3 Das Decorator-Muster

Objekte dekorieren

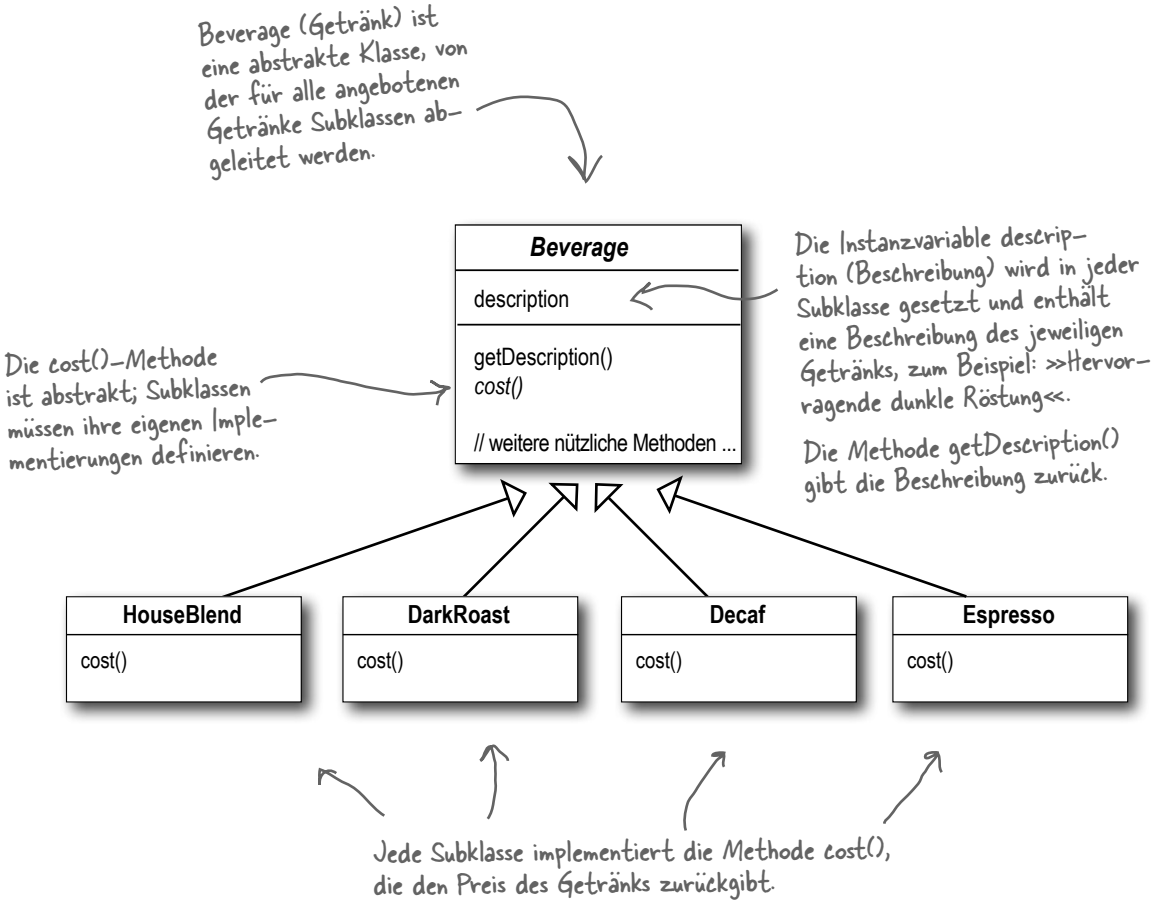
Nennen wir dieses Kapitel einfach »Vererbst du noch, oder designst du schon?«. Hier werfen wir einen weiteren Blick auf das typische Überstrapazieren von Vererbung. Sie werden lernen, wie Sie Ihre Klassen zur Laufzeit mit einer Form der Objektkomposition dekorieren können. Warum? Sobald Sie die Dekoration 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

Starbuzz Coffee hat sich einen Namen als am schnellsten wachsende Kaffeehauskette gemacht. Sie finden die Filialen nicht nur an der nächsten Straßenecke, sondern auch gegenüber.

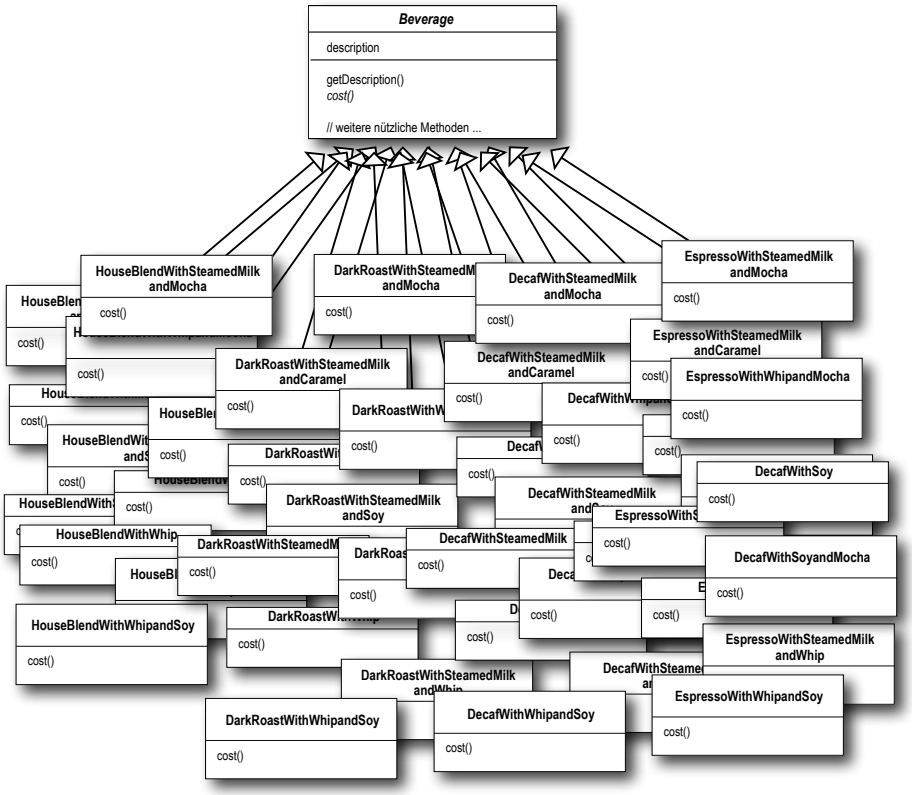
Wegen des rapiden Wachstums muss das Bestellsystem dringend aktualisiert werden, um mit dem wachsenden Getränkeangebot Schritt halten zu können.

Bei der Eröffnung sahen die Klassen etwa so aus:



Zum Kaffee kann man sich eine Vielzahl an Zutaten wie heiße Milch, Soja und Mocha (auch als Schokolade bekannt) bestellen, auf die zum Schluss noch Milchschaum (Whipped Milk) kommt. Natürlich berechnet Starbuzz die Zutaten extra, sodass sie alle ins Bestellsystem aufgenommen werden müssen.

Hier der erste Versuch ...



Jede cost()-Methode berechnet den Preis des Kaffees und der jeweiligen Zutaten einer Bestellung.

KOPF-NUSS

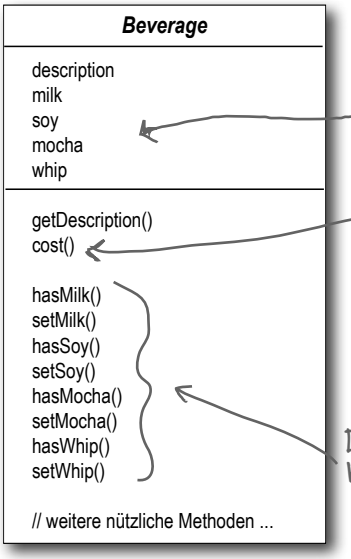
Offensichtlich hat sich Starbuzz einen wartungstechnischen Albtraum geschaffen. Was passiert, wenn der Milchpreis steigt? Was, wenn das Angebot um ein Karamell-Topping erweitert wird?

Sehen wir mal vom Wartungsproblem ab, welche unserer bisher behandelten Entwurfsprinzipien werden hier verletzt?

Tipp: Zwei werden ziemlich grundlegend verletzt!

Das ist doch blöd. Wofür brauchen wir diese ganzen Klassen? Können wir nicht einfach Instanzvariablen und Vererbung in der Superklasse nutzen, um den Überblick über die Zutaten zu behalten?

