O'REILLY®

Aktuell zu Java 8
2. Auflage

Entwurfsmuster

von Kopf bis Fuß

Vermeiden Sie Bindungsprobleme

Entdecken Sie die Geheimnisse des Pattern-Meisters

Finden Sie heraus, wie Sternback-Kaffee mit dem Decorator-Muster seinen Börsenwert verdoppeln konnte

Erfahren Sie, warum sich Ihre Freunde bezüglich des Factory-Musters vermutlich irren

Laden Sie sich die entscheidenden Muster direkt ins Hirn

Lesen Sie, wie sich Jörgs Liebesleben verbessert hat, seitdem er das Vererben einschränkt

Eric Freeman & Elisabeth Robson mit Kathy Sierra & Bert Bates

Übersetzung von Lars Schulten & Elke Buchholz

Der Inhalt (in der Übersicht)

Der 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?

Willkommen bei den Entwurfsmustern

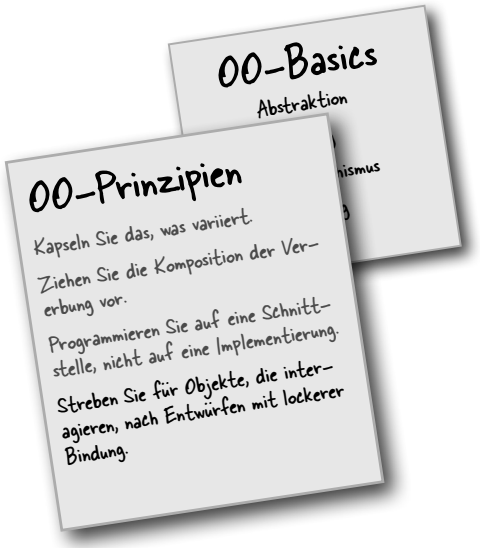
1 Willkommen bei den Entwurfsmustern

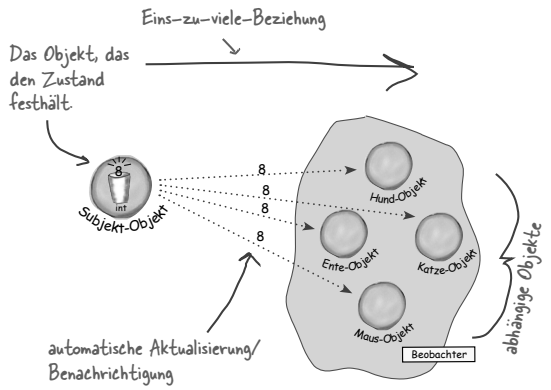
Irgendjemand hat Ihre Probleme bereits gelöst. In diesem Kapitel lernen Sie, warum (und wie) Sie die Erfahrungen und Lektionen verwerten können, die andere Entwickler gelernt haben, die in den gleichen Entwurfsschwierigkeiten steckten und den Trip überlebt haben. Dazu werden wir einen Blick auf die Verwendung und die Vorteile von Entwurfsmustern werfen, uns einige grundlegende OO-Entwurfsprinzipien ansehen und ein Beispiel dafür durchgehen, wie ein bestimmtes Muster funktioniert. Am besten arbeiten Sie mit Mustern, indem Sie *Ihr Gehirn mit ihnen aufladen* und dann in Ihren Entwürfen und in bestehenden Anwendungen die *Punkte erkennen*, an denen Sie sie *anwenden können*. Anstelle von *Code*-Wiederverwendung bieten Ihnen Muster *Erfahrungs*-Wiederverwendung.

Das Observer-Muster

2 Ihre Objekte auf dem Laufenden halten

Verpassen Sie es nicht, wenn etwas Interessantes passiert! Wir haben ein Muster, das Ihre Objekte auf dem Laufenden hält, wenn etwas passiert, das Sie interessieren könnte. Objekte können sogar zur Laufzeit entscheiden, ob sie informiert werden möchten. Das Observer-Muster ist eins der Muster, die im JDK am häufigsten verwendet werden. Und es ist unglaublich nützlich. In diesem Kapitel sehen wir uns außerdem Eins-zu-viele-Beziehungen und lockere Bindungen an. Mit dem Observer-Muster werden Sie zum Mittelpunkt der Muster-Party.





Das Decorator-Muster

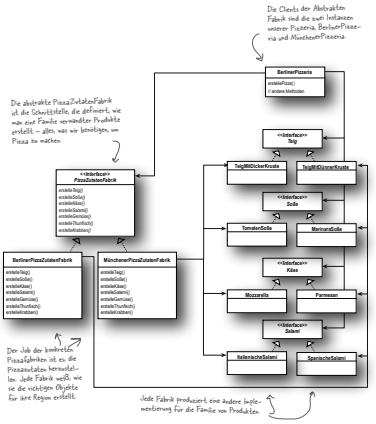
3 Objekte dekorieren

Nennen wir dieses Kapitel einfach »Vererbst du noch oder designst du schon?«. Wir untersuchen noch einmal einen typischen Fall überstrapazierter Vererbung, und Sie werden lernen, wie Sie Ihre Klassen mithilfe einer Form der Objekt-Zusammensetzung erst zur Laufzeit »dekorieren«. Warum? Wenn Ihnen die Techniken des Dekorierens einmal vertraut sind, können Sie Ihren Objekten (oder den Objekten anderer) neue Aufgaben geben, ohne den Code der zugrunde liegenden Klasse ändern zu müssen.

Das Factory-Muster

4 Backen in OO-Qualität

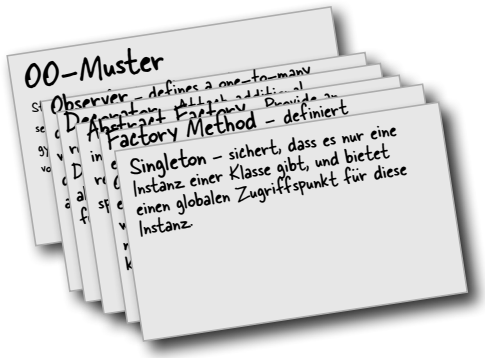
Machen Sie sich bereit, ein paar locker gebundene OO-Entwürfe zu backen. Das Erstellen von Objekten hat mehr zu bieten als die simple Verwendung des new-Operators. Sie werden lernen, dass Instantiierung eine Aktivität ist, die nicht immer in der Öffentlichkeit verübt werden sollte und oft zu Bindungsproblemen führen kann. Und das wollen Sie doch nicht, oder? Lernen Sie, wie Sie das Factory-Muster vor lästigen Abhängigkeiten retten kann.



Das Singleton-Muster

5 Ein einzigartiges Objekt

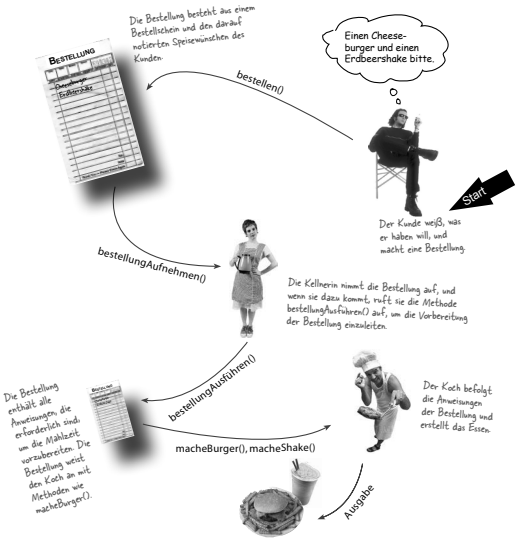
Unser nächster Halt ist das Singleton-Muster, unsere Fahrkarte zur Erstellung einzigartiger Objekte von Klassen, von denen es nur eine einzige Instanz geben kann. Vielleicht freut es Sie zu erfahren, dass das Singleton-Muster in Bezug auf das Klassendiagramm das einfachste aller Muster ist. Das Diagramm enthält tatsächlich nur eine einzige Klasse! Aber machen Sie es sich nicht zu bequem. Trotz der Einfachheit in Bezug auf das Klassendiagramm werden wir auf eine Reihe Buckel und Schlaglöcher in seiner Implementierung stoßen. Sie schnallen sich also besser an.



Das Command-Muster

6 Aufrufe einkapseln

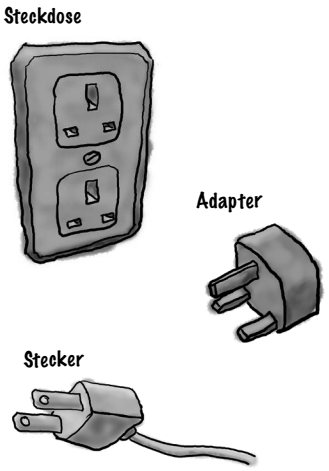
In diesem Kapitel heben wir die Kapselung noch einmal auf ein ganz neues Niveau: Wir werden Methodenaufrufe einkapseln. Ja, wirklich. Indem wir den Methodenaufruf kapseln, können wir Teile von Berechnungen einfrieren, damit das Objekt, das die Berechnung aufruft, sich nicht darum kümmern muss, wie diese Dinge gemacht werden. Es verwendet einfach unsere eingefrorene Methode, um sie ausführen zu lassen. Mit diesen eingekapselten Methodenaufrufen können wir außerdem einige unverschämt geschickte Dinge tun, sie beispielsweise speichern, um sie zu protokollieren, oder wiederverwenden, um unserem Code eine Rückgängig-Funktionalität zu spendieren.

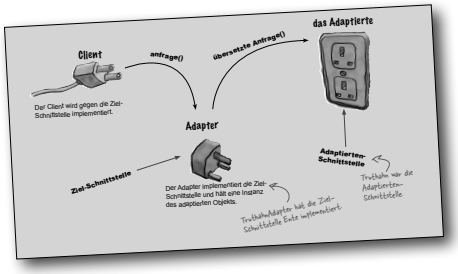


Die Adapter- und Facade-Muster

7 Anpassungsfähigkeit beweisen

In diesem Kapitel werden wir uns an unmöglichen Dingen versuchen – einen rechteckigen Pflock in ein rundes Loch zu stecken beispielsweise. Klingt unmöglich? Nicht, wenn man Design-Patterns hat. Erinnern Sie sich an das Decorator-Muster? Wir haben Objekte umhüllt, um ihnen neue Verantwortlichkeiten zu geben. Jetzt werden wir einige Objekte mit einem anderen Ziel einpacken: um ihren Schnittstellen den Anschein zu verleihen, dass sie wie etwas aussehen, das sie nicht sind. Warum sollten wir das tun? Wir haben damit die Möglichkeit, ein Design, das eine bestimmte Schnittstelle erwartet, an eine Klasse anzupassen, die eine andere Schnittstelle implementiert. Und das ist nicht alles. Da wir gerade dabei sind, werden wir uns noch ein weiteres Muster ansehen, das Objekte umhüllt, um ihre Schnittstelle zu vereinfachen.

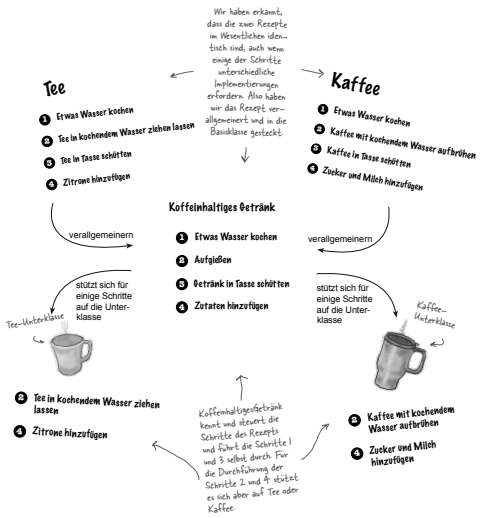




Das Template Method-Muster

8 Algorithmen einkapseln

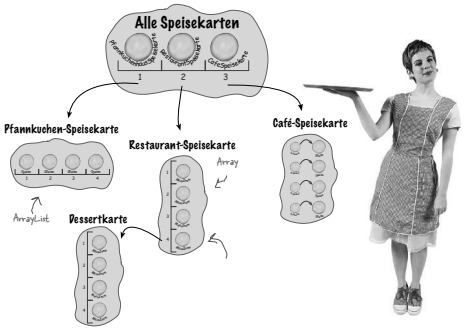
Wir sind auf dem totalen Kapselungstrip. Wir haben die Objekt-Erstellung eingekapselt, Methodenaufrufe, komplexe Schnittstellen, Enten, Pizzas ... was könnte als Nächstes kommen? Wir werden dazu übergehen, Teile von Algorithmen zu kapseln, damit Unterklassen sich jederzeit in eine Berechnung einkapseln können, wenn sie das möchten. Außerdem werden wir einiges über ein Entwurfsprinzip lernen, das von Hollywood inspiriert ist.



