu entwickeln (»Head First« heißt die amerikanische Reihe, aus der dieses Buch stammt). Und ein großes Dankeschön auch an die treibende Kraft hinter Head First, **Tim O'Reilly**. Dank ebenfalls an die kluge Head First-»Serienmutter« **Kyle Hart**, an »InDesign-König« **Ron Bilodeau**, an Rockstar **Ellie Volckhausen** für ihr geniales Titeldesign, an **Melanie Yarbrough** für das Hüten der Herstellung, an **Colleen Gorman** für das gnadenlose Sprachkorrektorat und an **Bob Pfahler** für einen deutlich verbesserten Index. Vielen Dank schließlich an **Mike Hendrickson** und **Meghan Blanchette** dafür, dass sie sich für dieses Entwurfsmusterbuch eingesetzt und das Team zusammengestellt haben.

An unsere tapferen Fachgutachter:

Ganz besonders dankbar sind wir unserem Chef-Reviewer **Johannes deJong**. Du bist unser Held, Johannes. Überaus dankbar sind wir auch für die Beiträge des Co-Managers im **Javaranch**-Gutachterteam, des verstorbenen **Philippe Maquet**. Du ganz persönlich hast das Leben Tausender Entwickler bereichert und einen unauslöschlichen Eindruck in ihrem (und in unserem) Leben hinterlassen. **Jef Cumps** hat ein beängstigendes Talent, Probleme in unseren Rohfassungen der Kapitel zu finden, was für das Buch eine große Rolle gespielt hat. Danke, Jef! **Valentin Crettaz** (unser AOP-Fachmann), der schon seit dem allerersten Buch der Reihe dabei ist, hat uns (wie immer) gezeigt, wie sehr wir doch auf sein Fachwissen und seinen Einblick angewiesen sind. Du bist spitze, Valentin (aber leg die Krawatte ab!).

Zwei Neulinge im Head First-Fachgutachterteam, **Barney Marispini** und **Ike Van Atta**, haben ganz hervorragende Arbeit für das Buch geleistet – ihr habt uns wirklich entscheidende Rückmeldungen gegeben. Danke, dass ihr zum Team gestoßen seid.

Ausgezeichnete fachliche Hilfe haben wir auch von den Javaranch-Moderatoren und -Gurus **Mark Spritzler**, **Jason Menard**, **Dirk Schreckmann**, **Thomas Paul** und **Margarita Isaeva** erhalten. Und wie immer danken wir besonders dem Obercowboy von javaranch.com, **Paul Wheaton**.

Danke auch an die Teilnehmer der Endrunde im Javaranch-Wettbewerb »Pick the Head First Design Patterns Cover«. Der Gewinner, Si Brewster, hat uns mit seinem Beitrag überzeugt, die Frau auszuwählen, die Sie jetzt auf unserem Einband sehen. In der Endrunde waren außerdem Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni und Jeff Fisher.

Für die Aktualisierung in 2014 danken wir: George Hoffer, Ted Hill, Todd Bartoszkiewicz, Sylvain Tenier, Scott Davidson, Kevin Ryan, Rich Ward, Mark Francis Jaeger, Mark Masse, Glenn Ray, Bayard Fetler, Paul Higgins, Matt Carpenter, Julia Williams, Matt McCullough und Mary Ann Belarmino.

Danksagungen

Für die zweite Auflage:

An O'Reilly:

Zuallererst ist da **Mary Treseler**. Sie ist die Superheldin, ohne die gar nichts läuft. Wir sind ihr auf ewig für alles dankbar, was sie für O'Reilly, Head First und die Autoren tut. **Melissa Duffield** und **Michele Cronin** haben viele Wege freigeräumt, die zu dieser zweiten Auflage geführt haben. Ein großes Dankeschön auch an **Rachel Monaghan**, die unserem Text mit ihrem Sprachkorrektorat zu neuem Glanz verholfen hat. **Kristen Brown** sorgte dafür, dass dieses Buch online und im Druck gut aussieht. Die Magie von **Ellie Volckhausen** hat dieser zweiten Auflage zu einem neuen Cover verholfen. Vielen Dank an euch alle!

An unsere Fachgutachter für die zweite Auflage:

Wir danken unseren Fachgutachtern für die zweite Auflage dafür, dass sie diese Aufgabe 15 Jahre später übernommen haben. **David Powers** ist unser Gutachter der Wahl (ja **unser** – denken Sie nicht einmal daran, ihn zu bitten, *Ihr* Buch zu begutachten), weil ihm einfach nichts entgeht. Mit seinen detaillierten Kommentaren, Vorschlägen und Rückmeldungen ging **George Heineman** weit über das normale Maß hinaus und bekommt dafür den MVP-Award (Most Valuable Player) für diese Auflage. **Trisha Gee** und **Julian Setiawan** halfen mit ihren unschätzbar wertvollen Java-Kenntnissen dabei, all die peinlichen und schauderhaften Java-Fehler zu vermeiden. Herzlichen Dank an euch alle!

Ein ganz besonderes Dankeschön

Ein besonderes Dankeschön geht an **Erich Gamma**, der weit über seine Aufgabe als Gutachter hinausgegangen ist (er hat sogar ein Vorabmanuskript mit in den Urlaub genommen). Erich, dein Interesse an diesem Buch hat uns inspiriert, und dein technisches Gutachten hat es enorm verbessert. Vielen Dank auch an die gesamte **Gang of Four** für ihre Unterstützung, ihr Interesse und für einen besonderen Auftritt in Objectville. Weiteren Dank schulden wir **Ward Cunningham** und der Patterns-Community, die das Portland Pattern Repository angelegt haben, das uns beim Schreiben dieses Buchs eine Ressource von unschätzbarem Wert war.

Ein großes Dankeschön an **Mike Loukides**, **Mike Hendrickson** und **Meghan Blanchette**. Mike L. hat uns bei jedem Schritt dieses Wegs begleitet. Mike, deine fundierten Rückmeldungen haben geholfen, diesem Buch eine Form zu geben, und dein Ansporn brachte uns vorwärts. Mike H., danke für deine Beständigkeit über fünf Jahre, in denen du versucht hast, uns zum Schreiben eines Entwurfsmusterbuchs zu bewegen: Wir haben es schließlich geschafft und sind froh, dass wir auf Head First gewartet haben.

Es braucht ein ganzes Dorf, um so ein Fachbuch zu schreiben: **Bill Pugh** und **Ken Arnold** halfen mit ihrem Fachwissen zu Singleton. **Joshua Marinacci** unterstützte uns mit Tipps und Ratschlägen zu Swing. **John Brewer**s »Why a Duck?«-Artikel diente als Inspiration für SimUDuck (zum Glück mag auch er Enten). **Dan Friedman** inspirierte uns mit dem Beispiel zum Kleinen Singleton. **Daniel Steinberg** diente als unser »technischer Verbindungsoffizier« und als unser emotionales Unterstützernetzwerk. Danke auch an **James Dempsey** von Apple für die Erlaubnis, seinen MVC-Song zu benutzen. Und danke an **Richard Warburton**, der sicherstellte, dass unsere Java-8-Code-Updates für die aktualisierte Version dieses Buchs auf dem neuesten Stand waren.

Zum Schluss ein persönliches Dankeschön an das **Javaranch-Gutachterteam** für ihre erstklassigen Reviews und die herzliche Unterstützung. In diesem Buch steckt mehr von euch, als ihr denkt.

Das Schreiben eines »Von Kopf bis Fuß«-Buchs ist ein wilder Ritt mit zwei erstaunlichen Tour-Guides: **Kathy Sierra** und **Bert Bates**. Wenn man mit Kathy und Bert zusammenarbeitet, schmeißt man alle Konventionen zum Schreiben von Büchern über Bord und betritt eine Welt voller Geschichten, Lerntheorie, Kognitionswissenschaft und Popkultur, in der es grundsätzlich um die Leserinnen und Leser geht.

