Kapitel 1
Einführung

Wenn ich weiter gesehen habe als andere, so deshalb, weil ich auf den Schultern von Riesen stehe.
– Isaac Newton

Zunächst einmal: Herzlichen Dank, dass Sie sich für mein Buch entschieden haben. Sie können sich gar nicht vorstellen, wie ich mich darüber freue. Bevor wir *in medias res* gehen: Gestatten Sie mir bitte einige Anmerkungen dazu, was Sie in diesem Buch vorfinden.

1.1 Einleitung und allgemeine Hinweise

1.1.1 Für wen ist dieses Buch gedacht?

Kurz gesagt: Das Buch ist für alle, die mit dem Design, der Architektur oder der Entwicklung von Software zu tun haben, was einige spätere Themen einschließt, wie das Testen von Software. Es spielt keine Rolle, ob Sie Anfänger sind oder schon eine Weile entwickeln. Vielleicht sind nicht alle Muster gleich relevant für Sie, weil Sie zum Beispiel die Architektur Ihrer Anwendung gar nicht vorgeben. Aber auch dann hilft es in der täglichen Praxis, wenn Sie das zugrunde liegende Architekturmuster kennen.

Dieses Buch ist sowohl Lehr- als auch Nachschlagewerk, denn Sie werden vermutlich beides benötigen. Die einzelnen Kapitel und auch die einzelnen Abschnitte lassen sich weitgehend unabhängig voneinander lesen. Jeder Abschnitt erläutert ein einzelnes Entwurfsmuster, von den typischen Anwendungsfällen bis hin zur vollständigen Implementierung. Ich habe mich redlich bemüht, diese Informationshappen für Sie so leicht verdaulich wie nur möglich zu gestalten, aber Sie werden sehen, dass die einzelnen Muster unterschiedlich umfangreich und komplex sind. Unter diesem Gesichtspunkt ist das Buch also ein Lehrbuch.

Damit Sie sich schnell in einzelnen Entwurfsmustern zurechtfinden, ist jedes Muster strukturell identisch aufgebaut. Außerdem habe ich für Sie Musterimplementierungen zum Download bereitgestellt (*www.rheinwerk-verlag.de/3538*), sodass Sie das Buch auch als Nachschlagewerk nutzen können.

Um alle Beispiele verstehen zu können, benötigen Sie Grundkenntnisse in objektorientierter Entwicklung. Die Sprache ist dabei unwesentlich, wie Sie noch sehen werden, Sie können also in Java, Objective-C, C++, C#, VB.NET und vielen anderen Sprachen Nutzen aus den hier vorgestellten Mustern ziehen.

1.1.2 Muster erkennen

Die vielleicht wichtigste Fähigkeit ist die, typische Anwendungsfälle für Muster in den eigenen Aufgabenstellungen zu erkennen. Das Anwenden ist dann oft die einfachere (aber nicht unbedingt triviale) Aufgabe. Das wiederum erfordert ein wenig Übung und die Bereitschaft, die eigenen Probleme immer wieder auf Muster hin zu untersuchen. Mit der Zeit treten diese dann immer deutlicher aus bestehendem Code oder aus Spezifikationen hervor. Ich helfe Ihnen dabei, indem ich auf typische Lehrbuchbeispiele verzichte. Wenn ich also Fabrikmuster erläutere, dann bauen wir dort keine Pizzen oder Autos zusammen, sondern Objekte, die auch in Ihren eigenen Projekten vorkommen können.

1.1.3 Muster im weiteren Sinne

Neben den Entwurfsmustern im engeren Sinne, also den klassischen Mustern von Gamma, Johnson, Helm und Vlissides (siehe weiter hinten), finden Sie in diesem Buch noch zahlreiche Entwurfsmuster im weiteren Sinne. Manchmal sind es auch grundsätzliche Designprinzipien, die man kennen sollte, wie das *liskovsche Substitutionsprinzip*. Andere Muster haben mit Daten zu tun, zum Beispiel das von Martin Fowler geprägte *Data Transfer Object*. Wieder andere hängen mit bestimmten Produkten oder Technologien zusammen, wie *Dependency Injection* oder das *Model-View-Controller*-Muster.