Die Iterator- und Composite-Muster

9 Erfolgreiche Kollektionen

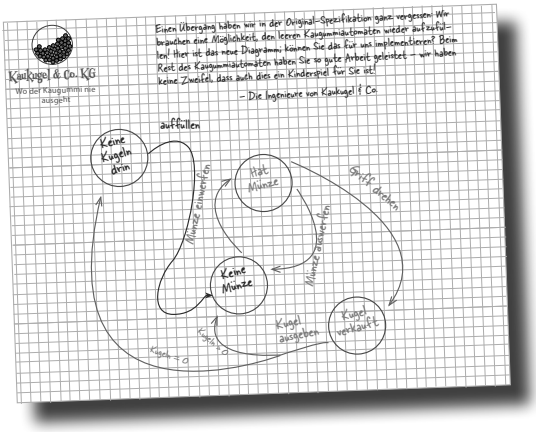
Es gibt viele Möglichkeiten, Objekte in eine Sammlung zu packen. Stecken Sie sie in ein Array-, Stack-, -List- oder Hashtable-Objekt. Sie haben die freie Auswahl. Und jede hat ihre Vor- und Nachteile. Aber irgendwann wird Ihr Client über diese Objekte iterieren wollen. Werden Sie ihm Ihre Implementierung zeigen, wenn er das tut? Wir hoffen ganz entschieden, dass Sie das nicht tun werden! Es wäre einfach nicht professionell. Sie müssen Ihre Karriere nicht riskieren. Sie werden sehen, wie Sie es Clients ermöglichen, über Ihre Objekt zu iterieren, ohne dass er je sieht, wie Sie Ihre Objekte speichern. Sie werden auch lernen, wie Sie Super Collections von Objekten pflegen, die mit einem einzigen Satz einige beeindruckende Datenstrukturen überspringen können. Und wenn Ihnen das immer noch nicht ausreicht, werden Sie außerdem ein oder zwei Dinge über Objektverantwortlichkeit lernen.



Das State-Muster

10 Die Zustände in Objekthausen

Eine kaum bekannte Tatsache ist: Das Strategy- und das State-Muster sind Zwillinge, die bei der Geburt getrennt wurden. Wie Sie schon wissen, hat das Strategy-Muster später ein supererfolgreiches Geschäft mit austauschbaren Algorithmen aufgebaut. Das State-Pattern hingegen hat einen anderen – vielleicht edelmütigeren – Weg eingeschlagen: Es hilft Objekten, ihr Verhalten mittels Veränderung ihres internen Zustands zu kontrollieren. Oft hört man es zu seiner Objekt-Klientel sagen: »Sprecht mir nach: Ich bin gut genug, ich bin klug genug, verdammt noch mal ...«

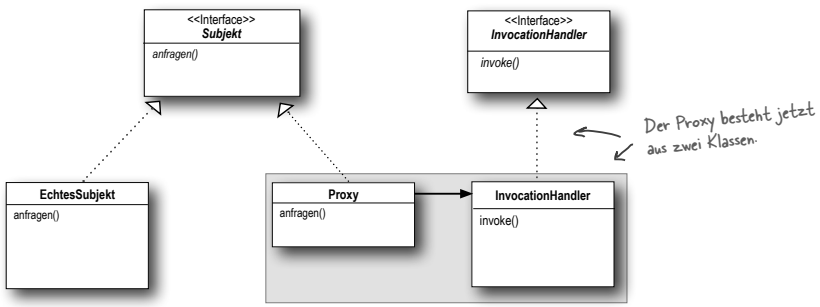


Das Proxy-Muster

11 Den Zugriff auf Objekte kontrollieren

Haben Sie schon mal »good cop – bad cop« gespielt?

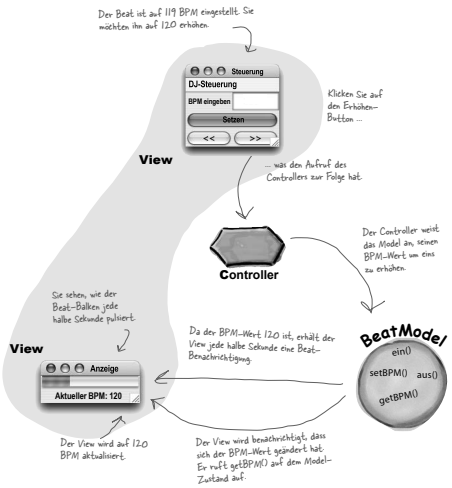
Sie sind der gute Polizist und helfen den Menschen nett und freundlich. Aber Sie möchten einfach nicht *jedem* zu Diensten sein, und deshalb haben Sie den bösen Polizisten, der den *Zugang zu Ihnen kontrolliert.* Genau das tun Proxys: Sie kontrollieren und steuern den Zugang zu etwas anderem. Wie Sie sehen werden, können Proxys sich auf ganz unterschiedliche Art und Weise vor ihre zugehörigen Objekte stellen. Proxys haben schon komplette Methodenaufrufe über das Internet für ihre Objekte durchgeführt; manchmal sind sie aber auch nur geduldige Stellvertreter für ziemlich faule Objekte.

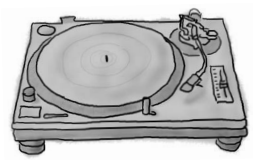


12 Muster von Mustern

Wer hätte je gedacht, dass Entwurfsmuster zusammenarbeiten könnten?

Sie sind ja schon Zeuge der erbitterten Auseinandersetzungen am Kamin geworden (und dabei haben Sie noch nicht mal die Seiten mit den Kämpfen auf Leben und Tod gesehen, die wir auf Druck des Verlags wieder herausnehmen mussten). Mal ehrlich, hätten Sie geglaubt, dass Muster gut miteinander auskommen können? Also, ob Sie es glauben oder nicht: Einige der leistungsfähigsten OO-Designs setzen mehrere Muster gemeinsam ein. Machen Sie sich also bereit für Ihren nächsten Muster-Qualifikationslevel, denn jetzt stehen zusammengesetzte Muster auf dem Plan.

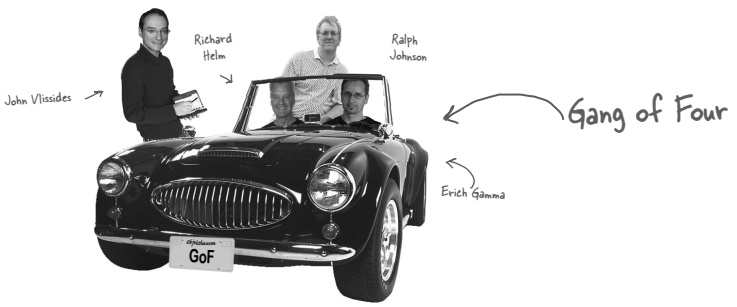




Besser leben mit Mustern

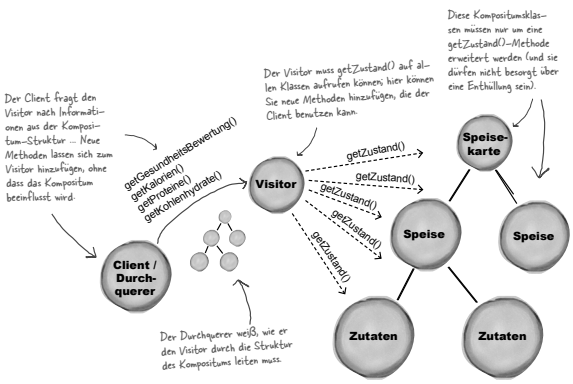
13 Entwurfsmuster in der realen Welt

Aaah, jetzt sind Sie bereit für eine strahlende neue Welt voller Entwurfsmuster! Aber bevor Sie all die tollen Chancen nutzen, die sich Ihnen jetzt bieten, müssen wir noch ein paar Einzelheiten besprechen, die Sie in der realen Welt beachten müssen – ja, ein bisschen komplizierter als hier in Objekthausen wird es schon! Schauen Sie mal auf die nächste Seite: Dort haben wir einen schönen Leitfaden, der Ihnen die Eingewöhnung erleichtern wird.



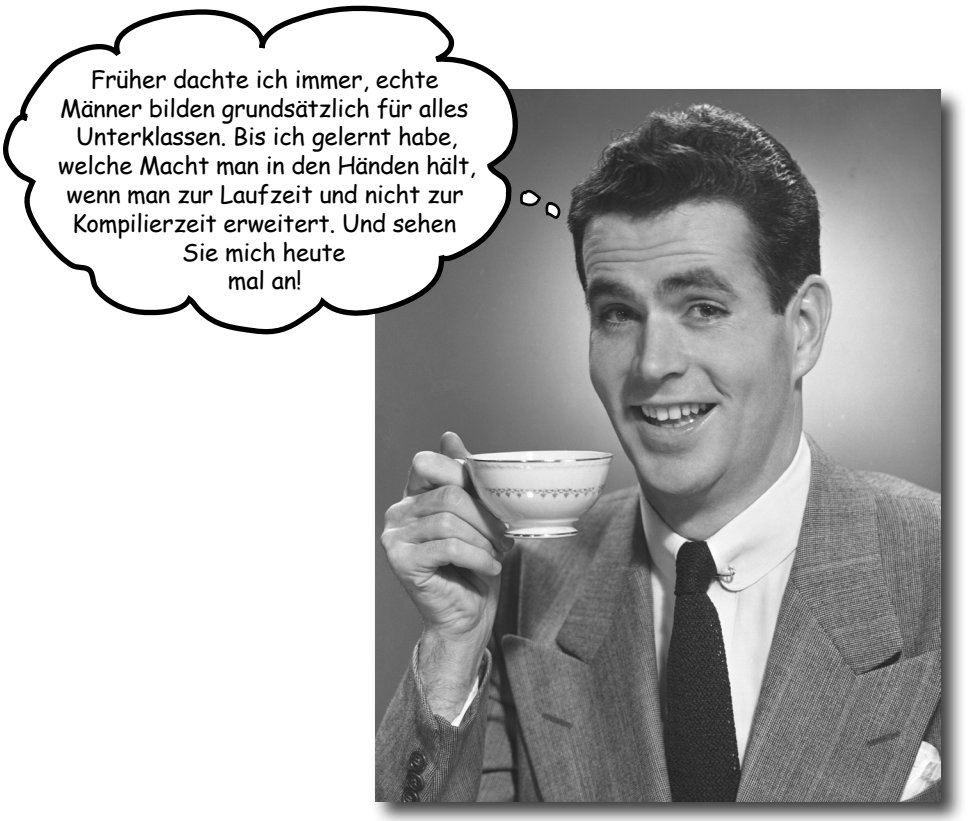
14 Anhang: Übrig gebliebene Muster

Nicht jeder kann eine Berühmtheit sein. In den letzten zehn Jahren hat sich eine Menge geändert. Seit die 1. Auflage von *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software* erschienen ist, haben Entwickler diese Muster Tausende von Malen angewendet. Die Muster, die in diesem Anhang zusammengefasst sind, sind vollwertige, ausgewiesene, offizielle GoF-Muster, sie werden nur nicht so oft verwendet wie die Muster, mit denen wir uns bis jetzt beschäftigt haben. Dennoch werden diese Muster mit vollem Recht als großartige Muster betrachtet, und wenn Sie in einer Situation sind, die danach verlangt, können Sie sie mit erhobenem Haupt anwenden. In diesem Anhang möchten wir Ihnen eine ungefähre Vorstellung davon vermitteln, worum es bei diesen Mustern geht.



3 Das Decorator-Muster

Objekte dekorieren



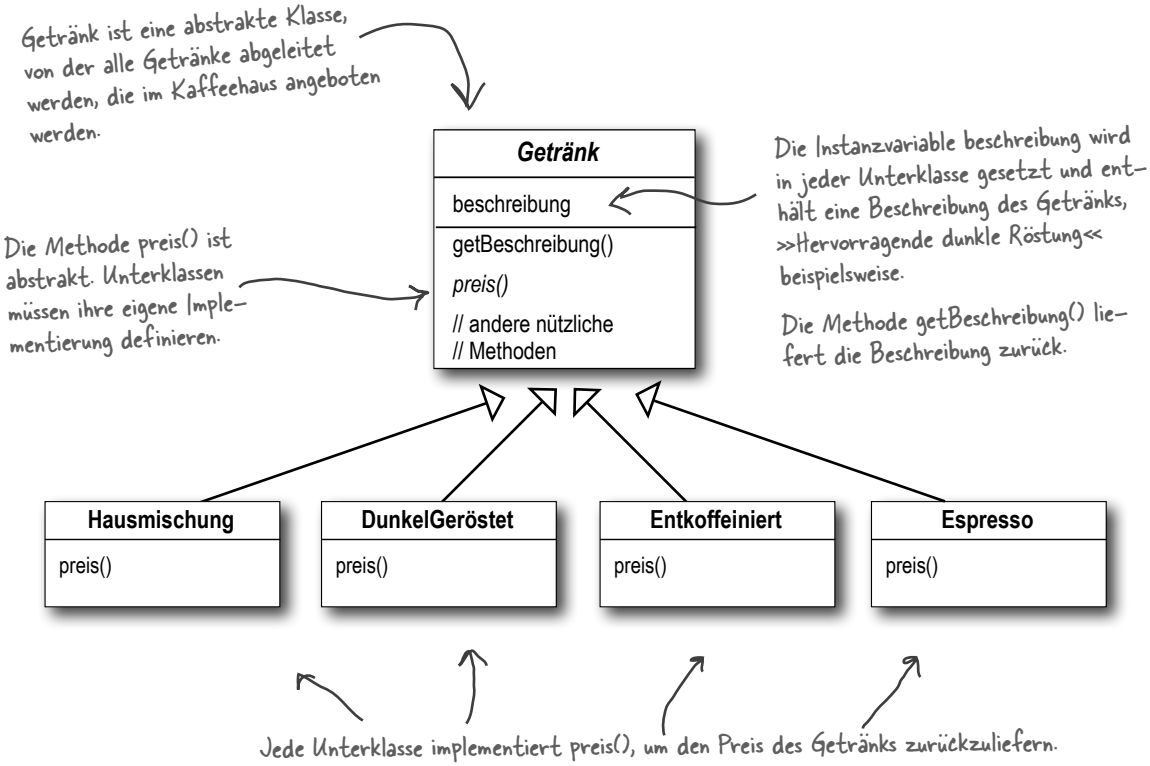
Nennen wir dieses Kapitel einfach »Vererbst du noch oder designst du schon?«. Wir untersuchen noch einmal einen typischen Fall überstrapazierter Vererbung, und Sie werden lernen, wie Sie Ihre Klassen mithilfe einer Form der Objekt-Zusammensetzung erst zur Laufzeit »dekorieren«. Warum? Wenn Ihnen die Techniken des Dekorierens einmal vertraut sind, können Sie Ihren Objekten (oder den Objekten anderer) neue Aufgaben geben, ohne den Code der zugrunde liegenden Klasse ändern zu müssen.