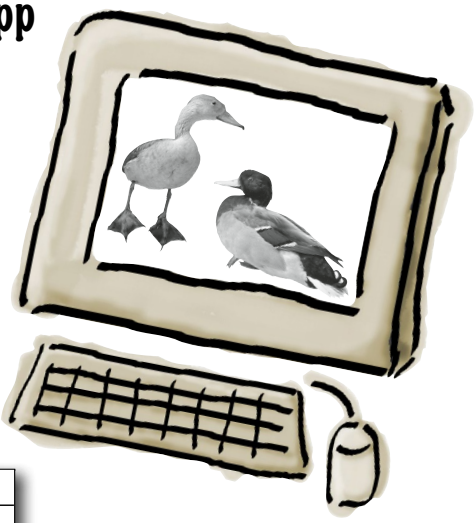
1 Einführung in Entwurfsmuster

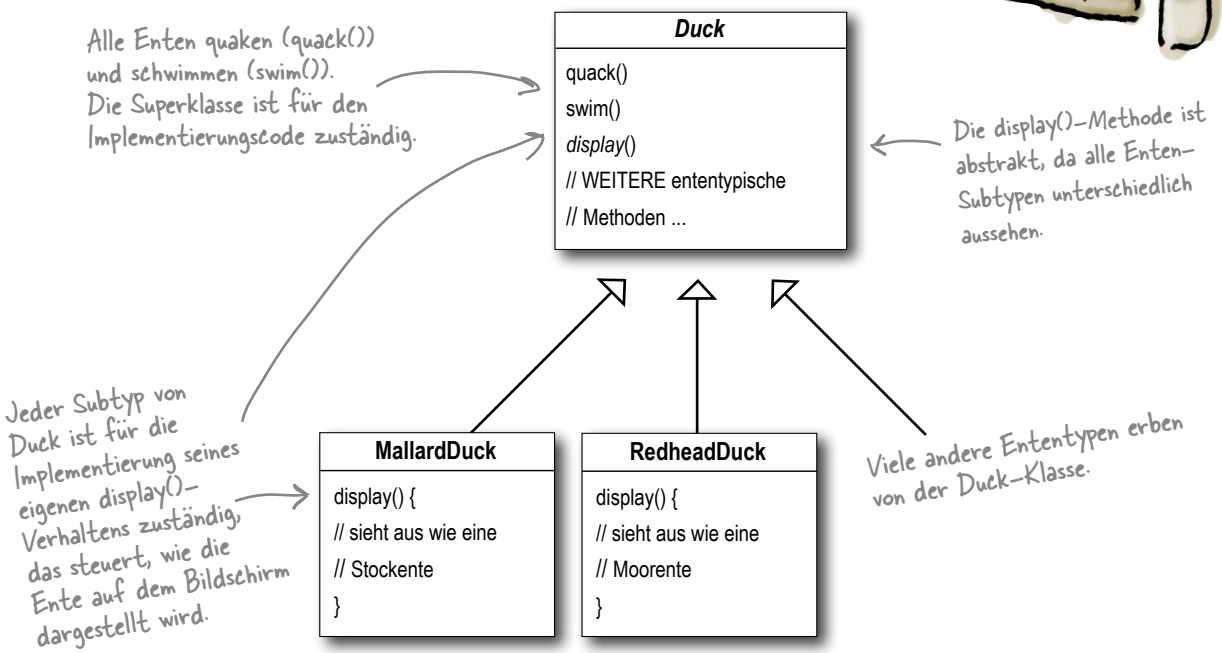
Willkommen bei den Entwurfsmustern

Irgendjemand hat Ihr Problem schon gelöst. In diesem Kapitel lernen Sie, warum (und wie) Sie die Weisheit und die Lehren anderer Entwickler nutzen können, die die gleichen Designprobleme bereits hatten und die Reise überlebt haben. Bevor dieses Kapitel zu Ende ist, kümmern wir uns um die Verwendung und die Vorteile der Entwurfsmuster, sehen uns ein paar grundsätzliche objektorientierte (OO-)Designprinzipien an und gehen mit Ihnen zusammen ein Beispiel für die Funktionsweise von Entwurfsmustern durch. Die beste Möglichkeit, die Muster zu verwenden, ist, sie *in Ihr Gehirn zu laden* und dann die Stellen in Ihren Designs und bestehenden Programmen zu *erkennen*, an denen der Einsatz sinnvoll ist. Im Gegensatz zur Codewiederverwendung können Sie mit Entwurfsmustern die *Erfahrung* anderer Menschen wiederverwenden.

Es begann mit einer einfachen SimUDuck-App

Joe arbeitet für ein Unternehmen, das das extrem erfolgreiche Ententeich-Simulationsspiel *SimUDuck* herstellt. Das Spiel kann viele Entenarten beim Schwimmen und Quaken zeigen. Die ursprünglichen Entwickler des Systems setzten klassische OO-Techniken ein. Sie erstellten eine Duck-(»Ente«-)Superklasse, von der alle anderen Ententypen erbten.

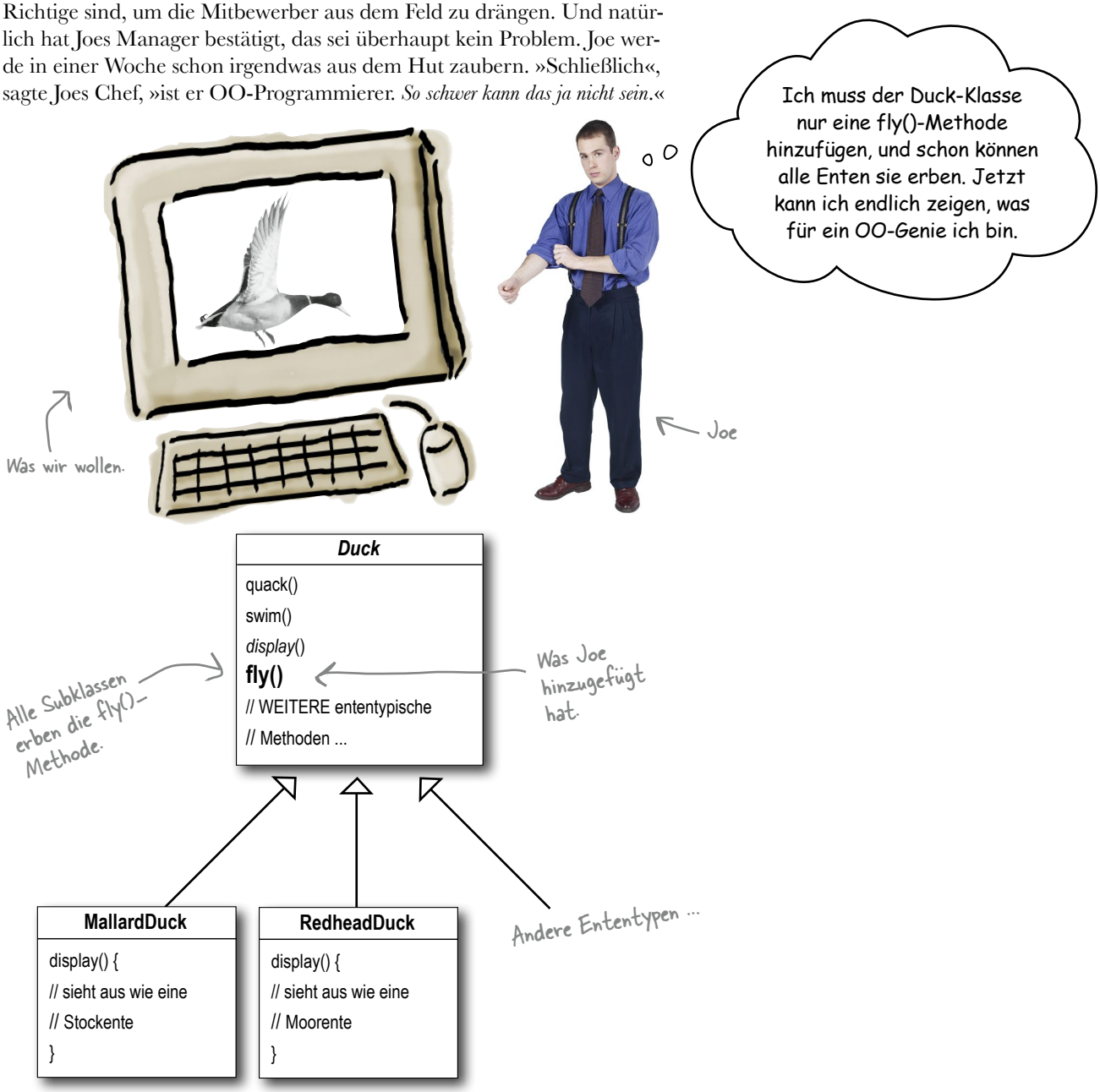




Im vergangenen Jahr sah sich das Unternehmen einem wachsenden Druck von Mitbewerbern ausgesetzt. Nach einem einwöchigen Brainstorming beim Golfen waren sich die Chefs darüber einig, dass die Zeit reif sei für eine große Innovation. Sie brauchten etwas *wirklich* Eindrucksvolles, das sie *in der folgenden Woche* bei der in Maui/Hawaii stattfindenden Aktionärsversammlung vorzeigen konnten.

Aber jetzt sollen die Enten FLIEGEN können

Die Geschäftsleitung hat entschieden, dass fliegende Enten genau das Richtige sind, um die Mitbewerber aus dem Feld zu drängen. Und natürlich hat Joes Manager bestätigt, das sei überhaupt kein Problem. Joe werde in einer Woche schon irgendwas aus dem Hut zaubern. »Schließlich«, sagte Joes Chef, »ist er OO-Programmierer. *So schwer kann das ja nicht sein.*«



Aber irgendetwas ging furchtbar schief ...

Joe, ich bin auf dem Aktionärstreffen. Sie haben gerade ein Demo gezeigt, und da flogen **Gummienten** über den Bildschirm. Soll das etwa ein Witz sein?

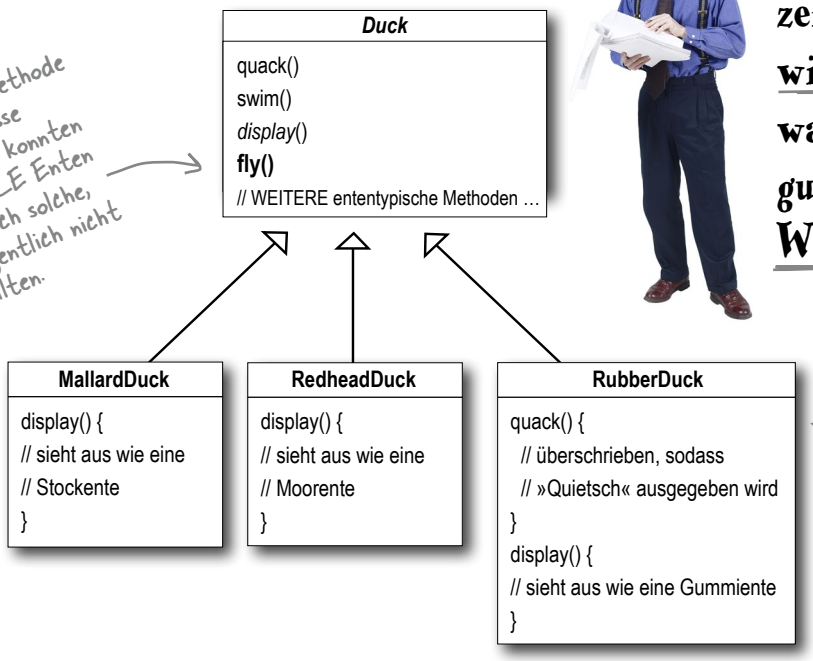
Was ist passiert?

Joe hatte übersehen, dass nicht *alle* Subklassen von Duck *fliegen* dürfen. Das neue Verhalten, um das Joe die Superklasse Duck erweitert hatte, war nun auch in Subklassen verfügbar, die es eigentlich nicht hätten haben sollen. Plötzlich konnten in SimUDuck auch leblose Objekte fliegen.
Ein lokalisiertes Code-Update hatte nicht lokale Nebenwirkungen (fliegende Gummienten)!

Okay, mein Design hat einen kleinen Fehler. Warum können die das nicht einfach als »Feature« bezeichnen? Ist doch irgendwie süß.

Joes Idee, Vererbung einzusetzen, um den Code wiederzuverwenden, war doch nicht so gut, wenn es um die Wartung geht.

Weil die fly()-Methode in der Superklasse definiert war, konnten plötzlich ALLE Enten fliegen – auch solche, die das eigentlich nicht können sollten.



Außerdem quaken Gummienten nicht. Daher haben wir auch die quack()-Methode überschrieben, sodass die Ente stattdessen quietscht.

Joe denkt über Vererbung nach ...

Ich könnte die fly()-Methode einfach wie die quack()-Methode in RubberDuck überschreiben.

RubberDuck

```
quack()  { // quietschen}
display() { // Gummiente }
fly() {
  // überschreiben, sodass
  // nichts getan wird
}
```

Aber was passiert, wenn ich das Programm um hölzerne Lockenten erweitere? Die sollen auch nicht fliegen oder quaken.

Hier eine weitere Klasse der Hierarchie. Wie RubberDuck fliegt und quakt sie nicht.

