

4 Das Factory-Muster

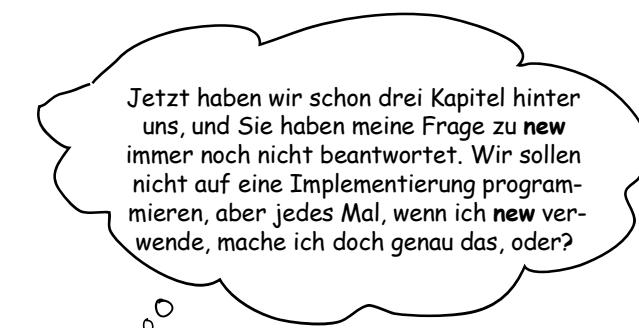


Backen in OO-Qualität



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.

Über »new« nachdenken



Denken Sie »konkret«, wenn Sie »new« sehen.

Ja, wenn Sie **new** verwenden, instantiierten Sie eindeutig eine konkrete Klasse, und das ist definitiv eine Implementierung und keine Schnittstelle. Und es bleibt eine gute Frage. Sie wissen jetzt ja, dass Ihr Code zerbrechlicher und weniger flexibel ist, wenn Sie ihn an eine konkrete Klasse binden.

```
Ente ente = new Stockente();
```

Wir möchten Interfaces verwenden, um unseren Code flexibel zu halten.

Aber wir müssen eine Instanz einer konkreten Klasse erstellen!

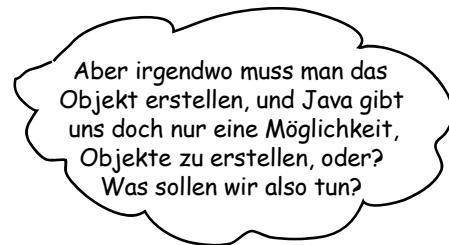
Wenn Sie einen Satz verwandter Klassen haben, müssen Sie oft Code wie diesen schreiben:

```
if (picknick) {  
    ente = new Stockente();  
} else if (jagd) {  
    ente = new Lockente();  
} else if (inBadewanne) {  
    ente = new Gummiente();  
}
```

Wir haben einen Haufen verschiedener Ente-Klassen, und wir wissen erst zur Laufzeit, welche wir instantiiieren müssen.

Wir haben hier verschiedene konkrete Klassen, die instantiiert werden. Dabei wird die Entscheidung, welche instantiiert werden soll, zur Laufzeit in Abhängigkeit von einem Satz bestimmter Bedingungen getroffen.

Wenn Sie derartigen Code sehen, wissen Sie, dass Sie diesen Code wieder öffnen müssen, wenn Änderungen oder Erweiterungen anstehen, um zu untersuchen, was hinzugefügt (oder gelöscht) werden muss. Oft taucht solcher Code in mehreren Teilen der Anwendung auf und macht Pflege und Aktualisierungen dadurch schwieriger und fehleranfällig.



Was ist der Haken an »new«?

In technischer Hinsicht ist **new** vollkommen in Ordnung. Schließlich ist es ein fundamentaler Bestandteil von Java. Der eigentlich Schuldige ist unser alter Freund VERÄNDERUNG und die Auswirkungen, die Veränderungen auf unsere Verwendung von **new** haben.

Sie wissen, dass Sie sich selbst vor vielen Veränderungen schützen können, die an einem System im Lauf der Zeit vorgenommen werden müssen, indem Sie auf eine Schnittstelle programmieren. Warum? Wenn Ihr Code auf eine Schnittstelle geschrieben ist, funktioniert er aufgrund von Polymorphismus mit jeder Klasse, die das entsprechende Interface implementiert. Aber wenn Sie Code haben, der viele konkrete Klassen verwendet, dann riskieren Sie Probleme, weil der Code eventuell geändert werden muss, wenn neue konkrete Klassen hinzugefügt werden. Ihr Code ist mit anderen Worten also nicht »für Veränderungen geschlossen«. Um ihn mit neuen Typen zu erweitern, müssen Sie ihn wieder öffnen.

Was können Sie also tun? In Zeiten wie diesen sollten Sie sich wieder auf ein OO-Entwurfsprinzip zurückbesinnen, um nach Hinweisen zu suchen. Denken Sie an unser erstes Prinzip. Es befasst sich mit Veränderung und weist uns an, *die Aspekte zu identifizieren, die sich ändern können, und sie von denen zu trennen, die gleich bleiben*.



Denken Sie daran, dass Entwürfe für >>Erweiterung offen, aber für Veränderungen geschlossen<< sein sollen – alles dazu finden Sie in Kapitel 3.



Wie könnten Sie alle Teile Ihrer Anwendung, die konkrete Klassen instantiierten, nehmen und vom Rest der Anwendung trennen oder einkapseln?

Identifizieren, was sich verändert

Die Aspekte identifizieren, die veränderlich sind

Nehmen wir an, Sie besitzen eine Pizzeria. Als innovationsfreudiger Pizzabesitzer aus Objekthausen könnten Sie Code wie diesen hier schreiben:

```
Pizza bestellePizza() {  
    Pizza pizza = new Pizza();  
  
    pizza.vorbereiten();  
    pizza.backen();  
    pizza.schneiden();  
    pizza.einpacken();  
  
    return pizza;  
}
```



Aus Flexibilitätsgründen sollte das eigentlich eine abstrakte Klasse oder ein Interface sein, aber beides könnten wir nicht direkt instantiiieren.

Aber mit einem Typ Pizza kommen Sie nicht aus

Sie fügen also etwas Code hinzu, der den passenden Pizzatyp *bestimmt* und sich dann darum kümmert, die Pizza zu machen:

```
Pizza bestellePizza(String typ) {  
    Pizza pizza;  
  
    if (typ.equals("Salami")) {  
        pizza = new SalamiPizza();  
    } else if (typ.equals("Spinat")) {  
        pizza = new SpinatPizza();  
    } else if (typ.equals("Schinken")) {  
        pizza = new SchinkenPizza();  
    }  
  
    pizza.vorbereiten();  
    pizza.backen();  
    pizza.schneiden();  
    pizza.einpacken();  
  
    return pizza;  
}
```

Jetzt übergeben wir bestellePizza den Typ der Pizza.

Auf Basis des Pizzatyps instantiiieren wir die konkrete Klasse und weisen sie der Instanzvariablen pizza zu. Beachten Sie, dass jeder der Pizzatypen das Interface Pizza implementieren muss.

Wenn wir eine Pizza haben, bereiten wir sie vor (das Übliche eben: Teig ausrollen, Soße draufgeben, Belag und Käse hinzufügen), dann backen, schneiden und verpacken wir sie! Jeder Untertyp von Pizza (SalamiPizza, SpinatPizza) weiß, wie er sich selbst zubereiten muss.

Aber es müssen noch mehr Pizzas hinzugefügt werden

Sie bemerken, dass Ihre Konkurrenten ihren Speisekarten ein paar beliebte Pizzas hinzugefügt haben: die Thunfischpizza und die Krabbenpizza. Natürlich müssen Sie mit der Konkurrenz mitziehen und fügen Ihrer Speisekarte diese Elemente hinzu. Außerdem haben Sie in letzter Zeit nicht viele Spinatpizzas verkauft und entschließen sich deshalb, diese von der Speisekarte zu streichen:

```
Pizza bestellePizza(String typ) {
    Pizza pizza;

    if (typ.equals("Salami")) {
        pizza = new SalamiPizza();
    } else if (typ.equals("Spinat")) {
        pizza = new SpinatPizza();
    } else if (typ.equals("Schinken")) {
        pizza = new SchinkenPizza();
    } else if (typ.equals("Thunfisch")) {
        pizza = new ThunfischPizza();
    } else if (typ.equals("Krabben")) {
        pizza = new KrabbenPizza();
    }

    pizza.vorbereiten();
    pizza.backen();
    pizza.schneiden();
    pizza.einpacken();
    return pizza;
}
```

Dieser Code ist NICHT für Veränderungen geschlossen. Wenn die Pizzeria ihr Pizzaangebot verändert, müssen wir den Code öffnen und modifizieren.

Das ist das, was veränderlich ist. Während sich das Pizzaangebot mit der Zeit ändert, müssen Sie diesen Code immer und immer wieder anpassen.

Das ist das, bei dem wir davon ausgehen, dass es gleich bleibt. Die Zubereitung, das Backen und das Verpacken einer Pizza haben sich zu einem Großteil seit Jahren nicht mehr geändert. Deswegen gehen wir nicht davon aus, dass sich dieser Code ändert, sondern nur die Pizzas, auf denen er operiert.

Ganz offensichtlich ist die Klärung, welche konkrete Klasse instantiiert werden soll, für das Durcheinander in unserer bestellePizza()-Methode verantwortlich und verhindert, dass sie gegenüber Veränderungen geschlossen bleibt. Aber jetzt, da wir wissen, was veränderlich ist und was nicht, ist es wahrscheinlich an der Zeit, dieses einzukapseln.

Die Objekt-Erstellung kapseln

Jetzt wissen wir also, dass wir besser bedient sind, wenn wir die Objekt-Erstellung aus der Methode bestellePizza() herausziehen. Aber wie? Wir werden einfach den Code zum Erstellen herausziehen und in ein anderes Objekt verschieben, das sich nur damit befasst, Pizzas zu erstellen.

```
Pizza bestellePizza(String typ) {  
    Pizza pizza;  
  
    pizza.vorbereiten();  
    pizza.backen();  
    pizza.schneiden();  
    pizza.einpacken();  
    return pizza;  
}
```

Aber was kommt hier rein?

Zuerst ziehen wir den Code
zur Objekt-Erstellung aus der
Methode bestellePizza heraus.

```
if (typ.equals("Salami")) {  
    pizza = new SalamiPizza();  
} else if (typ.equals("Schinken")) {  
    pizza = new SchinkenPizza();  
} else if (typ.equals("Thunfisch")) {  
    pizza = new ThunfischPizza();  
} else if (typ.equals("Krabben")) {  
    pizza = new KrabbenPizza();  
}
```

Dann fügen wir diesen Code in ein Objekt
ein, das sich nur damit befasst, wie Pizzas
erstellt werden sollen. Wenn ein anderes
Objekt eine Pizza benötigt, muss es sich an
dieses Objekt wenden.



Wir haben einen Namen für dieses neue Objekt: Wir nennen es eine Fabrik.

Fabriken kümmern sich um die Details der Objekt-Erstellung. Wenn wir eine EinfachePizzaFabrik haben, wird unsere bestellePizza()-Methode einfach zu einem Client dieses Objekts. Jedes Mal, wenn es eine Pizza benötigt, bittet es die Pizzafabrik, eine herzustellen. Dann sind die Zeiten vorbei, in denen die Methode bestellePizza() etwas über den Unterschied zwischen SalamiPizza und KrabbenPizza wissen muss. Jetzt kümmert sich die Methode bestelle-Pizza() nur noch darum, dass sie eine Pizza erhält, die das Interface Pizza implementiert, damit sie vorbereiten(), backen(), schneiden() und einpacken() aufrufen kann.

Wir müssen hier immer noch ein paar Kleinigkeiten ausfüllen. Womit wird beispielsweise der Erstellungscode in bestellePizza() ersetzt? Implementieren wir einfach eine EinfachePizzaFabrik und finden wir es heraus ...

Eine einfache Pizzafabrik erstellen

Beginnen wir mit der Fabrik selbst. Was wir jetzt machen werden, ist, eine Klasse zu definieren, die die Objekt-Erstellung für alle Pizzas kapselt. Hier ist sie ...

```
Hier ist unsere neue Klasse, die EinfachePizzaFabrik.  
Sie hat nur eine einzige Lebensaufgabe: für ihre Kunden Pizzas herzustellen.
```

```
Erst definieren wir in der Fabrik die Methode erstellePizza(). Das ist die Methode, die alle Clients nutzen werden, um neue Objekte zu instantiiieren.
```

```
public class EinfachePizzaFabrik {  
    public Pizza erstellePizza(String typ) {  
        Pizza pizza = null;  
  
        if (typ.equals("Salami")) {  
            pizza = new SalamiPizza();  
        } else if (typ.equals("Schinken")) {  
            pizza = new SchinkenPizza();  
        } else if (typ.equals("Thunfisch")) {  
            pizza = new ThunfischPizza();  
        } else if (typ.equals("Krabben")) {  
            pizza = new KrabbenPizza();  
        }  
        return pizza  
    }  
}
```

```
Dies ist der Code, den wir auf der Methode bestellePizza() herausgeplückt haben.
```

```
Dieser Code wird, genau wie unsere ursprüngliche Methode bestellePizza(), immer noch durch den Pizzatyp parametrisiert.
```

Es gibt keine Dummen Fragen

F: Welche Vorteile bietet das? Es sieht eigentlich so aus, als würden wir das Problem nur in ein anderes Objekt verschieben.

A: Sie sollten immer daran denken, dass EinfachePizzaFabrik viele Clients haben kann. Bisher haben wir nur die Methode bestellePizza() gesehen. Es könnte jedoch auch eine PizzeriaSpeisekarte-Klasse geben, die die Pizzas abruft, um die aktuellen Beschreibungen und Preise einzusehen. Es könnte ebenso eine Klasse

PizzaKurier geben, die mit Pizzas anders umgeht als unsere Pizzeria-Klasse und trotzdem ein Client von EinfachePizzaFabrik ist.

Indem wir die Pizzaerstellung in einer Klasse kapseln, erreichen wir also, dass wir die Änderungen nur noch an einer Stelle durchführen müssen, wenn sich die Implementierung ändert. Vergessen Sie nicht, dass es uns auch darum ging, die konkreten Instantierungen aus unserem Client-Code zu entfernen.

F: Ich habe mal ein ähnliches Design wie das hier gesehen, bei dem eine Fabrik wie die hier als eine statische

Methode definiert wurde. Was ist der Unterschied?

A: Eine einfache Fabrik als eine statische Methode zu definieren ist eine gebräuchliche Technik, die üblicherweise als statische Fabrik bezeichnet wird. Warum eine statische Methode verwendet wird? Weil Sie dann kein Objekt instantiiieren müssen, um die Klasse verwenden zu können. Aber denken Sie daran, dass das auch den Nachteil hat, dass Sie keine Unterklassen bilden können, in denen sich das Verhalten der Erstellungsmethode ändern ließe.

Einfache Fabrik

Die Pizzeria-Klasse überarbeiten

Es ist nun an der Zeit, unseren Client-Code zu flicken. Wir möchten erreichen, dass wir uns die Pizzas von der Fabrik liefern lassen. Hier sind die Änderungen:

Jetzt übergeben wir Pizzeria eine Referenz auf eine EinfachePizzaFabrik.

```
public class Pizzeria {  
    EinfachePizzaFabrik fabrik  
  
    public Pizzeria(EinfachePizzaFabrik fabrik) {  
        this.fabrik = fabrik;  
    }
```

Pizzeria wird die Fabrik im Konstruktor übergeben.

```
    Pizza bestellePizza(String typ) {  
        Pizza pizza;  
  
        pizza = fabrik.erstellePizza(typ);  
  
        pizza.vorbereiten();  
        pizza.backen();  
        pizza.schneiden();  
        pizza.einpacken();  
        return pizza;  
    }  
    // andere Pizzeria-Methoden  
}
```

Und die Methode bestellePizza() nutzt die Fabrik, um ihre Pizzas zu erstellen, indem sie einfach den Typ der Bestellung weiterreicht.

Beachten Sie, dass wir den new-Operator durch eine Erstellungsmethode im Fabrik-Objekt ersetzt haben. Es gibt hier keine konkreten Instantierungen mehr!



Wir wissen, dass uns die Objekt-Komposition (unter anderem) ermöglicht, Verhalten zur Laufzeit dynamisch zu ändern, weil wir Implementierungen einschieben und rausziehen können. Wie könnten wir das in unserer Pizzeria verwenden? Was für Fabrik-Implementierungen können wir einschieben und rausziehen?

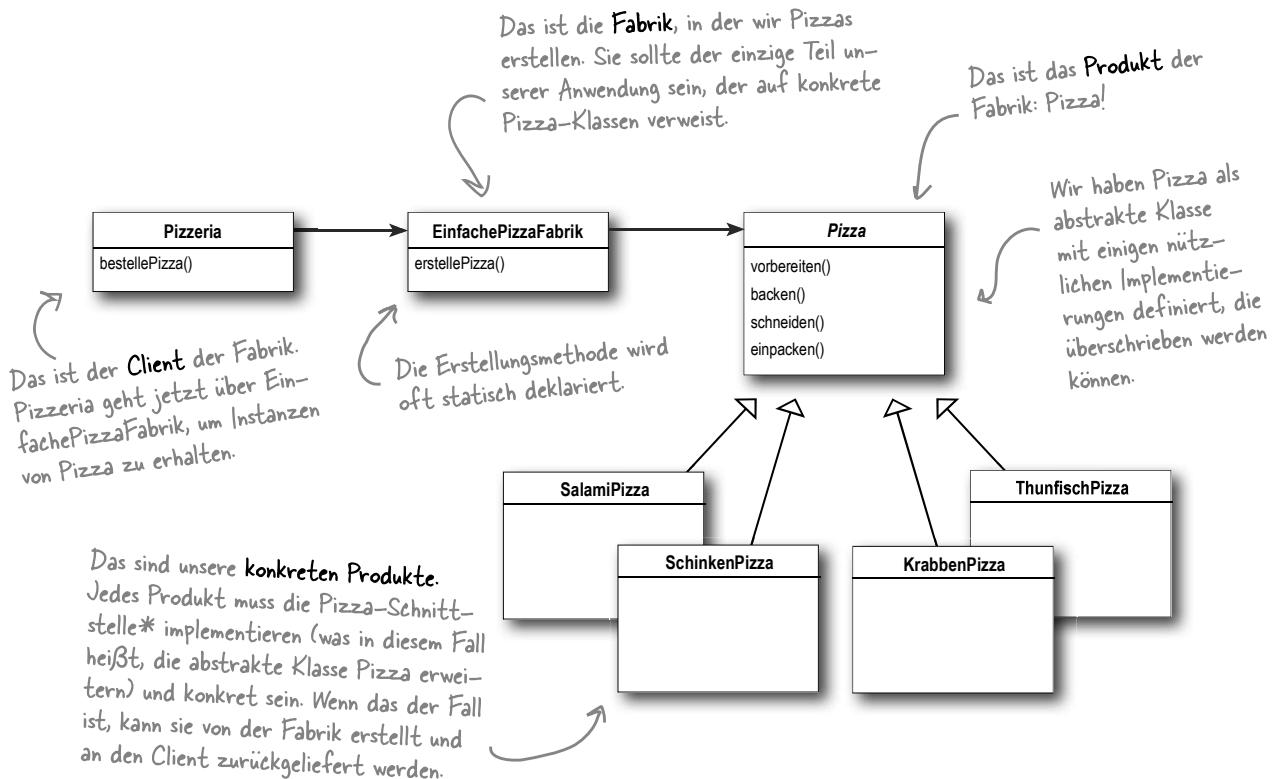
Münchener, Kölner (nicht zu vergessen nach Düsseldorfer) Art herstellen.
Wir wissen nicht, an was Sie so denken, aber wir denken an Pizza-Fabriken, die Pizzas nach Berliner,



Die Definition der einfachen Fabrik

Die einfache Fabrik ist eigentlich kein Entwurfsmuster, sondern eher ein Programmier-idiom. Aber sie wird verbreitet verwendet, deswegen verleihen wir ihr die Musterehrenmedaille. Einige Entwickler verwechseln dieses Idiom mit dem »Factory-Pattern«. Wenn das nächste Mal zwischen Ihnen und einem anderen Entwickler eine peinliche Stille herrscht, haben Sie also ein nettes Thema, um das Eis zum Schmelzen zu bringen.