Willkommen bei Sternback-Kaffee

Sternback-Kaffee hat sich einen Namen gemacht als die zurzeit am schnellsten expandierende Kaffeehaus-Kette. Wenn Sie bei Ihnen an der Ecke vor der Filiale stehen, müssen Sie nur einen Blick über die Straße werfen und werden dort bestimmt gleich auf die nächste stoßen.

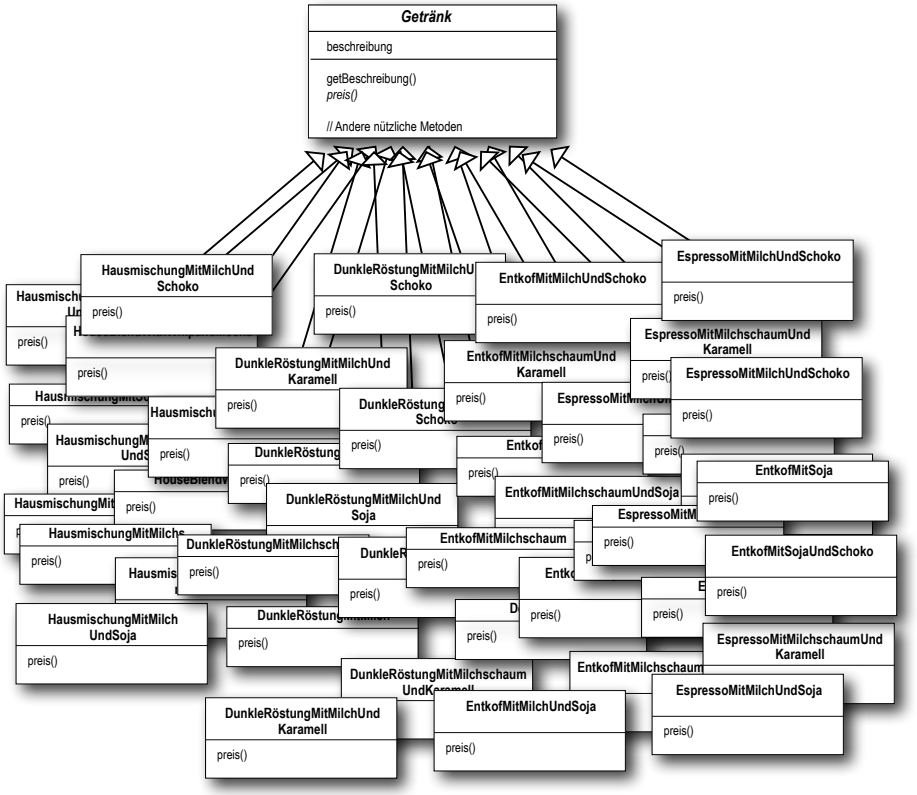
Aufgrund der rasanten Expansion kämpft das Unternehmen damit, sein Bestellsystem so zu aktualisieren, dass es dem Getränkeangebot entspricht.

Als Sternback das Geschäft aufnahm, hat man die Klassen folgendermaßen entworfen ...



Zum Kaffee kann man auch verschiedenste Zutaten wie heiße Milch, Soja oder Schokolade bestellen, und obendrauf kann man noch Milchschaum haben. Natürlich berechnet Sternback jede dieser Zutaten extra. Deswegen muss Sternback sehen, dass sie in das Bestellsystem integriert werden.

Hier ist der erste Versuch ...



Jede preis()-Methode berechnet den Preis des Kaffees zuzüglich aller bestellten Zutaten.

KOPF-NUSS

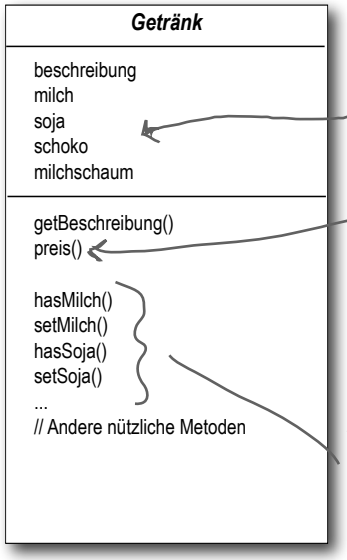
Es ist ziemlich offensichtlich, dass Sternback sich damit selbst einen Wartungsalbtraum beschert hat. Was passiert beispielsweise, wenn der Milchpreis steigt? Oder was machen sie, wenn sie eine neue Karamell-Garnierung einführen?

Wenn Sie das Wartungsproblem einmal außer Acht lassen, welche der Entwurfsprinzipien, die wir bisher behandelt haben, werden hier verletzt?

Hinweis: Zwei davon werden ziemlich grundlegend verletzt.

Das ist dämlich: Wozu brauchen wir all diese Klassen? Können wir nicht einfach in der Superklasse Instanzvariablen und Vererbung nutzen, um die Zutaten nachzuhalten?

Gut. Lassen Sie uns das versuchen. Wir beginnen mit der Basisklasse Getränk und fügen ihr Instanzvariablen hinzu, die angeben, ob ein Getränk Milch, Soja, Schoko oder Milchschaum enthält oder nicht.



Neue Boolesche Werte für die einzelnen Zutaten.

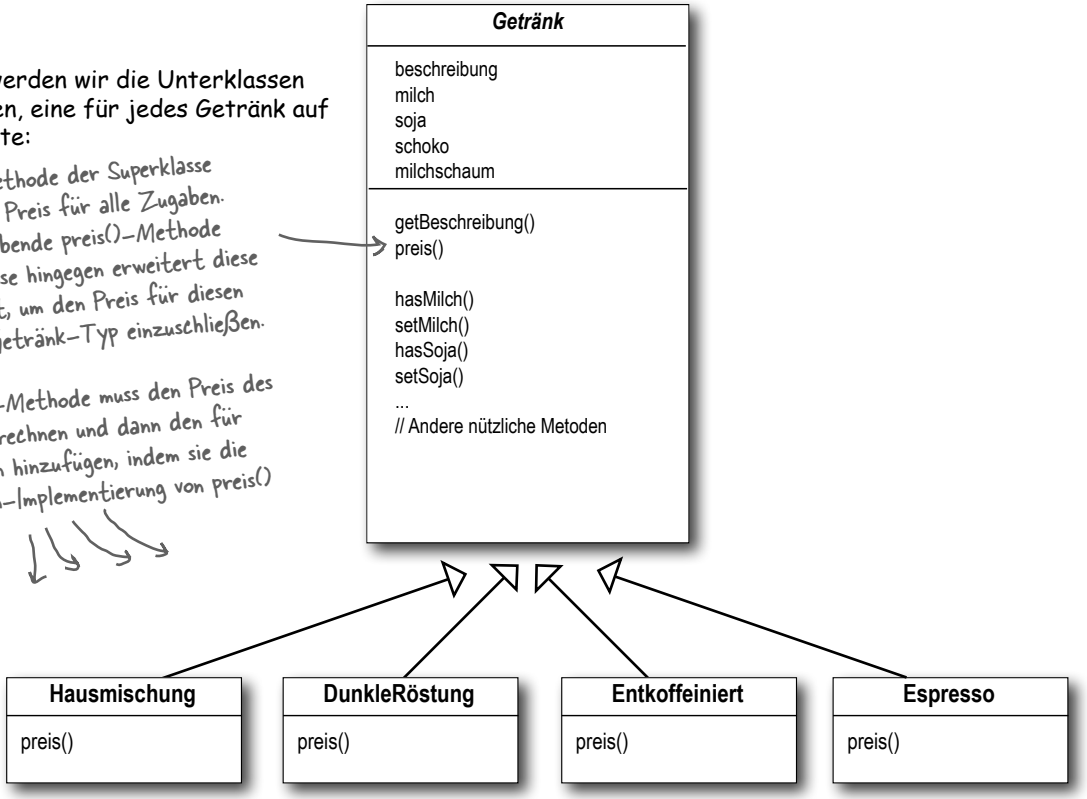
Jetzt implementieren wir die Methode preis() in Getränk (anstatt sie abstrakt zu lassen), damit sie den Preis für die Zutatenkombination einer bestimmten Getränk-Instanz berechnen kann. Unterklassen müssen preis() immer noch überschreiben, aber sie rufen auch die Version der Superklasse auf, damit sie den Preis für das Grundgetränk zuzüglich der hinzugefügten Zutaten berechnen können.

Diese Methoden lesen und setzen die Booleschen Werte für die Zutaten.

Jetzt werden wir die Unterklassen ergänzen, eine für jedes Getränk auf der Karte:

Die preis()-Methode der Superklasse berechnet den Preis für alle Zugaben. Die überschreibende preis()-Methode der Unterklasse hingegen erweitert diese Funktionalität, um den Preis für diesen bestimmten Getränk-Typ einzuschließen.

Jede preis()-Methode muss den Preis des Getränks berechnen und dann den für die Zutaten hinzufügen, indem sie die Superklassen-Implementierung von preis() aufruft.



Spitzen Sie Ihren Bleistift

Schreiben Sie die preis()-Methoden für die folgenden Klassen (Pseudo-Java genügt):

```
public class Getränk {
   public double preis() {




   }
}
```

```
public class DunkleRöstung extends Getränk {

   public DunkleRöstung() {
     beschreibung = "Hervorragende dunkle Röstung";
   }

   public double preis() {




   }
}
```

Sehen Sie? Nur fünf Klassen. Das ist bestimmt der richtige Weg.

Da bin ich nicht so sicher. Ich sehe bei diesem Ansatz einiges an Problempotenzial, wenn ich mir darüber Gedanken mache, wie sich der Entwurf in der Zukunft eventuell ändern muss.

Spitzen Sie Ihren Bleistift

Welche Anforderungen oder andere Faktoren könnten sich ändern, die Auswirkungen auf diesen Entwurf haben?

Preisänderungen bei den Zutaten könnten die Bearbeitung von bestehendem Code erforderlich machen.

Neue Zutaten würden uns zwingen, der Superklasse neue Methoden hinzuzufügen und ihre preis()–Methode zu verändern.

Vielleicht führen wir neue Getränke ein? Für manche dieser Getränke (Eistee?) könnten die Zutaten etwas unpassend sein, und trotzdem würde die Unterklasse Tee Methoden wie hasMilchschaum() erben.

Wie wir in Kapitel 1 gesehen haben, ist das ziemlich übel.

Was ist, wenn ein Kunde Doppelschoko wünscht?