DecoyDuck

```
quack() {
  // überschreiben, sodass
  // nichts getan wird
}

display() { // decoy duck}

fly() {
  // überschreiben, sodass
  // nichts getan wird
}
```

Spitzen Sie Ihren Bleistift

Welche der folgenden Punkte sind Nachteile beim Einsatz von *Vererbung*, um das Verhalten der Enten zu implementieren? (Kreuzen Sie alle zutreffenden Punkte an.)

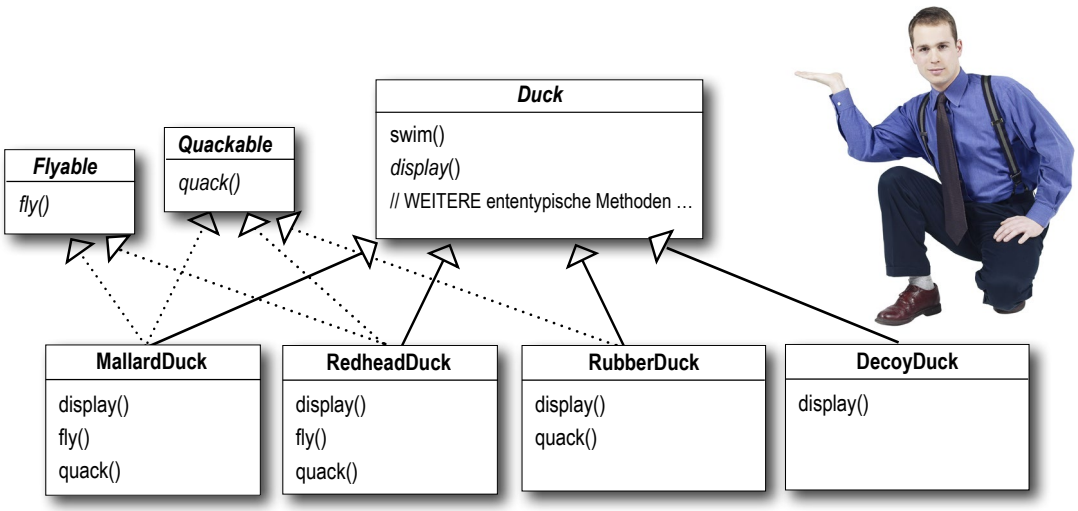
- ❑ A. Code wird in Subklassen dupliziert.
- ❑ B. Änderungen des Laufzeitverhaltens sind schwierig.
- ❑ C. Wir können Enten nicht das Tanzen beibringen.
- ❑ D. Es ist schwer, alle Verhaltensweisen der Enten genau zu kennen.
- ❑ E. Enten können nicht gleichzeitig fliegen und quaken.
- ❑ F. Änderungen können sich unbeabsichtigt auf andere Enten auswirken.

Wie wäre es mit einem Interface?

Joe begriff, dass Vererbung offenbar nicht die Antwort war. Er hatte nämlich gerade ein Memo erhalten, in dem die Geschäftsleitung mitteilte, dass das Produkt alle sechs Monate aktualisiert werden soll (wie genau, war noch nicht entschieden). Joe wusste, dass sich die Spezifikationen ständig ändern würden und dass er möglicherweise gezwungen wäre, die fly()- und quack()-Methoden für jede einzelne Duck-Subklasse im Programm zu überschreiben … *für immer*.

Er brauchte also einen besseren Weg, das Fliegen und Quaken nur für *bestimmte* (aber nicht *alle*) Enten anzupassen.

Ich könnte fly() aus der Duck-Superklasse entfernen und stattdessen ein **Flyable()-(»KannFliegen()«-)Interface** erstellen, das eine fly()-Methode enthält. Das Interface müsste nur von Enten implementiert werden, die tatsächlich eine fly()-Methode brauchen. Und weil nicht alle Enten quaken können, baue ich auch gleich ein Quackable()-Interface (»KannQuaken()«).



Was halten SIE von diesem Design?

Das ist so ziemlich das Dümmste, was ich je von dir gehört habe. **Hast du schon mal was von »dupliziertem Code« gehört?** Wenn du glaubst, das *Überschreiben von ein paar Methoden* sei schlimm, wie wird es dir erst gefallen, eine kleinere Änderung am Flugverhalten vorzunehmen – und zwar *bei allen 48 Subklassen für fliegende Enten*?

Was würden Sie an Joes Stelle tun?

Wir wissen, dass nicht *alle* Subklassen fliegen oder quaken sollen. Vererbung ist also die falsche Antwort. Wenn die Subklassen das Flyable- bzw. das Quackable-Interface implementieren, lösen wir immerhin einen Teil des Problems (keine fliegenden Gummienten mehr). Gleichzeitig wird die Codewiederverwendbarkeit für dieses Verhalten komplett zerstört, und wir tauschen einen Wartungsalptraum nur *durch einen anderen* aus. Und natürlich kann es verschiedene Arten von Flugverhalten unter den flugfähigen Enten geben …

An dieser Stelle warten Sie vermutlich auf ein Entwurfsmuster, das auf einem weißen Ross angaloppiert kommt und Rettung bringt. Aber wo bleibt da der Spaß? Nein. Wir finden unsere Lösung auf die altmodische Art – *indem wir gute alte OO-Designprinzipien anwenden.*

Wäre es nicht traumhaft, wenn wir Software so schreiben könnten, dass nötige Änderungen den bestehenden Code möglichst wenig beeinflussen? Wir bräuchten weniger Zeit zum Refaktorieren und hätten mehr Zeit, dem Programm neue coole Dinge beizubringen …

Die einzige Konstante in der Softwareentwicklung

Okay, worauf können Sie sich in der Softwareentwicklung immer verlassen?

Egal woran Sie arbeiten und was oder in welcher Sprache Sie programmieren – was ist die einzig wahre Konstante, die sich niemals ändert?

(Benutzen Sie einen Spiegel, um die Antwort zu lesen)

Ganz gleich, wie gut Ihr Programm entworfen ist: Im Laufe der Zeit muss es wachsen, sich verändern und schließlich *sterben*.

Spitzen Sie Ihren Bleistift

Viele Dinge können Veränderungen bewirken. Notieren Sie einige Gründe dafür, dass Sie den Code Ihrer Applikation ändern mussten (um Ihnen ein Beispiel zu geben, haben wir ein paar unserer eigenen Gründe schon aufgeschrieben.) Vergleichen Sie Ihre Antworten mit der Lösung am Ende des Kapitels, bevor Sie weitermachen.

Meine Kunden oder Benutzer haben sich für etwas anderes oder neue Funktionalitäten entschieden.

Mein Unternehmen ist zu einem anderen Datenbankanbieter gewechselt. Außerdem haben dessen Daten ein anderes Format. Argh!

Das Problem eingrenzen

Wir wissen also, dass Vererbung hier nicht so gut funktioniert, weil sich das Verhalten der Enten über die Subklassen verändert und nicht *alle* Subklassen dieses Verhalten überhaupt brauchen. Der Einsatz der Interfaces Flyable und Quackable klang eigentlich ganz gut – nur Enten, die tatsächlich fliegen, brauchen Flyable. Nur enthalten Java-Interfaces typischerweise keinen Implementierungscode. Die Codewiederverwendung fällt also weg. Soll bestimmtes Verhalten geändert werden, müssen Sie es oftmals in allen Subklassen, die es verwenden, aufspüren und anpassen, was schnell zu *neuen* Bugs führen kann!

Zum Glück gibt es für genau diese Situation ein Entwurfsprinzip.

Entwurfsprinzip

Ermitteln Sie die veränderlichen Teile Ihrer Applikation und trennen Sie sie von den unveränderlichen.

Das erste Entwurfsprinzip. Im Verlauf des Buchs werden wir noch einige weitere kennenlernen.

Anders gesagt: Wenn sich ein Teil Ihres Codes ändert, zum Beispiel mit jeder neuen Anforderung, wissen Sie, dass dieses Verhalten von den unveränderlichen Codeteilen getrennt werden muss.

Sie können sich das auch so vorstellen: ***Verkapseln Sie die Dinge, die sich ändern können, damit Sie diese Teile später anpassen können, ohne dass sich das auf die unveränderlichen Teile auswirkt.***

Trotz seiner Einfachheit bildet dieses Prinzip das Fundament für fast alle Entwurfsmuster. Sämtliche Muster schaffen die Möglichkeit, *bestimmte Teile eines Systems unabhängig von anderen zu ändern.*

Dann mal los. Gliedern wir das Entenverhalten aus den Duck-Klassen aus!

»Verkapseln« Sie die veränderlichen Dinge, sodass sie den Rest des Codes nicht beeinflussen.

Das Ergebnis? Weniger unerwünschte Folgen durch Codeänderungen und größere Flexibilität in Ihren Systemen!

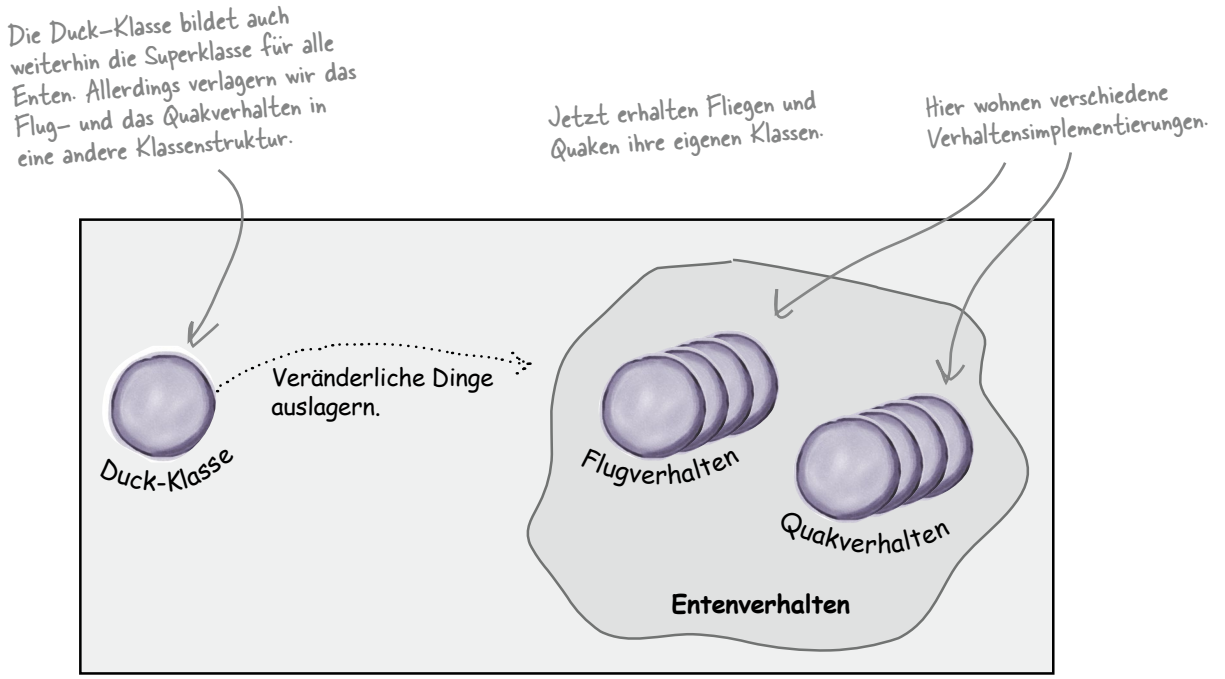
Veränderliches und Unveränderliches voneinander trennen

Wo sollen wir anfangen? Soweit wir wissen, funktioniert die Duck-Klasse bis auf die Probleme mit fly() und quack() eigentlich recht gut, und es gibt keine anderen Teile, die sich unterscheiden oder ständig ändern. Abgesehen von ein paar kleinen Anpassungen werden wir die Duck-Klasse größtenteils in Ruhe lassen.