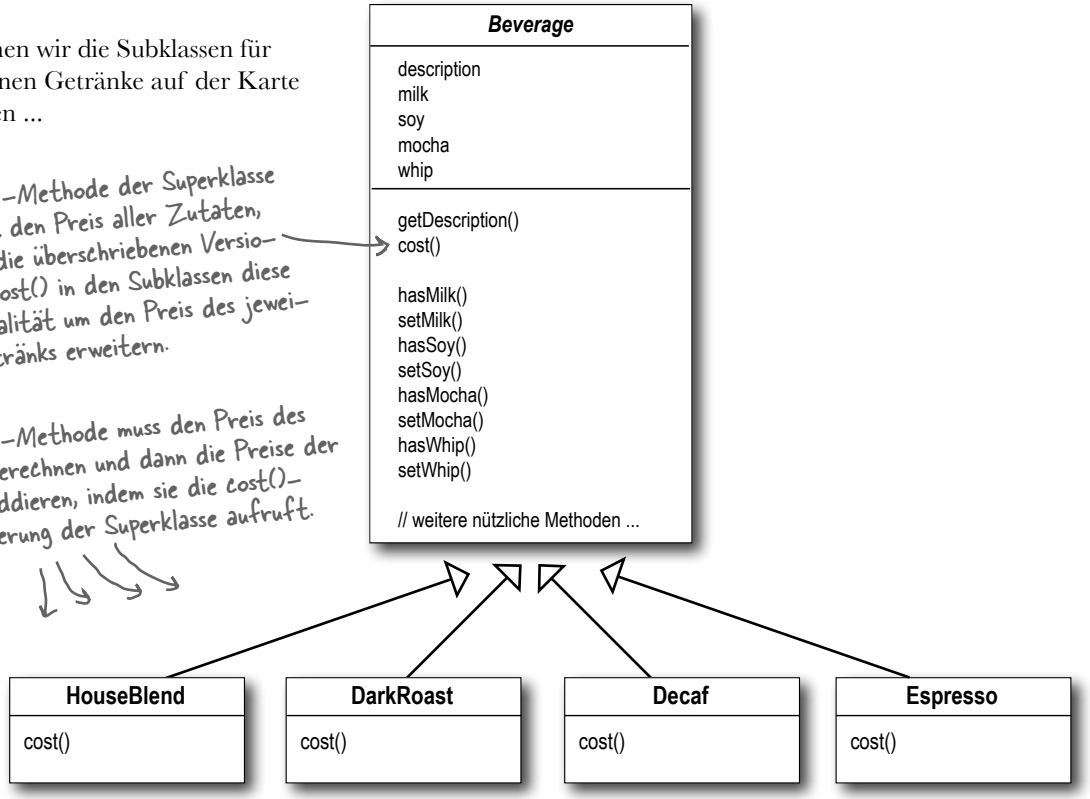
Wir können es zumindest versuchen. Beginnen wir mit der Basisklasse Beverage (Getränk) und erweitern sie um Instanzvariablen, die angeben, ob jedes Getränk Milch (milk), Soja (soy), Schokolade (mocha) und Milchschaum (whip) enthält ...



Jetzt können wir die Subklassen für die einzelnen Getränke auf der Karte hinzufügen ...

Die cost()-Methode der Superklasse berechnet den Preis aller Zutaten, während die überschriebenen Versionen von cost() in den Subklassen diese Funktionalität um den Preis des jeweiligen Getränks erweitern.

Jede cost()-Methode muss den Preis des Getränks berechnen und dann die Preise der Zutaten addieren, indem sie die cost()-Implementierung der Superklasse aufruft.



Spitzen Sie Ihren Bleistift

Schreiben Sie die cost()-Methoden für die folgenden Klassen (Pseudo-Java reicht aus):

```
public class Beverage {
    public double cost() {









    }
}
```

```
public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Hervorragende dunkle Röstung";
    }
    public double cost() {









    }
}
```

Spitzen Sie Ihren Bleistift

Welche Anforderungen oder andere Faktoren können sich ändern und sich auf unseren Entwurf auswirken?

Bei Preisänderungen der Zutaten müssen wir unseren Code anpassen.

Bei neuen Zutaten müssen wir neue Methoden hinzufügen und die cost()–Methode der Superklasse anpassen.

Es könnten neue Getränke dazukommen. Für einige (Eistee?) sind die Zutaten vielleicht unpassend. Trotzdem würde die Subklasse Tea (Tee) Methoden wie hasWhip() (MitMilchschaum) erben.

Wie wir aus Kapitel 1 wissen, ist das eine schlechte Idee!

Was passiert, wenn ein Kunde einen doppelten Mocha haben möchte?

Sie sind dran:

Meisterin und Schüler

Meisterin: *Wir haben uns lange nicht mehr gesehen, Grashüpfer. Hast du gründlich über Vererbung meditiert?*

Schüler: *Ja, Meisterin. Ich habe erkannt, dass Vererbung zwar mächtig ist, aber nicht immer zu den flexibelsten und wartbarsten Designs führt.*

Meisterin: *Aha! Ich sehe, du machst Fortschritte. Also sage mir, mein Schüler, wie willst du Wiederverwendung ohne Vererbung erreichen?*

Schüler: *Meisterin, ich habe gelernt, dass Verhalten durch Komposition und Delegation zur Laufzeit »vererbt« werden kann.*

Meisterin: *Sprich weiter ...*

Schüler: *Wenn ich Verhalten über Subklassen vererbe, wird es bei der Kompilierung festgelegt. Außerdem müssen alle Subklassen das gleiche Verhalten erben. Wenn ich das Objektverhalten dagegen durch Komposition verändere, funktioniert das dynamisch während der Laufzeit.*

Meisterin: *Sehr gut. Du beginnst, die Macht der Komposition zu erkennen.*

Schüler: *Ja, mit dieser Technik kann ich Objekten mehrere neue Aufgaben geben, inklusive solcher, die die Entwickler der Superklasse übersehen haben. Und ich muss nicht einmal ihren Code anfassen!*

Meisterin: *Und was hast du über die Auswirkungen der Komposition auf die Wartbarkeit des Codes gelernt?*

Schüler: *Genau darauf wollte ich hinaus, Meisterin. Durch die dynamische Komposition von Objekten kann Funktionalität durch das Schreiben von neuem Code hinzugefügt werden, anstatt bestehenden zu verändern. Dadurch ist die Gefahr neuer Programmierfehler und ungewollter Nebenwirkungen im bestehenden Code deutlich kleiner.*

Meisterin: *Sehr gut. Das reicht für heute, Grashüpfer. Ich möchte, dass du weiter über dieses Thema meditierst ... Bedenke, Code sollte (Veränderungen gegenüber) geschlossen sein wie die Lotosblüte am Abend, jedoch offen (für Erweiterungen) wie die Lotosblüte am Morgen.*

Das Offen/Geschlossen-Prinzip

Und damit kommen wir zu einem der wichtigsten Entwurfsprinzipien überhaupt:

Entwurfsprinzip

Klassen sollten offen für Erweiterungen, aber geschlossen für Änderungen sein.

Kommen Sie rein, wir haben *geöffnet*. Erweitern Sie Ihre Klassen mit beliebigem neuem Verhalten, wenn Sie wollen. Sobald sich Ihre Bedürfnisse oder Anforderungen ändern (und das werden sie), erstellen Sie einfach Ihre eigenen Erweiterungen.

Leider *geschlossen*. So ist das eben. Wir haben viel Zeit gebraucht, um unseren Code korrekt und fehlerfrei hinzubekommen. Daher dürfen Sie ihn auch nicht ändern. Er muss für Änderungen geschlossen bleiben. Wenn Ihnen das nicht gefällt, können Sie gerne mit dem Geschäftsführer sprechen.

Unsere Klassen sollen leicht erweiterbar sein, damit neues Verhalten ohne Änderungen am bestehenden Code hinzugefügt werden kann. Aber was haben wir davon? Entwürfe, die widerstandsfähig gegen Veränderungen, aber flexibel genug sind, um neue Funktionalität zu akzeptieren, mit der auf geänderte Anforderungen reagiert werden kann.

Es gibt keine Dummen Fragen

F: Offen für Erweiterungen, aber geschlossen für Änderungen? Das klingt widersprüchlich. Wie kann ein Entwurf beides sein?

A: Sehr gute Frage. Am Anfang scheint das tatsächlich unvereinbar. Je weniger etwas modifizierbar ist, desto schwerer ist es schließlich auch zu erweitern, oder?
Tatsächlich gibt es ein paar schlaue OO-Techniken, die es ermöglichen, ein System zu erweitern, ohne den zugrunde liegenden Code dafür zu ändern. Stellen Sie sich ein Observer-Muster (siehe Kapitel 2) vor. Durch das Hinzufügen neuer Beobachter können wir das Subjekt jederzeit erweitern, ohne den Code des Subjekts anpassen zu müssen. Daneben gibt es aber noch andere OO-Entwurfstechniken zur Erweiterung des Verhaltens.

F: Okay, das Observer-Muster habe ich verstanden. Aber wie kann ich allgemein etwas entwerfen, das gleichzeitig erweiterbar und trotzdem für Veränderungen geschlossen ist?

A: Viele der Muster bieten uns über lange Zeit bewährte Entwürfe, die unseren Code vor Änderungen schützen, indem sie einen bestimmten Erweiterungsmechanismus bereitstellen. In diesem Kapitel werden Sie ein gutes Beispiel dafür sehen, wie man das Decorator-Muster einsetzt, um dem Offen/Geschlossen-Prinzip zu genügen.

F: Wie kann ich dafür sorgen, dass mein gesamter Code dem Offen/Geschlossen-Prinzip folgt?

A: Das ist normalerweise nicht möglich. Das Erstellen eines OO-Designs, das offen für Erweiterungen ist, ohne Codeänderungen vorzunehmen, ist anstrengend und zeitaufwendig. Normalerweise können wir es uns nicht leisten, alle Teile unseres Entwurfs festzuzurren (das wäre vermutlich auch eine ziemliche Verschwendung). Folgt man dem Offen/Geschlossen-Prinzip, führt das normalerweise zu neuen Abstraktionsebenen, die unseren Code weiter verkomplizieren. Daher ist es besser, wenn Sie sich auf die Bereiche konzentrieren, die sich in Ihren Entwürfen am ehesten ändern, und wenden die Prinzipien darauf an.