Ich habe das Netz absichtlich ein wenig weiter ausgeworfen, weil seit der Vorstellung der ursprünglichen Muster schon zwei Jahrzehnte ins Land gingen und die Zeit in der Softwareentwicklung nun wirklich nicht stillstand. Ob Sie diese Muster nun Entwurfsmuster nennen oder nicht – jedenfalls sind es Muster, die Sie so (oder so ähnlich) in vielen Frameworks und Produkten finden und die Ihren eigenen Code genauso bereichern können, wie die Klassiker dies tun.

Auch über die Einteilung der Muster lässt sich in einigen Fällen streiten. Die Beschränkungen eines Buches verlangen aber, dass ich ein Muster irgendwo einordnen muss. Das Model-View-Controller-Muster ist so ein Beispiel. Es hätte problemlos auch als Architekturmuster durchgehen können. Aber zum Glück enthält dieses Buch ja auch noch einen Index.

1.1.4 Programmiersprachen und Frameworks

Die allermeisten Entwurfsmuster sind hinsichtlich der implementierenden Sprache und des verwendeten Frameworks nicht wählerisch. Allerdings benötigen Sie für einige der hier vorgestellten Muster wenigstens eine objektorientierte Sprache (OO-Sprache), die typische Eigenschaften besitzt – wie Vererbung, Schnittstellen und Polymorphie. Einige wenige Kapitel kommen weitgehend ohne Code aus, weil sie ein wenig weiter oben auf der Abstraktionspyramide angesiedelt sind, beispielsweise die Architekturmuster.

In der Implementierung gibt es natürlich Unterschiede. Sie finden dazu immer wieder Hinweise. So unterstützt beispielsweise C++ die Mehrfachvererbung, Java und C# tun das hingegen nicht, was dort selten jemand wirklich vermisst. Aus Gründen der Lesbarkeit und der Verständlichkeit musste ich mich natürlich dennoch für eine Sprache entscheiden, in der ich die Beispiele peu à peu aufbaue und erläutere. Die Wahl fiel diesmal nicht auf C#, das ich vorwiegend nutze, sondern auf die wohl meistverbreitete OO-Sprache überhaupt: Java. Ob Sie Java 5, 6, 7 oder 8 einsetzen, ist für die meisten Beispiele unerheblich. Aber auch hierzu finden Sie den einen oder anderen Tipp im Text. Außerdem dachte ich, es wäre für Sie vermutlich ungemein nützlich, den fertigen Code auch in anderen Sprachen vorzufinden, und habe die Muster daher in Java und C# implementiert. Sie können sie hier herunterladen: *www.rheinwerk-verlag.de/3538*.

Wenn Sie die Beispiele laden und ausführen möchten, dann benötigen Sie für Java üblicherweise Eclipse. Ich habe *Eclipse Luna* verwendet, und für C# verwende ich *Visual Studio*, Version 2012 oder 2013 – die Express-Edition genügt.

1.1.5 Babylon I: Deutsch vs. Englisch

Es ist schon ein Kreuz, gerade für uns Autoren: Wenn wir in unseren Beispielen vollständig aufs Englische bauen, leidet die Lesbarkeit doch merklich. Wollen wir alles eindeutschen, wirkt vieles gestelzt und manches gar lächerlich. Für manche Muster gibt es passable und auch gebräuchliche Übersetzungen, das »Beobachter«-Muster zum Beispiel. Andere Muster wiederum klingen im Deutschen gleich oder ähnlich, wie der »Adapter« oder der »Prototyp«. Wiederum andere Muster werden kaum eingedeutscht, wer sagt zum »Singleton«-Muster schon »Einzelstück«?