Um die »veränderlichen Teile von den unveränderlichen« zu trennen, erstellen wir zwei (von Duck vollkommen unabhängige) Klassensätze, eine für das *Fliegen* und eine für das *Quaken*. Jeder Satz an Klassen enthält die vollständige Implementierung für das jeweilige Verhalten. Das heißt, eine Klasse implementiert das *Quaken*, eine andere das *Quietschen* und eine weitere die *Stille*.

Wir wissen, dass sich fly() und quack() zwischen den verschiedenen Duck-Klassen unterscheiden.

Um diese Verhaltensweisen von der Duck-Klasse zu trennen, entfernen wir die jeweiligen Methoden aus der Klasse und erstellen einen neuen Satz Klassen, die das jeweilige Verhalten abbilden sollen.



Das Entenverhalten entwerfen

Wie müssen wir die Klassensätze entwerfen, die das Flug- und das Quakverhalten implementieren?

Grundsätzlich sollen die Dinge möglichst flexibel bleiben. Das Problem war schließlich nicht die mangelnde Flexibilität des Entenverhaltens. Außerdem wissen wir, dass wir den Instanzen von Duck Verhalten *zuweisen* wollen. Vielleicht möchten wir eine neue Stockente (MallardDuck) instanziieren und sie mit einem ganz bestimmten *Typ* Flugverhalten initialisieren. Und wenn wir schon dabei sind, können wir auch gleich dafür sorgen, dass das Verhalten einer Ente dynamisch geändert werden kann, oder? Anders gesagt: Die Duck-Klassen sollten Setter-Methoden enthalten, sodass wir das Flugverhalten von MallardDuck *zur Laufzeit ändern* können.

Mit diesen Zielen im Hinterkopf sehen wir uns das zweite Entwurfsprinzip an:

Entwurfsprinzip

Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.

Wir werden eine Schnittstelle einsetzen, um das jeweilige *Verhalten* abzubilden, zum Beispiel FlyBehavior (FlugVerhalten) und QuackBehavior (QuakVerhalten). Dabei implementiert jede Implementierung eines Verhaltens eines dieser Interfaces.

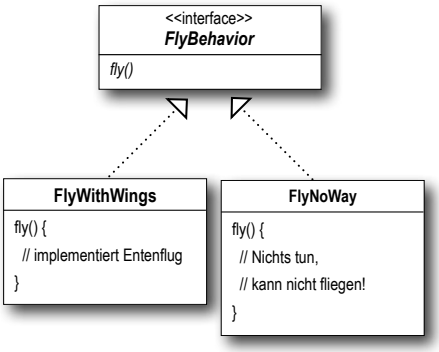
Diesmal wird das Fliegen und Quaken also nicht von der *Duck*-Klasse implementiert – stattdessen erstellen wir eine Reihe von Klassen, deren einziger Lebenszweck darin besteht, ein bestimmtes Verhalten abzubilden (zum Beispiel »Quietschen«). Dabei wird das Verhaltens-Interface nicht von der Duck-Klasse, sondern von dieser speziellen Verhaltensklasse implementiert.

Im Gegensatz dazu kam das Verhalten vorher von einer konkreten Implementierung in der Superklasse Duck oder einer spezialisierten Implementierung direkt in der Subklasse. In beiden Fällen verließen wir uns auf eine Implementierung. Diese mussten wir benutzen, ohne Raum für Änderungen des Verhaltens zu haben (außer mehr Code zu schreiben).

Bei unserem neuen Design wird das Verhalten der Duck-Subklassen durch *Schnittstellen* abgebildet (FlyBehavior und QuackBehavior). Dadurch ist die tatsächliche *Implementierung* (d. h. das spezifische konkrete in der Klasse programmierte Verhalten, das FlyBehavior und QuackBehavior implementiert) nicht mehr an die Duck-Subklasse gebunden.

Ab sofort lebt das Verhalten der Enten in einer eigenen Klasse, die ein bestimmtes Verhaltens-Interface implementiert.

So müssen die Duck-Klassen nichts über die Implementierungsdetails ihres eigenen Verhaltens wissen.



»Auf eine Schnittstelle programmieren« heißt eigentlich »auf einen Supertyp programmieren«

Abweichend vom englischen Original benutzen wir hier das deutsche Wort »Schnittstelle«, wenn es um das *Konzept* einer Schnittstelle geht. Daneben gibt es noch das Java-*Konstrukt* des *Interface*. Das heißt, Sie können auf eine Schnittstelle programmieren, ohne dafür ein Java-Interface zu verwenden. Hierbei geht es darum, das Konzept des Polymorphismus zu nutzen, indem wir auf einen Supertyp programmieren, wodurch das eigentliche Laufzeitobjekt nicht im Code »eingesperrt« ist. Wir könnten »auf einen Supertyp« also auch so formulieren: »Der deklarierte Typ der Variablen sollte ein Supertyp sein, üblicherweise eine abstrakte Klasse oder ein Interface, damit Objekte, die diesen Variablen zugewiesen wurden, eine beliebige konkrete Implementierung des Supertyps sein können. Das heißt, die deklarierende Klasse muss die tatsächlichen Objekttypen nicht mehr kennen!«

Aber das wissen Sie vermutlich schon. Um sicherzugehen, dass wir alle von der gleichen Sache sprechen, hier ein kleines Beispiel für die Verwendung polymorpher Typen. Stellen Sie sich eine abstrakte Klasse namens Animal (Tier) mit den zwei konkreten Implementierungen Dog (Hund) und Cat (Katze) vor. **Die Programmierung auf eine Implementierung** könnte dann so aussehen:

```
Dog d = new Dog();
d.bark();
```

Indem wir deklarieren, dass die Variable »d« den Typ Dog hat (eine konkrete Implementierung von Animal), sind wir gezwungen, auf eine konkrete Implementierung zu programmieren.

Hier ein Beispiel für die Programmierung auf eine Schnittstelle/einen Supertyp:

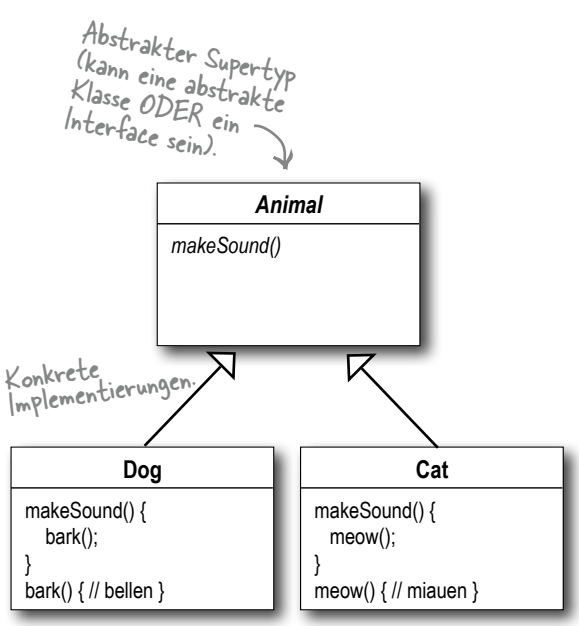
```
Animal animal = new Dog();
animal.makeSound();
```

Wir wissen, dass dies ein Dog ist. Jetzt können wir die Animal-Referenz jedoch polymorph einsetzen.

Noch besser: Anstatt die Instanziierung des Subtyps (wie new Dog()) hartzucodieren, **weisen wir das konkrete Implementierungsobjekt zur Laufzeit zu**.

```
a = getAnimal();
a.makeSound();
```

Wir wissen nicht, WAS der tatsächliche Subtyp von Animal ist. Uns ist nur wichtig, dass er weiß, wie er auf makeSound() reagieren muss.

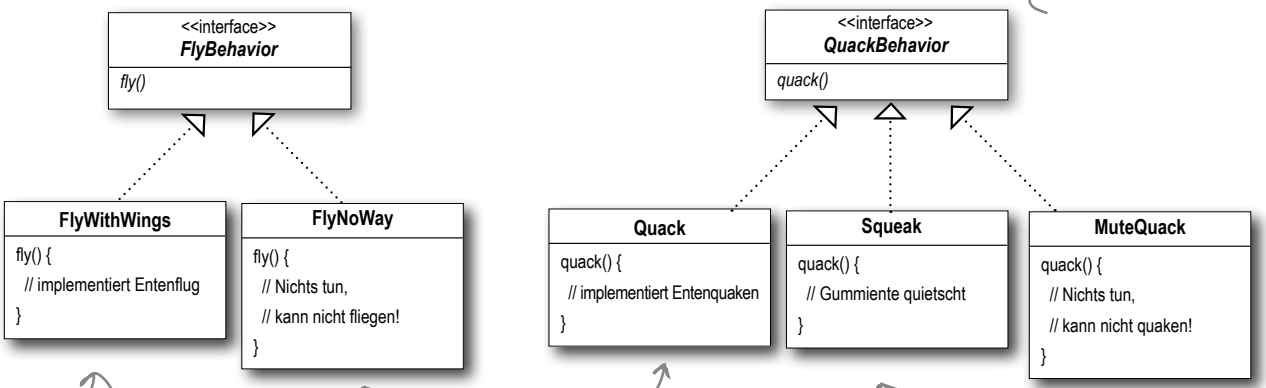


Das Entenverhalten implementieren

Unten sehen Sie die beiden Interfaces FlyBehavior (Flugverhalten) und QuackBehavior (Quakverhalten) sowie die dazugehörigen Klassen, die das jeweilige konkrete Verhalten implementieren.