F: Woher weiß ich, welche Veränderungen am wichtigsten sind?

A: Zum Teil ist das eine Frage der Erfahrung im Entwerfen von OO-Systemen, zum Teil eine Frage der Beherrschung Ihres jeweiligen Arbeitsgebiets. Die Beschäftigung mit anderen Beispielen wird Ihnen helfen, die Bereiche Ihrer Entwürfe zu identifizieren, die Änderungen unterworfen sein werden.

Obwohl es wie ein Widerspruch klingt, gibt es Techniken, mit denen der Code ohne direkte Änderungen erweitert werden kann.

Entscheiden Sie sorgfältig, welche Codeteile erweitert werden müssen: Wenden Sie das Offen/Geschlossen-Prinzip NICHT ÜBERALL an. Sonst kann es leicht zu Verschwendung und zu komplexem, schwer verständlichem Code führen.

Wir stellen vor: das Decorator-Muster

Inzwischen wissen wir, dass die Darstellung unserer Getränke und Zutaten mithilfe von Vererbung nicht sehr gut funktioniert hat. Entweder explodiert die Zahl der Klassen bei einem sehr starren Design, oder wir erweitern die Basisklasse um Funktionalität, die einige Subklassen gar nicht brauchen.

Um das zu vermeiden, gehen wir anders vor: Wir beginnen mit einem Getränk und »dekorieren« es zur Laufzeit mit den Zutaten. Will der Kunde beispielsweise die »dunkle Röstung« (Dark Roast) mit Schokolade (Mocha) und Milchschaum (Whip), können wir folgendermaßen vorgehen:

1. **Mit einem DarkRoast-Objekt beginnen.**
2. **Mit einem Mocha-Objekt dekorieren.**
3. **Mit einem Whip-Objekt dekorieren.**
4. **Die cost()-Methode aufrufen und sich auf die Delegation verlassen, um den Gesamtpreis der Zutaten zu berechnen.**

So weit, so gut, aber wie »dekoriert« man ein Objekt, und was hat die Delegation damit zu tun? Tipp: Stellen Sie sich die Dekorator-Objekte als »Wrapper« vor. Sehen wir mal, wie das funktioniert ...

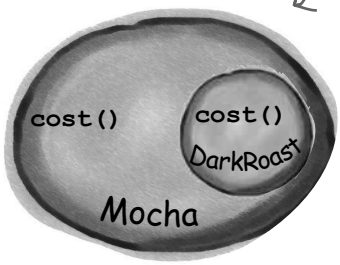
Eine Getränkebestellung mit Dekoratoren aufbauen

1 Wir beginnen mit unserem DarkRoast-Objekt.

Wie wir wissen, erbt DarkRoast von Beverage und besitzt eine cost()-Methode, die den Getränkepreis berechnet.

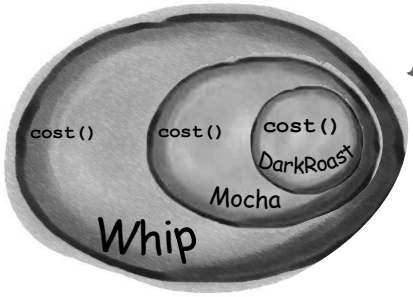
2 Der Kunde möchte Mocha, also erstellen wir ein passendes Objekt und packen das DarkRoast-Objekt darin ein.

Das Mocha-Objekt ist ein Dekorator. Sein Typ spiegelt das dekorierte Objekt, in diesem Fall Beverage. (Mit >>spiegeln<< meinen wir, es hat den gleichen Typ.)



Mocha hat ebenfalls eine cost()-Methode. Durch Polymorphismus können wir jedes in Mocha verpackte Beverage auch als Beverage behandeln (weil Mocha ein Subtyp von Beverage ist).

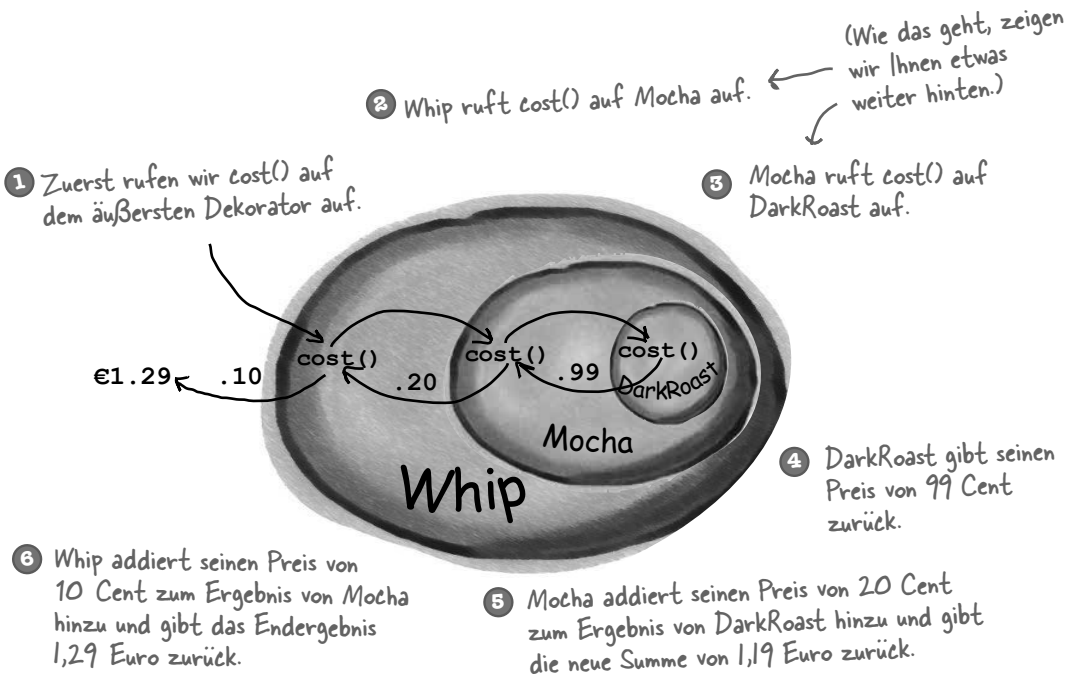
3 Außerdem will der Kunde Milchschaum. Also erstellen wir einen Whip-Dekorator und packen Mocha darin ein.



Whip ist ein Dekorator. Also spiegelt er ebenfalls den Typ von DarkRoast und enthält eine cost()-Methode.

Ein in Mocha und Whip verpackter (>>wrapped<<) DarkRoast ist also weiterhin ein Beverage (Getränk). Wir können daher alles damit machen, was auch mit DarkRoast möglich ist, zum Beispiel seine cost()-Methode aufrufen.

❹ Jetzt können wir den Preis berechnen. Hierfür rufen wir cost() am äußersten Dekorator (Whip) auf. Whip delegiert die Preisberechnung an die von ihm dekorierten Objekte und so weiter. Das funktioniert so:



Das wissen wir bisher über Dekoratoren:

- Dekoratoren besitzen den gleichen Supertyp wie die von ihnen dekorierten Objekte.
- Sie können ein Objekt mit mehreren Dekoratoren umgeben.
- Da der Dekorator den gleichen Supertyp hat wie das von ihm dekorierte Objekt, können wir ein dekoriertes Objekt anstelle des »verpackten« Originalobjekts weitergeben.
- Der Dekorator fügt sein eigenes Verhalten vor und/oder nach der Delegation an das von ihm dekorierte Objekt hinzu, um den Rest der Aufgabe zu erfüllen.

Der wichtigste Punkt!

- Objekte können jederzeit dekoriert werden. Das heißt, Objekte können zur Laufzeit dynamisch mit beliebig vielen Dekoratoren versehen werden.