Die Muster selbst sind daher im Steckbrief jeweils zweisprachig aufgeführt, und auch der Wegweiser ist zweisprachig aufgebaut. Gleiches gilt für den Index, die erste Anlaufstelle, wenn Sie ein Muster suchen sollten. Ansonsten halte ich mich an unsere schöne Muttersprache, die ich vor allem für die Beispiele gebrauche. Und dort, wo es mir unumgänglich schien, verzeihen Sie mir hoffentlich ein (kleines) Maß an Denglisch.

1.1.6 Babylon II: Verwendung von Begriffen

Entwurfsmuster stehen nicht unbedingt im Ruf, leicht verständlich zu sein. Das liegt auch daran, dass die verwendete Sprache manchmal mit der von Entwicklern kollidiert.

Wenn Entwickler »Schnittstelle« sagen, dann meinen sie entweder eine Schnittstelle zwischen zwei Systemen, zum Beispiel eine Webservice-Schnittstelle, oder das Sprachmerkmal Schnittstelle (Interface), so wie in:

```
public interface MyInterface
…
public class MyClass implements MyInterface
…
```

Im weiteren Sinne – und so wird der Begriff bei Entwurfsmustern oft verwendet – ist eine *Schnittstelle* einfach alles, was eine Klasse ihrem Verwender anbietet, also zum Beispiel die Methoden `BucheBetrag()` und `HoleKontostand()`, egal ob in der Implementierung tatsächlich eine Schnittstelle verwendet wird. Wenn also die Rede davon ist, dass eine Fassade eine Schnittstelle anbietet, die eine Menge von Schnittstellen anderer Systeme kapselt, dann wissen Sie das jetzt einzuordnen.

Ein *Algorithmus* ist eine Handlungsvorschrift, die von der konkreten Programmiersprache unabhängig ist. Im Sinne der Mustersprache meine ich damit einfach Quellcode, der eine Aufgabe ausführt, zum Beispiel eine bestimmte Implementierung zum Abrufen von Datensätzen. Wenn das Strategie-Muster also den Algorithmus unabhängig vom Verwender austauschbar macht, dann heißt das also, dass zum Beispiel die Daten in der einen Implementierung aus einer Datenbank kommen können und in einer anderen Implementierung aus einer XML-Datenbank – und dass das Strategie-Muster dafür eine Art »Umschaltmechanismus« anbietet.

1.1.7 UML

Wenn Sie sich ein wenig mit der UML, der Unified Modeling Language, auskennen, sind Sie im Vorteil, denn für viele Muster finden Sie das zugehörige UML-Diagramm im Buch. Der für uns relevante Teil ist das UML-Klassendiagramm (siehe Abbildung 1.1). Die UML bietet noch viele weitere Diagramme an, die aber für das Verständnis hier nicht erforderlich sind.

Besonders wichtig sind für unser Thema:

- die Sichtbarkeit von Operationen und Attributen (z. B. public/private)
- Vererbung
- abstrakte Klassen

- statische Klassen und Operationen (Klassenattribute, statische Methoden)
- Schnittstellen
- Komposition und Aggregation