Dass die einfache Fabrik kein ECHTES Muster ist, bedeutet aber noch lange nicht, dass wir uns nicht ansehen sollten, wie sie zusammengebaut ist. Werfen wir einen Blick auf das Klassendiagramm für unsere neue Pizzeria:



Betrachten Sie die einfache Fabrik als Aufwärmübung. Als Nächstes werden wir zwei Schwerlast-Muster untersuchen, die beide Fabriken sind. Aber keine Angst, es gibt noch mehr Pizza!

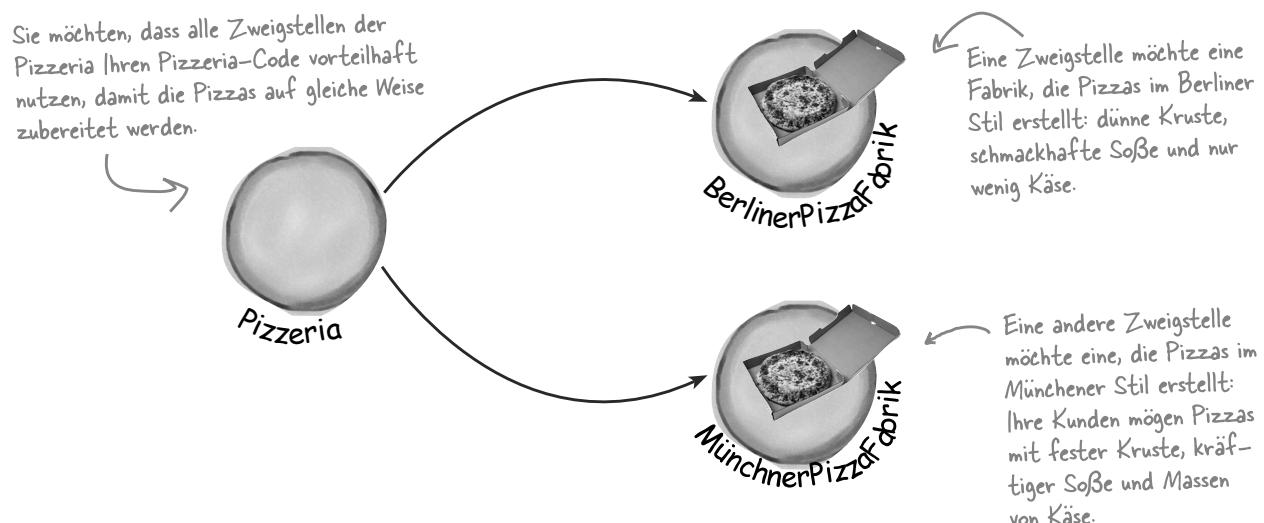
*Noch mal zur Erinnerung: Bei Entwurfsmustern bedeutet die Phrase »eine Schnittstelle implementieren« (oder »ein Interface implementieren«) NICHT immer: »Schreiben Sie eine Klasse, die über das Schlüsselwort 'implements' in der Klassendeklaration ein Java-Interface implementiert.« In der allgemeinen Verwendung der Aussage wird eine konkrete Klasse, die eine Methode von einem Supertypen implementiert (der eine Klasse ODER ein Interface sein kann), immer noch als Klasse betrachtet, die »die Schnittstelle« dieses Supertyps »implementiert«.

Pizzeria-Kette

Von der Pizzeria zur Pizzeria-Kette

Ihre Pizzeria in Objekthausen ist so gut eingeslagen, dass Sie die Wettbewerber verdrängt haben und jetzt jeder eine Pizzeria in seiner Nachbarschaft haben will. Als Haupthaus wollen Sie die Qualität der Zweigstellen sicherstellen und fordern, dass diese Ihren erprobten Code verwenden.

Was aber ist mit den regionalen Unterschieden? Vielleicht möchte jede Zweigstelle (Berlin, München und Köln, um nur ein paar zu nennen) ein unterschiedliches Sortiment an Pizzas anbieten, das davon abhängig ist, wo sich die Zweigstelle befindet und was die lokalen Pizzaliebhaber bevorzugen.



Wir haben einen Ansatz gesehen ...

Wenn wir die EinfachePizzaFabrik nehmen und drei unterschiedliche Fabriken, BerlinerPizzaFabrik, MünchenerPizzaFabrik und KölnerPizzaFabrik, erstellen, können wir einfach die Pizzeria mit der geeigneten Fabrik zusammensetzen, und die Zweigstelle kann die Arbeit aufnehmen. Das ist ein Ansatz.

Sehen wir uns an, wie das aussehen würde ...

Das Factory-Muster

```
BerlinPizzaFabrik berlinFabrik = new BerlinPizzaFabrik();  
Pizzeria berlinPizza = new Pizzeria(berlinFabrik);  
berlinPizza.bestellen("Schinken");
```

Hier erzeugen wir eine Fabrik, um Pizzas im Berliner Stil zu machen.

Dann erstellen wir eine Pizzeria und übergeben ihr einen Referenz auf die Berliner Fabrik.

Und wenn wir Pizzas machen, erhalten wir Pizzas im Berliner Stil.

```
MünchenPizzaFabrik münchenFabrik= new MünchenPizzaFabrik();  
Pizzeria münchenPizza = new Pizzeria(münchenFabrik);  
münchenPizza.bestellen("Schinken");
```

Genau so erzeugen wir auch Münchener Pizzerias: Wir erzeugen eine Fabrik für Münchener Pizzas und erzeugen eine Pizzeria, die durch eine Münchener Fabrik zusammengesetzt wird. Wenn wir Pizzas machen, erhalten wir Pizzas, wie man sie in München mag.

Aber Sie hätten gern etwas mehr Qualitätskontrolle ...

Sie haben einen Testlauf der EinfacheFabrik-Idee durchgeführt und dabei festgestellt, dass die Zweigstellen Ihre Fabrik verwenden haben, um Pizzas zu erstellen, für den restlichen Vorgang dann aber ihre selbst gestrickten Prozeduren verwendet haben: Sie backen die Dinge etwas anders, vergessen, die Pizza zu schneiden, und verwenden Verpackungen anderer Hersteller.

Beim erneuten Durchdenken des Problems erkennen Sie, dass Sie eigentlich ein Framework erstellen möchten, das Pizzeria und Pizzaerstellung zusammenbindet und trotzdem ermöglicht, dass die Dinge flexibel bleiben.

In unserem früheren Code vor EinfachePizzaFabrik hatten wir den Code zur Pizzaerstellung an die Pizzeria gebunden. Dieser Code war aber nicht sonderlich flexibel. Wie also kriegen wir unsere Pizza und können sie auch noch essen?

Ich mache schon seit Jahren Pizza und dachte mir, ich füge der Pizziaprozedur meine eigenen »Verbesserungen« hinzu ...



Etwas, das Sie in einer guten Zweigstelle nicht wollen. Sie wollen lieber NICHT wissen, was er auf seine Pizza packt.

Die Unterklassen entscheiden lassen

Ein Framework für die Pizzeria

Es gibt eine Möglichkeit, alle Aktivitäten zur Pizzaerstellung in der Klasse Pizzeria zu lokalisieren und den Zweigstellen dennoch die Freiheit für eigene regionale Pizzastile zu geben.

Dazu werden wir Folgendes machen: Wir werden die Methode erstellePizza() wieder in Pizzeria stecken, diesmal aber als eine **abstrakte Methode**, und dann eine Pizzeria-Unterklasse für jeden regionalen Stil erstellen.

Sehen wir uns zuerst die Änderungen an Pizzeria an:

PizzaStore ist jetzt abstrakt (siehe unten, warum).
↓

```
public abstract class Pizzeria {  
  
    Pizza bestellePizza(String typ) {  
        Pizza pizza;  
        pizza = erstellePizza(typ);  
  
        pizza.vorbereiten();  
        pizza.backen();  
        pizza.schneiden();  
        pizza.einpacken();  
  
        return pizza;  
    }  
  
    abstract erstellePizza(String typ);  
}
```

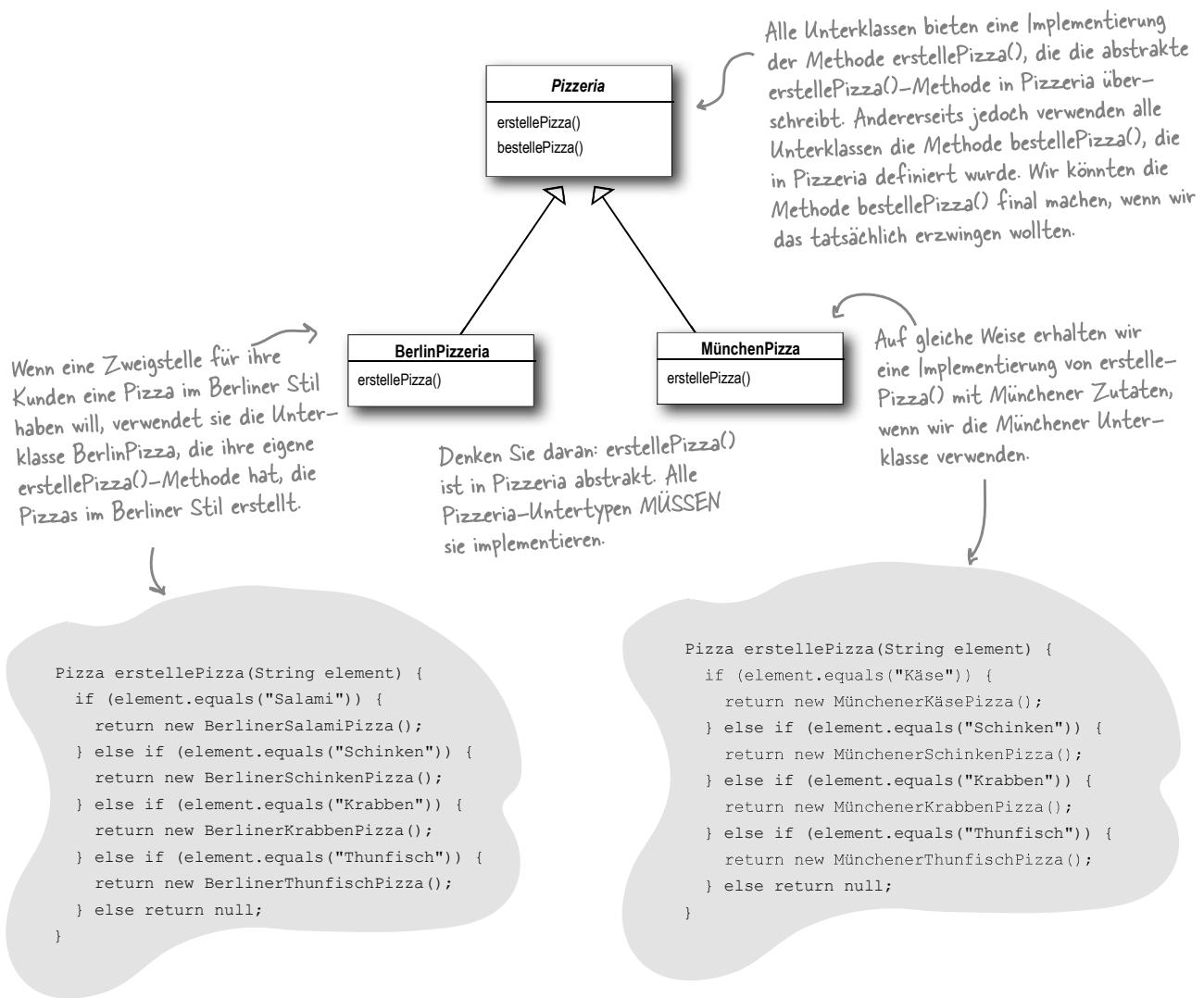
Jetzt ist erstellePizza wieder ein Aufruf einer Methode in Pizzeria anstatt in einem Fabrik-Objekt.
All das sieht völlig gleich aus ...
Jetzt haben wir unser Fabrik-Objekt in diese Methode verschoben.
Unsere >>Fabrikmethode<< ist nun eine abstrakte Methode in Pizzeria.

Jetzt wartet unsere Pizzeria nur noch auf Unterklassen. Wir werden Unterklassen für alle regionalen Typen (BerlinPizzeria, MünchenPizzeria, KölnPizzeria) erstellen, und jede Unterklasse wird selbst entscheiden, wie eine Pizza aussieht. Sehen wir uns an, wie das funktioniert.

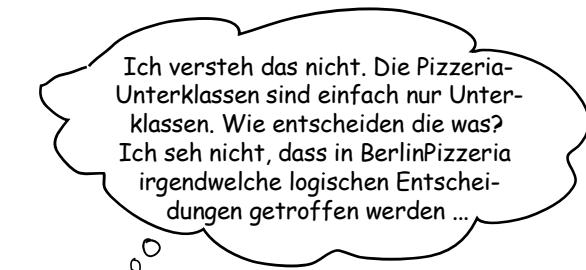
Die Unterklassen entscheiden lassen

Erinnern Sie sich daran, die Pizzeria hat bereits ein ausgefeiltes Bestellsystem in der Methode `erstellePizza()`. Sie möchten sicherstellen, dass dieses über alle Zweigstellen konsistent ist.

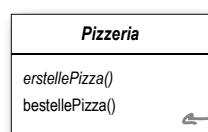
Was bei den regionalen Pizzerias variiert, ist die Art der Pizzas, die sie herstellen – Berlin hat dünne Krusten, München dicke und so weiter. Wir werden alle diese Variationen in die Methode `erstellePizza()` schieben und sie dafür verantwortlich machen, die richtige Art von Pizza zu erstellen. Das machen wir, indem wir jede Unterklasse von Pizzeria definieren lassen, wie die Methode `erstellePizza()` aussieht. Wir werden also eine Reihe konkreter Unterklassen von Pizzeria haben, die jeweils eigene Pizzavariationen besitzt, die alle in das Pizzeria-Framework passen und weiterhin die gut abgestimmte Methode `bestellePizza()` verwenden.



Wie entscheiden die Klassen?



Denken Sie darüber aus der Perspektive der bestellePizza()-Methode von Pizzeria nach: Sie wird in der abstrakten Pizzeria definiert, aber die konkreten Typen werden erst in den Unterklassen erstellt.



bestellePizza() wird in der abstrakten Pizzeria definiert, nicht in den Unterklassen. Die Methode hat also keine Ahnung, in welcher Unterklasse der Code wirklich ausgeführt und die Pizza erstellt wird.

Um das jetzt noch etwas weiter zu vertiefen: Die Methode bestellePizza() macht viele Dinge mit einem Pizza-Objekt (wie vorbereiten, backen, schneiden, verpacken). Weil Pizza abstrakt ist, hat bestellePizza() keine Vorstellung davon, welche konkreten Klassen einbezogen sind. Anders gesagt: Sie ist entkoppelt!



```
Pizza pizza = erstellePizza();
pizza.vorbereiten();
pizza.backen();
pizza.schneiden();
pizza.verpacken();
```

bestellePizza() ruft erstellePizza() auf, um tatsächlich ein Pizza-Objekt zu erhalten. Aber welche Art von Pizza wird sie erhalten? Die Methode bestellePizza() kann das nicht entscheiden. Sie weiß es nicht. Wer also entscheidet es?

Wenn bestellePizza() erstellePizza() aufruft, wird eine ihrer Unterklassen zu einer Aktion aufgerufen, und zwar, um eine Pizza zu erstellen. Welche Art von Pizza wird gemacht? Na, das wird durch die Wahl der Pizzeria festgelegt, bei der Sie die Pizza bestellen: BerlinPizzeria oder MünchenPizzeria.



Gibt es also Entscheidungen, die Unterklassen zur Laufzeit treffen? Nein, eigentlich nicht. Aus der Perspektive von bestellePizza() aber schon. Wenn Sie eine BerlinPizzeria wählen, entscheidet diese Unterklasse, was für eine Pizza erstellt wird. Die Unterklassen treffen also nicht wirklich eine »Entscheidung«: Sie haben entschieden, indem Sie die gewünschte Pizzeria ausgewählt haben – und die entscheidet dann, welche Art von Pizza gemacht wird.

Eröffnen wir also eine Pizzeria

Es hat seine Vorteile, eine Zweigstelle zu sein. Man erhält die ganze Pizzeria-Funktionalität umsonst. Die regionalen Zweigstellen müssen nur eine Unterklass von Pizzeria bilden und eine erstellePizza()-Methode anbieten, die ihre Art von Pizza implementiert. Hier werden wir uns um die drei wichtigen Pizzaarten für die Zweigstellen kümmern.

Dies ist der regionale Berliner Stil:

```

public class BerlinPizzeria extends Pizzeria {
    Pizza erstellePizza(String element) {
        if (element.equals("Salami")) {
            return new BerlinerSalamiPizza();
        } else if (element.equals("Schinken")) {
            return new BerlinerSchinkenPizza();
        } else if (element.equals("Krabben")) {
            return new BerlinerKrabbenPizza();
        } else if (element.equals("Thunfisch")) {
            return new BerlinerThunfischPizza();
        } else return null;
    }
}

```

Annotations pointing to the code:

- An arrow points from the text "erstellePizza() liefert eine Pizza, und die Unterklasse trägt die vollständige Verantwortung dafür, welche konkrete Pizza sie instanziert." to the line "return new BerlinerSalamiPizza();".
- An arrow points from the text "BerlinPizzeria erweitert Pizzeria und erbt also (unter anderem) die Methode bestellePizza()." to the line "super.bestellePizza(element);".
- An arrow points from the text "Wir müssen erstellePizza implementieren, da diese Methode in Pizzeria abstrakt ist." to the line "public Pizza erstellePizza(String element) {".
- An arrow points from the text "Dies ist der Ort, an dem wir unsere konkreten Klassen erstellen. Für jeden Pizzatyp erstellen wir einen Berliner Stil." to the line "if (element.equals("Salami")) {".
- An arrow points from the text "* Beachten Sie, dass die bestellePizza()-Methode der Superklasse keine Ahnung hat, was für Pizzas wir machen. Sie weiß nur, dass sie sie vorbereiten, backen, schneiden und verpacken kann!" to the line "return new BerlinerSalamiPizza();".

* Beachten Sie, dass die bestellePizza()-Methode der Superklasse keine Ahnung hat, was für Pizzas wir machen. Sie weiß nur, dass sie sie vorbereiten, backen, schneiden und verpacken kann!

Wenn wir unsere Pizzeria-Unterklassen erstellt haben, sollten wir daran gehen, ein oder zwei Pizzas zu bestellen. Aber bevor wir das tun, könnten Sie auf der nächsten Seite vielleicht mal versuchen, den Münchener und den Kölner Pizzeria-Stil zu erstellen.