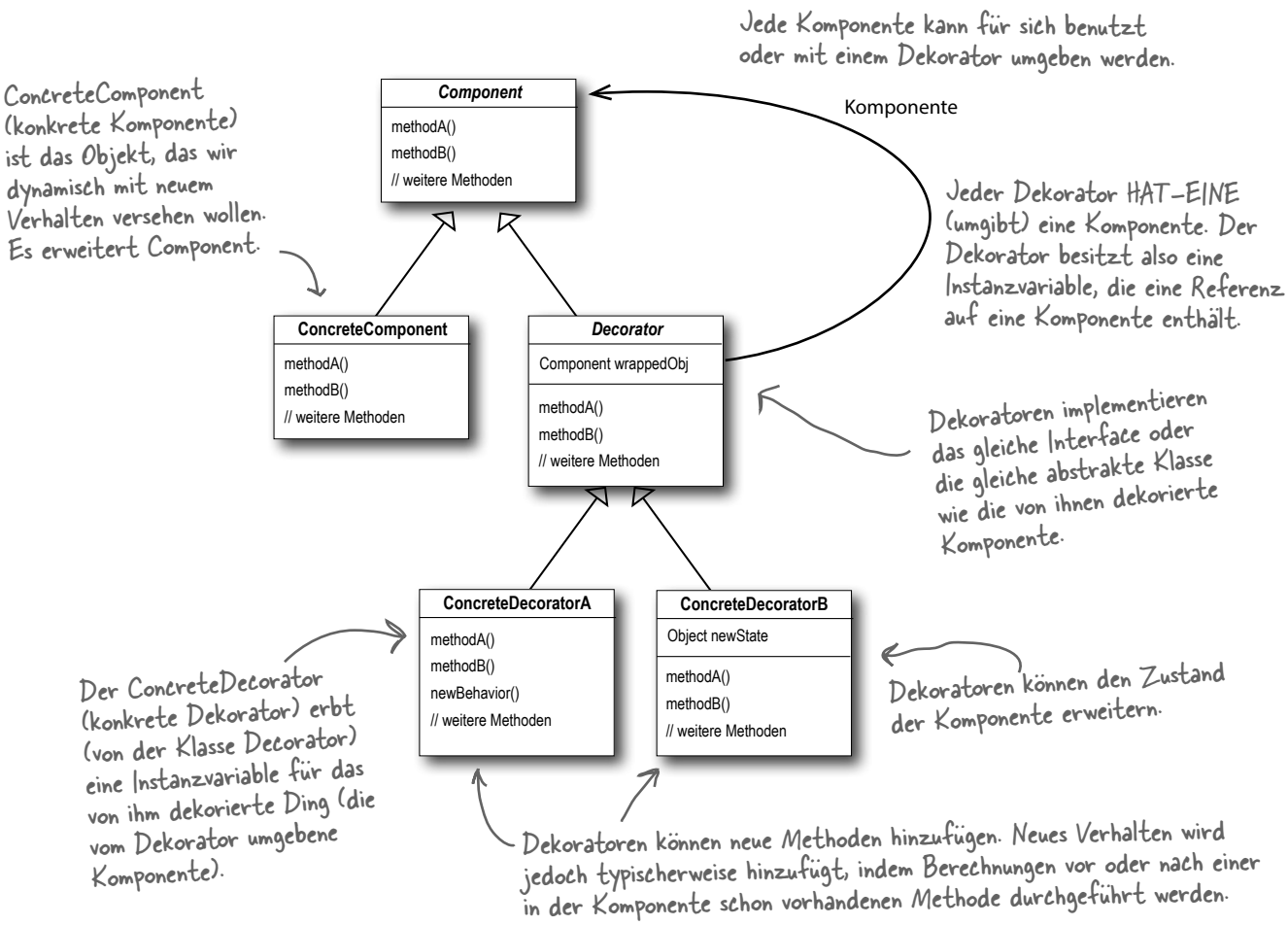
Um herauszufinden, wie das wirklich funktioniert, sehen wir uns die Definition des Decorator-Musters an und schreiben etwas Code.

Die Definition des Decorator-Musters

Sehen wir uns zuerst die Beschreibung des Decorator-Musters an:

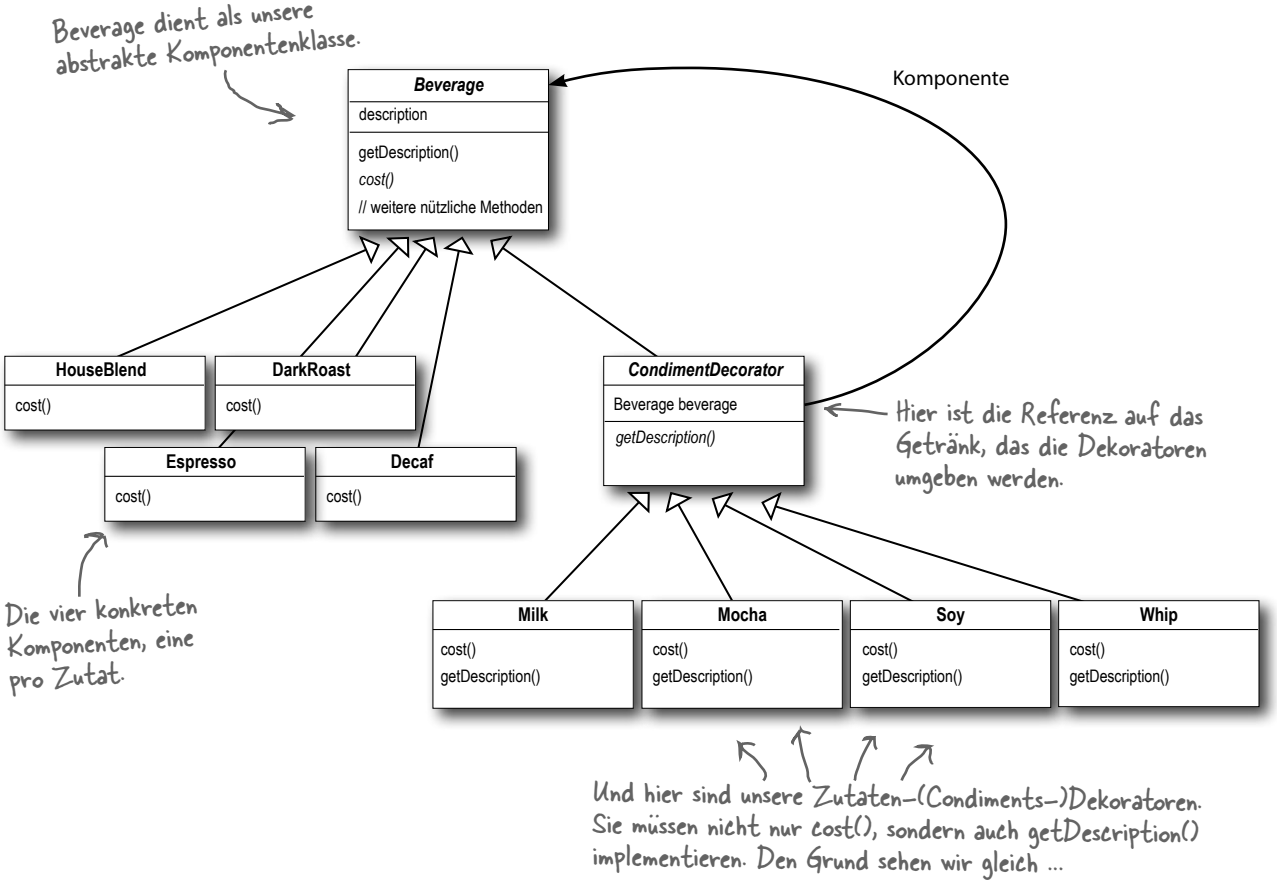
> **Das Decorator-Muster** erweitert ein Objekt dynamisch um weitere Verantwortlichkeiten. Dekoratoren bieten eine flexible Alternative zu Subklassen für die Erweiterung der Funktionalität.

Das beschreibt zwar die *Rolle* des Decorator-Musters, es gibt uns aber nur wenig Informationen darüber, wie wir es auf unsere eigene Implementierung *anwenden* können. Sehen wir uns hierzu das Klassendiagramm an, das etwas informativer ist (auf der folgenden Seite wenden wir die gleiche Struktur auf unser Getränkeproblem an.)



Unsere Getränke dekorieren

Jetzt können wir unsere Starbuzz-Getränke anhand des Decorator-Musters neu aufbauen ...

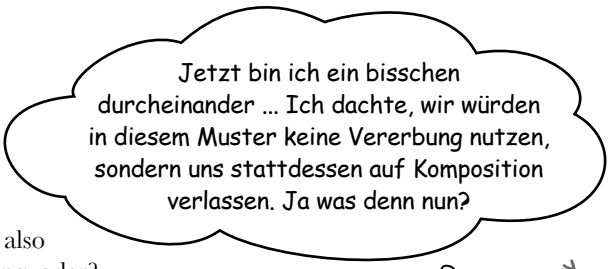


KOPF-NUSS

Bevor wir weitermachen, denken Sie darüber nach, wie Sie die cost()-Methoden für die Kaffees und die Zutaten implementieren würden. Überlegen Sie auch, wie Sie die getDescription()-Methode der Zutaten implementieren können.

Bürogespräch

Kleine Verwirrung hinsichtlich Vererbung und Komposition



Sue: Wie meinst du das?

Mary: Im Klassendiagramm erweitert der CondimentDecorator, also der Zutaten-Dekorator, die Klasse Beverage. Das ist doch Vererbung, oder?

Sue: Stimmt. Aber ich glaube, hier ist vor allem wichtig, dass die Dekoratoren den gleichen Typ haben wie die von ihnen dekorierten Objekte. Hier verwenden wir die Vererbung also, um *Übereinstimmung der Typen* zu erreichen, aber nicht, um das *Verhalten* zu übernehmen.

Mary: Ich verstehe, dass die Dekoratoren die gleiche »Schnittstelle« brauchen wie die von ihnen umgebenden Komponenten, weil sie ja deren Platz einnehmen sollen. Aber wo kommt das Verhalten denn dann ins Spiel?

Sue: Wenn wir einen Dekorator mit einer Komponente komponieren, fügen wir neues Verhalten hinzu. Dieses Verhalten erreichen wir aber nicht, indem wir aus einer Superklasse *erben*, sondern indem wir Objekte miteinander *komponieren*.

Mary: Also gut. Wir benutzen eine Subklasse der abstrakten Klasse Beverage, um den korrekten Typ zu bekommen, aber nicht um ihr Verhalten zu erben. Das Verhalten bekommen wir durch Komposition der Dekoratoren mit den Grundkomponenten und anderen Dekoratoren.

Sue: Das ist richtig.