Das Interface FlyBehavior wird von allen Fly-Klassen implementiert. Dadurch müssen alle neuen Klassen für Flugverhalten nur noch die fly()-Methode implementieren.

Das Gleiche machen wir hier für das Quakverhalten: Es gibt ein Interface, das nur eine quack()-Methode enthält, die implementiert werden muss.



Dies ist die Implementierung des Flugverhaltens für alle Enten mit Flügeln.

Und dies ist die Implementierung für alle Enten, die nicht fliegen können.

Tatsächlich quaken.

Quietschen statt quaken.

>>Stilles<< quaken.

Bei diesem Entwurf können andere Objekttypen unser Flug- und Quakverhalten wiederverwenden, weil die Verhaltensweisen nicht mehr in unseren Duck-Klassen versteckt sind!

Und wir können neues Verhalten hinzufügen, ohne die schon vorhandenen Verhaltensklassen verändern oder Duck-Klassen anfassen zu müssen, die Flugverhalten implementieren.

Wir können also die Vorteile der WIEDERVERWENDUNG nutzen, ohne den Ballast mitzuschleppen, den die Vererbung mitbringt.

Es gibt keine Dummen Fragen

F: Muss ich meine Applikation immer zuerst implementieren, um zu sehen, was sich ändert, und diese Dinge anschließend auslagern und verkapseln?

A: Nicht immer. Oft können Sie bereits beim Entwurf einer Applikation erkennen, welche Bereiche variabel sein werden, und können die dafür nötige Flexibilität schon in Ihren Code einbauen. Sie werden sehen, dass die Entwurfsprinzipien und -muster an jedem Punkt im Entwicklungszyklus verwendet werden können.

F: Sollte Duck nicht auch ein Interface sein?

A: In diesem Fall nicht. Ist alles miteinander verbunden, profitieren wir sogar davon, dass Duck kein Interface ist und spezifische Enten wie MallardDuck gemeinsame Eigenschaften und Methoden erben können. Nachdem wir die veränderlichen Dinge aus der Duck-Vererbungskette entfernt haben, können wir die Vorteile dieser Struktur problemlos nutzen.

F: Es erscheint seltsam, eine Klasse zu verwenden, die nur Verhalten enthält. Sollten Klassen nicht irgendetwas darstellen, also Zustand UND Verhalten besitzen?

A: In einem OO-System stellen Klassen tatsächlich Dinge dar, die Verhalten (Instanzvariablen) und Zustand besitzen. In diesem Fall ist es das Verhalten. Aber selbst Verhalten kann aus Zustand und Methoden bestehen: Das Flugverhalten könnte beispielsweise Instanzvariablen für die verschiedenen Attribute des Fliegens besitzen (Flügelschläge pro Minute, maximale Höhe, Geschwindigkeit usw.).

Spitzen Sie Ihren Bleistift

1. Was würden Sie bei unserem neuen Entwurf brauchen, um die SimUDuck-App um die Möglichkeit des Fliegens per Raketenantrieb zu erweitern?

2. Können Sie sich eine Klasse vorstellen, die das Quakverhalten nutzt, ohne eine Ente darzustellen?

Antworten:
1) Eine FlyRocketPowered-Klasse (Fliegen mit Raketenantrieb) erstellen, die das FlyBehavior-Interface implementiert.
2) Zum Beispiel eine Lockente (eine Art Pfeife, die Entengeräusche macht).

Das Entenverhalten integrieren

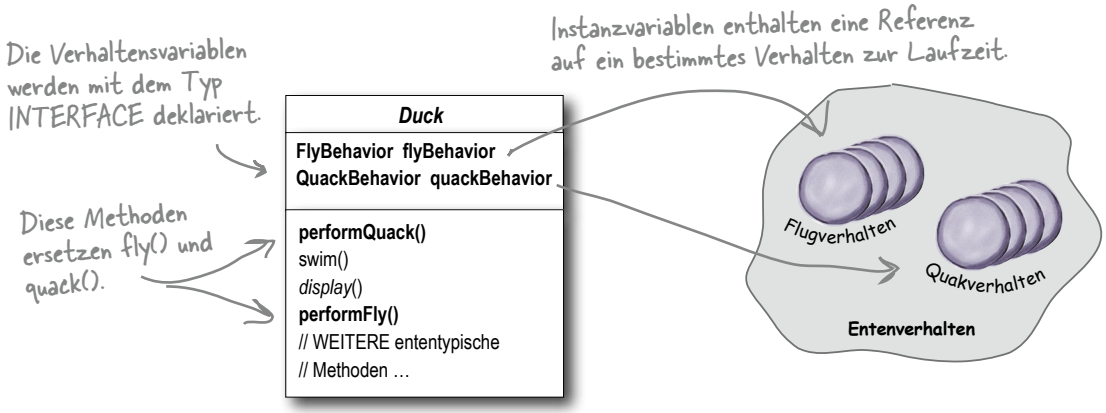
Der Schlüssel ist, dass eine Ente (ein Duck-Objekt) ihr Quak- und Flugverhalten jetzt delegiert, anstatt die in der Duck-Klasse definierten Quak- und Flugmethoden zu benutzen.

Und das geht so:

1. **Zuerst fügen wir zwei Instanzvariablen vom Typ FlyBehavior und QuackBehavior hinzu,** die wir entsprechend flyBehavior und quackBehavior nennen. Jedes konkrete Duck-Objekt weist diesen Variablen zur Laufzeit ein *spezifisches* Verhalten zu, zum Beispiel FlyWithWings (mit Flügeln fliegen) und Squeak (Quietschen) als spezifische Form des Quakens.

   Außerdem entfernen wir die Methoden fly() und quack() aus der Duck-Klasse (und allen Subklassen), weil wir das Verhalten in die Klassen FlyBehavior und QuackBehavior ausgelagert haben.

   Darüber hinaus ersetzen wir fly() und quack() in der Duck-Klasse durch ähnliche Methoden namens performFly() (Flug durchführen) und performQuack() (Quaken durchführen). Wie das geht, zeigen wir unten.



2. **Und jetzt implementieren wir performQuack():**

```
public abstract class Duck {
   QuackBehavior quackBehavior;
   // mehr

   public void performQuack() {
      quackBehavior.quack();
   }
}
```

Jede Ente enthält eine Referenz auf etwas, das das Interface QuackBehavior implementiert.

Anstatt das Quakverhalten selbst zu implementieren, delegiert das Duck-Objekt an das von quackBehavior referenzierte Objekt.

Gar nicht so schwer, oder? Um zu quaken, weist eine Ente (ein Duck-Objekt) das von quackBehavior referenzierte Objekt an, die entsprechende Aktion durchzuführen. In diesem Teil des Codes ist es uns egal, welche *Art von Objekt* die Ente tatsächlich ist. *Uns ist nur wichtig, dass sie etwas mit dem Befehl quack() anfangen kann.*

Mehr Integration ...

3 Und jetzt können wir uns darum kümmern, **wie** die Instanzvariablen flyBehavior und quackBehavior festgelegt werden. Sehen wir uns hierzu die Klasse MallardDuck an:

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("Ich bin eine echte Stockente.");
    }
}
```

Eine Stockende (ein MallardDuck-Objekt) benutzt die Klasse Quack, um zu quaken. Beim Aufruf von performQuack() wird die Verantwortung für das Quaken also an das Quack-Objekt delegiert, und wir bekommen ein echtes Quaken.

Als Typ für FlyBehavior verwendet sie FlyWithWings.

Wie gesagt, MallardDuck erbt die Instanzvariablen quackBehavior und flyBehavior von der Klasse Duck.

Das Quaken von MallardDuck ist ein echtes, ententypisches **Quaken**, kein **Quietschen** (squeak) und kein **stummes Quaken**. Bei der Instanziierung von MallardDuck benutzt ihr Konstruktor die geerbte Instanzvariable quackBehavior mit einer neuen Instanz des Typs Quack (eine konkrete Implementierungsklasse für QuackBehavior).

Das Gleiche gilt für das Flugverhalten der Ente: Der MallardDuck-Konstruktor initialisiert die geerbte Instanzvariable flyBehavior mit einer Instanz des Typs FlyWithWings (eine konkrete Implementierungsklasse für FlyBehavior).

Gut gesehen. Genau das machen wir *im Moment*.

Später im Buch wird unser Werkzeugkasten weitere Muster enthalten, mit denen wir auch das reparieren können.

Tatsächlich verwenden wir für das Verhalten zwar konkrete Klassen (indem wir eine Verhaltensklasse wie Quack oder FlyWithWings instanziieren und unserer Verhaltensreferenzvariablen zuweisen), das können wir zur Laufzeit aber *auf einfache Weise* ändern.

Insgesamt ist dieses Vorgehen also ziemlich flexibel. Allerdings ist diese flexible Instanziierung von Instanzvariablen alles andere als optimal. Die Instanzvariable quackBehavior hat den Typ Interface. Das heißt, wir könnten zur Laufzeit (durch die Magie des Polymorphismus) dynamisch eine andere QuackBehavior-Implementierungsklasse zuweisen.

Überlegen Sie, wie Sie eine Ente implementieren würden, deren Verhalten sich zur Laufzeit ändern kann. (Den Code hierfür finden Sie auf den folgenden Seiten.)

Den Entencode testen

1 Geben Sie die unten stehende Klasse Duck (Duck.java) und die vor zwei Seiten gezeigte Klasse MallardDuck (MallardDuck.java) ein und kompilieren Sie sie.

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   public Duck() { }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

   public void swim() {
      System.out.println("Alle Enten schwimmen, selbst Lockenten!");
   }
}
```

Zwei Referenzvariablen für die Typen der Verhaltens-Interfaces. Diese werden von allen Enten-Subklassen (im gleichen Package) geerbt.

Die Verhaltensklasse delegieren.

2 Geben Sie das unten stehende Interface FlyBehavior (FlyBehavior.java) und die beiden Klassen zur Implementierung des Verhaltens (FlyWithWings.java und FlyNoWay.java) ein und kompilieren Sie sie.

```
public interface FlyBehavior {
   public void fly();
}
```

Das Interface, das alle Klassen für das Flugverhalten implementieren.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("Ich fliege!!");
   }
}
```

Implementierung des Flugverhaltens für Enten, die fliegen.

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("Ich kann nicht fliegen");
   }
}
```

Implementierung des Flugverhaltens für Enten, die NICHT fliegen (wie Gummienten und Lockenten).