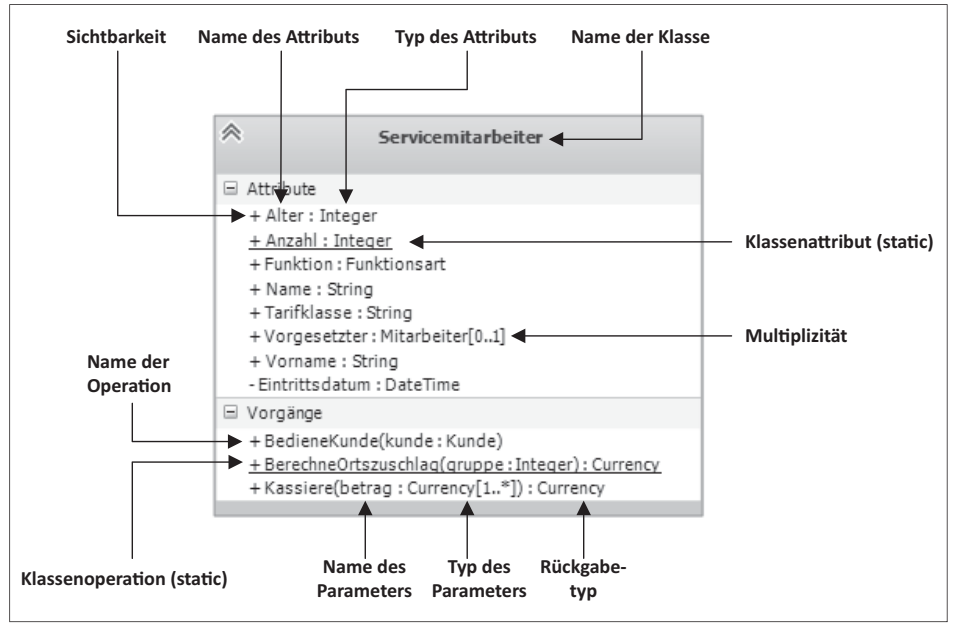

Abbildung 1.1 Anatomie einer Klasse in UML

1.1.8 Auf den Schultern von Riesen

Ehre, wem Ehre gebührt: Die meisten der hier vorgestellten Entwurfsmuster wurden von der Autorengemeinschaft *Erich Gamma*, *Richard Helm*, *Ralph Johnson* und *John Vlissides* bekannt gemacht, die heute liebevoll die *Gang of Four (GoF)* genannt wird. Erst ihr inzwischen legendäres Buch »Design Patterns – Elements of Reusable Object-Oriented Software« verhalf dem Thema und den darin beschriebenen 23 Mustern zum Durchbruch.

Viele der hier vorgestellten Muster gab es schon früher. Sie wurden verwendet, ohne dass die Entwickler einen Namen für sie gehabt hätten. Die vielleicht größte Leistung der GoF besteht darin, diese Muster erkannt, formalisiert, beschrieben und verbreitet zu haben.

Es gab noch viele weitere Väter, Mütter und Entdecker der hier vorgestellten Muster – zu viele, als dass ich sie hier alle aufzählen könnte. Ich habe mich dagegen entschieden, sie jeweils an Ort und Stelle zu würdigen, weil sonst schnell ein Geschichtsbuch entstanden wäre, denn viele Muster haben eine Historie und stammen oft von einer Autorengemeinschaft. Daher finden Sie im Anhang eine Literaturliste.

1.1.9 Der Aufbau eines Entwurfsmusters im Buch

Die meisten Entwurfsmuster sind auf dieselbe Art gegliedert. Den Anfang macht der Steckbrief, der den deutschen und englischen Namen der Muster nennt und – sofern vorhanden – auch weitere gebräuchliche Bezeichnungen auflistet sowie das Muster in seinen Ordnungsrahmen packt.

Danach folgt die allgemeine Beschreibung, noch bevor die Anwendungsfälle beschrieben werden. Das hat den Vorteil, dass Sie im weiteren Verlauf besser verstehen, warum ein Muster sich überhaupt für den einen oder anderen Einsatzweck eignet. Die Beschreibung umfasst auch meistens ein UML-Diagramm mit Legende.

Danach folgen wie gesagt die Anwendungsfälle, prototypische und ganz konkrete.

Das Kernstück ist dann die Implementierung. Ich erläutere das Muster Schritt für Schritt und gebe Hinweise zur Implementierung in verschiedenen Sprachen.

Den Rest des Abschnitts nehmen weitere Überlegungen zum Muster ein, zum Beispiel weitere Implementierungsdetails und die Abgrenzung zu anderen Mustern. Außerdem finden Sie hier den einen oder anderen Tipp und, sofern die Sache es notwendig macht, auch einmal eine Warnung.