Eine Fabrikmethode



Spitzen Sie Ihren Bleistift —————

BerlinPizzeria haben wir bereits zusammengeschustert. Nur zwei stehen noch aus, damit wir die Pizzeria-Kette ins Leben rufen können! Schreiben Sie hier die Implementierungen für MünchenPizzeria und KölnPizzeria:

Eine Fabrikmethode deklarieren

Mit nur ein paar Veränderungen an Pizzeria sind wir von der Instantiierung der konkreten Klassen über ein Objekt-Handle zu einem Satz von Unterklassen übergegangen, die jetzt diese Verantwortung übernehmen. Sehen wir uns das aus der Nähe an:

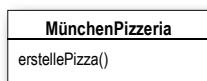
```
public abstract class Pizzeria {
    public Pizza bestellePizza(String typ) {
        Pizza pizza = erstellePizza(typ);
        pizza.vorbereiten();
        pizza.backen();
        pizza.schneiden();
        pizza.verpacken();

        return pizza;
    }

    protected abstract Pizza erstellePizza(String element);
}

// andere Methoden
}
```

Die Unterklassen von Pizzeria kümmern sich für uns in der Methode erstellePizza() um die Objekt-Instantiierung.



Die ganze Verantwortung für die Instantiierung von Pizzas wurde in eine Methode verschoben, die als Fabrik dient.



Code unter der Lupe

Eine Factory Method-Fabrikmethode kümmert sich um die Objekt-Erstellung und kapselt sie in einer Unterklasse. Damit wird der Client-Code in der Superklasse vom Objekt-Erstellungscode in der Unterklasse loskoppelt.

```
abstract Produkt fabrikMethode(String typ)
```

Eine Fabrikmethode kümmert sich um die Objekt-Erstellung und kapselt sie in einer Unterklasse. Das entkoppelt den Client-Code in der Superklasse von der Objekt-Erstellung in der Unterklasse.

Eine Fabrikmethode liefert ein Produkt zurück, das üblicherweise innerhalb der Methoden verwendet wird, die in der Superklasse definiert werden.

Eine Fabrikmethode isoliert den Client (den Code in der Superklasse, beispielsweise bestellePizza()) so, dass er nicht wissen muss, welche konkrete Art von Produkt tatsächlich erstellt wird.

Eine Fabrikmethode kann so parametrisiert sein (oder nicht), dass sie unter verschiedenen Formen eines Produkts auswählt.

Eine Pizza bestellen

Sehen wir uns an, wie es funktioniert: Bestellen wir Pizzas mit Factory Method



Wie also bestellen sie?

- ❶ Zuerst benötigen Tassilo und Ben eine Instanz von Pizzeria. Tassilo muss eine MünchenPizzeria instantiiieren, Ben eine BerlinPizzeria.
- ❷ Mit ihrer Pizzeria zur Hand rufen Ben und Tassilo jeweils die Methode bestellePizza() auf und übergeben ihr den gewünschten Typ Pizza (Salami, Krabben und so weiter).
- ❸ Um die Pizzas zu machen, wird die Methode erstellePizza() aufgerufen, die in den beiden Unterklassen BerlinPizzeria und MünchenPizzeria definiert ist. Ihren Definitionen entsprechend instantiierten BerlinPizzeria Pizzas im Berliner Stil und MünchenPizzeria Pizzas im Münchener Stil. In beiden Fällen wird die Pizza an die Methode bestellePizza() zurückgeliefert.
- ❹ Die Methode bestellePizza() hat keine Ahnung, was für eine Art Pizza erstellt wurde. Aber sie weiß, dass es eine Pizza ist, und macht, backt, schneidet und verpackt sie für Ben und Tassilo.

Sehen wir uns an, wie diese Pizzas auf Bestellung gemacht werden ...

Hinter den Kulissen

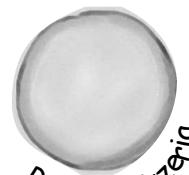


1

Gehen wir Bens Bestellung durch: Zuerst brauchen wir eine BerlinPizzeria:

```
Pizzeria berlinPizzeria = new BerlinPizzeria();
```

Erzeugt eine Instanz von BerlinPizzeria.



2

Jetzt haben wir eine Zweigstelle und können eine Bestellung aufnehmen:

```
berlinPizzeria.bestellePizza("Salami");
```

Auf der berlinPizzeria-Instanz wird die Methode bestellePizza aufgerufen (es wird die in Pizzeria definierte Methode ausgeführt).



3

Dann ruft die Methode bestellePizza() die Methode erstellePizza() auf:

```
Pizza pizza = erstellePizza("Salami");
```

Denken Sie daran, dass erstellePizza(), die Fabrikmethode, in der Unterklasse implementiert ist. In diesem Fall liefert sie eine Berliner Salamipizza zurück.



4

Schließlich haben wir eine rohe Pizza zur Verfügung und die Methode bestellePizza() kümmert sich darum, dass sie fertig gestellt wird:

```
pizza.vorbereiten();
pizza.backen();
pizza.schneiden();
pizza.verpacken();
```

Alle diese Methoden werden in der spezifischen Pizza definiert, die von der Fabrikmethode erstellePizza() zurückgeliefert wird, die in BerlinPizzeria definiert wurde.

Die Methode bestellePizza() erhält eine Pizza zurück, ohne zu wissen, welche konkrete Klasse es genau ist.

Die Pizza-Klassen

Uns fehlt nur noch eine Sache: Pizza!

Ohne ein paar Pizzas wird unsere Pizzeria nicht sehr beliebt werden. Implementieren wir also welche:

Wir beginnen mit einer abstrakten Pizza-Klasse, von der alle konkreten Pizzas abgeleitet werden.



```
public abstract class Pizza {  
    String name;  
    String teig;  
    String soße;  
    ArrayList beläge = new ArrayList();  
  
    void vorbereiten() {  
        System.out.println("Bereite " + name);  
        System.out.println("Werfe Teig ...");  
        System.out.println("Füge Soße hinzu ...");  
        System.out.println("Füge Beläge hinzu: ");  
        for (int i = 0; i < beläge.size(); i++) {  
            System.out.println(" " + beläge.get(i));  
        }  
    }  
  
    void backen() {  
        System.out.println("Backe 25 Minuten bei 350 Grad");  
    }  
  
    void schneiden() {  
        System.out.println("Schneide die Pizza diagonal in Stücke");  
    }  
  
    void verpacken() {  
        System.out.println("Packe die Pizza in die offizielle Pizzeria-Schachtel");  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Jede Pizza hat einen Namen, einen Typ Teig, einen Typ Soße und eine Anzahl Beläge.

Die abstrakte Klasse bietet ein paar elementare Vorgaben für das Backen, Schneiden und Verpacken.

Die Zubereitung besteht aus einer bestimmten Anzahl von Schritten in festgelegter Reihenfolge.

ZUR 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.

Wir brauchen nur noch ein paar konkrete Unterklassen ... wie wäre es damit, wenn wir mal Salamipizzas im Berliner und im Münchener Stil definierten:

```
public class BerlinerSalamiPizza extends Pizza {
    public BerlinerSalamiPizza() {
        name = "Salamipizza Berliner Art";
        teig = "Teig mit knuspriger Kruste";
        soße = "Marinara-Soße";

        beläge.add("Geriebener Parmesan");
        beläge.add("Salami in Scheiben");
    }
}
```

Eine Berliner Pizza hat eine eigene Marinara-Soße und eine dünne Kruste.

Und zwei Beläge, Salami und Parmesan!

```
public class MünchenerSalamiPizza extends Pizza {
    public MünchenerSalamiPizza() {
        name = "Deftige Salamipizza im Münchener Stil";
        teig = "Teig mit extra dicker Kruste";
        soße = "Tomatensoße";

        beläge.add("Mozzarella");
    }

    void schneiden() {
        System.out.println("Schneide die Pizza in rechteckige Stücke");
    }
}
```

Die Münchener Pizza verwendet eine Tomatensoße und hat eine extra dicke Kruste.

Auf die deftige Münchener Pizza kommt viel Mozzarella!

Für die Pizza im Münchener Stil wird außerdem die schneiden()-Methode überschrieben, damit die Pizza in rechteckige Stücke geschnitten wird.

Ein paar Pizzas machen

Sie haben lange genug gewartet, jetzt ist es Zeit für etwas Pizza

```
public class PizzaTestlauf {  
    public static void main(String[] args) {  
        Pizzeria berlinPizzeria = new BerlinPizzeria();  
        Pizzeria münchenPizzeria = new MünchenPizzeria();  
  
        Pizza pizza = berlinPizzeria.bestellePizza("Salami");  
        System.out.println("Ben hat eine " + pizza.getName() + " bestellt\n");  
  
        pizza = münchenPizzeria.bestellePizza("Salami");  
        System.out.println("Tassilo hat eine " + pizza.getName() + " bestellt\n");  
    }  
}
```

Erst erstellen wir zwei unterschiedliche Pizzerias.

Dann verwenden wir die eine, um Bens Bestellung zu erledigen.

Und die andere für Tassilos Bestellung.

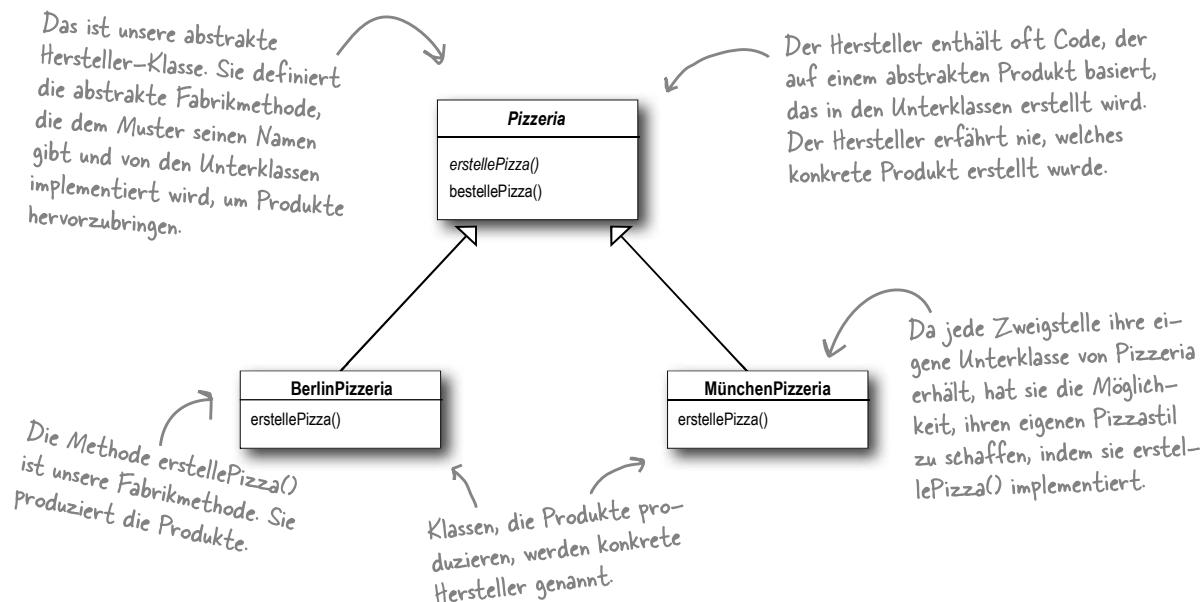
```
Datei Bearbeiten Fenster Hilfe WasWollenSieAufihrePizza?  
% java PizzaTestlauf  
--- Mache eine Salamipizza Berliner Art ---  
Bereite Salami-Pizza Berliner Art  
Werfe Teig ...  
Füge Soße hinzu ...  
Füge Beläge hinzu:  
    Geriebener Parmesan  
    Salami in Scheiben  
Backe 25 Minuten bei 350  
Schneide die Pizza diagonal in Stücke  
Packe die Pizza in die offizielle Pizzeriaschachtel  
Ben hat eine Salami-Pizza Berliner Art bestellt  
  