Den Entencode testen (Fortsetzung)

3 **Geben Sie das unten stehende Interface QuackBehavior (QuackBehavior.java) und die drei Klassen zur Implementierung des Verhaltens (Quack.java, MuteQuack.java und Squeak.java) ein und kompilieren Sie sie.**

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quak");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Stille >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Quietsch");
   }
}
```

4 **Geben Sie die Testklasse (MiniDuckSimulator.java) ein und kompilieren Sie sie.**

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

Hiermit wird die durch MallardDuck geerbte Methode performQuack() aufgerufen, die dann an das Quakverhalten des Objekts delegiert (d. h., sie ruft quack() an der geerbten quackBehavior-Referenz der Ente auf).

Das Gleiche machen wir mit der durch MallardDuck geerbten performFly()-Methode.

5 **Führen Sie den Code aus!**

```
Datei Bearbeiten Fenster Hilfe Blablablablabla
%java MiniDuckSimulator
Quack
Ich fliege!!
```

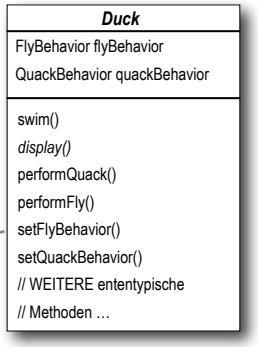
Verhalten dynamisch festlegen

Es wäre eine Schande, wenn wir all dieses dynamische Talent unserer Enten nicht auch nutzen würden! Stellen Sie sich vor, Sie wollten den Verhaltenstyp der Ente mit einer Setter-Methode auf der Duck-Klasse festlegen, anstatt sie mit dem Konstruktor der Ente zu instanziieren.

1 Die Duck-Klasse um zwei neue Enten erweitern.

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Wir können diese Methoden jederzeit aufrufen, um das Verhalten einer Ente flugs zu ändern.

Anmerkung der Lektorin: Unnötiges Wortspiel. Raus damit.

2 Einen neuen Duck-Typ erstellen (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("Ich bin eine Modellente");
    }
}
```

Unsere Modellente beginnt ihr Leben auf dem Boden ... ohne eine Möglichkeit zu fliegen.

3 Einen neuen FlyBehavior-Typ erstellen (FlyRocketPowered.java).

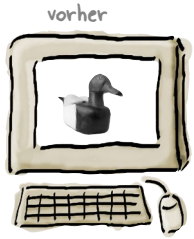
Das ist in Ordnung, wir erstellen ein raketengetriebenes Flugverhalten.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("Ich fliege mit Raketenantrieb!");
    }
}
```

4 **Passen Sie die Testklasse (MiniDuckSimulator.java) an, erweitern Sie sie um ModelDuck (Modellente) und versehen Sie die neue Ente mit einem Raketenantrieb.**

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```



Der erste Aufruf von performFly() delegiert an das im Konstruktor von ModelDuck definierte flyBehavior-Objekt, das eine Instanz von FlyNoWay ist.

Das ruft die geerbte Verhaltens-Setter-Methode des Modells auf und ... voilà! Plötzlich hat das Modell einen Raketenantrieb!

Wenn das funktioniert, hat die Modellente ihr Flugverhalten dynamisch verändert! DAS können Sie nicht tun, wenn sich die Implementierung in der Duck-Klasse befindet.

5 **Führen Sie den Code aus!**

Datei Bearbeiten Fenster Hilfe Abrakadabra

```
%java MiniDuckSimulator

Quak

Ich fliege!!