Sie sind dran:

Meister und Schüler

Meister: *Grashüpfer, seit unserem letzten Treffen ist einige Zeit verstrichen. Hast du gründlich über die Vererbung meditiert?*

Schüler: *Ja, Meister. Vererbung ist zwar mächtig, aber ich habe gelernt, dass sie nicht immer zu den flexibelsten oder wartbarsten Entwürfen führt.*

Meister: *Aha! Du hast kleine Fortschritte gemacht. Also sag mir, mein Schüler, wie du ohne Vererbung Wiederverwendbarkeit erreichen willst.*

Schüler: *Meister, ich habe gelernt, dass es Wege gibt, Verhalten zur Laufzeit durch Komposition und Delegierung zu »vererben«.*

Meister: *Weiter bitte.*

Schüler: *Wenn ich Verhalten durch Ableitung erbe, wird dieses Verhalten statisch zur Kompilierzeit festgelegt. Außerdem müssen alle Unterklassen das gleiche Verhalten erben. Aber wenn ich das Verhalten eines Objekts durch Komposition erweitere, kann ich das dynamisch zur Laufzeit tun.*

Meister: *Sehr gut, Grashüpfer. Du beginnst, die Macht der Zusammensetzung zu verstehen.*

Schüler: *Ja. Mit dieser Technik kann ich Objekten mehrere neue Aufgaben geben. Das können sogar Aufgaben sein, an die der Entwickler der Superklasse nie gedacht hat. Und ich muss seinen Code dafür nicht anrühren.*

Meister: *Was hast du über die Auswirkungen der Komposition auf die Wartbarkeit von Code gelernt?*

Schüler: *Ja, darauf wollte ich gerade eingehen. Durch die dynamische Komposition von Objekten kann ich neue Funktionalitäten hinzufügen, indem ich neuen Code schreibe, anstatt bestehenden Code zu ändern. Weil ich keinen bestehenden Code ändern muss, sinkt die Gefahr erheblich, dass ich in bereits vorhandenen Code Fehler einbaue oder unerwünschte Nebeneffekte verursache.*

Meister: *Sehr gut. Das reicht für heute, Grashüpfer. Ich möchte gern, dass du dich aufmachst und noch mehr über dieses Thema meditierst ... Denke daran: Wie die Lotusblüte am Abend sollte Code (für Veränderungen) geschlossen sein und doch wie die Lotusblüte am Morgen (für Erweiterung) offen bleiben.*

Das Offen/Geschlossen-Prinzip

Grashüpfer ist auf dem Weg zur Erkenntnis eines der wichtigsten Entwurfsprinzipien:

Entwurfsprinzip

Klassen sollten für Erweiterung offen, aber für Veränderung geschlossen sein.

Treten Sie ein. Es ist geöffnet. Wenn Sie wollen, erweitern Sie unsere Klassen mit jedem neuen Verhalten, das Ihnen gefällt. Wenn sich Ihre Bedürfnisse oder Anforderungen ändern (und wir wissen, dass sie das werden), gehen Sie einfach weiter und schreiben Ihre eigenen Erweiterungen.

Leider haben wir geschlossen. Ja, so ist es. Wir haben sehr viel Zeit damit verbracht, diesen Code richtig und fehlerfrei zu machen. Deswegen können wir nicht zulassen, dass Sie bestehenden Code modifizieren. Er muss für Änderungen geschlossen bleiben. Falls Ihnen das nicht passt, können Sie sich an den Geschäftsführer wenden.

Unser Ziel ist es zu ermöglichen, dass Klassen leicht erweitert werden können, um neue Verhalten zu integrieren, ohne bestehenden Code zu verändern. Was es uns bringt, wenn wir das erreichen? Entwürfe, die für Verändungen offen und flexibel genug sind, um neue Funktionalitäten aufzunehmen und so geänderten Anforderungen gerecht zu werden.

Es gibt keine Dummen Fragen

F: **Für Erweiterungen offen und für Veränderungen geschlossen? Das klingt ziemlich widersprüchlich. Wie kann ein Entwurf beides zugleich sein?**

A: Das ist eine sehr gute Frage. Natürlich klingt das zunächst widersprüchlich. Je weniger etwas modifizierbar ist, um so schwerer ist es schließlich auch zu erweitern, oder?

Aber es stellt sich dennoch heraus, dass es ein paar clevere OO-Techniken gibt, die die Erweiterung von Systemen ermöglichen, auch wenn wir den zugrunde liegenden Code nicht ändern können. Denken Sie an das Observer-Muster (aus Kapitel 2) ... indem wir neue Beobachter hinzufügen, können wir das Subjekt jederzeit ändern, ohne dem Code des Subjekts etwas hinzuzufügen. Sie werden noch eine ganze Reihe weiterer Möglichkeiten kennen lernen, Verhalten mit OO-Entwurfstechniken zu erweitern.

F: **Okay, ich verstehe Observable. Aber ganz allgemein: Wie entwerfe ich etwas so, dass es erweiterbar und dennoch für Veränderungen geschlossen ist?**

A: Viele der Muster bieten uns erprobte Entwürfe, die Ihren Code vor Veränderung schützen, indem sie Erweiterungsmöglichkeiten bieten. In diesem Kapitel werden Sie ein gutes Beispiel dafür sehen, wie man das Decorator-Muster verwendet, um dem Offen/Geschlossen-Prinzip zu genügen.

F: **Wie erreiche ich, dass jeder Teil meines Entwurfs dem Offen/Geschlossen-Prinzip folgt?**

A: Normalerweise ist das nicht möglich. Es braucht viel Zeit und Anstrengung, OO-Entwürfe flexibel und für Erweiterung offen zu gestalten, ohne dass dazu bestehender Code modifiziert werden muss. In der Regel können wir es uns nicht leisten, jeden Teil unseres Entwurfs festzunageln (und es wäre wahrscheinlich auch Verschwendung). Dem Offen/Geschlossen-Prinzip zu folgen führt in der Regel zur Einführung neuer Abstraktionsschichten und macht unseren Code deswegen komplexer. Sie sollten sich in Ihren Entwürfen auf die Bereiche konzentrieren, in denen Änderungen am wahrscheinlichsten sind, und diese Prinzipien dort anwenden.

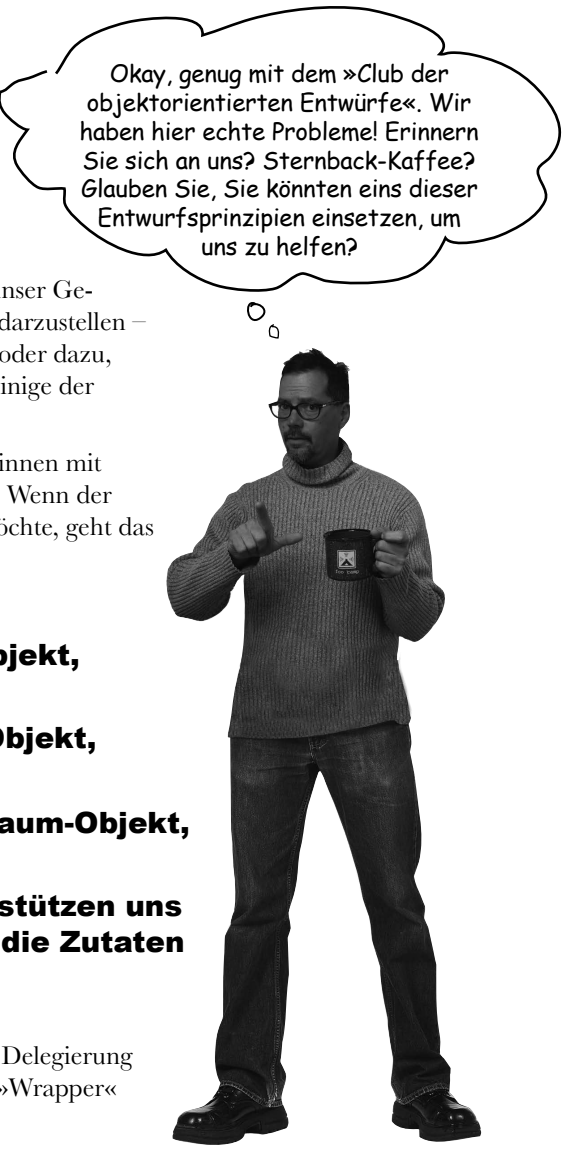
F: **Woher weiß ich, welche Bereiche da wichtiger sind?**

A: Zum Teil ist das eine Frage der Erfahrung im Entwurf von OO-Systemen, zum Teil eine Frage der Beherrschung des Gebiets, auf dem man arbeitet. Andere Beispiele nachzuvollziehen wird Ihnen helfen, in Ihren eigenen Entwürfen solche Bereiche zu identifizieren, die sich bestimmt ändern.

Auch wenn es wie ein Widerspruch klingt: Es gibt Techniken, die es ermöglichen, Code zu erweitern, ohne ihn direkt zu modifizieren.

Seien Sie vorsichtig, wenn Sie die Codebereiche auswählen, die erweitert werden müssen. Das Offen/Geschlossen-Prinzip ÜBERALL anzuwenden ist Verschwendung, unnötig und kann zu komplexem, schwer verständlichem Code führen.

Dürfen wir vorstellen: das Decorator-Muster!



Gut, wir haben eingesehen, dass es nicht so toll funktioniert, unser Getränke-plus-Zutaten-Preissystem über das Vererbungsschema darzustellen – das führt zu einer Klassen-Explosion, einem starren Entwurf oder dazu, dass wir der Basisklasse Funktionalitäten hinzufügen, die für einige der Unterklassen nicht geeignet sind.

Deswegen werden wir stattdessen Folgendes machen: Wir beginnen mit einem Getränk und »dekorieren« es zur Laufzeit mit Zutaten. Wenn der Kunde eine dunkle Röstung mit Schoko und Milchschaum möchte, geht das beispielsweise so:

1. **Wir nehmen ein DunkleRöstung-Objekt,**
2. **dekorieren es mit einem Schoko-Objekt,**
3. **dekorieren es mit einem Milchschaum-Objekt,**
4. **rufen die Methode preis() auf und stützen uns auf Delegierung, um den Preis für die Zutaten hinzuzufügen.**

Gut! Aber wie »dekoriert« man ein Objekt, und wie spielt die Delegierung da rein? Ein Hinweis: Stellen Sie sich Dekorierer-Objekte als »Wrapper« vor. Lassen Sie uns mal sehen, wie das funktioniert ...

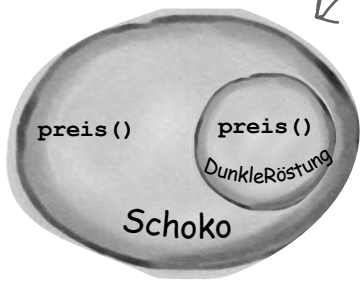
Ein Getränk mit Dekorierern aufbauen

1 Wir beginnen mit unserem DunkleRöstung-Objekt.

Denken Sie daran, dass DunkleRöstung von Getränk erbt und eine preis()-Methode hat, die den Preis eines Getränks berechnet.

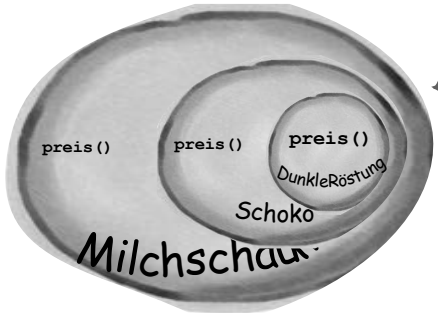
2 Der Kunde möchte Schoko, also erzeugen wir ein Schoko-Objekt und packen es um DunkleRöstung.

Das Schoko-Objekt ist ein Dekorierer. Es ist ein Typ, der das Objekt widerspiegelt, das es dekoriert, in diesem Fall ein Getränk. (Mit >>widerspiegeln<< meinen wir, dass es den gleichen Typ hat.)



Schoko hat also ebenfalls eine preis()-Methode, und durch Polymorphie können wir jedes Getränk, das in ein Schoko eingepackt ist, ebenfalls als ein Getränk behandeln (weil Schoko ein Untertyp von Getränk ist).

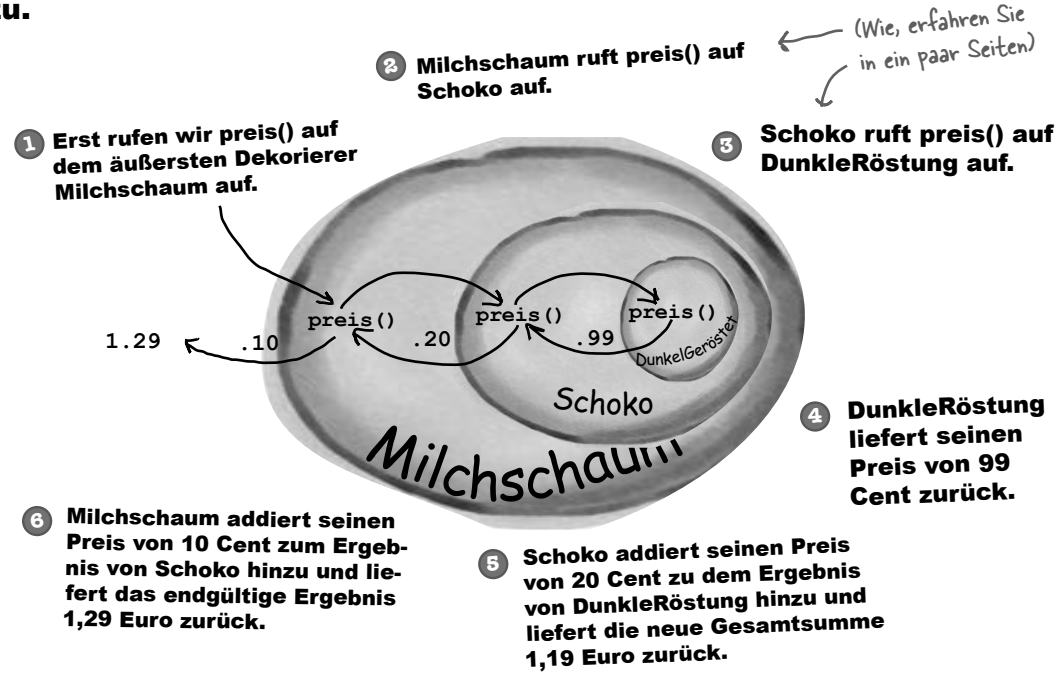
3 Der Kunde möchte außerdem Milchschaum, also erstellen wir einen Milchschaum-Dekorierer und packen Schoko damit ein.