--- Mache eine deftige Salamipizza im Münchener Stil ---  
Bereite deftige Salamipizza im Münchener Stil  
Werfe Teig ...  
Füge Soße hinzu ...  
Füge Beläge hinzu:  
    Mozzarella  
Backe 25 Minuten bei 350  
Schneide die Pizza in rechteckige Stücke  
Packe die Pizza in die offizielle Pizzeriaschachtel  
Tassilo hat eine deftige Salamipizza im Münchener Stil bestellt  
%
```

Beide Pizzas werden vorbereitet, die Zutaten hinzugefügt und die Pizzas dann gebacken, geschnitten und verpackt. Unsere Superklasse musste sich nie um die Details kümmern. Das hat alles die Unterklasse erledigt, indem sie einfach die richtige Pizza instantiiert hat.

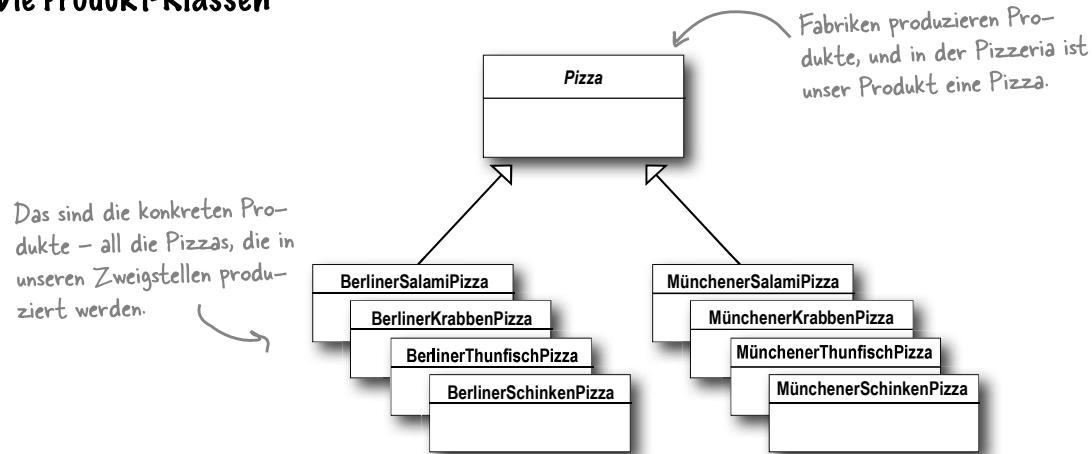
Jetzt ist es endlich Zeit, dem Factory Method-Muster zu begegnen

Alle Factory-Muster kapseln die Objekt-Erstellung. Das Factory Method-Muster kapselt die Objekt-Erstellung, indem es die Unterklassen entscheidet lässt, welche Objekte erstellt werden. Sehen wir uns diese Klassendiagramme an, um zu sehen, wer die Mitspieler in diesem Muster sind:

Die Hersteller-Klassen



Die Produkt-Klassen

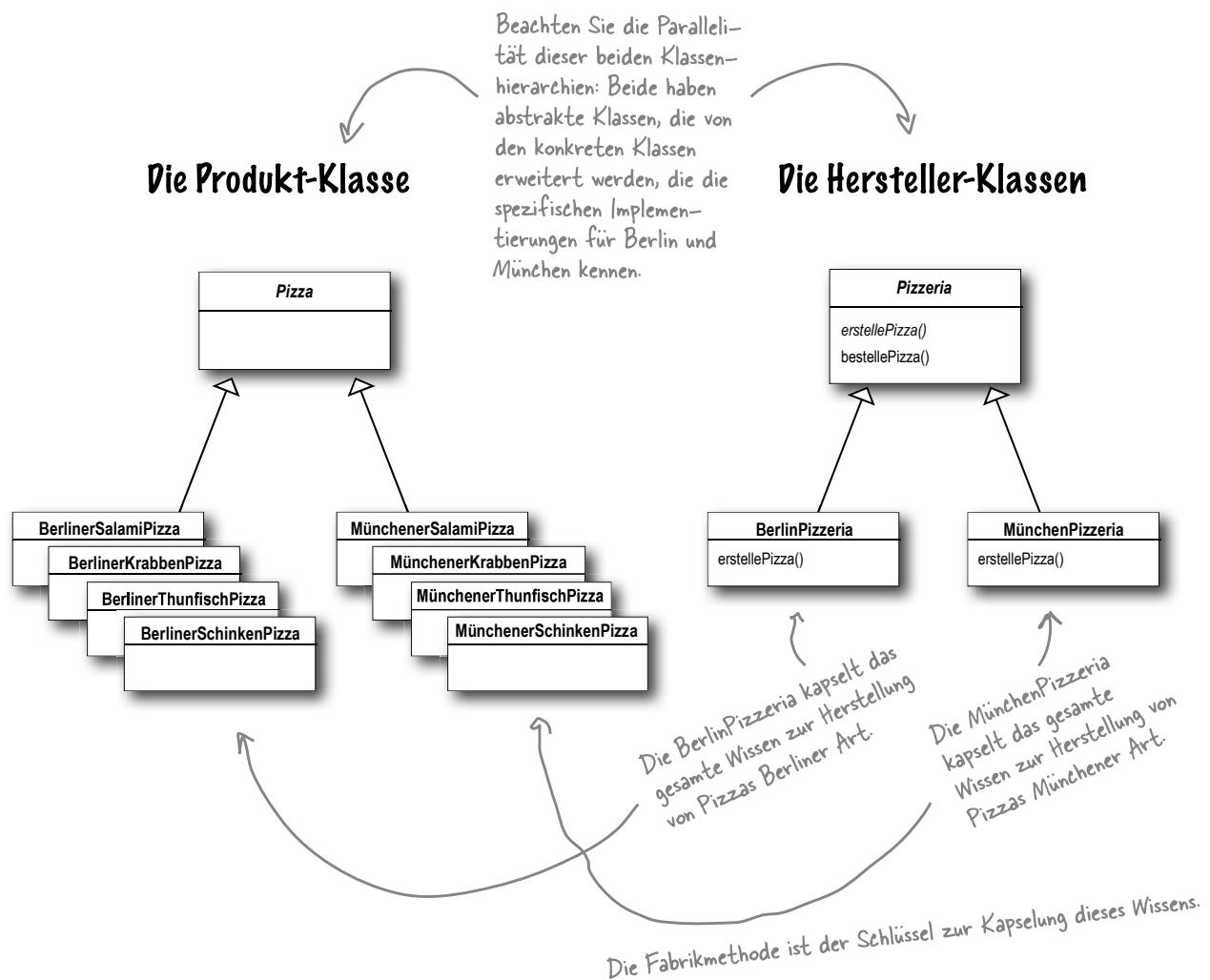


Hersteller und Produkte

Eine andere Perspektive: parallele Klassenhierarchien

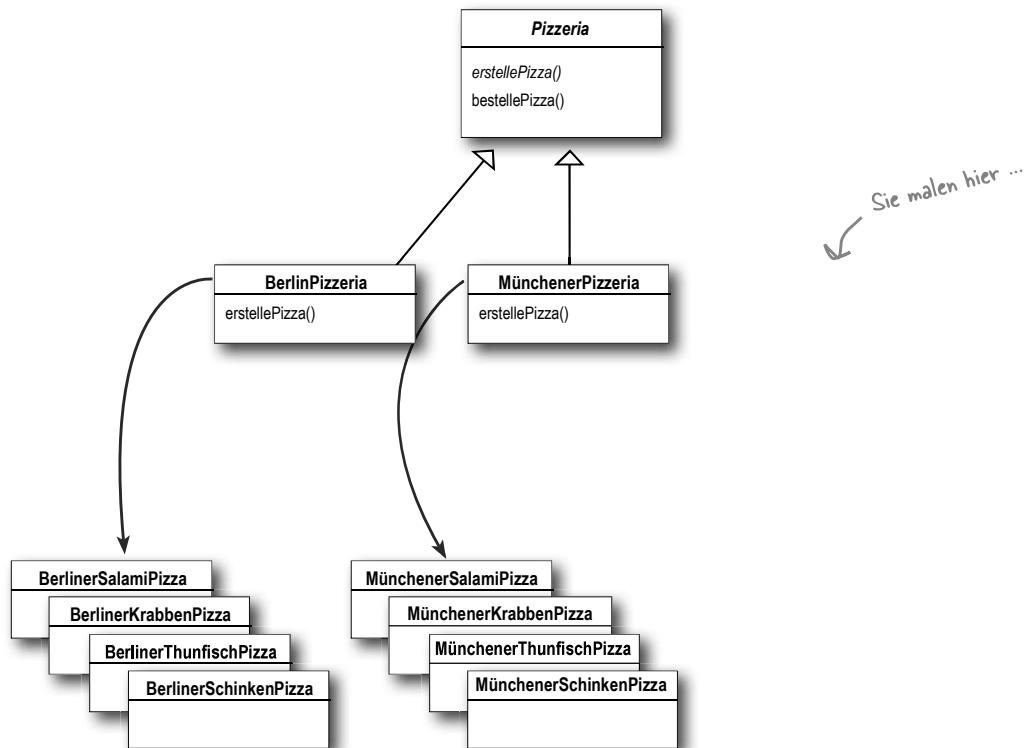
Wir haben gesehen, dass die Fabrikmethode ein Framework bietet, indem sie eine `erstellePizza()`-Methode bereitstellt, die mit einer Fabrikmethode kombiniert ist. Auch in der Art und Weise, wie dieses Muster die Kenntnis über die Produkte in den einzelnen Herstellern kapselt, lässt sich dieses Muster als Framework betrachten.

Sehen wir uns die beiden parallelen Klassenhierarchien und ihr Verhältnis zueinander an:



Entwurfspuzzle

Wir brauchen noch einen anderen Typ Pizza für die verrückten Kölner (auf *nette* Weise verrückt natürlich). Zeichnen Sie einen weiteren parallelen Satz von Klassen, um unserer Pizzeria einen regionalen Typ für Köln hinzuzufügen.



Gut, und jetzt schreiben Sie die fünf *seltsamsten* Dinge auf, die Sie sich auf einer Pizza vorstellen können. Dann können Sie in Köln das Pizzageschäft aufnehmen!

Die Definition des Factory Method-Musters

Die Definition des Factory Method-Musters

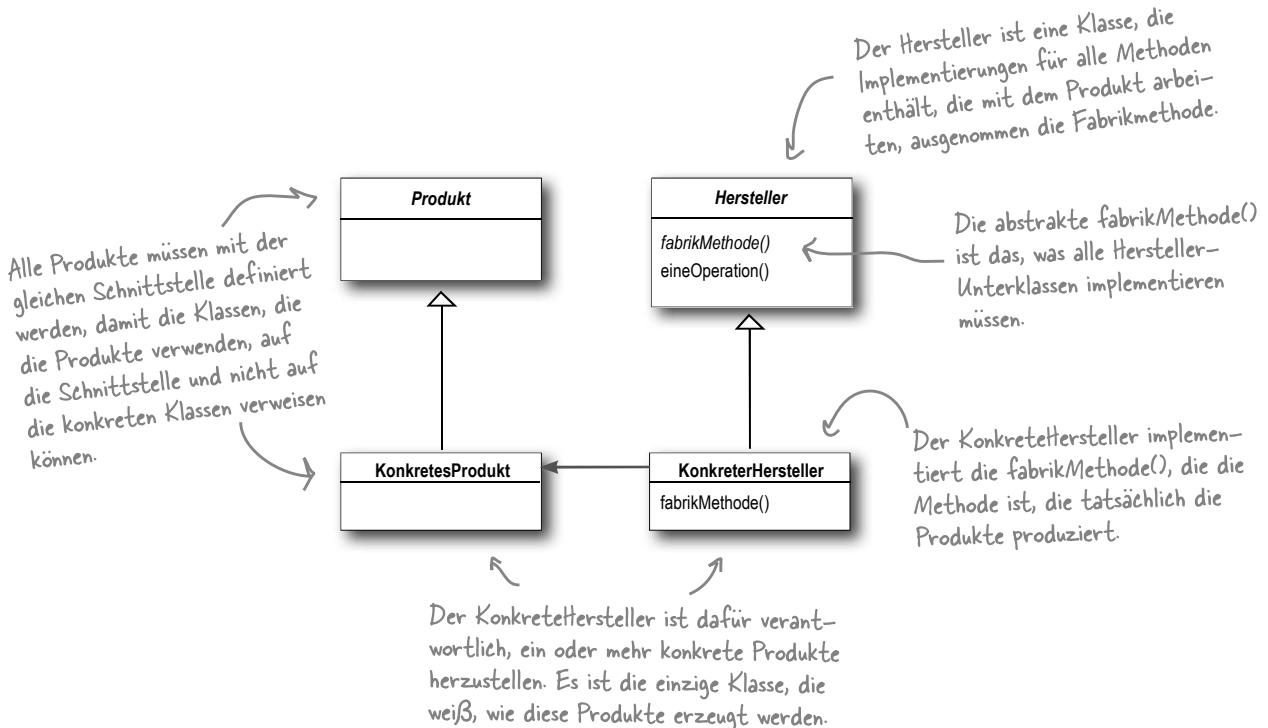
Es ist an der Zeit, die offizielle Definition des Factory Method-Musters abzuliefern:

Das Factory Method-Muster definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instantiiert werden. Factory Method ermöglicht einer Klasse, die Instantiiierung in Unterklassen zu verlagern.

Wenn Sie das Klassendiagramm unten betrachten, können Sie erkennen, dass der abstrakte Hersteller Ihnen eine Schnittstelle mit einer Methode zum Erstellen von Objekten bietet. Diese Methode wird auch als »Fabrikmethode« bezeichnet. Alle anderen Methoden, die in der abstrakten Hersteller-Klasse implementiert sind, sind so geschrieben, dass sie auf den Produkten operieren, die die Fabrikmethode produziert. Erst die Unterklassen bieten tatsächlich eine Implementierung der Fabrikmethode und produzieren damit Produkte.

Wie in der offiziellen Definition hören Sie häufig Entwickler davon sprechen, dass das Factory Method-Muster die Unterklassen entscheidet, welche Klasse instantiiert wird. Von »entscheiden« sprechen sie nicht, weil das Muster den Unterklassen erlaubt, zur Laufzeit eigenständige Entscheidungen zu treffen, sondern weil die Hersteller-Klasse so geschrieben ist, dass sie keine Kenntnis von den Produkten hat, die tatsächlich hergestellt werden. Das wird nur durch die Wahl der Unterklasse entschieden, die verwendet wird.

Sie könnten sie fragen, was sie mit »entscheiden« meinen; aber wir sind uns sicher, dass Sie das jetzt besser verstehen als diese Entwickler!



Es gibt keine
Dummen Fragen

F: Was ist der Vorteil des Factory Method-Musters, wenn man nur einen konkreten Hersteller hat?

A: Wenn man nur einen konkreten Hersteller hat, ist das Factory Method-Muster nützlich, weil die Implementierung des Produkts von seiner Verwendung entkoppelt wird. Wenn Sie zusätzliche Produkte hinzufügen oder die Implementierung eines Produkts ändern, hat das keine Auswirkungen auf Ihren Hersteller (weil der Hersteller nicht fest an ein KonkretesProdukt gekoppelt ist).

F: Könnte man sagen, dass unsere Berliner und Münchener Pizzerias anhand der einfachen Fabrik implementiert wurden? Sie sehen so aus.

A: Sie sind ähnlich, werden aber auf andere Weise verwendet. Auch wenn die Implementierungen der konkreten Pizzerias der EinfachePizzaFabrik stark ähneln, sollten Sie daran denken, dass die konkreten Pizzerias eine Klasse erweitern, in der erstellePizza() als eine abstrakte Methode definiert wurde. Die Pizzerias können selbst entscheiden, wie sie das Verhalten der Methode erstellePizza() definieren. Bei einer einfachen Fabrik ist die Fabrik ein anderes Objekt, das mit der Pizzeria zusammengesetzt wird.

F: Sind die Fabrikmethode und der Hersteller immer abstrakt?

A: Nein. Sie können eine Basis-Fabrikmethode definieren, die ein konkretes Produkt herstellt. Dann haben Sie jederzeit die Möglichkeit, Produkte zu erstellen, auch wenn es keine Unterklassen des Herstellers gibt.

F: Jede Pizzeria kann auf Basis des übergebenen Typs vier verschiedene Typen von Pizzas machen. Machen alle konkreten Hersteller mehrere Produkte, oder machen sie manchmal auch nur eins?

A: Was wir hier implementieren, nennt man eine parametrisierte Fabrikmethode. Eine solche kann, wie Sie erkannt haben, auf Basis eines übergebenen Parameters

verschiedene Objekte erstellen. Häufig aber produziert eine Fabrik auch nur ein Objekt und ist nicht parametrisiert. Beides sind gültige Ausprägungen des Musters.

F: Ihre parametrisierten Typen scheinen nicht »typsicher« zu sein. Es wird ja bloß ein String übergeben! Was ist, wenn ich eine »KrapfenPizza« bestelle?

A: Da haben Sie absolut Recht; dies würde das verursachen, was man branchenüblich einen »Laufzeifehler« nennt. Es gibt einige ausgefeilte Techniken, die eingesetzt werden können, um Parameter »typsicherer« zu machen, oder anders gesagt, sicherzustellen, dass Fehler in Parametern zur Kompilierzeit abgefangen werden können. Sie können beispielsweise Objekte erstellen, die die Parameter-typen repräsentieren, statische Konstanten oder, in Java 5, Enums verwenden.

F: Ich bin immer noch etwas verwirrt in Bezug auf den Unterschied zwischen der einfachen Fabrik und dem Factory Method-Muster. Sie scheinen mir sehr ähnlich. Der einzige Unterschied ist, dass bei Factory Method die Klasse, die die Pizzas liefert, eine Unterklasse ist. Können Sie das erklären?

A: Sie haben damit Recht, dass die Unterklassen eine starke Ähnlichkeit zur einfachen Fabrik aufweisen. Sie sollten sich die einfache Fabrik aber als eine einmalige Angelegenheit vorstellen. Bei Factory Method erstellen Sie aber ein Framework, in dem die Unterklassen entscheiden, welche Implementierung verwendet wird. Beispielsweise bietet die Methode erstellePizza() in Factory Method ein allgemeines Framework zur Erstellung von Pizzas, das sich auf eine Fabrikmethode stützt, um die konkreten Klassen zu erstellen, die sich tatsächlich um die Herstellung einer Pizza kümmern. Indem Sie Unterklassen von Pizzeria bilden, entscheiden Sie, welche konkreten Produkte in die Erstellung der Pizza eingehen, die erstellePizza() zurückliefert. Im Vergleich dazu liefert Ihnen die einfache Fabrik zwar eine Möglichkeit, die Objekt-Erstellung zu kapseln, aber trotzdem nicht die Flexibilität, die Factory Method bietet, weil Sie keine Möglichkeit haben, die Produkte zu variieren, die Sie herstellen.

Meister und Schüler



Meister und Schüler

Meister: Sage mir, Grashüpfer, wie deine Ausbildung vorangeht.

Schüler: Meister, ich habe mein Studium des »Kapseln«, was veränderlich ist, vorangetrieben.

Meister: Fahre fort ...

Schüler: Ich habe gelernt, dass man den Code kapseln kann, der Objekte erstellt. Hat man Code, der konkrete Klassen instantiiert, ist das ein Bereich häufiger Veränderungen. Ich habe eine Technik gelernt, die man als »Fabrik« bezeichnet, und die ermöglicht es, das Verhalten der Instantiierung zu kapseln.

Meister: Und diese »Fabriken«. Welche Vorteile bieten sie?

Schüler: Viele. Indem ich meinen gesamten Erstellungscode in ein Objekt oder eine Methode packe, verhindere ich Verdopplungen in meinem Code und biete einen Ort an, an dem die Wartung erfolgen kann. Das heißt auch, dass Clients sich nur auf Schnittstellen stützen und nicht auf die konkreten Klassen, die zur Instantiierung der Objekte erforderlich sind. Wie ich während meiner Studien erfahren habe, erlaubt mir dieses, auf eine Schnittstelle, nicht auf eine Implementierung zu programmieren. Und das macht meinen Code flexibler und in Zukunft einfacher erweiterbar.

Meister: Ja Grashüpfer, deine OO-Instinkte entwickeln sich. Hast du heute Fragen an deinen Lehrer?

Schüler: Meister, ich weiß, dass ich durch die Kapselung der Objekt-Erstellung Code auf Abstraktionen schreiben und meinen Client-Code von den eigentlichen Implementierungen entkoppeln kann. Aber mein Fabrik-Code muss trotzdem noch konkrete Klassen verwenden, um echte Objekte zu instantiieren. Streue ich mir da nicht bloß Sand in die Augen?

Meister: Im Leben ist es so, dass man Objekte erstellt, Grashüpfer. Wir müssen Objekte erstellen, sonst werden wir nie auch nur ein einziges Java-Programm schreiben. Aber kennen wir diese Lebensweisheit, können wir unseren Code so entwerfen, dass wir diesen Code so einschließen wie eine Sanduhr den Sand, den du in deine Augen streuen wolltest. Haben wir ihn eingeschlossen, können wir den Code für die Objekt-Erstellung schützen und pflegen. Lassen wir unseren Erstellungscode einfach herumliegen, wird ihn der erste Wind in alle Himmelsrichtungen verwehen.

Schüler: Meister, ich sehe die Wahrheit in diesen Worten.

Meister: So wie ich wusste, dass du sie sehen würdest. Jetzt, bitte, gehe und meditiere über Objekt-Abhängigkeiten.

Eine sehr abhängige Pizzeria

 Spitzen Sie Ihren Bleistift

Tun wir mal so, als hätten Sie noch nie etwas von einer OO-Factory gehört. Hier ist eine Version der Pizzeria, die keine Fabrik verwendet. Zählen Sie, von wie vielen konkreten Pizza-Objekten diese Klasse abhängig ist. Von wie vielen Objekten wäre sie abhängig, wenn Sie dieser Pizzeria Pizzas nach Kölner Art hinzufügen würden?