1.1.10 Die Implementierung und die verwendete Programmiersprache

Die allermeisten Beispiele in diesem Buch sind mit Hilfe von Java umgesetzt. Wenn Sie einigermaßen fit in Java sind, könnten Sie vielleicht das eine oder andere im Code bemängeln. Einige Beispiele:

- Diese oder jene Variable kann treffender mit `final` gekennzeichnet werden.
- Klassen, die Schnittstellen implementieren, können die Methoden dieser Schnittstellen mittels `@override` annotieren, nicht nur Klassen, die Methoden ihrer Oberklassen überschreiben.
- Modernere Java-Versionen machen einem ein paar Dinge einfacher, beispielsweise gibt es seit Java 7 das *try-with-resources*-Statement, was den Code unter anderem kürzer und eleganter macht.
- Decimal-Datentypen eignen sich besser als ein `float` dazu, mit Währungsbeträgen zu rechnen.

Alles richtig, und die Liste ließe sich noch fortsetzen. Dass ich in vielen Fällen die Sprache nicht vollständig ausgereizt habe hat einen einfachen Grund: Klarheit durch Fokussierung auf die Muster selbst. Viele Entwurfsmuster sind schon umfangreich genug, ich wollte es meinen Leserinnen und Lesern ersparen, bei jedem Konstrukt die Frage zu stellen, ob dieses zum Muster, oder nur zur Implementierung gehört. Außerdem lässt sich der Code so viel leichter auf andere Zielsprachen übertragen und die Beschreibung wird dadurch kürzer.

Die Methodennamen in den UML-Diagrammen sind überwiegend großgeschrieben, während ich mich im Java-Quellcode an die gängigen Konventionen gehalten habe. Das ist in jeder Sprache anders, in C# beispielsweise beginnen öffentliche Methoden mit einem Großbuchstaben und Schnittstellen mit einem »I«.

Die Muster an sich verwenden allgemeine Begriffe. Das Befehlsmuster kennt z. B. eine Klasse `Befehl` mit einer Methode `FuehreAus()`. Im Praxisbeispiel wird daraus die Klasse `Spielzug`, und auch die Methoden sind so benannt, wie es zum Praxisbeispiel passt. Damit Sie beides leichter zusammenbringen, sind viele UML-Diagramme zweimal vorhanden: Einmal mit der allgemeinen Form des Musters und ein zweites Mal auf das Muster angewendet.

Zu einigen der beschriebenen Alternativen und Erläuterungen gibt es in der Fachwelt hitzige Diskussionen: Darf man in C++ das Schlüsselwort `friend` verwenden (und wann?) und welches Serialisierungsformat ist das beste? Auch hier habe ich die Alternativen zwar beschrieben, aber nicht immer ausführlich bewertet; aus Platzgründen, aber auch, weil es zu weit vom Thema wegführen würde.

In den Beispielsdateien finden Sie übrigens noch weitere Kommentare, die ich aus Platzgründen im Buch weggelassen habe. Sollte Ihnen etwas unklar sein, dann werfen Sie doch einen Blick in das Eclipse-Projekt.

1.1.11 Herstellerspezifische Technologien

An einigen Stellen, vor allem wenn es praktisch wird, musste ich mich entscheiden: Mit welchem SQL-Server soll ich die Beispiele aus dem Kapitel zu den Datenmustern umsetzen? Im Beispiel verwende ich den SQL-Server von Microsoft. Nicht ohne an dieser Stelle zu erwähnen, dass es viele weitere Produkte gibt, kommerzielle wie freie, die ich hätte verwenden können. Dieses Buch bevorzugt weder einen bestimmten Hersteller noch eine bestimmte Technologie, sondern versucht, anhand praktischer Beispiele den Bogen zu spannen, von der Erläuterung eines Musters bis hin zu den Details der Implementierung.