Ich kann nicht fliegen

Ich fliege mit
Raketenantrieb!
```

Um das Verhalten der Ente zur Laufzeit zu ändern, müssen Sie nur die Setter-Methoden der Ente für dieses Verhalten aufrufen.

Das große Ganze: Verkapseltes Verhalten

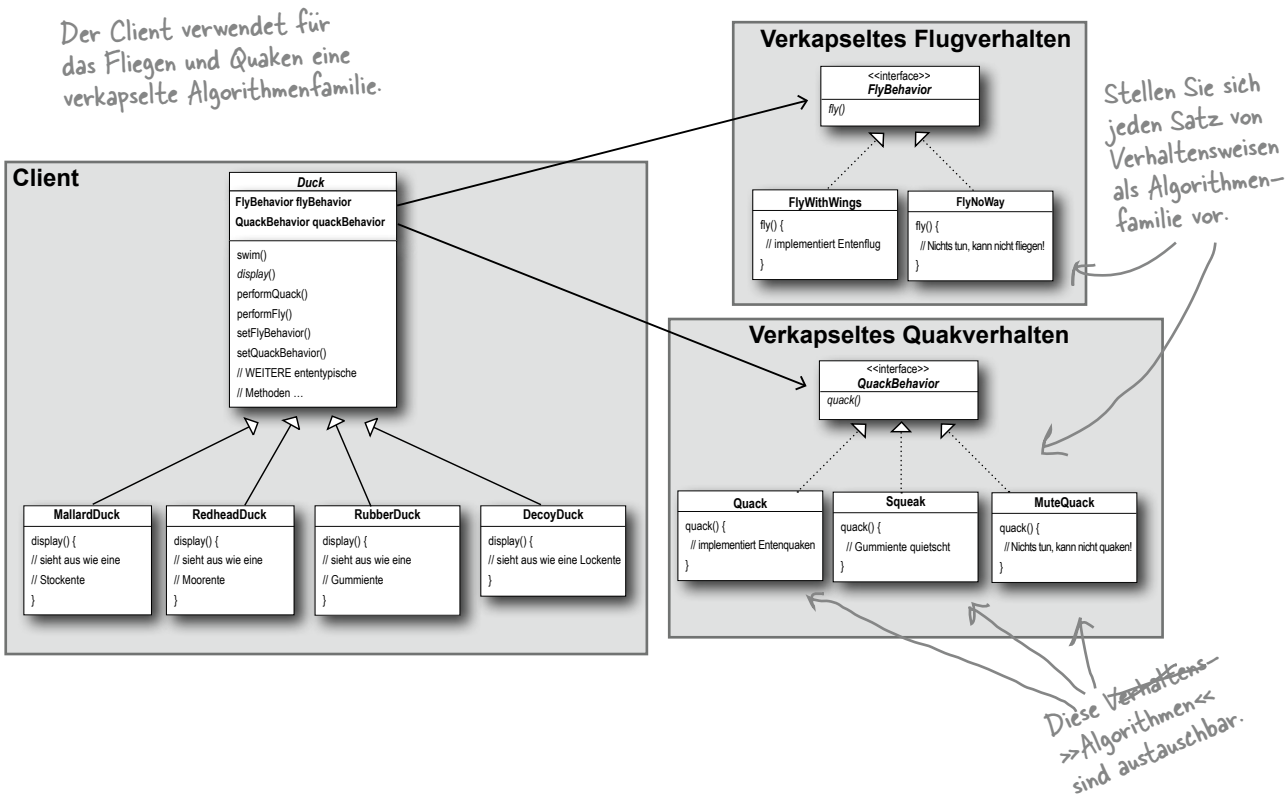
Nachdem wir uns detailliert mit dem Entwurf des Entensimulators beschäftigt haben, ist es nun Zeit, einen Blick auf das große Ganze zu werfen.

Unten sehen Sie die komplette überarbeitete Klassenstruktur. Es ist alles dabei, was Sie wahrscheinlich erwarten: Enten, die Duck erweitern, Flugverhalten, das FlyBehavior erweitert, und Quakverhalten, das QuackBehavior erweitert.

Außerdem fällt auf, dass wir die Dinge jetzt etwas anders beschreiben. Anstatt sich das Entenverhalten als eine Reihe von *Verhaltensweisen* vorzustellen, betrachten wir sie jetzt als eine *Algorithmenfamilie*. Überlegen Sie: Im Entwurf von SimUDuck stehen die Algorithmen für Dinge, die eine Ente tun würde (verschiedene Arten zu fliegen und zu quaken). Wir könnten diese Techniken aber genauso einfach für eine Reihe von Klassen einsetzen, die beispielsweise die Mehrwertsteuerberechnung für verschiedene EU-Staaten implementieren.

Achten Sie besonders genau auf die *Beziehungen* zwischen den Klassen. Am besten nehmen Sie einen Stift zur Hand und schreiben an die Pfeile im Diagramm, welche Art der Beziehung jeweils vorliegt (IST-EIN, HAT-EIN und IMPLEMENTIERT).

Machen Sie das auf jeden Fall!



HAT-EIN ist besser als IST-EIN

Die HAT-EIN-Beziehung ist recht interessant: Jede Ente hat ein FlyBehavior und ein QuackBehavior, an das sie ihr Flug- und Quakverhalten delegiert.

Wenn Sie die Klassen auf diese Weise kombinieren, verwenden Sie *Komposition*. Anstatt es zu *erben*, bekommen die Enten ihr Verhalten durch Komposition mit dem passenden Verhaltensobjekt.

Dies ist eine wichtige Technik. Tatsächlich ist sie sogar das Fundament für unser drittes Entwurfsprinzip:

Entwurfsprinzip

Ziehen Sie Komposition der Vererbung vor.

Wie Sie gesehen haben, sorgt der Einsatz von Komposition für den Aufbau eines Systems für deutlich mehr Flexibilität. Sie können nicht nur eine Algorithmenfamilie in einen eigenen Satz an Klassen verkapseln, sondern auch *das Verhalten zur Laufzeit ändern*, solange das Objekt, mit dem Sie komponieren, die korrekte Verhaltensschnittstelle implementiert.

Komposition kommt in vielen Entwurfsmustern zum Einsatz. Im Verlauf dieses Buchs werden Sie noch jede Menge weitere Beispiele zu den Vor- und Nachteilen sehen.

KOPF-NUSS

Eine Lockente ist eine Art Pfeife, mit der Jäger das Quaken der Ente nachahmen. Wie würden Sie eine Lockente implementieren, die *nicht* von der Duck-Klasse erbt?

Meisterin und Schüler

Meisterin: *Erzähle mir, was du über den Weg der Objektorientierung gelernt hast.*

Schüler: *Meisterin, ich habe gelernt, dass das Versprechen des Wegs der Objektorientierung in der Wiederverwendung liegt.*

Meisterin: *Sprich weiter, Grashüpfer ...*

Schüler: *Meisterin, durch Vererbung können alle guten Dinge wiederverwendet werden. So können wir die Entwicklungszeit so drastisch reduzieren, als schnitten wir den Bambus in den Wäldern.*

Meisterin: *Wann wird mehr Zeit mit dem Code verbracht? Vor oder nach dem Ende der Entwicklung?*

Schüler: *Die Antwort ist »danach«, Meisterin. Wir verbringen immer mehr Zeit mit der Pflege des Codes als mit seiner Entwicklung.*

Meisterin: *Ist die Wiederverwendbarkeit also wichtiger als Wartbarkeit und Erweiterbarkeit?*

Schüler: *Meisterin, ich glaube, in Euren Worten liegt viel Wahrheit.*

Meisterin: *Ich sehe, du hast noch viel zu lernen, junger Grashüpfer. Gehe und meditiere weiter über die Vererbung. Wie du siehst, hat auch Vererbung ihre Probleme, und es gibt mehr als einen Weg, die Wiederverwendbarkeit zu erlangen.*

Da wir gerade von Entwurfsmustern sprechen ...

Herzlichen Glückwunsch zu Ihrem ersten Muster!

Sie haben soeben Ihr erstes Entwurfsmuster verwendet, um die SimUDuck-App zu überarbeiten: das STRATEGY-Muster.

Dank dieses Musters ist der Simulator auf alle Änderungen vorbereitet, die sich die Chefs auf ihrer nächsten Geschäftsreise nach Maui ausdenken.

Damit Sie das Muster wirklich lernen, sind wir mit Ihnen den langen, steinigen Weg gegangen. Zur Belohnung kommt hier die formale Definition dieses Musters:

> **Das Strategy-Muster** definiert eine Familie an Algorithmen, verkapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen, variieren zu lassen.

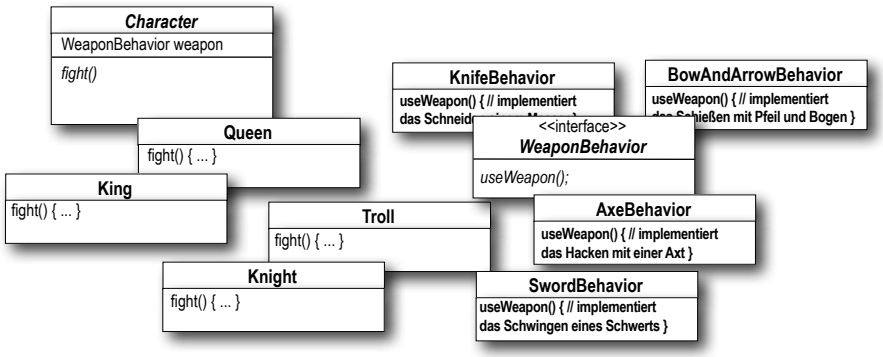
Nehmen Sie DIESE Definition, um Freunde zu beeindrucken und Vorgesetzte zu beeinflussen.

Entwurfspuzzle

Die unten gezeigten Klassen und Interfaces für ein Action-Abenteuerspiel sind durcheinandergeraten. Es gibt Klassen für Spielcharaktere und für Waffenverhalten, das die Charaktere im Spiel einsetzen können. Jeder Charakter kann zeitgleich immer nur eine Waffe benutzen, aber die Waffen jederzeit während des Spiels wechseln. Ihre Aufgabe ist es, alles zu sortieren ...
(Die Antworten finden Sie am Ende des Kapitels.)

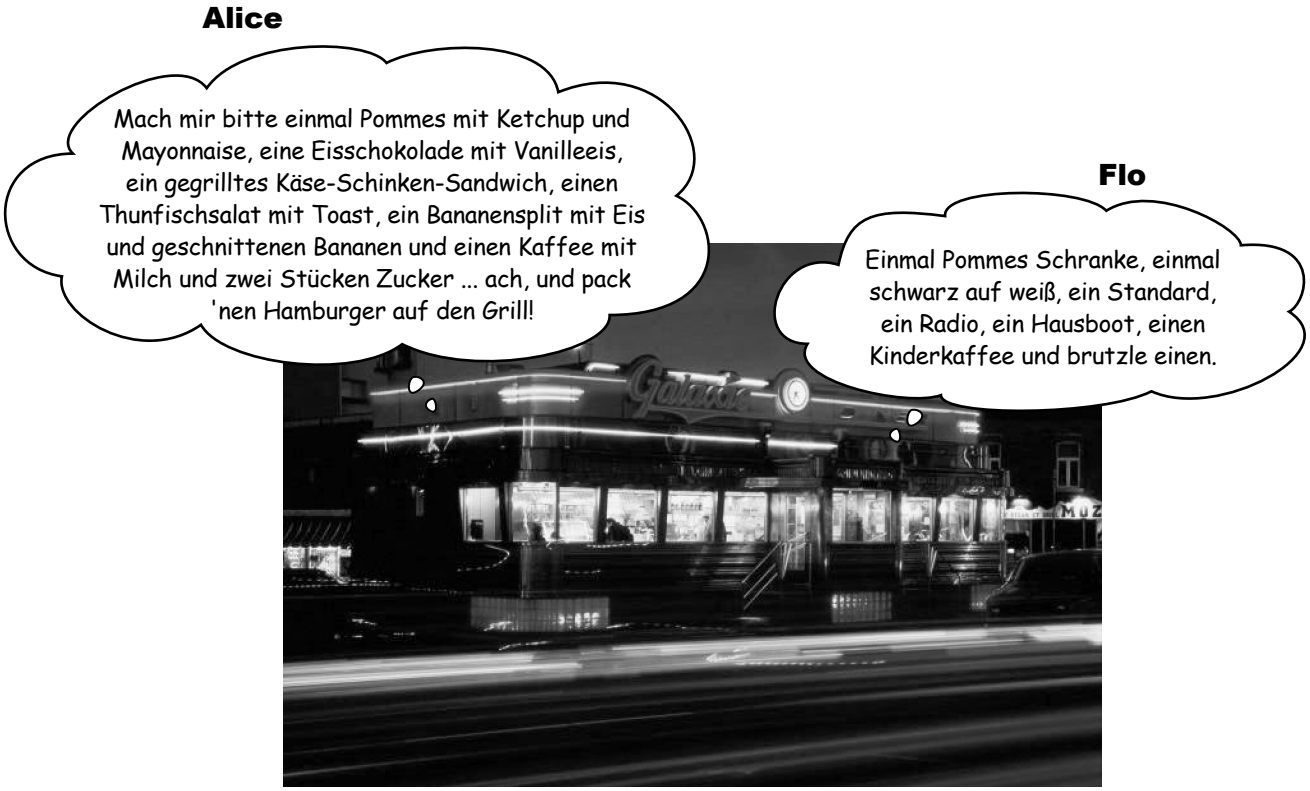
Ihre Aufgabe:

1. Ordnen Sie die Klassen.
2. Identifizieren Sie eine abstrakte Klasse, ein Interface und die acht Klassen.
3. Verbinden Sie die Klassen mithilfe von Pfeilen.
   a. Zeichnen Sie diesen Pfeil für Vererbung (»erweitert«).
   b. Zeichnen Sie diesen Pfeil für ein Interface (»implementiert«).
   c. Zeichnen Sie diesen Pfeil für eine HAT-EINE-Beziehung.
4. Fügen Sie die Methode setWeapon() (Waffe wählen) in die richtige Klasse ein.




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Im Bistro um die Ecke aufgeschnappt ...



Was ist der Unterschied zwischen diesen beiden Bestellungen? Es gibt keinen! Beides ist die gleiche Bestellung, nur dass Alice doppelt so viele Wörter braucht und die Geduld eines schlecht gelaunten Fast-Food-Kochs auf die Probe stellt.

Was hat Flo, das Alice fehlt? Ein gemeinsames Vokabular mit dem Koch. Das erleichtert nicht nur die Kommunikation, sondern der Koch muss sich auch weniger Dinge merken, weil er die Muster für die Speisen schon im Kopf hat.

Durch Entwurfsmuster steht Ihnen ein gemeinsames Vokabular mit anderen Entwicklern zur Verfügung. Sobald Sie die Vokabeln gelernt haben, können Sie viel leichter kommunizieren und Entwickler, die die Muster noch nicht kennen, dazu ermuntern, sie ebenfalls zu lernen. Außerdem erweitern Sie Ihr architektonisches Denken, indem Sie auf *Muster*ebene statt auf der kleinkarierten *Objekt*ebene denken.

Im Büro nebenan aufgeschnappt ...

Meine Broadcast-Klasse erfasst alle Objekte, die abonniert haben. Sobald neue Daten eintreffen, schickt sie eine Nachricht an die Abonnenten. Das Tolle daran ist, dass die Abonnenten sich jederzeit registrieren und sogar selbst wieder entfernen können. Das ist so richtig dynamisch und locker gebunden.

KOPF-NUSS

Fallen Ihnen noch weitere gemeinsame Vokabulare ein, die außerhalb von OO-Design und Bistrobestellungen benutzt werden? (Tipp: Wie wäre es mit Automechanikern, Tischlern, Gourmetköchen oder Fluglotsen?) Welche Qualitäten werden zusammen mit der Fachsprache weitergegeben?

Überlegen Sie, welche Aspekte der OO-Programmierung zusammen mit den Musternamen vermittelt werden. Welche Qualitäten werden mit dem Namen »Strategy-Muster« kommuniziert?

Rick, warum sagst du nicht einfach, dass du das Observer-Muster nutzt?

Gute Idee. Wenn du in Mustern kommunizierst, wissen andere Entwickler sofort und genau, worum es geht. Pass nur auf, dass dich das Mustervirus nicht erwischt. Du merkst es daran, dass du Entwurfsmuster für ein »Hallo Welt«-Programm einsetzt.

Die Stärke eines gemeinsamen Mustervokabulars

Wenn Sie anhand von Mustern kommunizieren, benutzen Sie nicht nur eine gemeinsame FACHSPRACHE.

Gemeinsame Mustervokabulare sind MÄCHTIG. Wenn Sie mit einem Entwickler in Ihrem Team anhand von Mustern kommunizieren, geben Sie nicht nur den Musternnamen weiter, sondern auch eine Reihe von Qualitäten, Eigenschaften und Beschränkungen, für die dieses Muster steht.