```
public class AbhangigePizzeria {

    public Pizza erstellePizza(String art, String typ) {
        Pizza pizza = null;
        if (art.equals("Berlin")) {
            if (typ.equals("Salami")) {
                pizza = new BerlinerSalamiPizza();
            } else if (typ.equals("Schinken")) {
                pizza = new BerlinerSchinkenPizza();
            } else if (typ.equals("Krabben")) {
                pizza = new BerlinerKrabbenPizza();
            } else if (typ.equals("Thunfisch")) {
                pizza = new BerlinerThunfischPizza();
            }
        } else if (art.equals("München")) {
            if (typ.equals("Salami")) {
                pizza = new MünchenerSalamiPizza();
            } else if (typ.equals("Schinken")) {
                pizza = new MünchenerSchinkenPizza();
            } else if (typ.equals("Krabben")) {
                pizza = new MünchenerKrabbenPizza();
            } else if (typ.equals("Thunfisch")) {
                pizza = new MünchenerThunfischPizza();
            }
        } else {
            System.out.println("Fehler: Ungültiger Pizzatyp");
            return null;
        }
        pizza.vorbereiten();
        pizza.backen();
        pizza.schneiden();
        pizza.verpacken();
        return pizza;
    }
}
```

Kümmert sich um
alle Pizzas Berliner
Art

Kümmert sich um
alle Pizzas Mün-
chener Art

Hier können Sie Ihre
Antwort notieren:

Anzahl

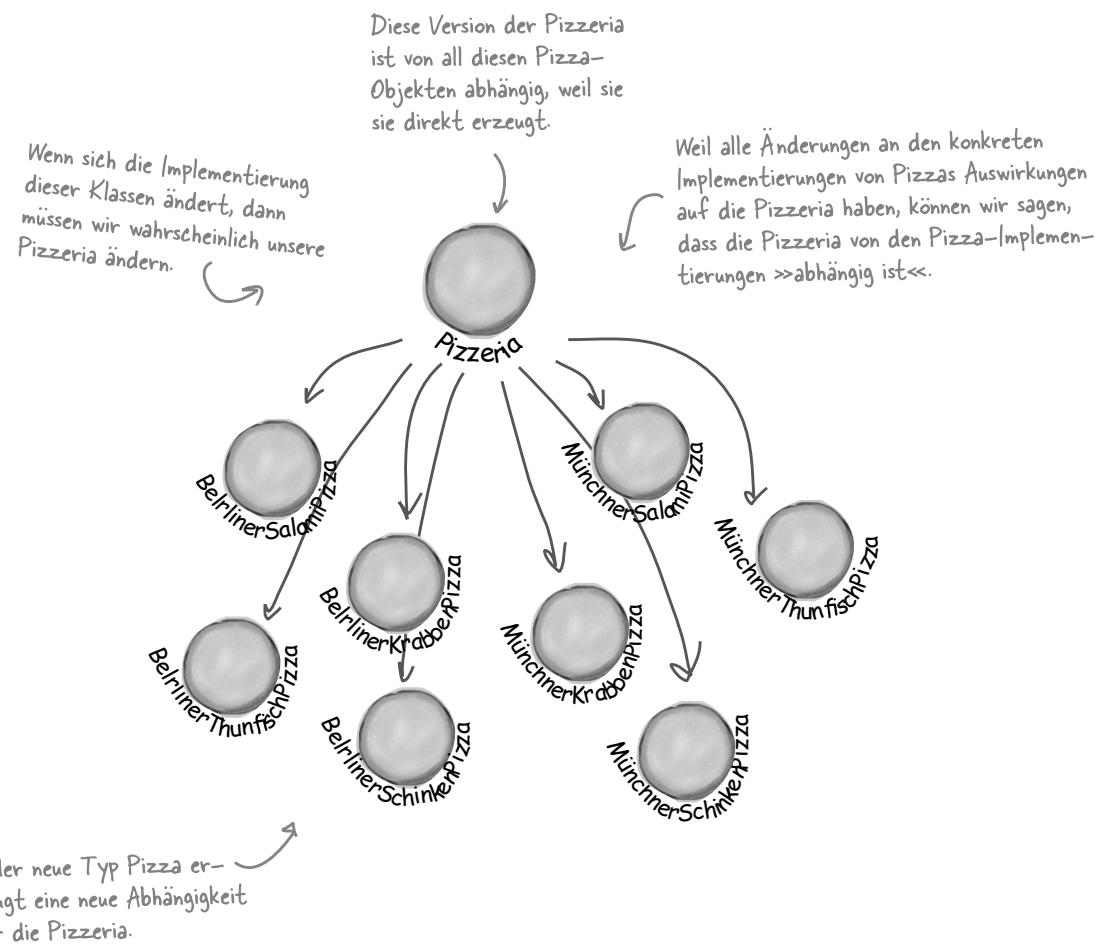
Anzahl mit Kölner Art

Objekt-Abhängigkeiten

Ein Blick auf Objekt-Abhängigkeiten

Wenn Sie ein Objekt direkt instantiiieren, sind Sie von seiner konkreten Klasse abhängig. Sehen Sie sich noch einmal unsere sehr abhängige Pizzeria auf der vorangegangenen Seite an. Sie erstellt alle Pizza-Objekte direkt in der Pizzeria-Klasse, anstatt das an eine Fabrik zu delegieren.

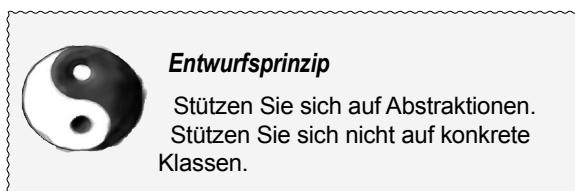
Wenn wir ein Diagramm zeichnen, das diese Version der Pizzeria und alle Objekte repräsentiert, von denen sie abhängig ist, sähe das so aus:



Das Prinzip der Umkehrung der Abhängigkeiten

Es sollte jetzt ziemlich offensichtlich sein, dass es eine »gute Sache« ist, wenn wir in unserem Code die Abhängigkeiten von konkreten Klassen reduzieren. Und tatsächlich gibt es auch ein OO-Entwurfsprinzip, das eine formale Fassung dieser Feststellung bietet. Es hat sogar einen großen formellen Namen: *Dependency Inversion Principle* (oder für die, die eine knappe deutsche Form bevorzugen: »Abhängigkeitsumkehrungsprinzip«).

Hier ist das allgemeine Prinzip:



Ein weiteres Schlagwort, das Sie einsetzen können, um die Vorgesetzten im Raum zu beeindrucken! Ihre Gehaltserhöhung wird die Kosten für dieses Buch sicher locker wieder einspielen. Und außerdem gewinnen Sie die Bewunderung Ihrer Entwicklerkollegen.

Zunächst scheint das dem »Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung« ziemlich ähnlich, stimmt's? Es ist ihm ähnlich. Aber das Prinzip der Umkehrung von Abhängigkeiten macht eine noch deutlichere Aussage zur Abstraktion. Es schlägt vor, dass unsere hochstufigen Komponenten nicht von unseren niedrigstufigen Elementen abhängig sein sollen. Stattdessen sollten sie sich *beide* auf Abstraktionen stützen.

Aber was zum Teufel heißt das?

Na, schauen wir doch einfach noch einmal auf unser Pizzeria-Diagramm auf der vorangegangenen Seite. Pizzeria ist unsere »hochstufige Komponente«, die Pizza-Implementierungen sind unsere »niedrigstufigen Komponenten«, und Pizzeria ist offensichtlich von den konkreten Pizza-Klassen abhängig.

Jetzt sagt uns dieses Prinzip, dass wir unseren Code auf Abstraktionen stützen sollen, nicht auf konkrete Klassen. Das gilt für unsere hochstufigen und unsere niedrigstufigen Module gleichermaßen.

Wie wir das erreichen? Überlegen wir noch mal, wie wir dieses Prinzip auf unsere Implementierung der sehr abhängigen Pizzeria anwenden würden.

Eine »hochstufige« Komponente ist eine Klasse mit einem Verhalten, das auf Basis anderer, »niedrigstufiger« Komponenten basiert. Zum Beispiel ist Pizzeria eine hochstufige Komponente, weil ihr Verhalten in Abhängigkeit von den Pizzas definiert ist – sie erstellt all die verschiedenen Pizza-Objekte, bereitet sie vor, backt sie, schneidet sie und verpackt sie. Die Pizzas hingegen, die sie verwendet, sind niedrigstufige Komponenten.

Abhängigkeitsumkehrungsprinzip

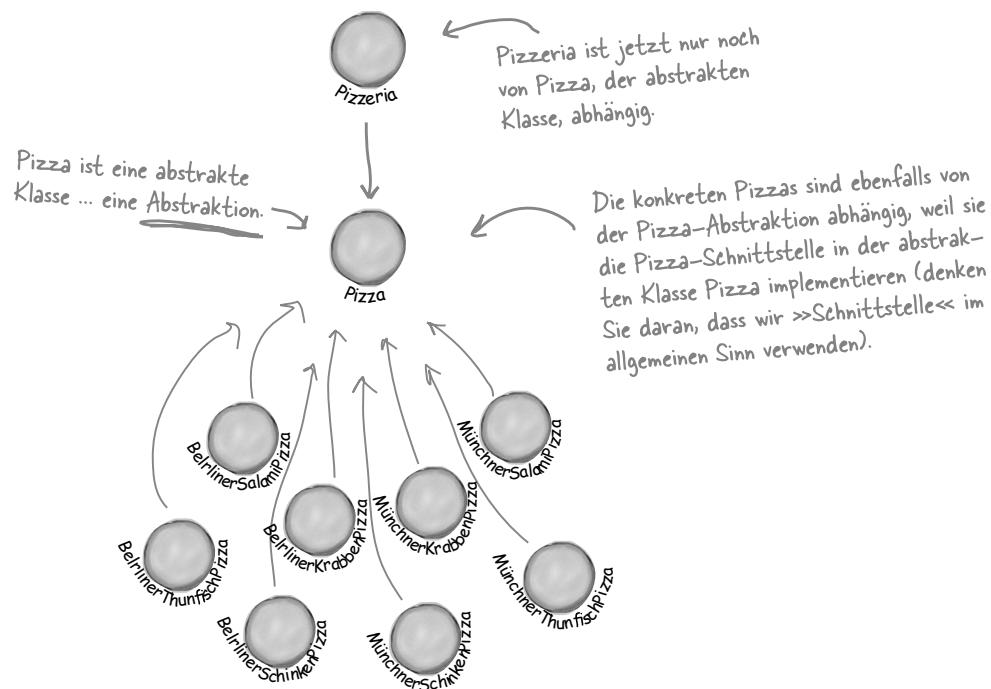
Das Prinzip anwenden

Das Hauptproblem mit der sehr abhängigen Pizzeria ist, dass sie von allen Pizzatypen abhängig ist, weil sie die konkreten Typen in der Methode bestellePizza() tatsächlich instantiiert.

Obwohl wir eine Abstraktion erstellt haben, Pizza, erzeugen wir im Code trotzdem noch konkrete Pizzas. Diese Abstraktion leistet also nicht besonders viel für uns.

Wie können wir diese Instantierungen aus der Methode bestellePizza herauskriegen? Na, wie wir wissen, ist es genau das, was uns das Factory Method-Muster ermöglicht.

Nachdem wir Factory Method auf unser Diagramm angewendet haben, sieht dieses folgendermaßen aus:



Sie bemerken sicher, dass nach der Anwendung der Factory Method die hochstufigen Komponenten, die Pizzeria, und die niedrigstufigen Komponenten, die Pizzas, gleichermaßen von Pizza, der Abstraktion, abhängig sind. Factory Method ist nicht die einzige Technik dafür, dem Prinzip der Umkehrung der Abhängigkeiten treu zu bleiben, aber es ist eine der mächtigsten.



Wo ist die »Umkehrung« im Prinzip der Umkehrung der Abhängigkeiten?

Die »Umkehrung« taucht in der Bezeichnung Prinzip der Umkehrung der Abhängigkeiten auf, weil es die Art und Weise umkehrt, wie Sie sich einen OO-Entwurf üblicherweise vorstellen. Sehen Sie sich das Diagramm auf der vorigen Seite an. Achten Sie darauf, dass die niedrigstufigen Komponenten jetzt von einer hochstufigen Abstraktion abhängig sind. Die hochstufige Komponente ist gleichermaßen an diese Abstraktion gebunden. Die von oben nach unten laufende Karte der Abhängigkeiten, die wir vor ein paar Seiten gezeichnet haben, hat sich umgekehrt, denn jetzt sind sowohl die hochstufigen als auch die niedrigstufigen Komponenten von der Abstraktion abhängig.

Gehen wir doch noch mal die Überlegungen durch, die hinter einem typischen Entwurfsvorgang stehen, und überlegen, wie es unsere Überlegungen zu dem Entwurf umkehren kann, wenn wir das Prinzip darin einführen ...

Stellen Sie Ihr Denken auf den Kopf

Stellen Sie Ihr Denken auf den Kopf



Also: Sie müssen eine Pizzeria implementieren. Was ist der erste Gedanke, der Ihnen in den Kopf kommt?

Sollte Ihre Pizzeria nichts über die konkreten Pizzatypen wissen müssen, weil sie dann von all diesen konkreten Klassen abhängig wäre!

Und jetzt wollen wir Ihr Denken einmal umkehren ... anstatt oben zu beginnen, beginnen Sie bei den Pizzas und überlegen sich, was Sie abstrahieren können.

Richtig! Sie denken an die Abstraktion *Pizza*. Also gehen Sie jetzt doch mal zurück und denken Sie noch mal über die Pizzeria nach.

Warm. Aber dazu müssten Sie sich auf eine Fabrik stützen, um diese konkreten Klassen aus der Pizzeria herauszukriegen. Wenn Sie das erledigt haben, sind Ihre verschiedenen konkreten Pizzatypen *und* Ihre Pizzeria nur noch von einer Abstraktion abhängig. Wir sind von einem Entwurf ausgegangen, bei dem die Pizzeria von konkreten Klassen abhängig war, und haben diese Abhängigkeiten (und Ihr Denken) umgekehrt.

Ein paar Richtlinien, die Ihnen bei der Befolgung des Musters helfen

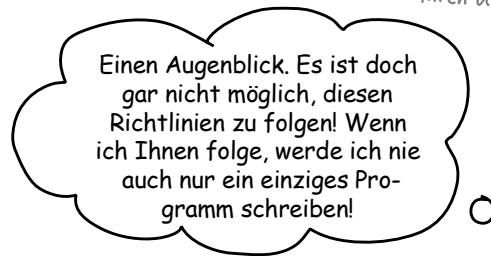
Die folgenden Richtlinien können Ihnen helfen, OO-Entwürfe zu vermeiden, die das Prinzip von der Umkehrung der Abhängigkeiten verletzen:

- Keine Variable sollte eine Referenz auf eine konkrete Klasse halten.
- Keine Klasse sollte von einer konkreten Klasse abgeleitet sein.
- Keine Methode sollte eine implementierte Methode einer ihrer Basisklassen überschreiben.

Wenn Sie `new` verwenden, halten Sie eine Referenz auf eine konkrete Klasse. Verwenden Sie eine Factory, um das zu umgehen!

Wenn Sie von einer konkreten Klasse ableiten, sind Sie von einer konkreten Klasse abhängig. Leiten Sie von einer Abstraktion wie einem Interface oder einer abstrakten Klasse ab.

Wenn Sie eine implementierte Methode überschreiben, war Ihre Basisklasse keine Abstraktion, die als Ausgangspunkt tauglich ist. Zweck der in der Basisklasse implementierten Methoden ist, dass sie von allen Unterklassen geteilt werden.



Sie haben vollkommen Recht! Wie viele unserer Prinzipien ist auch dieses eher eine Richtlinie, nach der Sie streben sollten, als eine Regel, die Sie immer befolgen müssen. Klar verletzen alle jemals geschriebenen Java-Programme diese Richtlinien!

Aber wenn Sie diese Richtlinien verinnerlichen und sie beim Entwerfen im Hinterkopf haben, wissen Sie, wann Sie das Prinzip verletzen und dass Sie dazu auch einen guten Grund haben. Wenn Sie eine Klasse haben, bei der Änderungen unwahrscheinlich sind und Sie das genau wissen, bricht die Welt nicht zusammen, wenn Sie in Ihrem Code eine konkrete Klasse instantiiieren. Denken Sie darüber nach. Wir instantiieren ständig String-Objekte, ohne darüber nachzudenken. Verletzen wir damit das Prinzip? Ja. Ist das in Ordnung? Ja. Warum? Weil es sehr unwahrscheinlich ist, dass String sich jemals ändern wird.

Wenn es aber wahrscheinlich ist, dass sich eine von Ihnen geschriebene Klasse ändert, dann haben Sie ein paar gute Techniken wie das Factory Method-Muster, um die Änderung zu kapseln.



Zutatenfamilien

Inzwischen in der Pizzeria

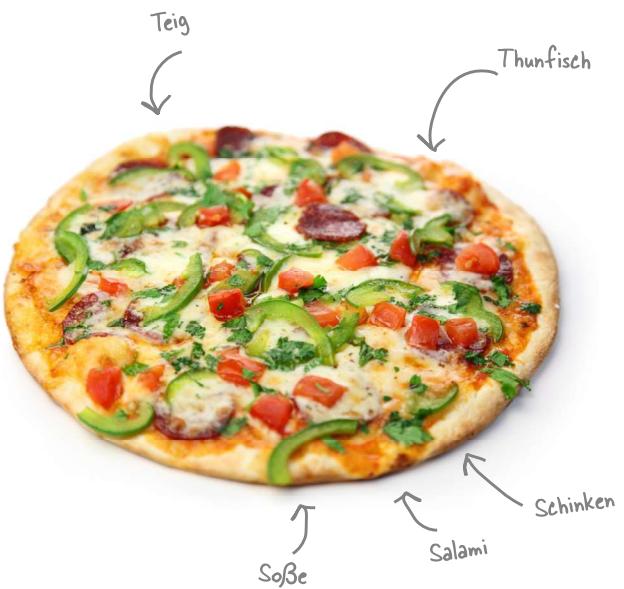
Der Entwurf für die Pizzeria nimmt langsam Gestalt an: Er besitzt ein flexibles Framework und macht gute Arbeit, wenn es um die Befolgung der Entwurfsprinzipien geht.

Der Schlüssel zum Erfolg der Pizzeria Objekthausen waren immer die frischen, hochwertigen Zutaten. Aber während des Einsatzes des neuen Frameworks haben Sie festgestellt, dass Ihre Zweigstellen zwar Ihren *Prozeduren* folgen, einige Zweigstellen für ihre Pizzas aber schlechtere Zutaten verwendet haben, um die Kosten zu reduzieren und ihre Gewinnspanne zu erhöhen. Sie wissen, dass Sie etwas tun müssen, weil das dem Objekthausener Markenzeichen langfristig schaden wird!

Konsistenz bei Ihren Zutaten sichern

Wie also sichern Sie, dass alle Zweigstellen hochwertige Zutaten verwenden? Sie bauen eine Fabrik, die sie herstellt und an Ihre Zweigstellen ausliefert!

Dieser Plan hat allerdings einen Haken: Ihre Zweigstellen befinden sich in unterschiedlichen Regionen, und etwas, das in Berlin eine rote Soße ist, ist in München noch lange keine rote Soße. Es gibt also einen Satz von Zutaten, der nach Berlin geliefert werden muss, und einen *anderen* Satz, der nach München geliefert werden muss. Sehen wir uns das aus der Nähe an:



 **Münchener Pizza-Speisekarte**

salamipizza
Tomatensoße, italienische Salami, Mozzarella

schinkenpizza
Tomatensoße, Tiroler Speck, Zwiebeln, Schmand, Mozzarella

Krabbenpizza
Tomatensoße, Krabben, Knoblauch, Mozzarella

Thunfischpizza
Tomatensoße, Thunfisch, schwarze Oliven, Kapern, Mozzarella

Wir haben die gleichen Produktfamilien (Teig, Soße, Salami, Schinken, Thunfisch), aber regional unterschiedliche Implementierungen.

 **Berliner Pizza-Speisekarte**

salamipizza
Marinara-Soße, spanische Salami, Parmesan

schinkenpizza
Marinara-Soße, Rucola, Parmaschinken, Mozzarella, Knoblauch, Parmesan

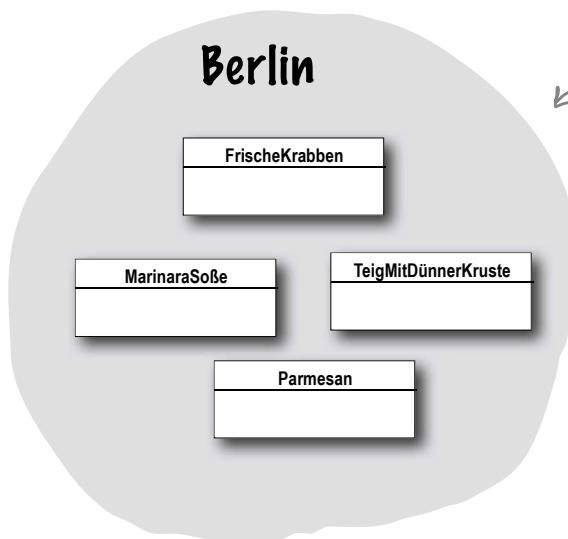
Krabbenpizza
Marinara-Soße, Krabben, Knoblauch, Parmesan

Thunfischpizza
Marinara-Soße, Thunfisch, Zwiebeln, Parmesan

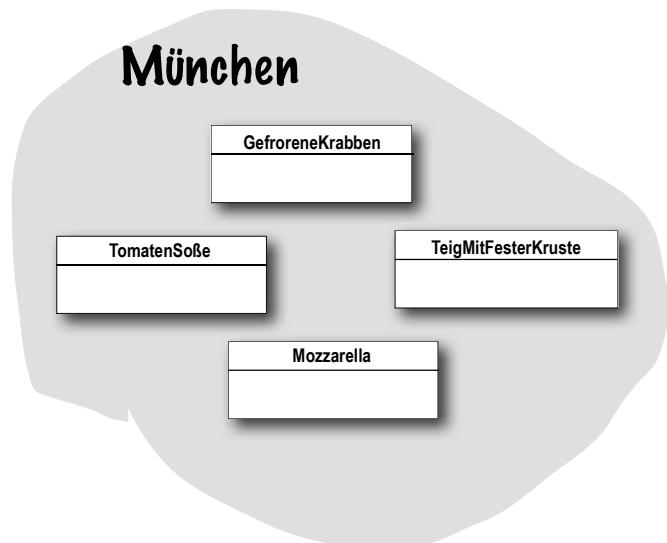
Zutatenfamilien

Berlin verwendet einen Satz von Zutaten, München einen anderen. Bei der Beliebtheit der Pizzeria Objekthausen dauert es sicher nicht lange, bevor Sie einen weiteren Satz regionaler Zutaten nach Köln verschicken müssen ... und was kommt danach? Düsseldorf?

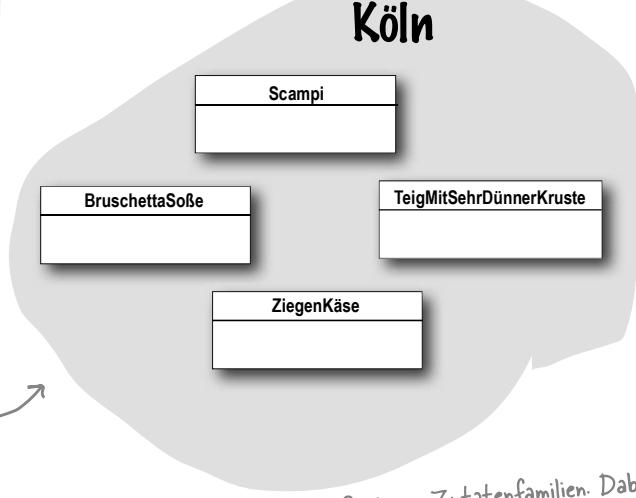
Damit das funktioniert, müssen Sie herausfinden, wie Sie mit Familien von Zutaten umgehen.



Jede Familie besteht aus einem Typ Teig, einem Typ Käse und einem Meeresfrüchtebelag (und ein paar weiteren, die wir hier nicht gezeigt haben, wie Gemüse und Gewürze).



Alle Objekthausener Pizzas werden aus den gleichen Komponenten zubereitet, aber jede Region hat andere Implementierungen dieser Komponenten.



Insgesamt bilden diese drei Regionen Zutatenfamilien. Dabei implementiert jede Region eine vollständige Zutatenfamilie.

Zutatenfabrik

Die Zutatenfabriken aufbauen

Jetzt werden wir eine Fabrik bauen, um unsere Zutaten herzustellen. Die Fabrik wird dafür verantwortlich sein, alle Zutaten in der Zutatenfamilie herzustellen. Anders gesagt, die Fabrik muss Teig, Soße, Käse und so weiter herstellen ... Wie wir mit den regionalen Unterschieden umgehen, werden Sie in Kürze sehen.

Beginnen wir damit, ein Interface für die Fabrik zu erstellen, die all unsere Zutaten herstellt:

```
public interface PizzaZutatenFabrik {  
    public Teig erstelleTeig();  
    public Soße erstelleSoße();  
    public Käse erstelleKäse();  
    public Salami erstelleSalami();  
    public Gemüse[] erstelleGemüse();  
    public Thunfisch erstelleThunfisch();  
    public Krabben erstelleKrabben();  
}  
  
}   
  


Massen neuer Klassen, jeweils eine pro Zutat.