Mary: Jetzt verstehe ich! Und weil wir Objektkomposition nutzen, erhalten wir viel mehr Flexibilität beim Mischen der Zutaten und Getränke. Ziemlich schick.

Sue: Ja, denn wenn wir Vererbung nutzen, kann unser Verhalten nur statisch in der Kompilierungsphase festgelegt werden. Das heißt, wir bekommen nur das Verhalten, das die Superklasse uns bietet oder das wir überschreiben. Durch die Komposition können wir Dekoratoren ganz nach Gusto mischen und zusammenstellen – und zwar zur *Laufzeit*.

Mary: Ich hab's. Wir können jederzeit neue Dekoratoren implementieren, um neues Verhalten hinzuzufügen. Hätten wir uns auf die Vererbung verlassen, müssten wir jedes Mal, wenn wir neues Verhalten brauchen, den bestehende Code ändern.

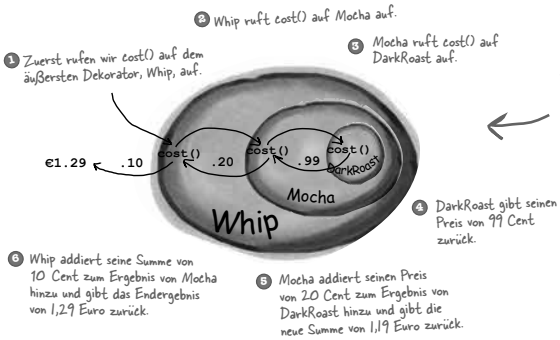
Sue: Genau.

Mary: Eine Frage habe ich noch: Wenn wir nur den Typ der Komponente erben müssen, warum benutzen wir dann kein Interface anstelle der abstrakten Klasse der Beverage-Klasse?

Sue: Erinnerst du dich noch? Als wir diesen Code bekommen haben, besaß Starbuzz schon eine abstrakte Beverage-Klasse. Traditionell gibt das Decorator-Muster eine abstrakte Komponente vor. Aber in Java könnten wir natürlich ein Interface nutzen. Allerdings versuchen wir, existierenden Code nach Möglichkeit nicht zu verändern. Wir müssen also nichts »reparieren«, wenn die abstrakte Klasse genauso gut funktioniert.

Neue Barista einarbeiten

Zeichnen Sie einmal, was passiert, wenn ein »doppelter Mocha Soy Latte mit Milchschaum« bestellt wird. Die korrekten Preise finden Sie in der Getränkekarte. Benutzen Sie für Ihre Zeichnung das Format, das wir ein paar Seiten weiter vorn verwendet haben.



Diese Abbildung galt dem Getränk »Dark Rost Mocha Whip«.

Spitzen Sie Ihren Bleistift

Hier kommt Ihre Zeichnung hin.

Starbuzz Coffee

Kaffees	
House Blend	0,89
Dark Roast	0,99
Decaf	1,05
Espresso	1,99
Zutaten	
Steamed Milk	0,10
Mocha	0,20
Soy	0,15
Whip	0,10

TIPP: Sie können einen »doppelten Mocha Soy Latte mit Milchschaum« machen, indem Sie die Hausmischung (House Blend), Sojamilch (Soy), zwei Portionen Schoko (Mocha) und Whip kombinieren!

Den Starbuzz-Code schreiben

Und jetzt können wir unseren Entwurf endlich in echten Code umsetzen.

Wir beginnen mit der Beverage-Klasse. Gegenüber dem Originalentwurf von Starbuzz sind keine Änderungen nötig. Schauen wir mal:

```
public abstract class Beverage {
    String description = "Unbekanntes Getränk";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage ist eine abstrakte Klasse mit den beiden Methoden getDescription() und cost().

getDescription() ist schon für uns implementiert. cost() müssen wir dagegen in den Subklassen implementieren.

Beverage ist ganz einfach. Die abstrakten Klassen für die Zutaten (die Dekoratoren) können wir auch gleich implementieren:

Zuerst müssen die Zutaten (Condiments) mit Beverage austauschbar sein. Also erweitern wir die Klasse Beverage.

```
public abstract class CondimentDecorator extends Beverage {
    Beverage beverage;
    public abstract String getDescription();
}
```

Hier ist das Getränk (Beverage), das mit Dekoratoren umgeben werden soll. Übrigens benutzen wir den Supertyp Beverage, um auf das Getränk (Beverage-Objekt) zu verweisen, damit der Dekorator jede Art von Getränk umgeben kann.

Außerdem müssen die Dekoratoren für die Zutaten die getDescription()-Methode reimplementieren. Auch hier werden wir den Grund gleich sehen ...

Getränke programmieren

Nachdem wir die Basisklasse fertig haben, können wir jetzt ein paar Getränke implementieren. Wir beginnen mit Espresso. Wir müssen eine Beschreibung für das jeweilige Getränk festlegen und die cost()-Methode implementieren.

Zuerst erweitern wir die Klasse Beverage. Schließlich handelt es sich hier um ein Getränk.

```
public class Espresso extends Beverage {

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

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

Für die Beschreibung definieren wir dies im Konstruktor der Klasse. Nicht vergessen: Die Instanzvariable description wird von Beverage geerbt.

Zum Schluss müssen wir noch den Preis eines Espressos berechnen. In dieser Klasse brauchen wir uns nicht um die Addition der Zutaten zu künmern, sondern müssen nur den Preis von 1,99 Euro zurückgeben.

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

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

Hier ist noch ein Getränk. Wir müssen nur die korrekte Beschreibung »House Blend Coffee« hinzufügen und den richtigen Preis von 89 Cent zurückgeben.

Starbuzz Coffee

Kaffees	
House Blend	0,89
Dark Roast	0,99
Decaf	1,05
Espresso	1,99
Zutaten	
Steamed Milk	0,10
Mocha	0,20
Soy	0,15
Whip	0,10

Die anderen zwei Beverage-Klassen (DarkRoast und Decaf) können Sie exakt auf die gleiche Weise erstellen.

Zutaten programmieren

Wenn Sie sich das Klassendiagramm für das Decorator-Muster noch einmal ansehen, werden Sie feststellen, dass wir unsere abstrakte Komponente (Beverage), unsere konkreten Komponenten (HouseBlend) und unseren abstrakten Dekorator (CondimentDecorator) bereits fertig haben. Jetzt müssen wir die konkreten Dekoratoren implementieren. Hier haben wir Mocha:

Mocha ist ein Dekorator, also erweitern wir CondimentDecorator.

Nicht vergessen: CondimentDecorator erweitert (extends) Beverage.

```
public class Mocha extends CondimentDecorator {

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return beverage.cost() + .20;
    }
}
```

Wir werden Mocha mit einer Referenz auf ein Beverage instanziieren.

Bedenken Sie, dass diese Klasse die Beverage-Instanzvariable erbt, die das Getränk enthält, das wir umgeben.

Diese Instanzvariable setzen wir auf dem Objekt, das wir umgeben. Hier übergeben wir das >>verpackte<< Getränk an den Konstruktor des Dekorators.

Unsere Beschreibung soll nicht nur das Getränk, zum Beispiel >>Dark Roast<<, ausgeben, sondern auch alle Zutaten, die es dekorieren (also beispielsweise: >>Dark Roast, Mocha<<). Daher delegieren wir zuerst an das Objekt, das wir dekorieren, um seine Beschreibung zu bekommen, und hängen ihr anschließend >>, Mocha<< an.

Jetzt müssen wir den Preis unseres Getränks mit Mocha berechnen. Zuerst delegieren wir den Aufruf an das dekorierte Objekt, damit es den Preis berechnen kann. Danach addieren wir den Preis für Mocha hinzu.

Auf der folgenden Seite werden wir das Getränk tatsächlich instanziieren und mit allen seinen Zutaten (Dekoratoren) umgeben, aber zuerst ...

Schreiben und kompilieren Sie den Code für die anderen Zutaten Sojamilch (Soy) und Milchschaum (Whip). Die brauchen Sie, um die Applikation fertigzustellen und zu testen.

Den Kaffee servieren

Glückwunsch. Endlich können wir uns zurücklehnen, ein paar Kaffees bestellen und das flexible Design bewundern, das wir mit dem Decorator-Muster geschaffen haben.