Milchschaum ist ein Dekorierer, er spiegelt also ebenfalls den Typ von DunkleRöstung wider und enthält eine preis()-Methode.

Eine in Schoko und Milchschaum eingepackte DunkleRöstung ist also immer noch ein Getränk. Wir können damit also alles machen, was wir mit einer DunkleRöstung machen können, seine preis()-Methode aufrufen eingeschlossen.

4 Jetzt ist es an der Zeit, den Preis für den Kunden zu berechnen. Das machen wir, indem wir preis() auf dem äußersten Dekorierer, Milchschaum, aufrufen, und Milchschaum delegiert die Berechnung des Preises dann an die Objekte, die es dekoriert. Wenn es einen Preis erhalten hat, fügt es den Preis für Milchschaum hinzu.



Das also wissen wir bisher

- Dekorierer haben den gleichen Supertyp wie die Objekte, die sie dekorieren.
- Sie können ein oder mehr Objekte verwenden, um ein Objekt einzupacken.
- Da der Dekorierer den gleichen Supertyp wie das dekorierte Objekt hat, können wir das dekorierte Objekt anstelle des ursprünglichen (jetzt eingepackten) Objekts herumreichen.
- Der Dekorierer fügt sein eigenes Verhalten hinzu, bevor und/oder nachdem der Aufruf an das dekorierte Objekt delegiert wurde, um die Arbeit abzuschließen.

Wichtiger Punkt!

- Objekte können jederzeit dekoriert werden. Wir können Objekte also zur Laufzeit dynamisch mit so vielen Dekorierern dekorieren, wie es uns gefällt.

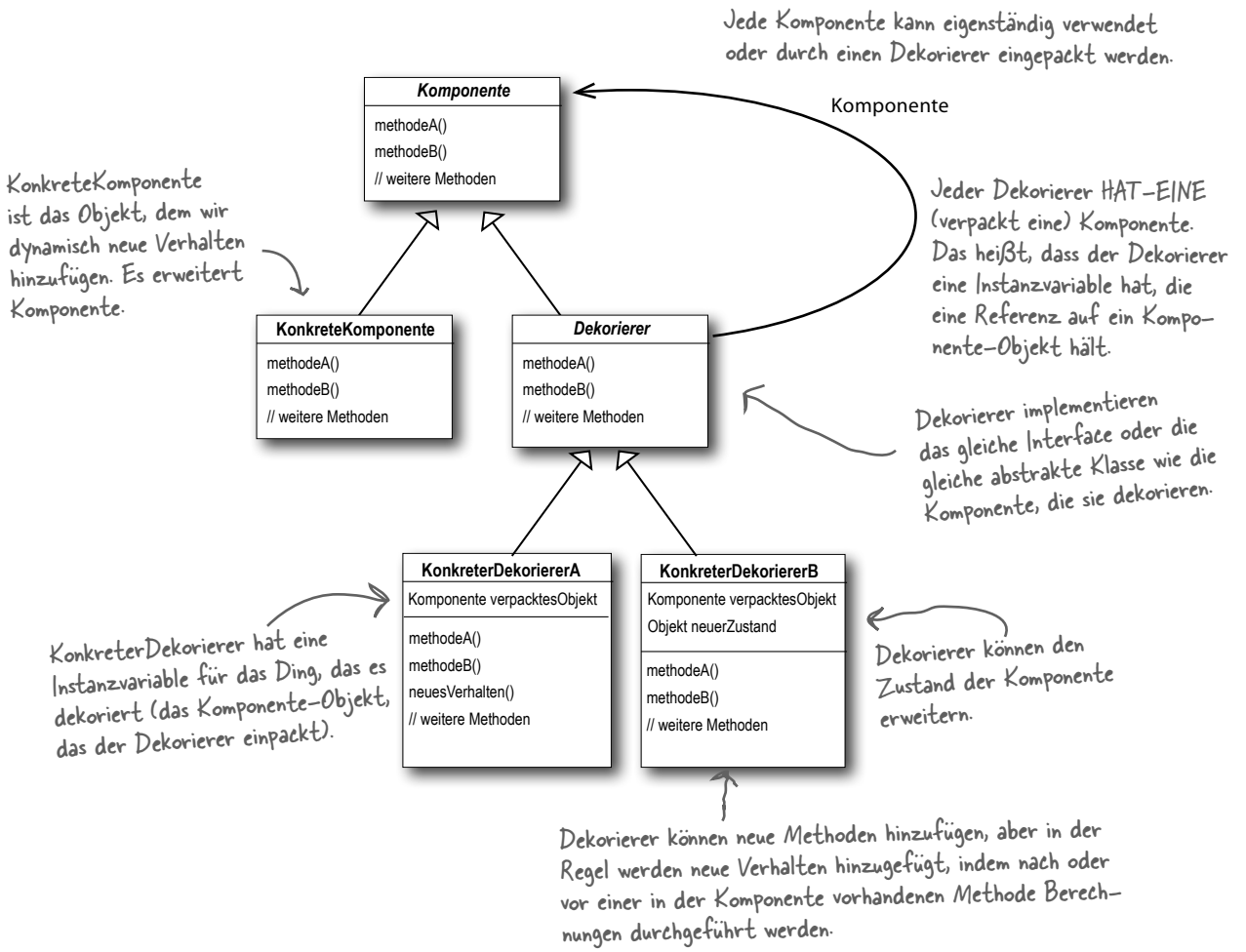
Sehen wir uns jetzt an, wie das alles wirklich funktioniert, indem wir uns die Definition für das Decorator-Muster ansehen und etwas Code schreiben.

Die Definition des Decorator-Musters

Werfen wir zunächst einen Blick auf die Beschreibung des Decorator-Musters:

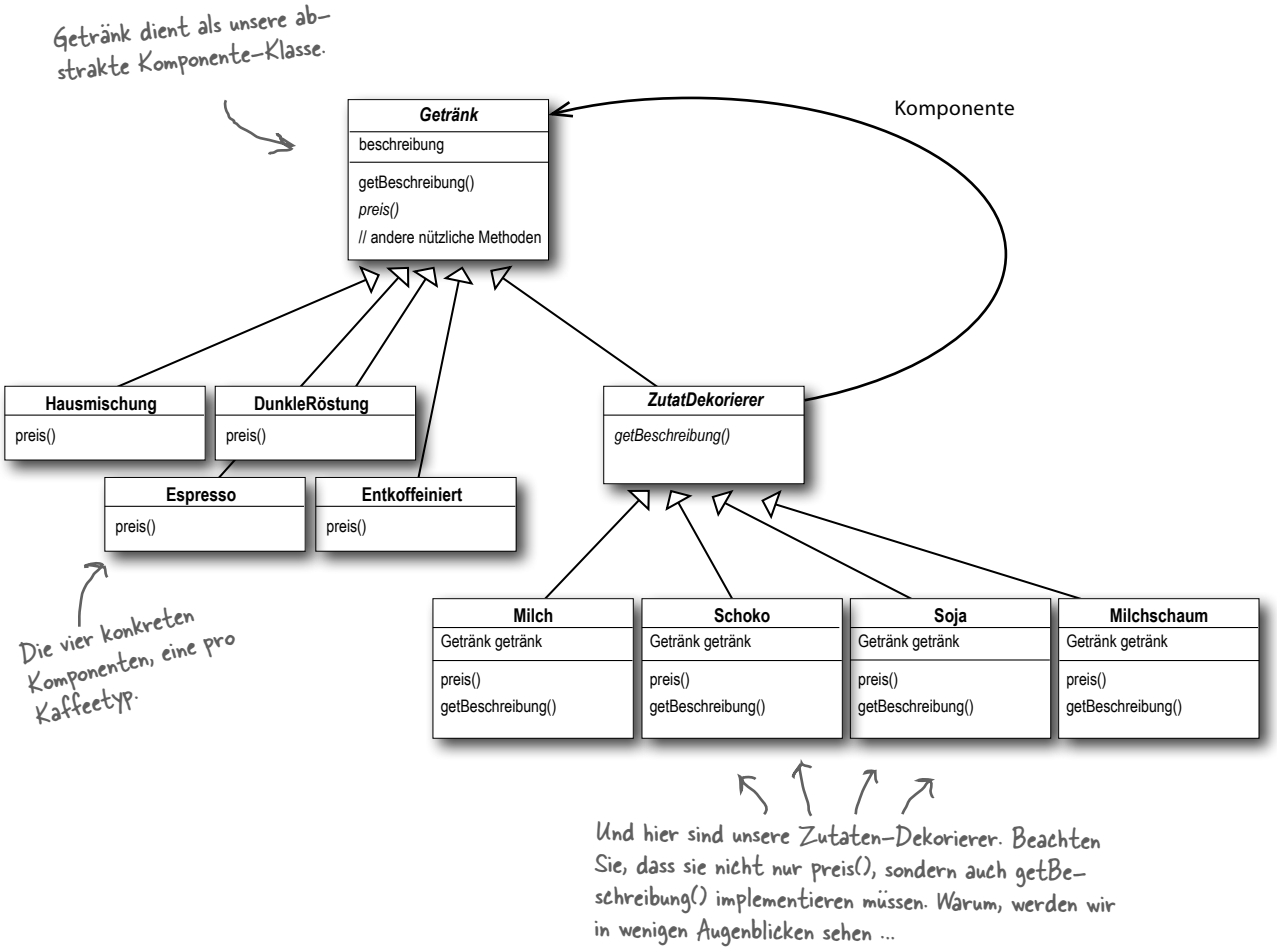
> **Das Decorator-Muster** fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität.

Das beschreibt zwar die *Rolle* des Decorator-Musters, erklärt aber nicht besonders gut, wie wir das Muster auf unsere eigene Implementierung *anwenden*. Werfen wir einen Blick auf das Klassendiagramm, das etwas informativer ist (auf der nächsten Seite sehen wir uns an, wie diese Struktur auf das Getränke-Problem angewandt wird).



Unsere Getränke dekorieren

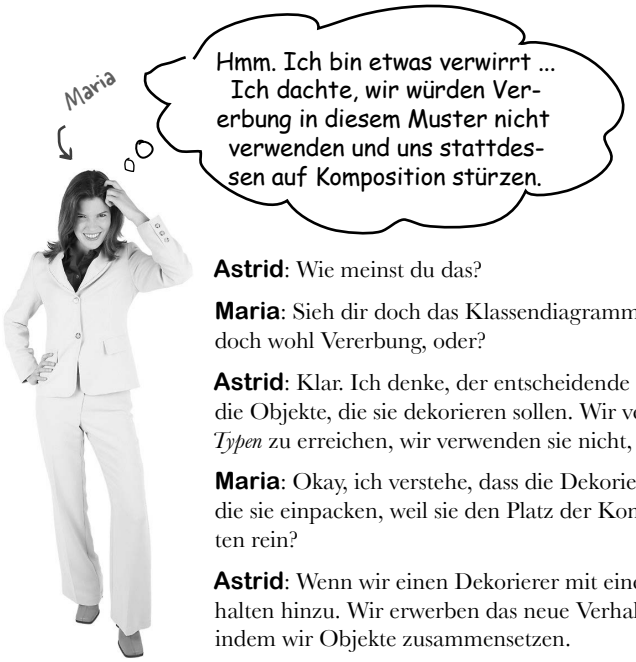
So weit, so gut, fügen wir also unsere Sternback-Getränke in dieses Framework ein ...



Überlegen Sie, bevor wir fortfahren, wie Sie die preis()-Methode für die Kaffees und die Zutaten implementieren würden. Überlegen Sie auch, wie Sie die getBeschreibung-Methode für die Zutaten implementieren würden.

Gespräch im Büro

Kleine Verwirrung hinsichtlich Vererbung und Komposition



Astrid: Wie meinst du das?

Maria: Sieh dir doch das Klassendiagramm an. ZutatDekorierer erweitert die Klasse Getränk. Das ist doch wohl Vererbung, oder?

Astrid: Klar. Ich denke, der entscheidende Punkt ist, dass die Dekorierer den gleichen Typ haben, wie die Objekte, die sie dekorieren sollen. Wir verwenden die Vererbung hier also, um das *Übereinstimmen der Typen* zu erreichen, wir verwenden sie nicht, um *Verhalten* zu bekommen.

Maria: Okay, ich verstehe, dass die Dekorierer die gleiche »Schnittstelle« benötigen wie die Komponente, die sie einpacken, weil sie den Platz der Komponente einnehmen sollen. Aber wo kommt da das Verhalten rein?