Für jede Zutat definieren wir in unserem Interface eine Erstellungsmethode.



Gäbe es allgemeines >>Zutatenwerkzeug<<, das in jeder Instanz der Fabrik implementiert werden müsste, hätten wir daraus auch ein abstrakte Klasse machen können ...


```

Und das ist das, was wir machen werden:

- ➊ Wir bauen eine Fabrik für jede Region. Dazu erstellen Sie eine Unterklasse der PizzaZutatenFabrik, die jede der Erstellungsmethoden implementiert.
- ➋ Wir implementieren einen Satz von Zutaten-Klassen, die mit der Fabrik verwendet werden, wie Parmesan, Paprika und TeigMitDickerKruste. Diese Klassen können von Regionen geteilt werden, wenn das passend ist.
- ➌ Dann müssen wir das alles zusammenkitten, indem wir unsere neuen Zutatenfabriken in den alten Pizzeria-Code einarbeiten.

Die Berliner Zutatenfabrik aufbauen

Hier ist also die Implementierung für die Berliner Zutatenfabrik. Diese Fabrik ist auf Marinara-Soße, Parmesan, frische Krabben usw. spezialisiert:

Die Berliner Zutatenfabrik implementiert die Schnittstelle für alle Zutatenfabriken.

```
public class BerlinerPizzaZutatenFabrik implements PizzaZutatenFabrik {

    public Teig erstelleTeig() {
        return new TeigMitDünnerKruste();
    }

    public Soße erstelleSoße() {
        return new MarinaraSoße();
    }

    public Käse erstelleKäse() {
        return new Parmesan();
    }

    public Salami erstelleSalami() {
        return new SpanischeSalami();
    }

    public Gemüse[] erstelleGemüse() {
        Gemüse gemüse[] = { new Knoblauch(), new Zwiebeln(), new Pilze(), new Paprika() };
        return gemüse;
    }

    public Thunfisch erstelleThunfisch() {
        return new ThunfischStücke();
    }

    public Krabben erstelleKrabben() {
        return new FrischeKrabben();
    }
}
```

Berlin hat einen guten Draht zur Nordsee und erhält frische Krabben. In München müssen sie sich mit gefrorenen zufrieden geben.

Für alle Zutaten in der Zutatenfamilie erstellen wir eine Berliner Version.

Für die Gemüse liefern wir ein Array mit Gemüsen zurück. Wir haben die Gemüse hier hartcodiert. Das könnten wir raffinierter machen. Aber da das uns beim Lernen des Factory-Musters nicht weiterhilft, haben wir die Sache hier einfach gelassen.

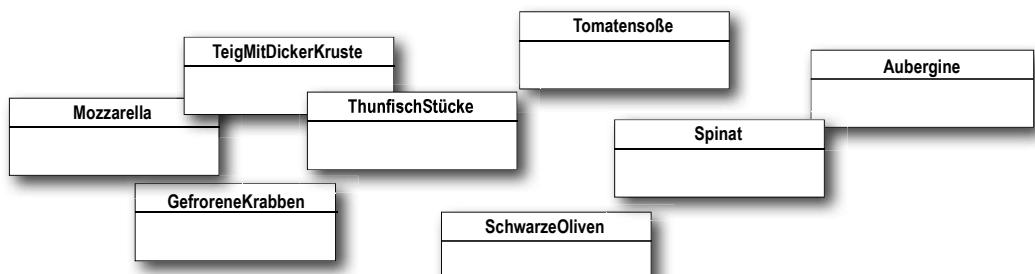
Bester, ökologisch gefangener Thunfisch in Stücken. Er wird von Berlin und München gemeinsam verwendet. Denken Sie daran, ihn zu verwenden, wenn Sie selbst das Vergnügen haben, die Münchener Fabrik zu implementieren.

Eine Fabrik bauen



Spitzen Sie Ihren Bleistift

Schreiben Sie die MünchenerPizzaZutatenFabrik. Sie können die unten angegebenen Klassen in Ihrer Implementierung referenzieren.



Die Pizza überarbeiten

Jetzt haben wir unsere Fabriken hochgefahren und sind bereit, hochwertige Zutaten herzustellen. Daher müssen wir unsere Pizzas überarbeiten, damit sie nur die Zutaten verwenden, die von unseren Fabriken produziert werden. Wir beginnen mit der abstrakten Klasse Pizza:

```
public abstract class Pizza {
    String name;
    Teig teig;
    Soße soße;
    Salami salami;
    Gemüse gemüse[];
    Käse käse;
    Thunfisch thunfisch;
    Krabben krabben;

    abstract void vorbereiten();

    void backen() {
        System.out.println("Backe 25 Minuten bei 350");
    }

    void schneiden() {
        System.out.println("Schneide die Pizza diagonal in Stücke");
    }

    void verpacken() {
        System.out.println("Packe die Pizza in die offizielle Pizzeria-Schachtel");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // hier kommt der Code zum Ausgeben der Pizzas rein
    }
}
```

Jede Pizza hält einen Satz von Zutaten, die bei ihrer Zubereitung verwendet werden.

Wir haben die Methode vorbereiten() jetzt abstrakt gemacht. Hier werden wir die Zutaten für die Pizza sammeln, die natürlich von der Zutatenfabrik kommen.

Mit Ausnahme der Methode vorbereiten() bleiben unsere anderen Methoden unverändert.

Zutaten entkoppeln

Die Pizza überarbeiten (Fortsetzung)

Jetzt haben wir eine abstrakte Pizza, von der wir ausgehen können, und nun ist es an der Zeit, die Pizzas Berliner und Münchener Art zu erstellen – nur werden sie diesmal die Zutaten direkt von der Fabrik erhalten. Die Zeiten, in denen die Zweigstellen bei den Zutaten schlampen können, sind vorbei!

Als wir den Factory Method-Code geschrieben haben, hatten wir eine BerlinerSalamiPizza und eine MünchenerSalamiPizza. Wenn Sie sich die beiden Klassen anschauen, sehen Sie, dass sie sich nur in ihrer Verwendung der regionalen Zutaten unterscheiden. Die Pizzas werden auf gleiche Weise gemacht (Teig + Soße + Käse). Das Gleiche gilt für die anderen Pizzas: Vegetarisch, Krabben und so weiter. Alle folgen den gleichen Zubereitungsschritten, sie haben einfach nur unterschiedliche Zutaten.

Sie sehen also, dass wir gar keine zwei Klassen für jede Pizza benötigen. Um die regionalen Unterschiede kümmert sich die Zutatenfabrik für uns. Hier ist die Salamipizza:

```
public class SalamiPizza extends Pizza {  
    PizzaZutatenFabrik zutatenFabrik;  
  
    public SalamiPizza(PizzaZutatenFabrik zutatenFabrik) {  
        this.zutatenFabrik = zutatenFabrik;  
    }  
  
    void vorbereiten() {  
        System.out.println("Mache " + name);  
        teig = zutatenFabrik.erstelleTeig();  
        soße = zutatenFabrik.erstelleSoße();  
        salami = zutatenFabrik.erstelleSalami();  
        käse = zutatenFabrik.erstelleKäse();  
    }  
}
```



Die Methode `vorbereiten()` geht die Schritte durch, die erforderlich sind, um eine Salami-pizza zu erstellen, und immer, wenn sie eine Zutat benötigt, bittet sie die Fabrik, diese herzustellen.

Um eine Pizza zu machen, brauchen wir eine Fabrik, die die Zutaten liefert. Also wird jeder Pizza-Klasse im Konstruktor eine Fabrik übergeben, die in einer Instanzvariablen gespeichert wird.

← Das ist der Ort, an dem gezaubert wird!



Code unter der Lupe

Der Pizza-Code verwendet die Fabrik, mit der er zusammengesetzt wurde, um die Zutaten zu produzieren, die in der Pizza verwendet werden. Welche Zutaten produziert werden, ist von der verwendeten Fabrik abhängig. Der Klasse Pizza ist das egal. Sie weiß, wie man Pizzas macht. Die Bindung an die verschiedenen regionalen Zutaten ist jetzt aufgehoben. Deswegen kann sie einfach wieder verwendet werden, wenn es Fabriken für Ostfriesland, Schwaben oder wo auch immer gibt.

```
soße = zutatenFabrik.erstelleSoße();
```

Wir setzen die Instanzvariable von Pizza so, dass sie auf die bestimmte Soße verweist, die in dieser Pizza verwendet wird.

Das ist unsere Zutatenfabrik. Pizza kümmert sich nicht darum, was für eine Fabrik verwendet wird. Es muss nur eine Zutatenfabrik sein.

Die Methode erstelleSoße() liefert die Soße zurück, die in dieser Region verwendet wird. Wird die Methode auf einer Berliner Zutatenfabrik aufgerufen, erhalten wir eine Marinara-Soße.

Befassen wir uns ebenfalls mit der Krabbenpizza:

```
public class KrabbenPizza extends Pizza {
    PizzaZutatenFabrik zutatenFabrik;

    public KrabbenPizza(PizzaZutatenFabrik zutatenFabrik) {
        this.zutatenFabrik = zutatenFabrik;
    }

    void vorbereiten() {
        System.out.println("Mache " + name);
        teig = zutatenFabrik.erstelleTeig();
        soße = zutatenFabrik.erstelleSoße();
        käse = zutatenFabrik.erstelleKäse();
        krabben = zutatenFabrik.erstelleKrabben();
    }
}
```

KrabbenPizza verstaat ebenfalls eine Zutatenfabrik.

Um eine Krabbenpizza zu machen, sammelt die Methode vorbereiten() die erforderlichen Zutaten von der lokalen Fabrik.

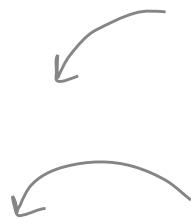
Ist die Fabrik eine Berliner Fabrik, sind die Krabben frisch, sonst sind sie tiefgefroren.

Rückkehr zur Pizzeria

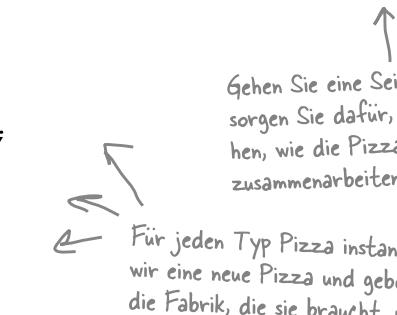
Wir haben es fast geschafft. Wir müssen nur noch einen kurzen Ausflug zu unseren Zweigstellen machen, um sicherzustellen, dass sie die richtigen Pizzas verwenden. Wir müssen ihnen auch noch eine Referenz auf ihre lokalen Zutatenfabriken spendieren:

```
public class BerlinerPizzeria extends Pizzeria {  
  
    protected Pizza erstellePizza(String item) {  
        Pizza pizza = null;  
        PizzaZutatenFabrik zutatenFabrik =  
            new BerlinerPizzaZutatenFabrik();  
  
        if (item.equals("Salami")) {  
  
            pizza = new SalamiPizza(zutatenFabrik);  
            pizza.setName("Salamipizza Berliner Art");  
  
        } else if (item.equals("Schinken")) {  
  
            pizza = new SchinkenPizza(zutatenFabrik);  
            pizza.setName("Schinkenpizza Berliner Art");  
  
        } else if (item.equals("Krabben")) {  
  
            pizza = new KrabbenPizza(zutatenFabrik);  
            pizza.setName("Krabbenpizza Berliner Art");  
  
        } else if (item.equals("Thunfisch")) {  
  
            pizza = new ThunfischPizza(zutatenFabrik);  
            pizza.setName("Thunfischpizza Berliner Art");  
  
        }  
        return pizza;  
    }  
}
```

Die Berliner Pizzeria wird mit einer Berliner Zutatenfabrik zusammen- gesetzt. Diese wird verwendet, um die Zutaten für alle Pizzas Berliner Art herzustellen.



Jetzt übergeben wir jeder Pizza die Fabrik, die verwendet werden soll, um die erforderlichen Zu- taten zu produzieren.



Gehen Sie eine Seite zurück und sorgen Sie dafür, dass Sie verste- hen, wie die Pizzas und die Fabrik zusammenarbeiten!

Für jeden Typ Pizza instantiiieren wir eine neue Pizza und geben ihr die Fabrik, die sie braucht, um ihre Zutaten zu erhalten.



Vergleichen Sie diese Version der Methode erstelle- Pizza() mit der Factory Method-Implementierung weiter vorn in diesem Kapitel.

Was wir gemacht haben

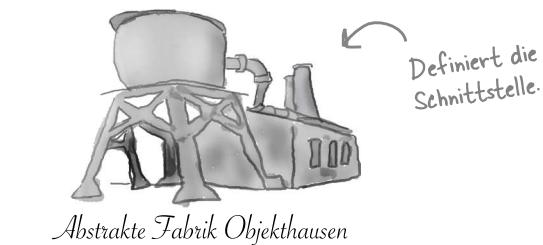
Das war eine ganz ordentliche Reihe von Code-Änderungen. Was genau haben wir da gemacht?

Wir haben ein Mittel geliefert, für Pizzas eine Familie von Zutaten zu erzeugen, indem wir einen neuen Typ Fabrik eingeführt haben, der als Abstract Factory (Abstrakte Fabrik) bezeichnet wird.

Eine Abstract Factory gibt uns eine Schnittstelle zur Erzeugung einer Familie von Produkten. Wenn wir Code schreiben, der diese Schnittstelle verwendet, entkoppeln wir unseren Code von der bestimmten Fabrik, die die Produkte erzeugt. Das ermöglicht uns, eine Vielzahl von Fabriken zu implementieren, die Produkte produzieren, die für unterschiedliche Kontexte – wie verschiedene Regionen, Betriebssysteme, Look-and-Feels – gedacht sind.

Weil unser Code nicht mehr an die tatsächlichen Produkte gebunden ist, können wir andere Fabriken einsetzen, um andere Verhalten zu erreichen (beispielsweise um in unserem Fall statt Marinara-Soße Tomatensoße zu erhalten).

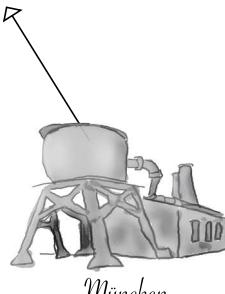
Eine Abstrakte Fabrik bietet eine Schnittstelle für eine Familie von Produkten. Was ist eine Familie? In unserem Beispiel sind das all die Dinge, die wir brauchen, um Pizza zu machen: Teig, Soße, Käse, Fleischsorten und Gemüse.



Abstrakte Fabrik Objekthausen

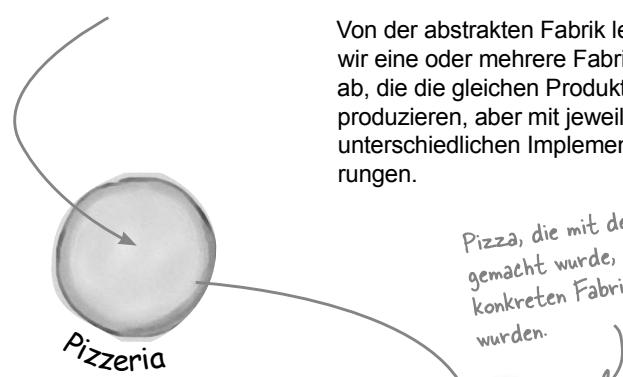


Bietet die Implementierungen für die Produkte.



München

Von der abstrakten Fabrik leiten wir eine oder mehrere Fabriken ab, die die gleichen Produkte produzieren, aber mit jeweils unterschiedlichen Implementierungen.



Pizza, die mit den Zutaten gemacht wurde, die von einer konkreten Fabrik hergestellt wurden.

Dann schreiben wir unseren Code so, dass er die Fabrik verwendet, um die Produkte herzustellen. Indem wir unterschiedliche Fabriken übergeben, erhalten wir eine Vielzahl von Implementierungen dieser Produkte. Aber unser Client-Code bleibt immer gleich.

Noch ein paar Pizzas bestellen

Mehr Pizza für Ben und Tassilo

Ben und Tassilo kriegen einfach nicht genug von der Objekt-hausener Pizza! Was sie allerdings nicht wissen, ist, dass ihre Bestellungen jetzt die neuen Zutatenfabriken verwenden.



Der erste Teil des Bestellvorgangs hat sich gar nicht geändert. Sehen wir uns wieder Bens Bestellung an:

1 Zuerst brauchen wir eine BerlinPizzeria:

```
Pizzeria berlinPizzeria = new BerlinPizzeria();
```

Erzeugt eine Instanz von BerlinPizzeria.



2 Jetzt haben wir eine Zweigstelle und können eine Bestellung aufnehmen:

```
berlinPizzeria.bestellePizza("Salami");
```

Auf der berlinPizzeria-Instanz wird die Methode bestellePizza aufgerufen.

bestellePizza("Salami")

3 Dann ruft die Methode bestellePizza() die Methode erstellePizza() auf:

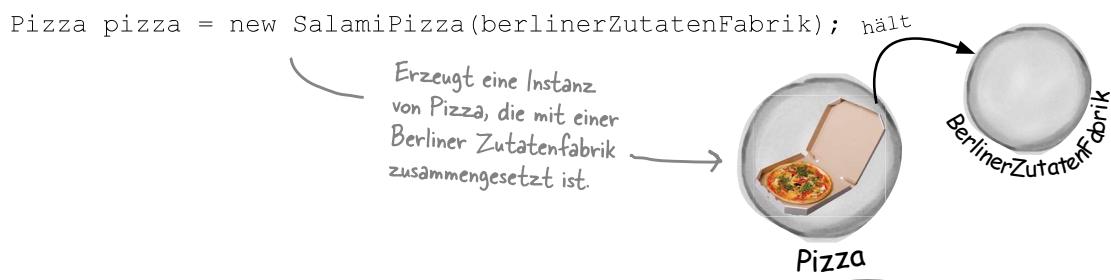
```
Pizza pizza = erstellePizza("Salami");
```

Von hier ab ändern sich die Dinge, weil wir eine Zutatenfabrik verwenden.



4 Wenn die Methode erstellePizza() aufgerufen wird, kommt unsere Zutatenfabrik ins Spiel:

Die Zutatenfabrik wird in der Pizzeria ausgewählt und instantiiert und dann an den Konstruktor der einzelnen Pizzas übergeben.



5 Als Nächstes müssen wir die Pizza vorbereiten. Ist die Methode vorbereiten() aufgerufen, wird die Fabrik aufgefordert, die Zutaten vorzubereiten:

```
void vorbereiten() {
    teig = zutatenFabrik.erstelleTeig();
    soße = zutatenFabrik.erstelleSoße();
    salami = zutatenFabrik.erstelleSalami();
    käse = zutatenFabrik.erstelleKäse();
}
```

Für Bens Pizza wird die Berliner Zutatenfabrik verwendet. Also erhalten wir auch die Berliner Zutaten.



6 Und schließlich haben wir die vorbereitete Pizza in der Hand, die von bestellePizza() gebacken, geschnitten und verpackt wird.

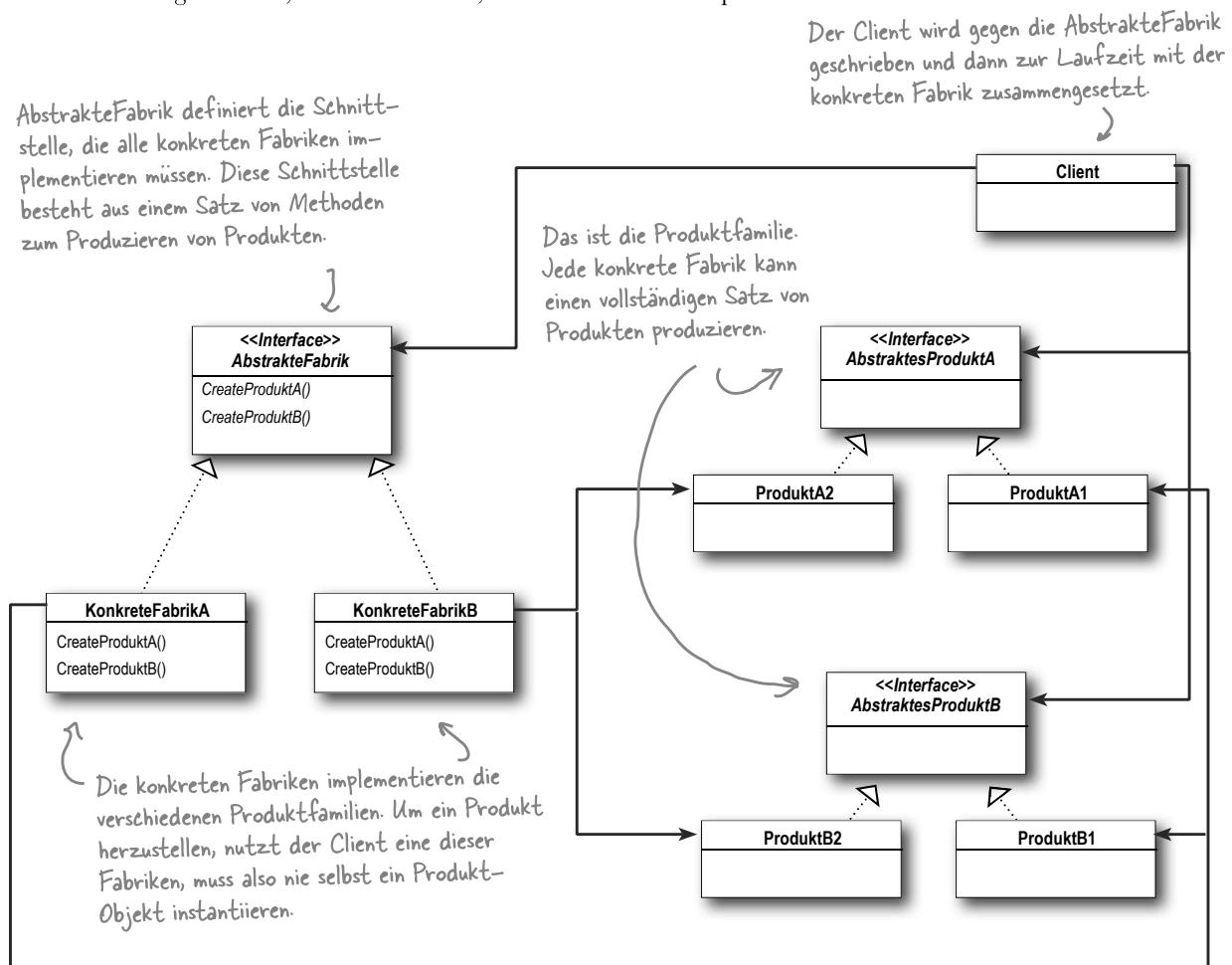
Die Definition der abstrakten Fabrik

Die Definition des Abstract Factory-Musters

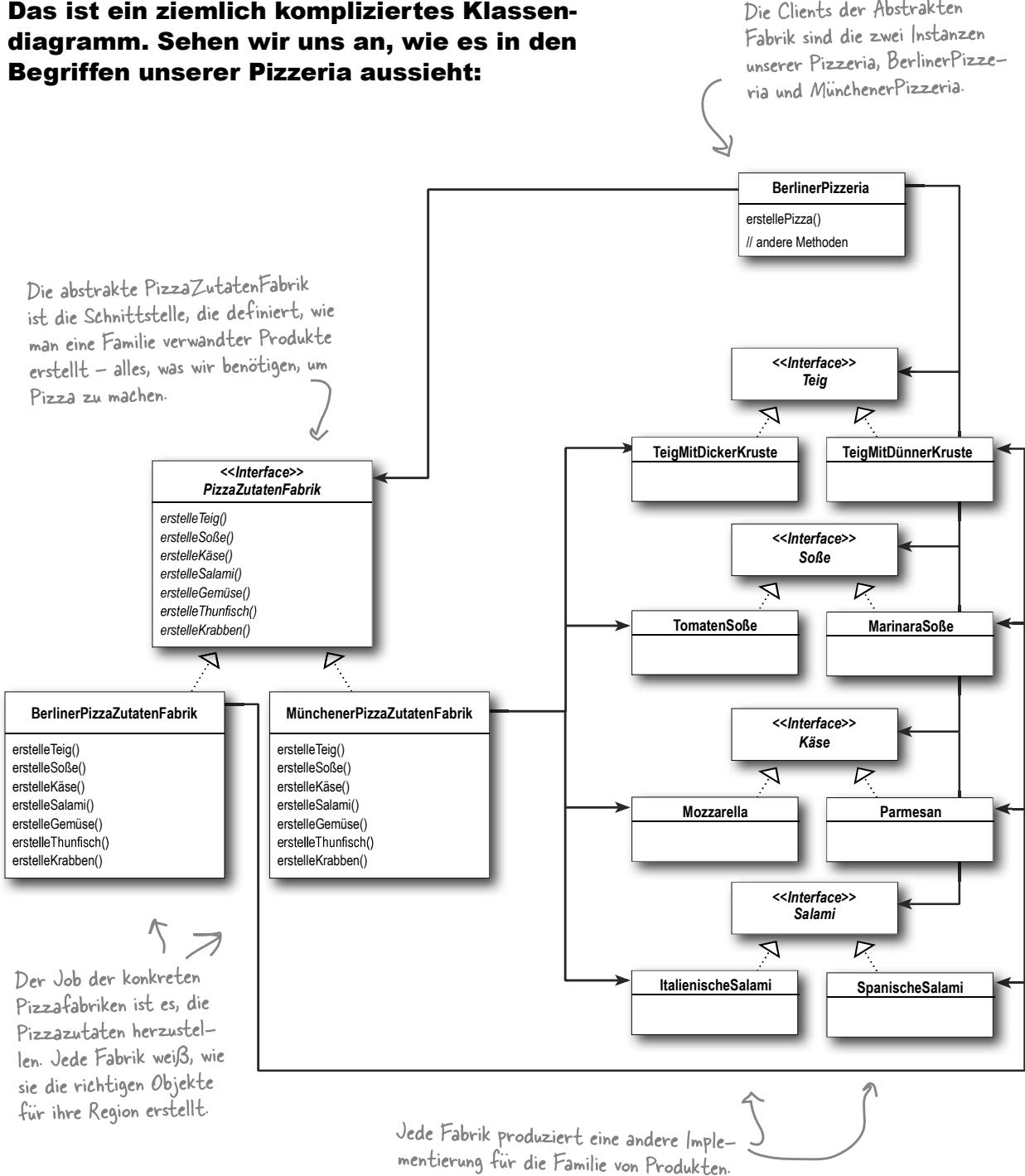
Nun fügen wir unserer Musterfamilie ein weiteres Factory-Muster hinzu, eins, das es uns ermöglicht, Familien von Produkten zu bilden. Sehen wir uns die offizielle Definition für dieses Muster an:

Das Abstract Factory-Muster bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.

Ganz sicher wissen wir inzwischen, dass Abstract Factory es einem Client ermöglicht, eine abstrakte Schnittstelle zu verwenden, um einen Satz verwandter Produkte zu erstellen, ohne etwas über die konkreten Produkte zu wissen, die tatsächlich produziert werden. Auf diese Weise wird der Client von den Einzelheiten der konkreten Produkte entkoppelt. Sehen wir uns das Klassendiagramm an, um nachzusehen, wie all das zusammenpasst:



Das ist ein ziemlich kompliziertes Klassendiagramm. Sehen wir uns an, wie es in den Begriffen unserer Pizzeria aussieht:



Interview mit den Factory-Mustern



Mir ist aufgefallen, dass alle Methoden in der Abstrakten Fabrik eigentlich wie Fabrikmethoden aussehen (erstelleTeig(), erstelleSoße() usw.). Jede Methode ist als abstrakt deklariert und wird von den Unterklassen so überschrieben, dass sie irgendein Objekt erzeugt. Entspricht das nicht dem Factory Method-Muster?

Ist das eine Factory Method, die sich da in der Abstract Factory verbirgt?

Gut aufgepasst! Es ist tatsächlich häufig so, dass die Methoden einer Abstrakten Fabrik als Fabrikmethoden implementiert werden. Macht doch Sinn, oder? Aufgabe einer Abstrakten Fabrik ist die Definition einer Schnittstelle zur Erstellung eines Satzes von Produkten. Jede Methode in dieser Schnittstelle ist dafür verantwortlich, ein konkretes Produkt zu erstellen, und wir implementieren eine Unterklasse der Abstrakten Fabrik, um diese Implementierungen zu liefern. Fabrikmethoden sind ein natürliches Mittel zur Implementierung von Produkt-Methoden in Ihren abstrakten Fabriken.



Muster unter der Lupe

Interview der Woche:
Factory Method und Abstract Factory

Von Kopf bis Fuß: Wow, ein Interview mit zwei Mustern auf einmal! Das ist Neuland für uns.

Factory Method: Wissen Sie, ich bin nicht so sicher, ob es mir gefällt, hier mit Abstract Factory zusammengeworfen zu werden. Dass wir beide Factory-Muster sind, heißt noch lange nicht, dass wir nicht beide unsere eigenen Interviews bekommen sollten.

Von Kopf bis Fuß: Da reagieren Sie aber gleich ein wenig zu gereizt, oder? Wir wollten Sie beide in das Interview einbeziehen, um dem Leser die mögliche Verwirrung zu nehmen, wer von Ihnen wer ist. Sie sind einander ja doch recht ähnlich, und ich habe gehört, dass manche Leute Sie gelegentlich verwechseln.

Abstract Factory: Das stimmt, ich bin gelegentlich mit Factory Method verwechselt worden, und ich weiß, dass du, Factory Method, ähnliche Probleme hattest. Wir sind beide echt gut, wenn es darum geht, Anwendungen von spezifischen Implementierungen zu entkoppeln. Nur tun wir es auf unterschiedliche Weise. Ich verstehe schon, warum die Leute uns manchmal verwechseln.

Factory Method: Es geht mir einfach auf die Nerven. Schließlich verwende ich zum Erstellen Klassen und du Objekte. Das ist doch was ganz anderes!

Von Kopf bis Fuß: Können Sie uns das genauer erklären?

Factory Method: Klar. Wir beide, Abstract Factory und ich, erstellen Objekte – das ist unser Job. Aber ich mach das über Vererbung ...

Abstract Factory: ... und ich über Objekt-Zusammensetzung.

Factory Method: Richtig. Und das bedeutet, dass man, wenn man mit mir Objekte erstellen will, eine Klasse erweitern und eine Fabrikmethode überschreiben muss.

Von Kopf bis Fuß: Und diese Fabrikmethode? Was macht die?

Factory Method: Die stellt natürlich Objekte her! Was ich sagen will, ist, dass das Wichtige am Factory Method-Muster ist, dass man eine Unterklasse verwendet, die für einen die Objekt-Erstellung erledigt. So müssen die Clients nur den abstrakten Typen kennen, den sie verwenden. Um die konkreten Typen kümmert sich schon die Unterklasse. Anders gesagt: Ich entkoppeln die Clients von den konkreten Typen.

Abstract Factory: Und genau das mache ich auch. Nur auf etwas andere Weise.

Von Kopf bis Fuß: Machen Sie ruhig weiter, Abstract Factory. Sie sagten etwas über Objekt-Komposition?

Abstract Factory: Ich liefere einen abstrakten Typ zur Erstellung einer Familie von Produkten. Unterklassen dieses Typs definieren, wie diese Produkte produziert werden. Um die Fabrik zu verwenden, instantiiieren sie eine und übergeben sie an irgendwelchen Code, der gegen den abstrakten Typ geschrieben ist. Also sind meine Clients, wie bei Factory Method, von den tatsächlichen konkreten Produkten entkoppelt, die sie verwenden.

Von Kopf bis Fuß: Ah, verstehe. Ein weiterer Vorteil ist also, dass Sie einen Satz verwandter Produkte gruppieren.

Abstract Factory: Das ist richtig.

Von Kopf bis Fuß: Was passiert, wenn Sie diesen Satz verwandter Produkte erweitern müssen, beispielweise um ein weiteres hinzuzufügen? Müssen Sie dazu nicht Ihre Schnittstelle ändern?

Abstract Factory: Das stimmt. Meine Schnittstelle muss sich ändern, wenn neue Produkte hinzugefügt werden. Ich weiß, dass man das im Allgemeinen nicht so gern hat ...

Factory Method: <kichert>

Abstract Factory: Was gibt es da zu kichern, Factory Method?

Factory Method: Ach komm schon, das ist eine wichtige Sache! Deine Schnittstelle ändern. Das heißt, dass du den Code anpacken und die Schnittstelle in jeder Unterklasse ändern musst! Das klingt nach einer ganzen Menge Arbeit.

Abstract Factory: Und? Ich brauche eine große Schnittstelle, weil ich verwendet werde, um eine ganze Familie von Produkten herzustellen. Du erstellst nur ein einziges Produkt und brauchst deswegen keine große Schnittstelle, sondern bloß eine einzige Methode.

Von Kopf bis Fuß: Abstract Factory, ich habe gehört, dass Sie oft Fabrikmethoden verwenden, um Ihre konkreten Fabriken zu implementieren?

Abstract Factory: Ja. Ich gebe zu, dass meine konkreten Fabriken oft eine Fabrikmethode implementieren, um ihre Produkte herzustellen. Aber bei mir werden die nur verwendet, um Produkte herzustellen ...

Factory Method: ... während ich im abstrakten Hersteller in der Regel Code implementiere, der die konkreten Typen verwendet, die die Unterklassen erstellen.

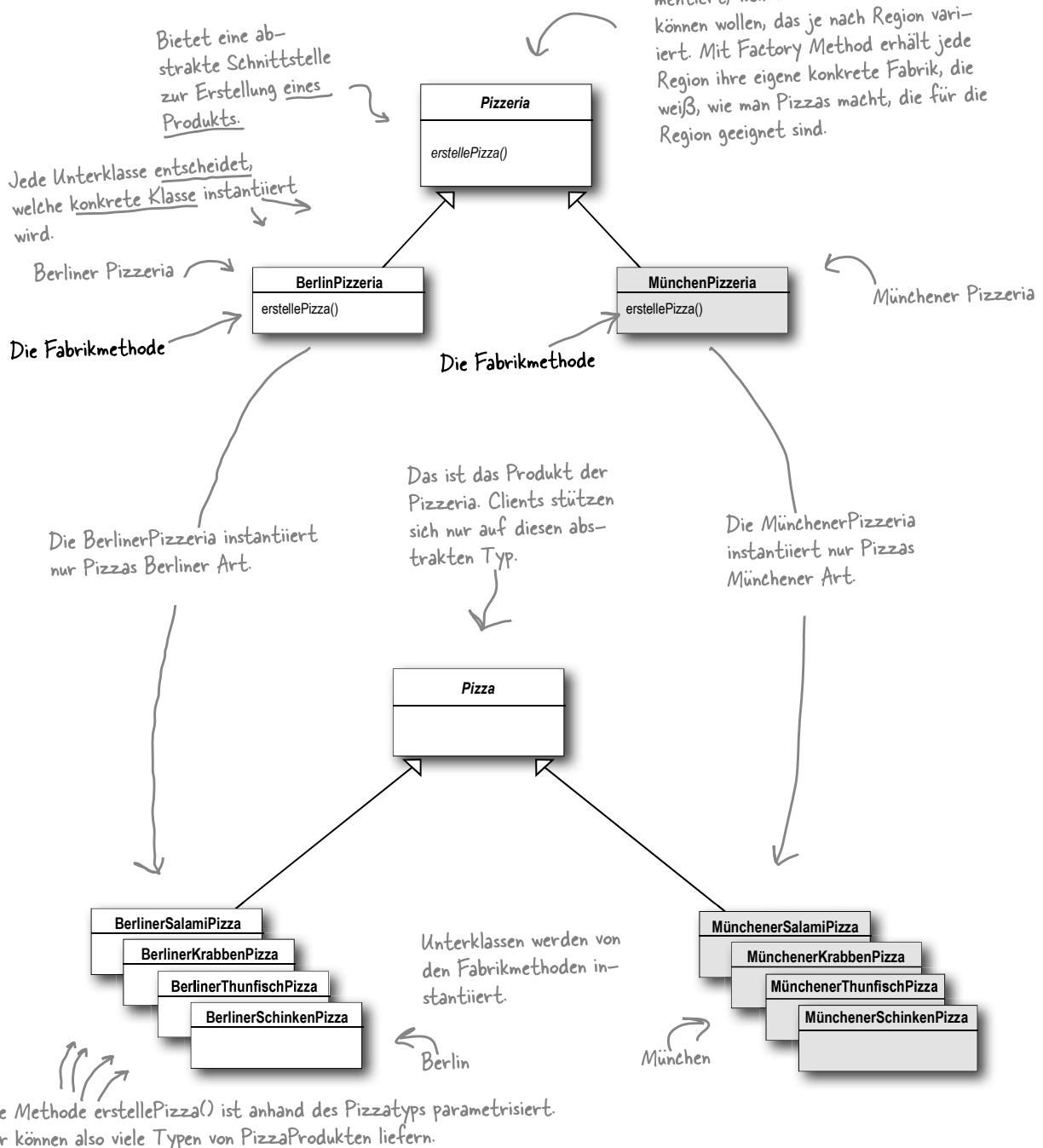
Von Kopf bis Fuß: Das klingt „als würden Sie beide jeweils gute Arbeit leisten. Ich bin sicher, dass die Leute gern verschiedene Möglichkeiten zur Auswahl haben. Schließlich sind Fabriken so nützlich, dass man sie in den unterschiedlichsten Situationen einsetzen möchte. Beide kapseln Sie die Objekt-Erstellung, um Anwendungen locker gebunden zu halten und weniger abhängig von Implementierungen zu machen. Und das ist, egal ob man Factory Method oder Abstract Factory verwendet, eine klasse Sache. Möchte vielleicht jeder von Ihnen zum Abschluss noch etwas sagen?

Abstract Factory: Danke. Denken Sie an mich, Abstract Factory, und verwenden Sie mich, wenn Sie Familien von Produkten haben, die Sie herstellen müssen, und sicherstellen möchten, dass Ihre Clients Produkte verwenden, die zusammengehören.

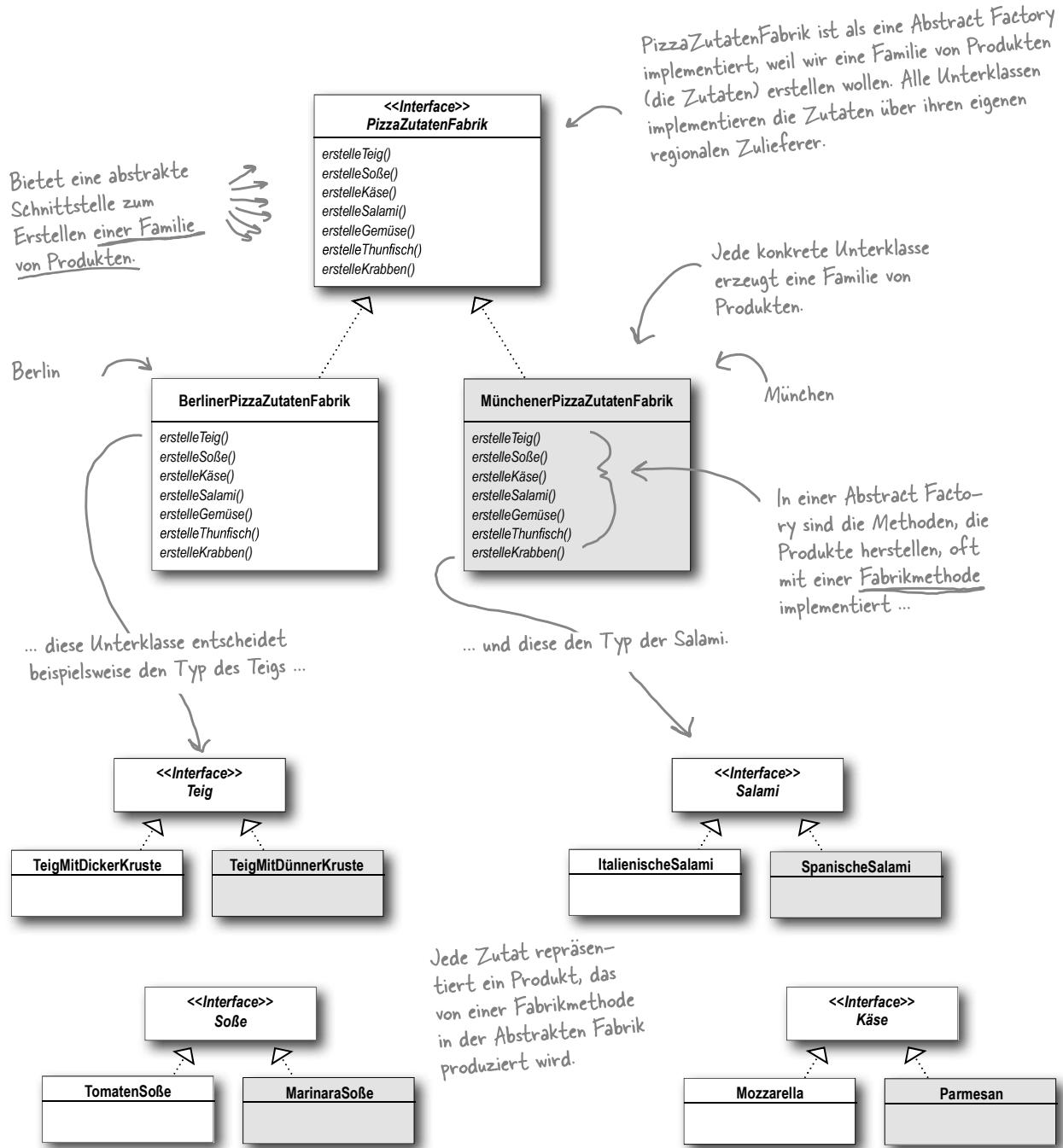
Factory Method: Ich bin Factory Method. Verwenden Sie mich, um Ihren Client-Code von den konkreten Klassen zu entkoppeln, die Sie instantiierten müssen, oder wenn Sie nicht im Voraus alle konkreten Klassen kennen, die Sie benötigen. Wenn Sie mich verwenden wollen, bilden Sie einfach eine Unterklasse von mir und implementieren meine Fabrikmethode!

Muster im Vergleich

Factory Method und Abstract Factory im Vergleich



Das Factory-Muster



Die Produkt-Unterklassen erstellen parallele Sätze von Produktfamilien. Hier haben wir eine Berliner und eine Münchener Zutatenfamilie.



Werkzeuge für Ihren Design-Werkzeugkasten

In diesem Kapitel haben wir Ihrem Werkzeugkasten zwei weitere Werkzeuge hinzugefügt: Factory Method und Abstract Factory. Beide Muster kapseln die Objekt-Erstellung und ermöglichen Ihnen, Ihren Code von den konkreten Typen zu entkoppeln.



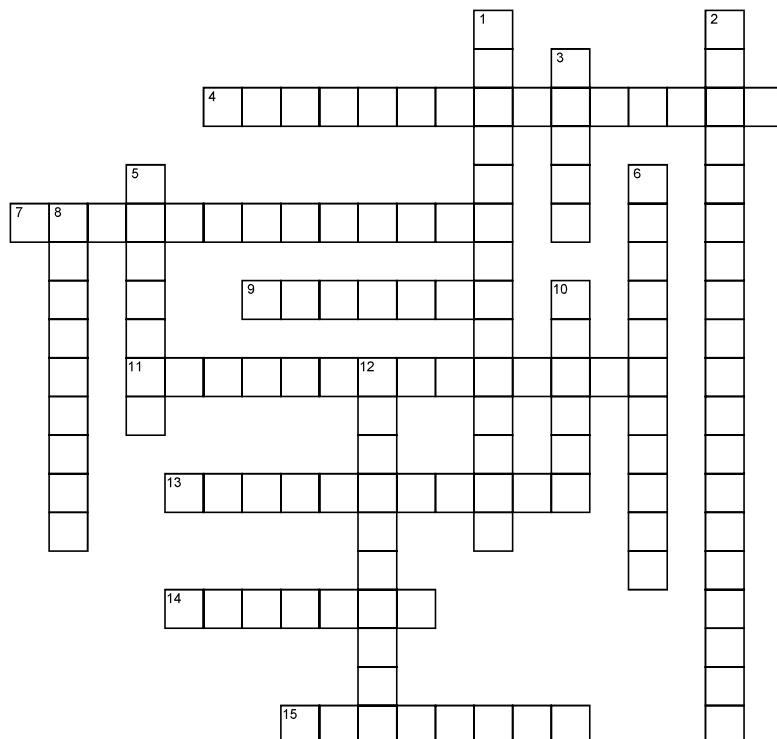
Punkt für Punkt

- Alle Factories kapseln die Objekt-Erstellung.
- Die einfache Fabrik ist eine unkomplizierte Möglichkeit, Clients von konkreten Klassen zu entkoppeln, ist aber kein echtes Entwurfsmuster.
- Factory Method stützt sich auf Vererbung: Die Objekt-Erstellung wird an Unterklassen delegiert, die die Fabrikmethode implementieren, um Objekte zu erstellen.
- Abstract Factory stützt sich auf Objekt-Komposition: Die Objekt-Erstellung ist in Methoden implementiert, die in der Fabrik-Schnittstelle vorgegeben werden.
- Alle Factory-Muster fördern lockere Bindung, indem sie für Ihre Anwendung die Abhängigkeit von konkreten Klassen reduzieren.
- Der Zweck von Factory Method ist es, einer Klasse zu ermöglichen, die Instantiierung bis in ihre Unterklassen zu verzögern.
- Der Zweck der Abstract Factory ist es, Familien verwandter Objekte zu erstellen, ohne dabei von den konkreten Klassen abhängig zu sein.
- Das Prinzip der Umkehrung der Abhängigkeiten leitet uns an, Abhängigkeiten von konkreten Typen zu vermeiden und Abstraktionen anzustreben.
- Factories sind eine mächtige Technik, um auf Abstraktionen statt auf konkreten Klassen zu programmieren.



Entwurfsmuster-Kreuzworträtsel

Das war ein langes Kapitel. Nehmen Sie sich ein Stück Pizza und entspannen Sie sich, während Sie dieses Kreuzworträtsel lösen. Alle Lösungswörter stammen aus diesem Kapitel.



Waagerecht

- 4 Wenn Sie new verwenden, programmieren Sie auf eine _____.
- 7 ErstellePizza() ist eine _____.
- 9 Abstract Factory erstellt eine _____ von Produkten.
- 11 Kein echtes Muster und trotzdem nützlich.
- 13 In Simple Factory und Abstract Factory haben wir _____ verwendet, in Factory Method Vererbung.
- 14 Alle Factory-Muster ermöglichen uns, die Objekt-Erstellung zu _____.
- 15 Dieser Käse wird für alle Pizzas Berliner Art verwendet.

Senkrecht

- 1 Bei Abstract Factory ist jede Zutatenfabrik eine _____.
- 2 Bei Factory Method ist jede Zweigstelle ein _____.
- 3 Bei Factory Method, endlich mal ein FIO, stützen sich Pizzeria und die konkreten Pizzas auf diese Abstraktion.
- 5 In Berlin sind sie frisch, in München tiefgefroren.
- 6 Wer entscheidet bei Factory Method, welche Klasse instantiiert wird?
- 8 Wenn eine Klasse ein Objekt einer konkreten Klasse instantiiert, ist sie von diesem Objekt _____.
- 10 Ben mag Pizza dieser Stadt.
- 12 Rolle der Pizzeria beim Factory Method-Muster.

Lösungen zu den Übungen

Spitzen Sie Ihren Bleistift

Lösung

BerlinPizzeria haben wir bereits zusammengeschustert. Nur zwei stehen noch aus, und wir können die Pizzeria-Kette ins Leben rufen! Schreiben Sie hier die Implementierungen für MünchenPizzeria und KölnPizzeria:

Diese beiden Pizzerias sind mit der Berliner Pizzeria fast
identisch – sie machen nur andere Arten von Pizza.

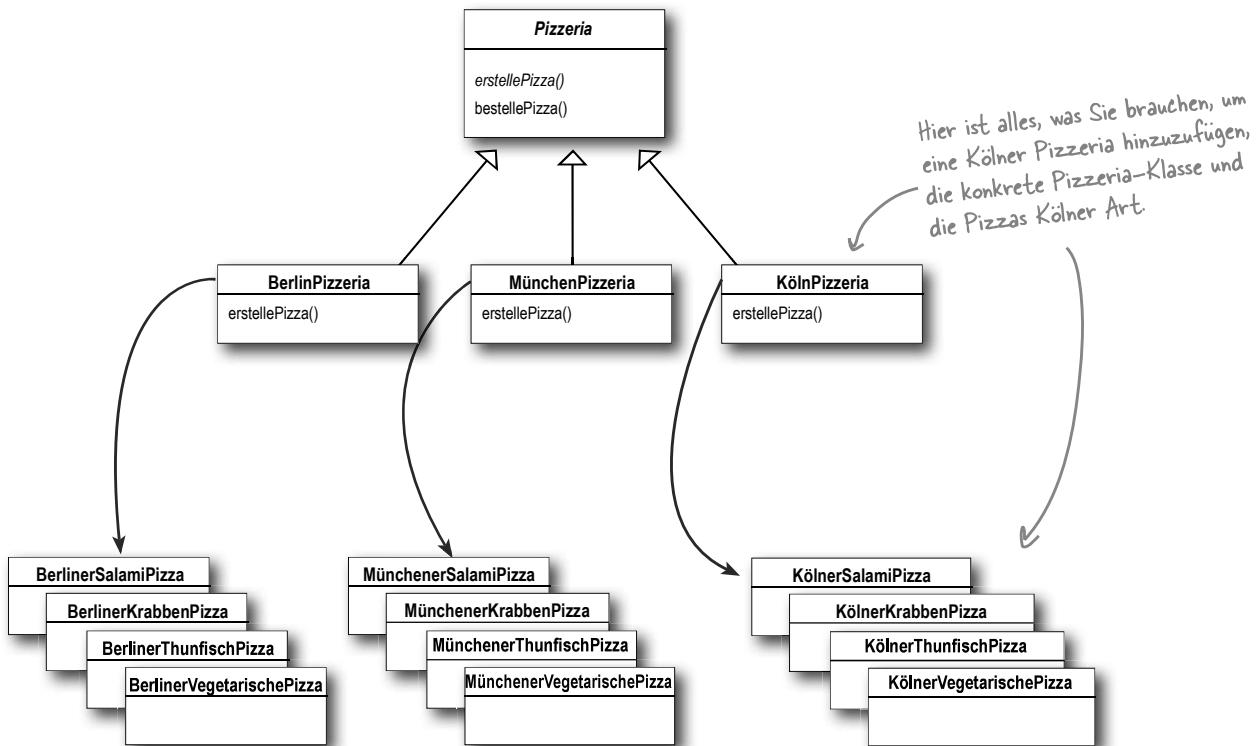
```
public class MünchenPizzeria extends Pizzeria {  
    protected Pizza erstellePizza(String element) {  
        if (element.equals("Salami")) {  
            return new MünchenerSalamiPizza();  
        } else if (element.equals("Schinken")) {  
            return new MünchenerSchinkenPizza();  
        } else if (element.equals("Krabben")) {  
            return new MünchenerKrabbenPizza();  
        } else if (element.equals("Thunfisch")) {  
            return new MünchenerThunfischPizza();  
        } else return null;  
    }  
}  
  