Hier etwas Code*, um die Bestellungen durchzuführen:

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " €" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " €" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " €" + beverage3.cost());
    }
}
```

Einen Espresso ohne Zutaten bestellen und seine Beschreibung und seinen Preis ausgeben.

Ein DarkRoast-Objekt erstellen.

Mit einem Mocha umgeben.

Mit einem zweiten Mocha umgeben.

Mit Milchschaum (Whip) umgeben.

Und zum Schluss noch einen House Blend mit Sojamilch, Mocha und Milchschaum.

* Eine deutlich bessere Möglichkeit für die Erstellung dekorierter Objekte werden wir sehen, sobald wir uns mit dem Factory- und dem Builder-Entwurfsmuster beschäftigen. Das Builder-Muster finden Sie im Anhang.

Jetzt können wir die Bestellungen ausführen:

```
Datei Bearbeiten Fenster Hilfe  WolkenInMeinemKaffee
% java StarbuzzCoffee
Espresso €1.99
Dark Roast Coffee, Mocha, Mocha, Whip €1.49
House Blend Coffee, Soy, Mocha, Whip €1.34
%
```

Es gibt keine Dummen Fragen

F: Ich mache mir Sorgen um Code, der auf eine bestimmte konkrete Komponente testet, wie etwa HouseBlend, und dann eine Aktion durchführt, zum Beispiel die Anwendung eines Rabatts. Sobald ich HouseBlend mit Dekoratoren umgeben habe, funktioniert das nicht mehr.

A: Das ist vollkommen richtig. Wenn sich Ihr Code auf den Typ einer konkreten Komponente verlässt, wird er durch die Dekoratoren nicht mehr funktionieren. Solange Sie Ihren Code nur auf Basis des abstrakten Komponententyps schreiben, bleibt die Verwendung der Dekoratoren für Ihren Code transparent. Sobald Sie dagegen beginnen, auf Basis konkreter Komponenten zu schreiben, sollten Sie Ihr Applikationsdesign und die Verwendung von Dekoratoren noch einmal überdenken.

F: Kann es nicht leicht passieren, dass ein Client eines Getränks bei einem Dekorator hängen bleibt, der nicht der äußerste Dekorator ist? Bei einem DarkRoast mit Mocha, Soja und Milchschaum ist es beispielsweise einfach, Code zu schreiben, der irgendwie eine Referenz auf Soja anstatt auf Milchschaum bekommt, was dazu führt, dass der Milchschaum nicht Teil der Bestellung ist.

A: Man könnte natürlich sagen, dass man bei der Verwendung des Dekorator-Musters mehr Objekte verwalten muss. Das könnte dazu führen, dass es durch Programmierfehler zu der von Ihnen beschriebenen Art von Problemen kommt. Allerdings erstellen wir Dekoratoren typischerweise mit anderen Mustern wie Factory und Builder. Sobald wir diese besprochen haben, werden Sie feststellen, dass die Erstellung der konkreten Komponente mit ihrem Dekorator »gut verkapselt« ist und nicht zu Problemen dieser Art führt.

F: Können Dekoratoren denn von anderen Dekoratoren in der Kette wissen? Vielleicht soll meine getDescription()-Methode »Whip, Double Mocha« ausgeben und nicht »Double Mocha, Whip«. Dafür müsste der äußerste Dekorator aber alle anderen Dekoratoren kennen, die er umgibt.

A: Dekoratoren sind dafür gedacht, das von ihnen umgebene Objekt mit Verhalten zu versehen. Wenn Sie mehrere Ebenen in die Dekoratorenkette hineinsehen müssen, nutzen Sie den Dekorator über seine eigentliche Aufgabe hinaus. Dennoch ist das grundsätzlich möglich. Vielleicht haben Sie einen CondimentPrettyPrint-Dekorator, der die abschließende Beschreibung parst und der »Mocha, Whip, Mocha« als »Whip, Double Mocha« ausgeben kann. Um das zu erleichtern, könnte getDescription() eine ArrayList mit Beschreibungen zurückgeben.

Spitzen Sie Ihren Bleistift

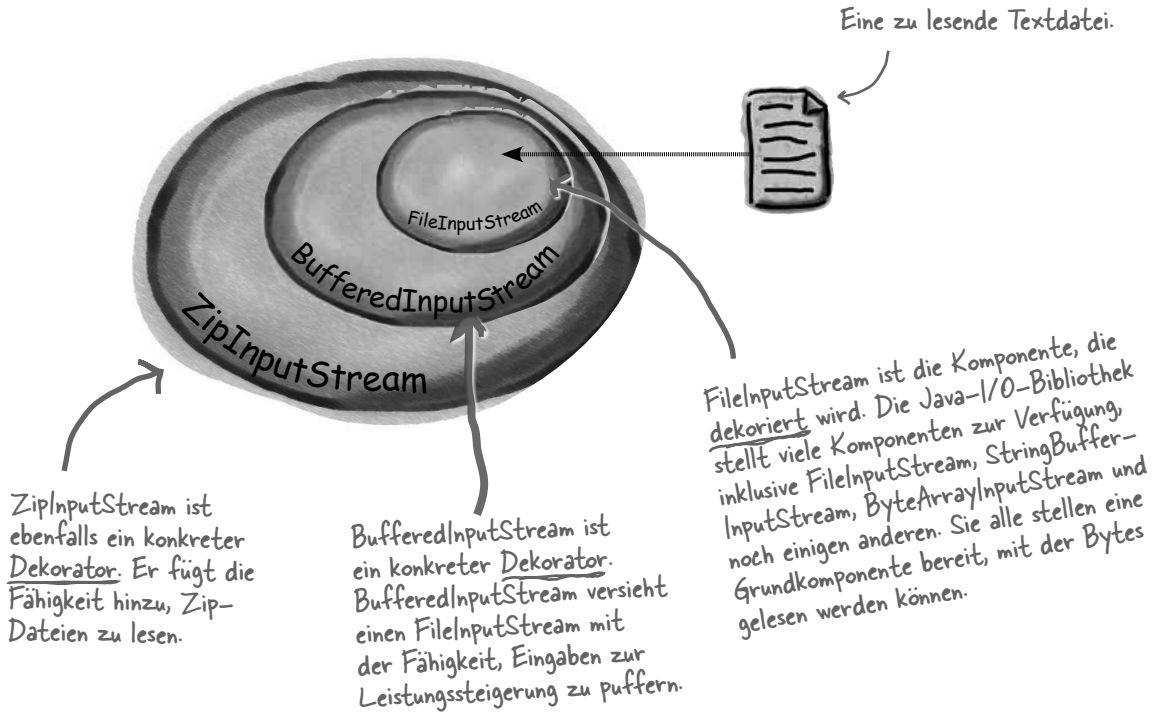
Unsere Freunde bei Starbuzz bieten ihre Getränke jetzt in verschiedenen Größen an. Sie können einen Kaffee nun in Tall, Grande und Venti bestellen (auf Deutsch: klein, mittel und groß). Starbuzz sah dies als einen wesentlichen Bestandteil der Kaffee-Klasse an und hat die Methoden setSize() und getSize() bereits in die Beverage-Klasse integriert. Außerdem sollten die einzelnen Zutaten je nach Größe unterschiedliche Preise haben. Sojamilch kostet für Tall, Grande und Venti also 10, 15 und 20 Cent. Die aktualisierte Beverage-Klasse sehen Sie unten.

Wie würden Sie die Dekorator-Klassen für diese veränderten Anforderungen anzupassen?