Astrid: Wenn wir einen Dekorierer mit einer Komponente zusammensetzen, fügen wir ein neues Verhalten hinzu. Wir erwerben das neue Verhalten nicht, indem wir es von der Superklasse erben, sondern indem wir Objekte zusammensetzen.

Maria: Gut. Wir leiten also eine Unterklasse von der abstrakten Klasse Getränk ab, damit wir den richtigen Typ erhalten, nicht um ihr Verhalten zu erben. Das Verhalten kommt durch die Komposition von Dekorierern mit den Basiskomponenten und den anderen Dekorierern zu Stande.

Astrid: Stimmt.

Maria: Mensch, ich hab's! Und weil wir Objekt-Komposition verwenden, kriegen wir viel mehr Flexibilität zum Mischen und Zusammenstellen von Zutaten und Getränken. Sehr ausgefeilt.

Astrid: Genau, wenn wir uns auf Vererbung stützen, dann kann das Verhalten nur statisch zur Kompilierzeit festgelegt werden. Anders gesagt, wir bekommen nur das Verhalten, das uns die Superklasse gibt oder das wir überschreiben. Bei Komposition können wir Dekorierer mischen und zusammenstellen, wie wir wollen ... und das *zur Laufzeit*.

Maria: Und wenn ich das richtig verstehe, können wir jederzeit neue Dekorierer implementieren, um neue Verhalten hinzuzufügen. Würden wir uns auf Vererbung stützen, müssten wir jedes Mal, wenn wir neue Verhalten hinzufügen wollen, bestehenden Code ändern.

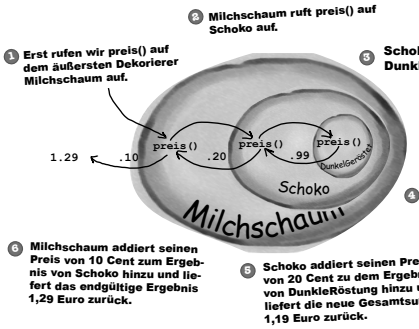
Astrid: Ganz genau.

Maria: Eine Frage habe ich aber noch. Wenn alles, was wir erben, der Typ der Komponente ist, warum haben wir für Getränk dann kein Interface verwendet anstatt einer abstrakten Klasse?

Astrid: Erinnerst du dich darin, dass Sternback bereits eine abstrakte Getränk-Klasse *hatte*, als wir den Code bekommen haben? Traditionellerweise gibt das Decorator-Muster eine abstrakte Komponente vor, aber in Java könnten wir natürlich ein Interface verwenden. Aber wir versuchen immer, Änderungen an bestehendem Code zu vermeiden, und »reparieren« ihn hier deswegen nicht, wenn die abstrakte Klasse genauso gut funktioniert.

Neue Angestellte einarbeiten

Erstellen Sie eine Abbildung, die die Bestellung von »Doppel-Schoko-Soja-Kaffee mit Milchschaum« darstellt. Entnehmen Sie die richtigen Preise der Speisekarte und zeichnen Sie Ihre Abbildung in dem Format, das wir oben (ein paar Seiten weiter vorn) verwendet haben:



Diese Abbildung wurde für das Getränk »Schoko-Dunkle Röstung mit Milchschaum« erstellt.

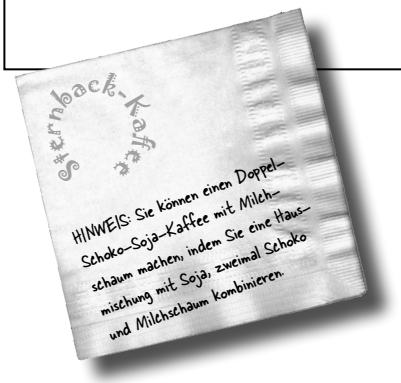
Spitzen Sie Ihren Bleistift

Zeichnen Sie Ihre Abbildung hier.

Sternback-Kaffee

Kaffees	
Hausmischung	0,89
Dunkle Röstung	0,99
Entkoffeiniert	1,05
Espresso	1,99

Zutaten	
Heiße Milch	0,10
Schoko	0,20
Soja	0,15
Milchschaum	0,10



Den Sternback-Code schreiben

Jetzt ist es an der Zeit, diesen Entwurf zu etwas echtem Code zu verquirlen.

Beginnen wir mit der Klasse Getränk, die gegenüber Sternbacks Originalentwurf nicht verändert werden muss. Werfen wir einen Blick darauf:

```
public abstract class Getränk {
    String beschreibung = "Unbekanntes Getränk";

    public String getBeschreibung() {
        rreturn beschreibung;
    }

    public abstract double preis();
}
```

Getränk ist eine abstrakte Klasse mit den beiden Methoden getBeschreibung() und preis().

getBeschreibung() ist bereits implementiert, aber preis() muss in den Unterklassen implementiert werden.

Getränk ist eigentlich ziemlich simpel. Implementieren wir also auch die abstrakte Klasse für die Zutaten (Dekorierer):

Da die Zutaten mit Getränk austauschbar sein müssen, erweitern wir die Klasse Getränk.

```
public abstract class ZutatDekorierer extends Getränk {
    public abstract String getBeschreibung();
}
```

Außerdem verlangen wir, dass die Zutaten-Dekorierer alle die Methode getBeschreibung() neu implementieren. Noch mal: Warum, werden wir gleich sehen ...

Getränke programmieren

Jetzt haben wir unsere Basisklassen erledigt und können ein paar Getränke implementieren. Wir beginnen mit Espresso. Denken Sie daran, dass wir eine Beschreibung für dieses bestimmte Getränk setzen und außerdem die Methode preis() implementieren müssen.

Zuerst erweitern wir die Klasse Getränk, weil Espresso ein Getränk ist.

```
public class Espresso extends Getränk {

    public Espresso() {
        beschreibung = "Espresso";
    }

    public double preis() {
        return 1.99;
    }
}
```

Um die Beschreibung kümmern wir uns, indem wir sie im Konstruktor für die Klasse setzen. Erinnern Sie sich daran, dass die Instanzvariable beschreibung von Getränk geerbt wird.

Außerdem müssen wir den Preis eines Espresso berechnen. Da wir uns jetzt nicht mehr darum sorgen müssen, in dieser Klasse die Zutaten hinzuzufügen, liefern wir einfach den Preis für einen Espresso von 1,99 € zurück.

```
public class Hausmischung extends Getränk {
    public Hausmischung() {
        beschreibung = "Hausmischung";
    }

    public double preis() {
        return .89;
    }
}
```

Hier ist ein weiteres Getränk. Wir müssen nur die entsprechende Beschreibung, »Hausmischung«, setzen und dann den richtigen Preis zurückliefern: 89 Cent.

Die anderen beiden Getränke-Klassen (DunkleRöstung und Entkoffeiniert) können Sie auf genau die gleiche Weise erstellen.

Sternback-Kaffee

Kaffees	
Hausmischung	0,89
Dunkle Röstung	0,99
Entkoffeiniert	1,05
Espresso	1,99

Zutaten	
Heiße Milch	0,10
Schoko	0,20
Soja	0,15
Milchschaum	0,10

Die Zutaten programmieren

Wenn Sie an unser Klassendiagramm für das Decorator-Muster zurückdenken, sehen Sie, dass wir jetzt die abstrakte Komponente (Getränk), die konkreten Komponenten (Hausmischung usw.) und den abstrakten Dekorierer (ZutatDekorierer) geschrieben haben. Jetzt ist es Zeit, die konkreten Dekorierer zu implementieren. Hier ist Schoko:

Schoko ist ein Dekorierer, also erweitern wir ZutatDekorierer.

Erinnern Sie sich, dass ZutatDekorierer Getränk erweitert.

```
public class Schoko extends ZutatDekorierer {
    Getränk getränk;

    public Schoko(Getränk getränk) {
        this.getränk = getränk;
    }

    public String getBeschreibung() {
        return getränk.getBeschreibung() + ", Schoko";
    }

    public double preis() {
        return .20 + getränk.preis();
    }
}
```

Wir werden Schoko mit einer Referenz auf ein Getränk instantiieren, indem wir:

(1) eine Instanzvariable verwenden, um das Getränk aufzunehmen, das wir einpacken, und

(2) eine Möglichkeit nutzen, diese Instanzvariable auf das Objekt zu setzen, das wir einpacken. Hier übergeben wir das einzupackende Getränk an den Konstruktor des Dekorierers.

Wir möchten, dass unsere Beschreibung nicht nur den Namen des Getränks – beispielsweise >>Dunkle Röstung<< – enthält, sondern auch jedes Element einschließt, das das Getränk dekoriert – beispielsweise >>Dunkle Röstung, Schoko<<. Deswegen delegieren wir zuerst den Aufruf an das Objekt, das wir dekorieren, um seine Beschreibung zu erhalten, und hängen dieser Beschreibung dann >>Schoko<< an.

Jetzt müssen wir den Preis unseres Getränks mit Schoko berechnen. Erst delegieren wir den Aufruf an das Objekt, das wir dekorieren, damit es den Preis berechnen kann. Dann fügen wir dem Ergebnis den Preis von Schoko hinzu.

Auf der nächsten Seite werden wir das Getränk wirklich instantiieren und mit all seinen Zutaten (Dekorierern) einpacken, aber erst ...

Schreiben und kompilieren Sie den Code für die anderen Zutaten Soja und Milchschaum. Sie benötigen dies, um die Anwendung fertig zu stellen und zu testen.

Kaffee servieren

Glückwunsch. Jetzt können Sie sich zurücklehnen, ein paar Kaffees bestellen und den flexiblen Entwurf bewundern, den Sie mit dem Decorator-Muster geschaffen haben.

Hier ist etwas Test-Code*, um Bestellungen auszuführen:

```
public class SternbackKaffee {

    public static void main(String args[]) {
        Getränk getränk = new Espresso();
        System.out.println(getränk.getBeschreibung()
                + " " + getränk.preis() + " €");

        Getränk getränk2 = new DunkleRöstung();
        getränk2 = new Schoko(getränk2);
        getränk2 = new Schoko(getränk2);
        getränk2 = new Milchschaum(getränk2);
        System.out.println(getränk2.getBeschreibung()
                + " " + getränk2.preis() + " €");

        Getränk getränk3 = new Hausmischung();
        getränk3 = new Soja(getränk3);
        getränk3 = new Schoko(getränk3);
        getränk3 = new Milchschaum(getränk3);
        System.out.println(getränk3.getBeschreibung()
                + " " + getränk3.preis() + " €");
    }
}
```

Einen Espresso ohne Zutaten bestellen und seine Beschreibung sowie die Kosten ausgeben lassen.

Ein DunkleRöstung-Objekt erstellen,

mit einmal Schoko einpacken,

mit noch mal Schoko einpacken

und dann in einen Milchschaum einpacken.

Schließlich erhalten wir eine Hausmischung mit Soja, Schoko und Milchschaum.

*Einen viel besseren Weg, dekorierte Objekte zu erstellen, werden wir sehen, wenn wir das Factory- und das Builder-Entwurfsmuster behandeln.

Und jetzt lassen wir die Bestellungen ausführen:

```
Datei Bearbeiten Fenster Hilfe FlöckchenInMeinemKaffee
%java SternbackKaffee
Espresso 1.99 €
Dunkle Röstung, Schoko, Schoko, Milchschaum 1.49 €
Hausmischung, Soja, Schoko, Milchschaum 1.34 €
%
```

Es gibt keine Dummen Fragen

F: Ich mache mir etwas Sorgen um Code, der auf eine bestimmte konkrete Komponente – beispielsweise Hausmischung – prüft und dann irgendetwas macht, wie einen Rabatt berechnen. Nachdem ich Hausmischung mit Dekorierern eingepackt habe, funktioniert so etwas nicht mehr.

A: Stimmt genau. Wenn Sie mit Code arbeiten, der auf den Typ einer konkreten Komponente angewiesen ist, zerbrechen Dekorierer diesen Code. Solange Sie nur Code auf Basis des abstrakten Komponententyps schreiben, bleibt die Verwendung von Dekorierern für Ihren Code transparent. Aber sobald Sie anfangen, Code auf Basis konkreter Komponenten zu schreiben, müssen Sie das Design Ihrer Anwendung und Ihren Einsatz von Dekorierern überdenken.

F: Könnte es nicht schnell passieren, dass ein Client eines Getränks bei einem Dekorierer hängen bleibt, der nicht der äußerste Dekorierer ist? Könnte man nicht leicht Code schreiben, der bei einer dunklen Röstung mit Schoko, Soja und Milchschaum am Ende eine Referenz auf Soja statt auf Milchschaum hat, was dazu führen würde, dass Milchschaum in die Bestellung nicht eingeschlossen wird?

A: Natürlich könnte man argumentieren, dass man bei der Verwendung des Decorator-Musters mehr Objekte verwalten muss und dass deswegen eine höhere Wahrscheinlichkeit besteht, dass Programmierfehler die Art von Problemen verursachen, von denen Sie sprechen. Üblicherweise werden Dekorierer allerdings mithilfe anderer Muster wie Factory und Builder erstellt. Wenn wir diese Muster behandelt haben, werden Sie sehen, dass die Erstellung der konkreten Komponente mit ihrem Dekorierer »gut gekapselt« ist und nicht zu derartigen Problemen führt.