>>Um die verschiedenen Verhaltensweisen unserer Enten zu implementieren, verwenden wir das Strategy-Muster.<< Damit wissen Sie, dass die Entenverhaltensweisen in ihren eigenen Klassen verkapselt wurden, die einfach erweitert und ausgetauscht werden können, bei Bedarf sogar zur Laufzeit.

Mit Mustern können Sie mit weniger mehr sagen. Wenn Sie in einer Beschreibung ein Muster verwenden, wissen andere Entwickler genau, welchen Entwurf Sie im Kopf haben.

Indem Sie auf Musterebene miteinander sprechen, können Sie länger »beim Entwurf« bleiben. Die Kommunikation über Softwaresysteme anhand von Mustern erlaubt Ihnen, die Diskussion auf Designebene zu führen, ohne in die Details der Implementierung von Objekten und Klassen eintauchen zu müssen.

Auf wie vielen Design-Meetings waren Sie schon, die sich schnell in Implementierungsdetails verloren haben?

Gemeinsame Vokabulare können Ihrem Entwicklungsteam Dampf machen. Ein Team, das gut aufeinander eingespielt mit Entwurfsmustern arbeitet, kommt schnell und ohne große Gefahr von Missverständnissen voran.

Wenn Ihr Team anfängt, Entwurfsideen anhand von Mustern miteinander zu teilen, bauen Sie eine Gemeinschaft von Musterbenutzern auf.

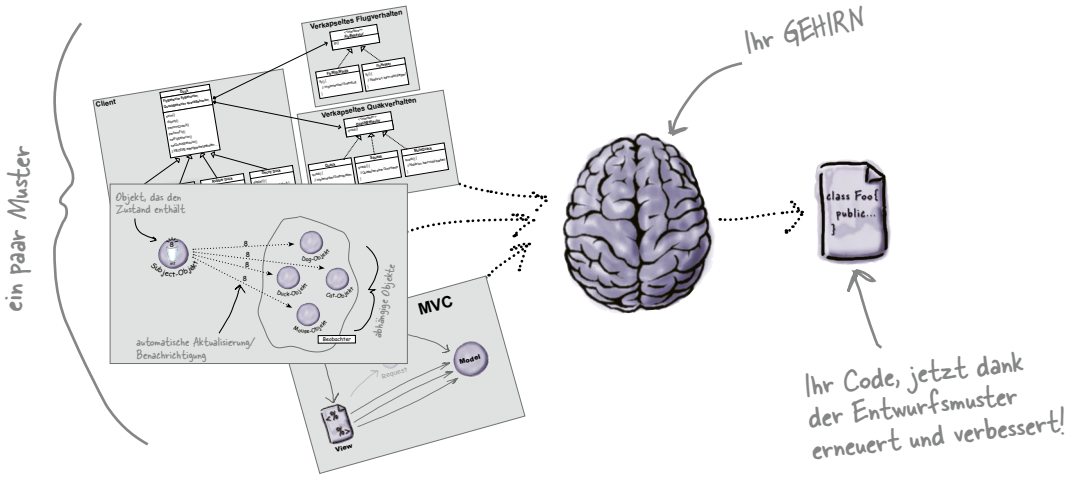
Gemeinsame Vokabulare motivieren mehr Nachwuchsentwickler darin, sich weiterzubilden. Nachwuchsentwickler bewundern die erfahrenen Entwickler. Wenn Seniorentwickler Entwurfsmuster einsetzen, werden jüngere Entwickler ebenfalls dazu motiviert, sie zu lernen. Schaffen Sie eine Community von Musterbenutzern in Ihrem Unternehmen.

Prüfen Sie, ob Sie in Ihrem Unternehmen eine Entwurfsmuster-Lerngruppe einrichten können. Vielleicht werden Sie sogar fürs Lernen bezahlt ...

Wie setze ich Entwurfsmuster ein?

Wir haben alle schon vorgefertigte Bibliotheken und Frameworks benutzt. Wir nehmen sie uns, schreiben etwas Code gegen ihre APIs, kompilieren sie in unsere Programme und profitieren von einer Menge Code, den andere geschrieben haben. Denken Sie an die Java-APIs und die Menge an Funktionalität, die sie Ihnen bieten: Netzwerk, GUI, IO und so weiter. Bibliotheken und Frameworks kommen einem Entwicklungsmodell nahe, bei dem wir uns die nötigen Bausteine zusammensuchen und direkt in unser Programm einstöpseln können. Aber ... sie helfen uns nicht dabei, unsere eigenen Applikationen so zu strukturieren, dass sie leichter verständlich, wartbarer und flexibler werden. Und da kommen die Entwurfsmuster ins Spiel.

Entwurfsmuster landen nicht direkt in Ihrem Code, sondern zuerst in Ihrem GEHIRN. Sobald Ihr Gehirn mit ausreichend Musterverständnis aufgeladen ist, können Sie damit beginnen, sie in Ihren neuen Entwürfen einzusetzen und Ihren alten Code zu überarbeiten, sofern Sie merken, dass er sich immer mehr in nicht mehr wartbaren Spaghetticode verwandelt.



Es gibt keine Dummen Fragen

F: Wenn Entwurfsmuster so toll sind, warum kann dann nicht einfach jemand eine Bibliothek daraus machen, damit ich das nicht mehr tun muss?

A: Entwurfsmuster leben auf einer höheren Abstraktionsebene als Bibliotheken. Entwurfsmuster helfen uns bei der Strukturierung von Klassen und Objekten, um bestimmte Probleme zu lösen. Wir müssen diese Entwürfe so anpassen, dass sie zu unserer jeweiligen Anwendung passen.

F: Sind Bibliotheken und Frameworks denn nicht auch Entwurfsmuster?

A: Frameworks und Bibliotheken sind keine Entwurfsmuster. Sie stellen spezifische Implementierungen bereit, die wir in unseren Code einbinden. Manchmal werden für diese Implementierungen jedoch Entwurfsmuster eingesetzt. Das ist klasse, denn wenn Sie das Konzept der Entwurfsmuster einmal verstanden haben, verstehen Sie auch die um die Muster herumstrukturierten APIs deutlich besser.

F: Gibt es also keine Bibliotheken mit Entwurfsmustern?

A: Nein. Später werden Sie aber Musterkataloge kennenlernen, die Muster enthalten, die Sie auf Ihre Applikationen anwenden können.

Skeptische Entwicklerin

Freundliche Muster-Meisterin

Entwicklerin: Okay, aber ist das nicht einfach gutes objektorientiertes Design? Muss ich mir wirklich über Entwurfsmuster Gedanken machen, wenn ich mich mit Abstraktion, Vererbung und Polymorphismus auskenne? Ist das nicht ganz einfach? Habe ich nicht genau deshalb die ganzen OO-Kurse belegt? Ich dachte, Entwurfsmuster sind etwas für Leute, die sich mit gutem OO-Design nicht auskennen.

Meisterin: Oh, nicht doch. Eines der größten Missverständnisse der objektorientierten Entwicklung ist, dass schon OO-Grundkenntnisse ausreichen, um flexible, wiederverwendbare und wartbare Systeme zu erstellen.

Entwicklerin: Ja stimmt das denn nicht?

Meisterin: Nein. Die Entwicklung von OO-Systemen mit diesen Eigenschaften ist nicht immer offensichtlich und wurde nur durch harte Arbeit überhaupt entdeckt.

Entwicklerin: Ich glaube, so langsam verstehe ich. Diese manchmal nicht offensichtlichen Wege, OO-Systeme zu konstruieren, wurden gesammelt ...

Meisterin: ... genau, in einer Reihe von Mustern, die wir als Entwurfsmuster bezeichnen.

Entwicklerin: Das heißt, wenn ich Muster kenne, kann ich mir die harte Arbeit sparen und direkt mit Entwürfen loslegen, die immer funktionieren?

Meisterin: Gewissermaßen. Aber vergiss nicht: Entwürfe sind eine Kunst für sich. Es wird immer Vor- und Nachteile geben. Aber, wenn du gut durchdachten und erprobten Entwurfsmustern folgst, bist du immer einen Schritt voraus.

Entwicklerin: Aber was kann ich tun, wenn ich kein passendes Muster finde?

Vergiss nicht: Das Wissen um Konzepte wie Abstraktion, Vererbung und Polymorphismus macht dich noch nicht zu einer guten OO-Designerin. Eine Design-Meisterin überlegt, wie sie flexible Entwürfe erschaffen kann, die wartbar sind und mit Veränderungen umgehen können.

Meisterin: Die Muster basieren auf bestimmten objektorientierten Prinzipien. Die Kenntnis dieser Prinzipien wird dir helfen, wenn du kein Muster findest, das zu deinem Problem passt.

Entwicklerin: Prinzipien? Sie meinen noch jenseits von Abstraktion, Verkapselung und ...?

Meisterin: Ja. Eines der Geheimnisse bei der Erstellung wartbarer OO-Systeme besteht darin, sich zu überlegen, wie sie sich in Zukunft verändern könnten, und den Prinzipien geht es genau um diese Fragen.

Werkzeuge für Ihren Entwurfs-Werkzeugkasten

Das erste Kapitel liegt fast hinter Ihnen, und ein paar Werkzeuge befinden sich schon in Ihrem OO-Werkzeugkasten. Bevor es mit Kapitel 2 weitergeht, sehen wir uns die Liste einmal an.

OO-Grundlagen

Abstraktion

Verkapselung

Polymorphismus

Vererbung

Wir gehen davon aus, dass Sie OO-Grundlagen wie Abstraktion, Verkapselung, Polymorphismus und Vererbung kennen. Wenn Ihr Wissen etwas eingerostet ist, nehmen Sie Ihr Lieblings-OO-Buch zur Hand und lesen noch einmal nach. Danach überfliegen Sie dieses Kapitel erneut.

OO-Prinzipien

Verkapseln, was variabel ist.
Komposition vor Vererbung.
Auf Schnittstellen programmieren, nicht auf Implementierungen.

Das sehen wir uns später noch einmal genau an und fügen unserer Liste weitere Prinzipien hinzu.

OO-Muster

Strategy – definiert eine Familie von Algorithmen, verkapselt sie und macht sie austauschbar. Durch das Strategy-Muster können die Algorithmen unabhängig von den sie einsetzenden Clients variieren.

Denken Sie im Verlauf des Buchs daran, wie Entwurfsmuster auf OO-Grundlagen und -Prinzipien aufbauen.

Eins geschafft, viele noch vor uns!

Punkt für Punkt

- Durch Kenntnis der OO-Grundlagen werden Sie nicht automatisch zu einem guten OO-Designer.
- Gute OO-Entwürfe sind wiederverwendbar, erweiterbar und wartbar.
- Muster zeigen Ihnen, wie man Systeme mit guten OO-Designmerkmalen erstellt.
- Muster sind bewährte objektorientierte Erfahrungen.
- Entwurfsmuster sind kein Code, sondern allgemeine Lösungen für Designprobleme, die für Ihre jeweilige Anwendung angepasst werden.
- Muster werden nicht *erfunden*, sondern *entdeckt*.
- Die meisten Muster und Prinzipien befassen sich mit *Veränderungen* in der Software.
- Bei den meisten Mustern können verschiedene Systemteile unabhängig voneinander verändert werden.
- Oft versuchen wir, die veränderlichen Dinge aus einem System auszulagern und zu verkapseln.
- Durch Muster erhalten Sie eine gemeinsame Sprache, die den Wert Ihrer Kommunikation mit anderen Entwicklern maximieren kann.