```
public abstract class Beverage {
        public enum Size { TALL, GRANDE, VENTI };
        Size size = Size.TALL;
        String description = "Unbekanntes Getränk";
        public String getDescription() {
               return description;
        }
        public void setSize(Size size) {
               this.size = size;
        }
        public Size getSize() {
               return this.size;
        }
        public abstract double cost();
}
```

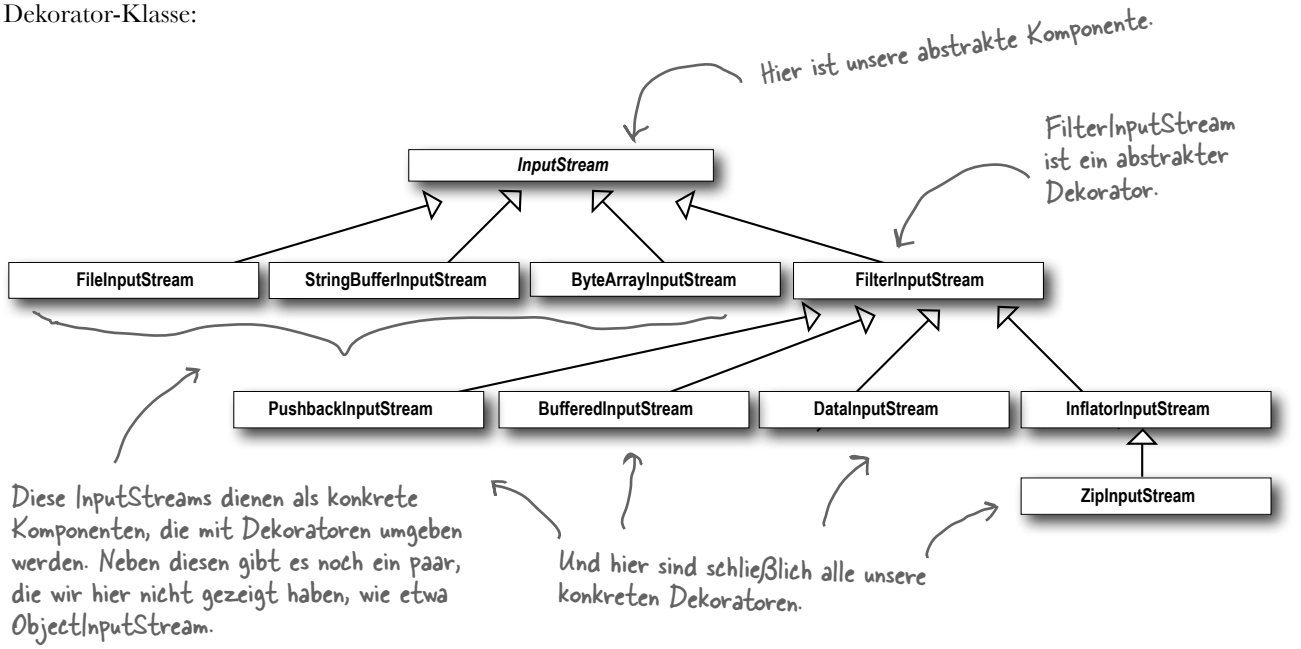
Dekoratoren in freier Wildbahn: Java I/O

Die große Zahl an Klassen im java.io-Package ist – *überwältigend*. Sie sind nicht alleine, wenn Ihnen beim ersten (und zweiten und dritten) Anblick dieser API ein »Wow!« entglitten ist. Nachdem Sie das Decorator-Muster kennen, sollten die I/O-Klassen deutlich mehr Sinn ergeben, weil das java.io-Package zu großen Teilen darauf basiert. Hier ein typischer Satz an Objekten, die Dekoratoren verwenden, um das Lesen aus einer Datei mit zusätzlicher Funktionalität zu versehen:



Die java.io-Klassen dekorieren

BufferedInputStream und ZipInputStream erweitern beide FilterInputStream, der wiederum InputStream erweitert. InputStream dient dabei als die abstrakte Dekorator-Klasse:



Wie Sie sehen, ist der Unterschied zum Starbuzz-Entwurf gar nicht so groß. Inzwischen sollten Sie nach einem Blick in die Dokumentation der java.io-API ohne große Probleme eigene Dekoratoren für die verschiedenen *Input*-Streams erstellen können.

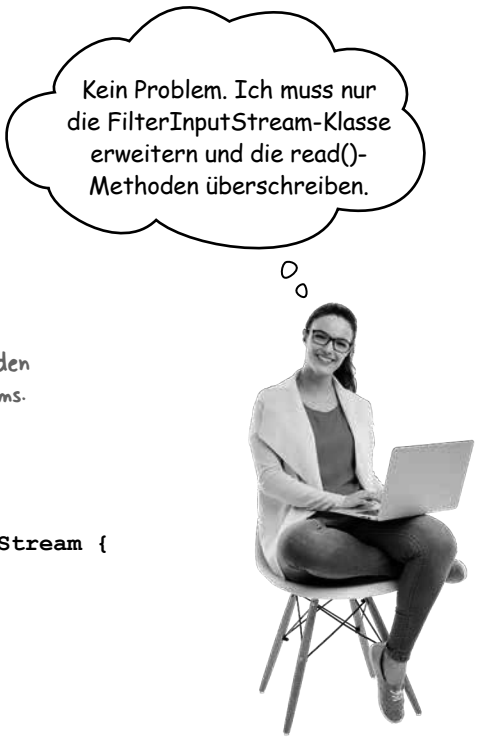
Sie werden merken, dass die *Output*-Streams auf die gleiche Weise gestaltet sind. Und möglicherweise haben Sie auch schon festgestellt, dass die Reader/Writer-Streams (für zeichenbasierte Daten) sehr große Ähnlichkeit mit dem Design der Stream-Klassen haben (mit ein paar Unterschieden und Inkonsistenzen, aber nah genug dran, um zu verstehen, was vor sich geht.)

Java I/O zeigt aber auch einen der Nachteile des Decorator-Musters auf: Entwürfe mit diesem Muster führen oft zu einer Vielzahl kleiner Klassen, die für einen Entwickler, der versucht, die Decorator-basierte API zu benutzen, schnell überwältigen können. Aber nachdem Sie jetzt wissen, wie das Decorator-Muster funktioniert, behalten Sie den Überblick. Wenn Sie eine externe Decorator-lastige API einsetzen, können Sie herausfinden, wie die Klassen organisiert sind, und das »Verpacken« nutzen, um das gewünschte Verhalten zu erreichen.

Einen eigenen Java-I/O-Dekorator schreiben

In Ordnung. Jetzt kennen Sie das Decorator-Muster und das I/O-Klassendiagramm. Damit sind Sie so weit, Ihren eigenen Input-Dekorator zu schreiben.

Wie wäre es hiermit: Schreiben Sie einen Dekorator, der alle Großbuchstaben im Eingabestream in Kleinbuchstaben umwandelt. Anders gesagt: Die Eingabe »Ich kenne das Decorator-Muster, also bin ich EIN HELD!« soll in »ich kenne das decorator-muster, also bin ich ein held!« umgewandelt werden.



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

Zuerst erweitern wir FilterInputStream, den abstrakten Dekorator für alle InputStreams.

```
public class LowerCaseInputStream extends FilterInputStream {

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

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

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

Jetzt müssen wir zwei read()-Methoden implementieren. Sie übernehmen ein Byte (oder ein Array mit Bytes, die jeweils für ein Zeichen stehen) und konvertieren jedes Byte in Kleinbuchstaben.

NICHT VERGESSEN: Unsere Codebeispiele enthalten keine import- und package-Anweisungen. Den vollständigen Quellcode finden Sie unter https://wickedlysmart.com/head-first-design-patterns.

Unseren neuen Java-I/O-Dekorator testen

Schreiben wir etwas Code, um den I/O-Dekorator zu testen:

```
public class InputTest {
    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 – zuerst mit einem BufferedInputStream und dann mit unserem brandneuen LowerCaseInputStream-Filter.

```
Ich kenne das Decorator-Muster, also
bin ich EIN HELD!
```

Datei test.txt

Diese Datei müssen Sie erstellen.

Nutzen Sie einfach den Stream, um die Zeichen bis zum Dateiende zu lesen und direkt wieder auszugeben.

Probelauf:

```
Datei Bearbeiten Fenster Hilfe  DecoratorsRule
% java InputTest
ich kenne das decorator-muster, also bin ich ein held!
%
```

Muster im Gespräch

Das Interview der Woche: Geheime Geständnisse eines Dekorators

Von Kopf bis Fuß: Willkommen, Decorator-Muster. Wir haben gehört, dass Sie in letzter Zeit ziemlich niedergeschlagen waren?

Decorator: Ich weiß, die Welt sieht mich als berühmtes Entwurfsmuster, aber wie alle habe auch ich meine Probleme.

Von Kopf bis Fuß: Möchten Sie uns vielleicht von einigen Ihrer Probleme erzählen?

Decorator: Na ja, wie Sie wissen, habe ich die Macht, Entwürfe flexibler zu machen, so viel ist sicher. Aber ich habe auch eine dunkle Seite. Manchmal erhält ein Entwurf durch mich viele kleine Klassen, was gelegentlich dazu führt, dass er für andere schwerer zu verstehen ist.

Von Kopf bis Fuß: Können Sie uns ein Beispiel geben?

Decorator: Nehmen wir zum Beispiel die I/O-Bibliotheken, die am Anfang für viele Menschen fast schon undurchschaubar sind. Würde man die Klassen dagegen als eine Reihe von Wrappern um einen InputStream betrachten, wäre das Leben viel einfacher.

Von Kopf bis Fuß: Das klingt doch gar nicht schlecht. Sie sind immer noch ein tolles Muster. Um das zu verbessern, muss die Öffentlichkeit doch nur richtig aufgeklärt werden, oder?

Decorator: Leider ist das noch nicht alles. Ich habe Typ-Probleme. Manchmal nimmt sich jemand ein Stück Clientcode, verlässt sich dabei auf bestimmte Typen und fügt, ohne wirklich darüber nachzudenken, Dekoratoren ein. Einer meiner Vorteile ist, *dass Sie Dekoratoren normalerweise transparent einfügen können, ohne dass der Client etwas davon merkt*. Aber wie gesagt, so mancher Code hängt von bestimmten Typen ab, und wenn Sie dann Dekoratoren einfügen, fliegt einem die Sache um die Ohren, und schlimme Dinge passieren.

Von Kopf bis Fuß: Ich denke, jeder weiß, dass man beim Einfügen von Dekoratoren vorsichtig sein muss. Das sollte aber kein Grund sein, dass Sie an sich selbst zweifeln.

Decorator: Ich versuche ja auch, das nicht zu tun. Ich habe aber darüber hinaus das Problem, dass das Einfügen von Dekoratoren die Komplexität des Codes erhöht, der für die Instanziierung einer Komponente gebraucht wird. Sobald Sie Dekoratoren nutzen, müssen Sie nicht nur die Komponente instanziieren, sondern sie auch noch mit wer weiß wie vielen Dekoratoren umgeben.

Von Kopf bis Fuß: Nächste Woche werde ich die Muster Factory und Builder interviewen. Wie ich höre, können Sie hierbei hilfreich sein?

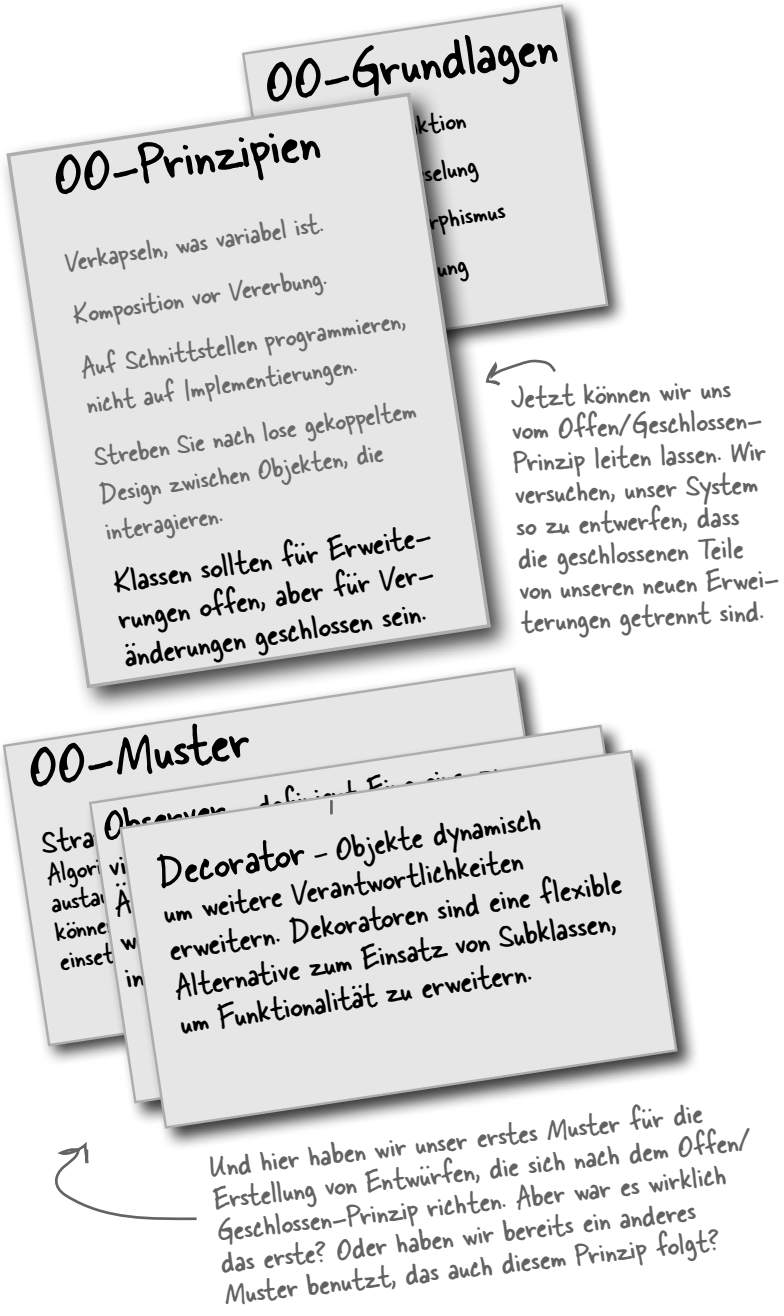
Decorator: Das stimmt. Ich sollte öfter mit diesen Typen reden.

Von Kopf bis Fuß: Nun, wir sind alle davon überzeugt, dass Sie ein großartiges Muster für die Erstellung flexibler Entwürfe sind und dabei dem Offen/Geschlossen-Prinzip treu bleiben. Also Kopf hoch und positiv denken!

Decorator: Ich tue mein Bestes, vielen Dank.

Werkzeuge für Ihren Entwurfs-Werkzeugkasten

Damit haben Sie ein weiteres Kapitel verinnerlicht und Ihren Werkzeugkasten um ein neues Prinzip und ein neues Muster erweitert.



Punkt für Punkt

- Vererbung ist eine Form der Erweiterung, aber nicht unbedingt die beste, um flexible Entwürfe zu erstellen.
- In unseren Entwürfen sollte es erlaubt sein, Verhalten zu erweitern, ohne vorhandenen Code anpassen zu müssen.
- Komposition und Delegation können oft genutzt werden, um zur Laufzeit neues Verhalten hinzuzufügen.
- Das Decorator-Muster stellt eine Alternative zu Subklassen für die Erweiterung von Verhalten dar.
- Zum Decorator-Muster gehört eine Reihe von Dekorator-Klassen, die konkrete Komponenten umgeben.
- Dekorator-Klassen spiegeln den Typ der von ihnen dekorierten Komponenten wider. (Tatsächlich haben sie den gleichen Typ wie die von ihnen dekorierten Komponenten, entweder durch Vererbung oder die Implementierung eines Interface.)
- Dekoratoren verändern das Verhalten ihrer Komponenten, indem sie vor und/oder nach (oder sogar anstelle von) Methodenaufrufen auf der Komponente neue Funktionalität hinzufügen.
- Sie können eine Komponente mit einer beliebigen Zahl von Dekoratoren umgeben.
- Dekoratoren sind typischerweise für den Client der Komponente transparent, zumindest solange sich der Client auf den konkreten Typ der Komponente verlässt.
- Dekoratoren können in einem Entwurf zu vielen kleinen Objekten führen. Ihr übermäßiger Einsatz hat schnell komplexen Code zur Folge.

Spitzen Sie Ihren Bleistift
Lösung

Schreiben Sie die cost()-Methoden für die folgenden Klassen (Pseudo-Java reicht aus). Hier ist unsere Lösung:

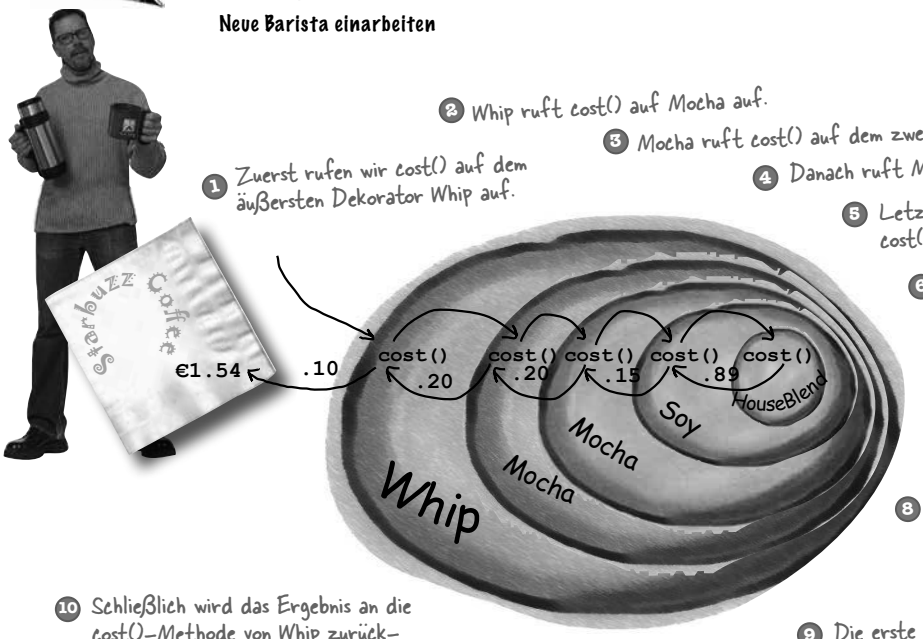
```
public class Beverage {

// Instanzvariablen für milkCost (Milchpreis), soyCost (Sojapreis),
// mochaCost (Mochapreis) und whipCost (Milchschaumpreis) sowie
// Getter und Setter für milk, soy, mocha und whip
// deklarieren.

public double cost() {

        double condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
}


public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Hervorragende dunkle Röstung";
    }

    public double cost() {
        return 1.99 + super.cost();
    }
}
```

Spitzen Sie Ihren Bleistift

Lösung

»Double Mocha Soja Latte mit Milchschaum«

Neue Barista einarbeiten



1. Zuerst rufen wir cost() auf dem äußersten Dekorator Whip auf.
2. Whip ruft cost() auf Mocha auf.
3. Mocha ruft cost() auf dem zweiten Mocha auf.
4. Danach ruft Mocha cost() auf Soja auf.
5. Letzte Zutat! Soja ruft cost() auf HouseBlend auf.
6. Die cost()-Methode von HouseBlend gibt 89 Cent zurück und wird vom Stack entfernt.
7. Die cost()-Methode von Soja addiert 15 Cent, gibt das Ergebnis zurück und wird vom Stack genommen.
8. Die cost()-Methode des zweiten Mocha addiert 20 Cent, gibt das Ergebnis zurück und wird vom Stack entfernt.
9. Die erste von Mochas cost()-Methoden addiert 20 Cent, gibt das Ergebnis zurück und wird vom Stack genommen.
10. Schließlich wird das Ergebnis an die cost()-Methode von Whip zurückgegeben, die 10 Cent addiert. Das Endergebnis ist 1,54 Euro.

Spitzen Sie Ihren Bleistift
Lösung

Unsere Freunde bei Starbuzz bieten ihre Getränke jetzt in verschiedenen Größen an. Sie können einen Kaffee nun in Tall, Grande und Venti bestellen (auf Deutsch: klein, mittel und groß). Starbuzz sah dies als einen wesentlichen Bestandteil der Kaffee-Klasse an und hat die Methoden setSize() und getSize() bereits in die Beverage-Klasse eingefügt. Außerdem sollten die einzelnen Zutaten je nach Größe unterschiedliche Preise haben. Sojamilch kostet für Tall, Grande und Venti also 10, 15 und 20 Cent. Hier ist unsere Lösung.

```
public abstract class CondimentDecorator extends Beverage {
    public Beverage beverage;
    public abstract String getDescription();

    public Size getSize() {
        return beverage.getSize();
    }
}


public class Soy extends CondimentDecorator {
    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (beverage.getSize() == Size.TALL) {
            cost += .10;
        } else if (beverage.getSize() == Size.GRANDE) {
            cost += .15;
        } else if (beverage.getSize() == Size.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

Wir haben die Dekoratoren um eine getSize()-Methode erweitert, die einfach die Größe eines Getränks zurückgibt.

Hier lesen wir die Größe aus (die sich bis zum konkreten Getränk durchzieht) und addieren den entsprechenden Preis.