F: Können Dekorierer Kenntnis von den anderen Dekorierern in der Kette haben? Nehmen wir an, ich hätte gern, dass meine getBeschreibung()-Methode »Milchschaum, Doppel-Schoko« statt »Schoko, Milchschaum, Schoko« ausgibt. Das würde erfordern, dass mein äußerster Dekorierer alle Dekorierer kennt, die er einpackt.

A: Dekorierer sollen den Objekten, die sie einpacken, Verhalten hinzufügen. Wenn Sie beginnen, auf mehrere Schichten in der Dekoriererkette zu blicken, dann strecken Sie Decorator über seinen eigentlichen Zweck. Trotzdem sind solche Sachen möglich. Stellen Sie sich beispielsweise einen ZutatPrettyPrint-Dekorierer vor, der die resultierende Beschreibung parst und »Schoko, Milchschaum, Schoko« als »Milchschaum, Doppel-Schoko« ausgeben kann. getBeschreibung() könnte natürlich auch eine ArrayList mit Beschreibungen zurückliefern, um das zu erleichtern.

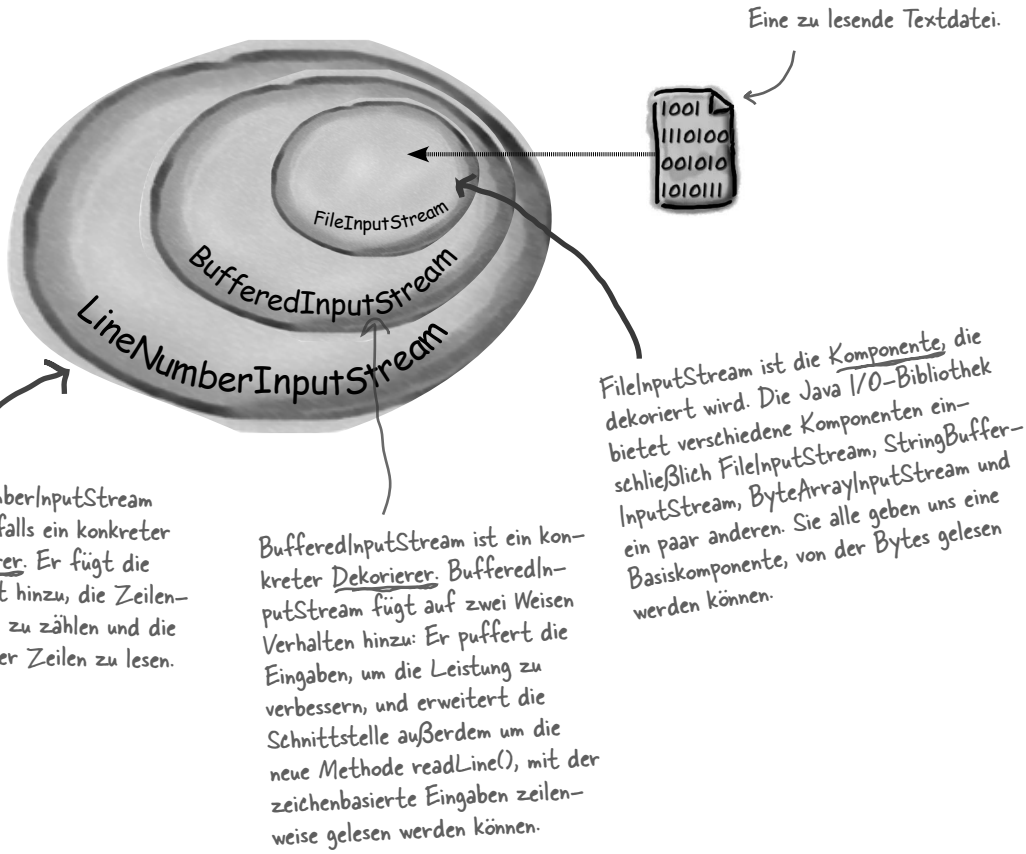
Spitzen Sie Ihren Bleistift

Unsere Freunde von Sternback haben ihren Speisekarten Größen hinzugefügt. Sie können Kaffee jetzt in den Größen Tall, Grande und Venti bestellen (Übersetzung: Klein, Mittel und Groß). Sternback betrachtete das als eine innere Komponente der Kaffee-Klasse. Sie haben der Klasse Getränk also zwei Methoden hinzugefügt: setGröße() und getGröße(). Sie möchten außerdem, dass Preise für die Zutaten größenabhängig sind. Soja soll also beispielsweise für Tall, Grande und Venti 10, 15 respektive 20 Cent kosten.
Wie würden Sie die Dekorierer-Klassen ändern, um dieser Anforderungsänderung zu entsprechen?

```
public abstract class Getränk {
  public enum Größe { TALL, GRANDE, VENTI };
  Größe größe = Größe.TALL;
  String beschreibung = "Unbekanntes Getränk";
  public String getBeschreibung() {
    return beschreibung;
  }
  public void setGröße(Größe größe) {
    this.größe = größe;
  }
  public Größe getGröße() {
    return this.größe;
  }
  public abstract double preis();
}
```

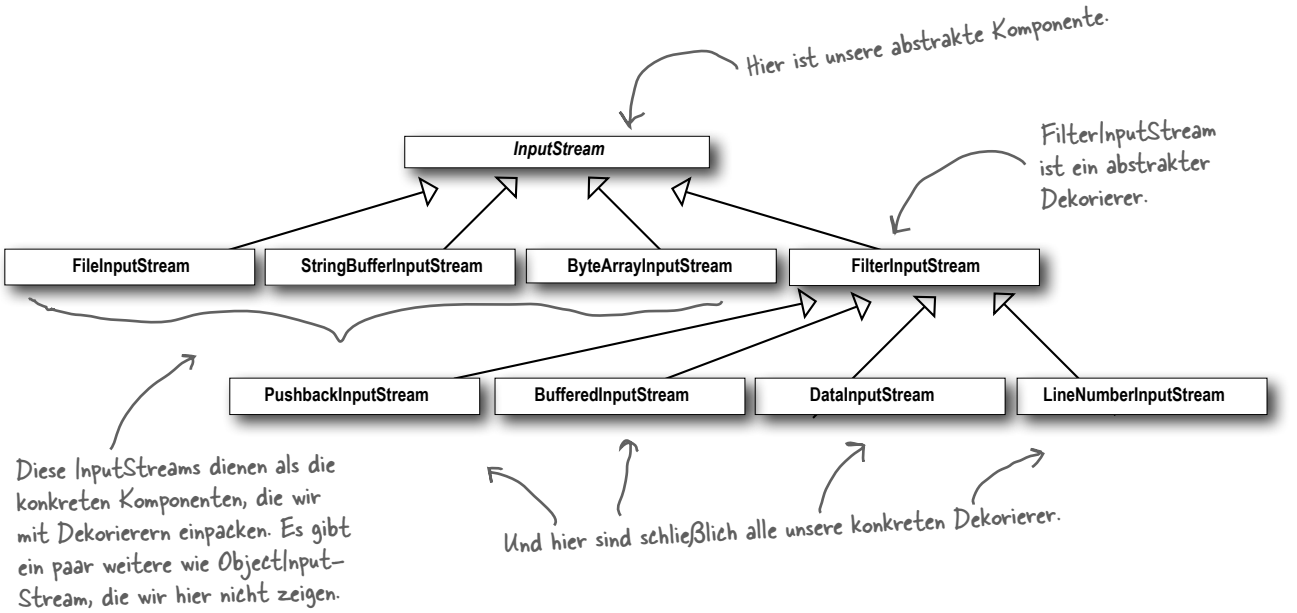
Dekorierer aus der Praxis: Java I/O

Die Vielzahl der Klassen im java.io-Package ist *überwältigend*. Glauben Sie nicht, Sie wären der Einzige, der beim ersten (und zweiten und dritten) Blick auf diese API »Wow« gesagt hat. Aber jetzt, da Sie das Decorator-Muster kennen, sollten Sie die I/O-Klassen besser verstehen, weil das java.io-Package zu einem Großteil auf dem Decorator-Muster basiert. Hier sehen Sie einen typischen Satz von Objekten, die Dekorierer verwenden, um Funktionalitäten für das Lesen aus einer Datei hinzuzufügen:



BufferedInputStream und ***LineNumber***InputStream erweitern beide ***Filter***InputStream, die als abstrakte Dekorierer-Klasse dient.

Die java.io-Klassen dekorieren



Sie sehen, dass sich das nicht so sehr vom Sternback-Design unterscheidet. Sie sollten gut vorbereitet dazu in der Lage sein, die Dokumentation der java.io-API durchzusehen und Dekorierer für die verschiedenen *Input*-Streams zu erstellen.

Sie werden feststellen, dass die *Output*-Streams das gleiche Design haben. Und wahrscheinlich haben Sie auch schon festgestellt, dass die Reader/Writer-Streams (für zeichenbasierte Daten) das Design der Stream-Klassen fast widerspiegeln (abgesehen von ein paar Unterschieden und Inkonsistenzen, aber doch ähnlich genug, um herauszufinden, was vor sich geht).

Aber Java I/O zeigt auch die *Nachteile* des Decorator-Musters auf: Designs, die dieses Muster einsetzen, führen oft zu vielen kleinen Klassen, die die Entwickler erschlagen können, die versuchen, die Decorator-basierte API zu verwenden. Aber jetzt, da Sie wissen, wie Decorator funktioniert, verlieren Sie sicher nicht mehr so leicht den Überblick, wenn Sie die Decorator-lastige API eines anderen verwenden und dazu in der Lage sind herauszufinden, wie die Klassen organisiert sind, damit Sie sie problemlos dekorieren können, um das Verhalten zu erreichen, das Sie wünschen.

Einen eigenen I/O-Dekorierer schreiben

Gut, jetzt kennen Sie das Decorator-Muster, und Sie haben das I/O-Klassendiagramm gesehen. Sie sollten eigentlich bereit sein, Ihren eigenen Input-Dekorierer zu schreiben.

Wie wäre es damit: Schreiben Sie einen Dekorierer, der alle Großbuchstaben im Eingabestrom in Kleinbuchstaben konvertiert. Anders gesagt: Wird der Satz »Ich kenne das Decorator-Muster und bin deswegen EIN KÖNIG!« gelesen, wandelt Ihr Dekorierer das in »ich kenne das decorater-muster und bin deswegen ein König!« um.

Vergessen Sie nicht, java.io zu importieren (wird nicht gezeigt).

Erweitern Sie zuerst FilterInputStream, den abstrakten Dekorierer für alle InputStreams.

```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int ergebnis = super.read(b, offset, len);
        for (int i = offset; i < offset+ergebnis; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return ergebnis;
    }
}
```

Jetzt müssen wir zwei read()-Methoden implementieren. Diese erwarten ein Byte (oder ein Array von Bytes) und konvertieren jedes Byte (das ein Zeichen repräsentiert) in Kleinbuchstaben, wenn es einen Großbuchstaben darstellt.

ERINNERUNG: Wir geben in den Code-Listings keine import- und package-Anweisungen an. Holen Sie sich den vollständigen Quellcode von unserer Website. Sie finden die URL auf Seite xxxi in der Einführung.

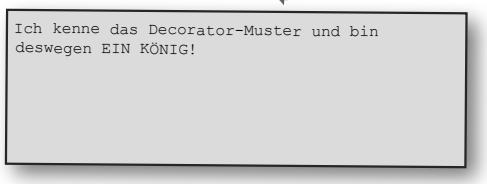
Den neuen Java I/O-Dekorierer testen

Schreiben Sie Test-Code, um den I/O-Dekorierer zu testen:

```
public class EingabeTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
    }
}
```

Richten Sie den FileInputStream ein und dekorieren Sie ihn erst mit einem BufferedInputStream und dann mit unserem brandneuen LowerCaseInput-Stream-Filter.

Nutzen Sie einfach den Stream, um die Zeichen bis zum Ende der Datei zu lesen und währenddessen auszugeben.



Datei test.txt

Diese Datei müssen Sie erstellen.

Setzen Sie es in Bewegung:

```
Datei Bearbeiten Fenster Hilfe DekoriererRegel
% java EingabeTest
ich kenne das decorater-muster und bin deswegen ein könig!
%
```

Muster unter der Lupe

Interview der Woche: Geständnis eines Dekorierers

Von Kopf bis Fuß: Herzlich willkommen, Decorator-Muster. Man hört, dass Sie in letzter Zeit ziemlich niedergeschlagen waren?

Decorator: Stimmt. Ich weiß, dass die Welt in mir ein berühmtes Entwurfsmuster sieht, aber, wissen Sie, auch ich habe meine Probleme wie alle anderen auch.

VKbF: Möchten Sie vielleicht einige Ihrer Probleme mit uns teilen?