public class KölnPizzeria extends Pizzeria {  
    protected Pizza erstellePizza(String element) {  
        if (element.equals("Salami")) {  
            return new KölnerSalamiPizza();  
        } else if (element.equals("Schinken")) {  
            return new KölnerSchinkenPizza();  
        } else if (element.equals("Krabben")) {  
            return new KölnerKrabbenPizza();  
        } else if (element.equals("Thunfisch")) {  
            return new KölnerThunfischPizza();  
        } else return null;  
    }  
}
```

Bei der Münchener Pizzeria müssen wir nur sicherstellen, dass wir Pizzas Münchener Art herstellen ...

... und für die Kölner Pizzeria, dass wir Pizzas Kölner Art machen.

Lösung des Design-Puzzles

Wir brauchen noch einen anderen Typ Pizza für die verrückten Kölner (auf *nette* Weise verrückt natürlich). Zeichnen Sie einen weiteren parallelen Satz von Klassen, um unserer Pizzeria einen regionalen Typ für Köln hinzuzufügen.



Gut, und jetzt schreiben Sie die fünf *seltsamsten* Dinge auf, die Sie sich auf einer Pizza vorstellen können. Dann können Sie in Köln das Pizzageschäft aufnehmen!

Hier unsere
Vorschläge ...

Reibekuchen Blutwurst Sauerkraut Rosinen Apfelmus

Lösung zu den Übungen

Ein sehr abhängige Pizzeria

 **Spitzen Sie Ihren Bleistift**

Tun wir mal so, als hätten Sie noch nie etwas von einer OO-Factory gehört. Hier ist eine Version der Pizzeria, die keine Fabrik verwendet. Zählen Sie, von wie vielen konkreten Pizza-Objekten diese Klasse abhängig ist. Von wie vielen Objekten wäre sie abhängig, wenn Sie dieser Pizzeria Pizzas nach Kölner Art hinzufügen würden?

```
public class AbhangigePizzeria {  
  
    public Pizza erstellePizza(String art, String typ) {  
        Pizza pizza = null;  
        if (art.equals("Berlin")) {  
            if (typ.equals("Salami")) {  
                pizza = new BerlinerSalamiPizza();  
            } else if (typ.equals("Schinken")) {  
                pizza = new BerlinerSchinkenPizza();  
            } else if (typ.equals("Krabben")) {  
                pizza = new BerlinerKrabbenPizza();  
            } else if (typ.equals("Thunfisch")) {  
                pizza = new BerlinerThunfischPizza();  
            }  
        } else if (art.equals("München")) {  
            if (typ.equals("Salami")) {  
                pizza = new MünchenerSalamiPizza();  
            } else if (typ.equals("Schinken")) {  
                pizza = new MünchenerSchinkenPizza();  
            } else if (typ.equals("Krabben")) {  
                pizza = new MünchenerKrabbenPizza();  
            } else if (typ.equals("Thunfisch")) {  
                pizza = new MünchenerThunfischPizza();  
            }  
        } else {  
            System.out.println("Fehler: Ungültiger Pizzatyp");  
            return null;  
        }  
        pizza.vorbereiten();  
        pizza.backen();  
        pizza.schneiden();  
        pizza.verpacken();  
        return pizza;  
    }  
}
```

Kümmert sich um
alle Pizzas Berliner
Art.

Kümmert sich um
alle Pizzas Mün-
chener Art.

Hier können Sie Ihre
Antwort notieren:

8

Anzahl

12

Anzahl mit Kölner Art

 **Spitzen Sie Ihren Bleistift** —————

Lösung

Schreiben Sie die MünchenerPizzaZutatenFabrik. Sie können die unten angegebenen Klassen in Ihrer Implementierung referenzieren.

```
public class MünchenerPizzaZutatenFabrik
    implements PizzaZutatenFabrik
{

    public Teig erstelleTeig() {
        return new TeigMitDickerKruste();
    }

    public Soße erstelleSoße() {
        return new TomatenSoße();
    }

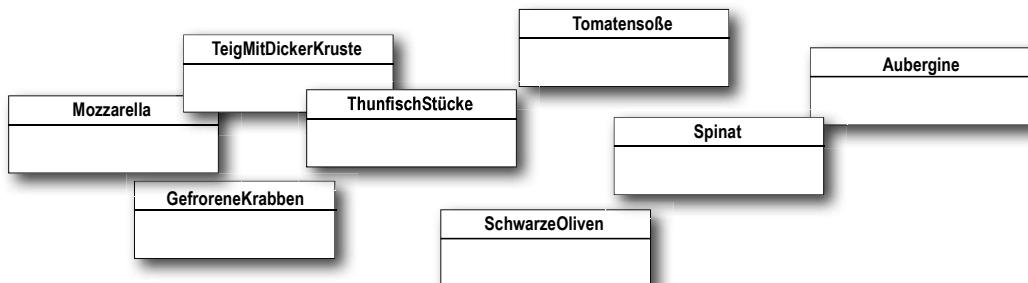
    public Käse erstelleKäse() {
        return new Mozzarella();
    }

    public Salami erstelleSalami() {
        return new ItalienischeSalami();
    }

    public Gemüse[] erstelleGemüse() {
        Gemüse gemüse[] = { new SchwarzeOliven(),
                            new Spinat(),
                            new Aubergine() };
        return gemüse;
    }

    public Thunfisch erstelleThunfisch() {
        return new ThunfischStücke();
    }

    public Krabben erstelleKrabben() {
        return new GefroreneKrabben();
    }
}
```



Lösung des Kreuzworträtsels



Entwurfsmuster-Kreuzworträtsel, Lösung