Decorator: Sicher. Ähm ... Sie wissen, dass ich die Macht habe, Entwürfen Flexibilität hinzuzufügen. Das zumindest steht fest. Aber ich habe auch eine *dunkle Seite*. Manchmal, wissen Sie, kann ich einem Entwurf viele kleine Klassen hinzufügen, und das führt gelegentlich zu Entwürfen, die für andere nicht so leicht zu verstehen sind.

VKbF: Können Sie uns vielleicht ein Beispiel geben?

Decorator: Nehmen Sie beispielsweise die Java I/O-Bibliotheken. Die sind berüchtigt dafür, wie schwer sich die Leute am Anfang damit tun, sie zu verstehen. Würden sie die Klassen einfach als einen Satz von Wrappern um einen InputStream betrachten, wäre das Leben viel leichter.

VKbF: Das klingt aber nicht so schlimm. Sie sind immer noch ein tolles Muster. Das zu verbessern ist doch nur eine Frage der allgemeinen Bildung des öffentlichen Verständnisses, oder?

Decorator: Ich fürchte, dass das nicht alles ist. Ich habe außerdem Typisierungsprobleme. Es ist doch so: Manchmal nehmen die Leute einfach etwas Client-Code, der sich auf bestimmte Typen stützt, und führen Dekorierer ein, ohne die Sache ordentlich zu durchdenken. Eine der tollen Sachen bei mir ist natürlich, ***dass man Dekorierer in der Regel transparent einfügen kann und der Client nie erfahren muss, dass er mit einem Dekorierer zu tun hat***. Aber wie ich schon sagte, ist mancher Code von bestimmten Typen abhängig, und wenn Sie dann versuchen, Dekorierer einzufügen, macht es einfach bumm! Da passieren die schlimmsten Dinge.

VKbF: Ja, ich glaube, jeder versteht, dass man beim Einfügen von Dekorierern sehr vorsichtig sein muss. Ich glaube nicht, dass das ein Grund ist, niedergeschlagen zu sein.

Decorator: Weiß ich. Ich versuche auch, es nicht zu sein. Aber ich habe auch noch das Problem, dass die Einführung von Dekorierern die Komplexität des Codes wachsen lassen kann, der benötigt wird, um die Komponente zu instantiieren. Wenn Sie Dekorierer einsetzen, müssen Sie nicht nur die Komponente instantiieren, sondern sie auch noch mit wer weiß wie vielen Dekorierern einpacken.

VKbF: In der kommenden Woche werden hier die Factory- und Builder-Muster zum Interview sitzen – ich habe gehört, dass die in dieser Hinsicht hilfreich sein können?

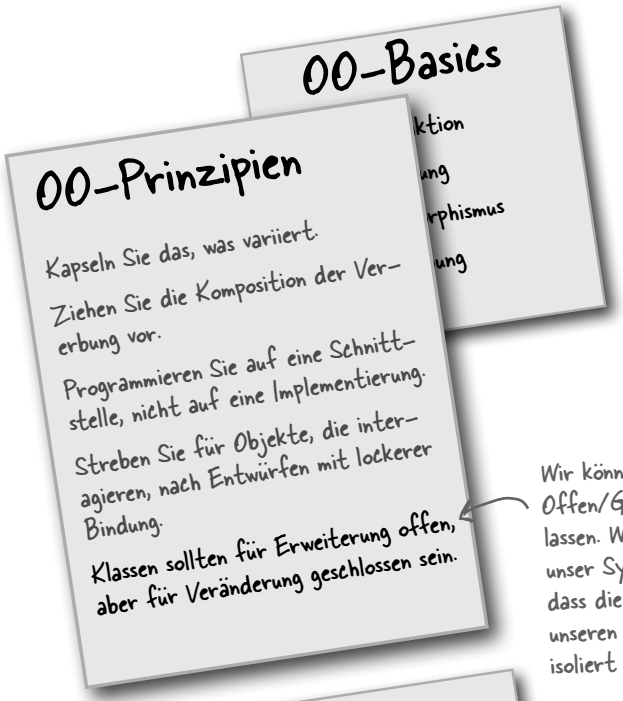
Decorator: Das ist richtig. Ich sollte mit den Typen öfter reden.

VKbF: Trotz allem werden Sie für uns alle ein tolles Muster bleiben, weil Sie für flexible Entwürfe sorgen und dem Offen/Geschlossen-Prinzip treu bleiben. Also Kopf hoch und denken Sie positiv!

Decorator: Ich tu mein Bestes, vielen Dank.

Werkzeuge für Ihren Design-Werkzeugkasten

Sie haben ein weiteres Kapitel verdaut und ein neues Prinzip sowie ein neues Muster in Ihrem Werkzeugkasten.



Punkt für Punkt

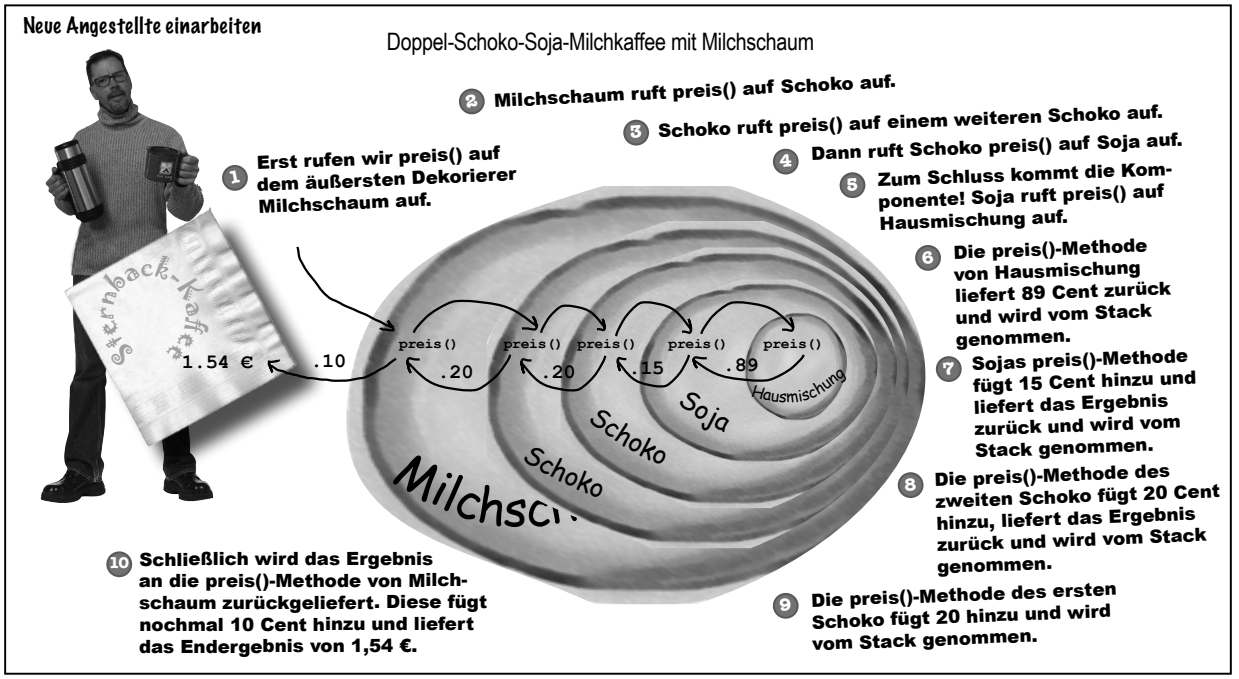
- Vererbung ist eine Form von Erweiterung, aber nicht notwendigerweise der beste Weg, um Ihren Entwürfen Flexibilität zu verleihen.
- Unsere Entwürfe sollten die Erweiterung von Verhalten ermöglichen, ohne dass dazu bestehender Code geändert werden müsste.
- Oft können Komposition und Delegierung verwendet werden, um zur Laufzeit neue Verhalten hinzuzufügen.
- Für die Erweiterung von Verhalten bietet das Decorator-Muster eine Alternative zur Ableitung von Unterklassen.
- Das Decorator-Muster schließt einen Satz von Dekorierer-Klassen ein, die verwendet werden, um konkrete Komponenten einzupacken.
- Dekorierer-Klassen spiegeln den Typ der Komponente wider, die sie dekorieren. (Sie haben sogar tatsächlich den gleichen Typ wie die Komponente, die sie dekorieren, entweder durch Vererbung oder durch die Implementierung eines Interface.)
- Dekorierer ändern das Verhalten der Komponenten, indem sie vor und/oder nach (oder auch anstelle von) Methodenaufrufen auf der Komponente neue Funktionalitäten hinzufügen.
- Sie können eine Komponente mit einer beliebigen Zahl von Dekorierern einpacken.
- Dekorierer sind für die Clients der Komponente üblicherweise transparent, außer wenn sich der Client auf den konkreten Typ der Komponente stützt.
- Dekorierer können in Ihren Entwürfen zu vielen kleinen Objekten führen, und eine übermäßige Verwendung kann den Code unübersichtlich machen.

Spitzen Sie Ihren Bleistift

Lösung

Schreiben Sie die preis()-Methoden für die folgenden Klasse (Pseudo-Java reicht). Hier ist unsere Lösung:

```
public class Getränk {

  // Deklarieren Sie die Instanzvariablen milchPreis,
  // sojaPreis, schokoPreis und milchschaumPreis sowie
  // Getter- und Setter-Methoden für Milch, Soja, Schoko
  // und Milchschaum.

  public float preis() {

    float zutatenPreis = 0.0;
    if (hasMilch()) {
      zutatenPreis += milchPreis;
    }
    if (hasSoja()) {
      zutatenPreis += sojaPreis;
    }
    if (hasSchoko()) {
      zutatenPreis += schokoPreis;
    }
    if (hasMilchschaum()) {
      zutatenPreis += milchschaumPreis;
    }
    return zutatenPreis;
  }
}



public class DunkleRöstung extends Getränk {

  public DunkleRöstung() {
    beschreibung = "Hervorragende dunkle Röstung ";
  }

  public float preis() {

    return 1.99 + super.preis();

  }
}
```


Neue Angestellte einarbeiten
Doppel-Schoko-Soja-Milchkaffee mit Milchschaum
Sternbach-Kaffee
1.54 €
.10
preis()
.20
preis()
.20
preis()
.15
preis()
.89
preis()
Hausmischung
Soja
Schoko
Schoko
Milchsch
1 Erst rufen wir preis() auf dem äußersten Dekorierer Milchschaum auf.
2 Milchschaum ruft preis() auf Schoko auf.
3 Schoko ruft preis() auf einem weiteren Schoko auf.
4 Dann ruft Schoko preis() auf Soja auf.
5 Zum Schluss kommt die Komponente! Soja ruft preis() auf Hausmischung auf.
6 Die preis()-Methode von Hausmischung liefert 89 Cent zurück und wird vom Stack genommen.
7 Sojas preis()-Methode fügt 15 Cent hinzu und liefert das Ergebnis zurück und wird vom Stack genommen.
8 Die preis()-Methode des zweiten Schoko fügt 20 Cent hinzu, liefert das Ergebnis zurück und wird vom Stack genommen.
9 Die preis()-Methode des ersten Schoko fügt 20 hinzu und wird vom Stack genommen.
10 Schließlich wird das Ergebnis an die preis()-Methode von Milchschaum zurückgeliefert. Diese fügt nochmal 10 Cent hinzu und liefert das Endergebnis von 1,54 €.

Spitzen Sie Ihren Bleistift

Lösung

Unsere Freunde von Sternback haben ihren Speisekarten Größen hinzugefügt. Sie können Kaffee jetzt in den Größen Tall, Grande und Venti bestellen (Übersetzung: klein, mittel und groß). Sternback betrachtete das als eine innere Komponente der Kaffee-Klasse. Sie haben der Klasse Getränk also zwei Methoden hinzugefügt: setGröße() und getGröße(). Sie möchten außerdem, dass Preise für die Zutaten größenabhängig sind. Soja soll also beispielsweise für Tall, Grande und Venti 10, 15 respektive 20 Cent kosten. Wie würden Sie die Dekorierer-Klassen ändern, um dieser Anforderungsänderung zu entsprechen?

```
public abstract class ZutatDekorierer extends Getränk {
  public Getränk getränk;
  public abstract String getBeschreibung();

  public Größe getGröße() {
    return getränk.getGröße();
  }
}

public class Soja extends ZutatDekorierer {

  public Soja(Getränk getränk) {
    this.getränk = getränk;
  }

  public String getBeschreibung() {
    return getränk.getBeschreibung() + ", Soja";
  }

  public double preis() {
    double preis = getränk.preis();
    if (getränk.getGröße() == Getränk.TALL) {
      preis += .10;
    } else if (getränk.getGröße() == Getränk.GRANDE) {
      preis += .15;
    } else if (getränk.getGröße() == Getränk.VENTI) {
      preis += .20;
    }
    return preis;
  }
}
```

Wir haben die getränk-Instanzvariable in den ZutatDekorierer verschoben und den Dekorierern eine getGröße()-Methode hinzugefügt, die einfach die Größe des Getränks zurückliefert.

Hier holen wir uns die Größe (die sich bis zum konkreten Getränk durchzieht) und fügen dann die entsprechenden Kosten hinzu.