1.2 Was sind Entwurfsmuster und was sind sie nicht?

So, nun zum Thema. Was ist eigentlich ein Entwurfsmuster genau? Versuchen wir eine Definition:

Definition Entwurfsmuster

Ein Entwurfsmuster ist eine *anpassbare Lösung* für ein *nicht triviales* und immer *wiederkehrendes Problem* in der *Entwicklung*, dem *Design* und der *Architektur* von Software. Es ist damit ein Element der *Wiederverwendung*.

Der Begriff Entwurfsmuster – häufig spreche ich hier auch nur von *Mustern* – ist inzwischen Allgemeingut. Weitere, überwiegend synonym verwendete Begriffe und Übersetzungen sind:

- Design Patterns
- Solution Patterns
- Lösungsschablonen
- Software Patterns

1.2.1 Was Entwurfsmuster sind

Die wichtigsten Eigenschaften aus dieser Definition im Überblick:

- **Problem**: Es muss ein zu lösendes Problem vorliegen, beispielsweise die Notwendigkeit, in einer Anwendung von einem Objekt nur eine einzige Instanz zu haben.
- **Wiederverwendung**: Das Problem muss häufig genug vorkommen, damit sich das Wissen um und über Entwurfsmuster auch lohnt.
- **Lösung**: Das Muster muss das Problem natürlich lösen, und zwar auf eine anerkannte und im Laufe der Zeit erprobte, gut verstandene und elegante Art und Weise.
- **Nicht trivial**: Das zu lösende Problem muss eine gewisse Komplexität aufweisen, darf also nicht offensichtlich oder mit einfachsten Mitteln zu lösen sein. So kann ein Anwender eines Musters Nutzen daraus ziehen, dass jemand anderes das Problem bereits vollständig durchdacht und elegant und stimmig gelöst hat.
- **Konkret**: Sowohl das Problem als auch die Lösung müssen konkret sein. Es geht also nicht um abstrakte Grundsatzfragen der Informatik, sondern um ganz konkrete und praktische Aufgabenstellungen in der Entwicklung von Software. Auch die Lösung ist nicht lediglich eine amorphe Skizze, sondern ein fertig einsetzbares Konstrukt, auch wenn es noch der Überführung in Quellcode bedarf.
- **Anpassbar**: Die meisten Muster sind, wie gesagt, recht konkret. Aber es gibt fast immer auch den einen oder anderen Freiheitsgrad in der Implementierung und damit die Möglichkeit, das Muster an die eigenen Bedürfnisse anzupassen. Und auch die verwendete Sprache macht Unterschiede in der Implementierung notwendig.

So viel zu den Eigenschaften. Mindestens genauso interessant ist aber die gegenteilige Betrachtung, also die Antwort auf die Frage, worin sich Entwurfsmuster von anderen wiederverwendbaren Elementen in der Softwareentwicklung abgrenzen.

1.2.2 Was Entwurfsmuster nicht sind

Entwurfsmuster, dieser früher aufgrund der Arbeit der GoF hinreichend genau definierte Begriff, wird heute ein wenig inflationär gebraucht und manchmal auch missbraucht. Die folgenden Abschnitte sind meine Highlights gängiger Missverständnisse.

Algorithmen

Auch wenn viele Algorithmen größer und komplexer als Entwurfsmuster sind, sind sie doch auch immer ein oder zwei Spuren konkreter. Algorithmen sind eindeutige Handlungsvorschriften, zum Beispiel zum Sortieren von Listen oder zum Durchsuchen von Bäumen. Die Implementierungen sind daher sehr nah am Algorithmus.

Wenn Sie so wollen, dann gleicht ein Algorithmus einem Schritt-für-Schritt-Rezept, während ein Entwurfsmuster eher eine Schablone oder eine Blaupause ist. Diese Schablone ist zwar auch konkret, aber längst nicht so konkret wie ein Algorithmus.

Ein Algorithmus verlangt also im Wesentlichen die technische Fähigkeit, ihn elegant in die Zielsprache zu übertragen, während ein Entwurfsmuster ein wenig mehr geistige Transferleistung voraussetzt und Ihnen mehr Freiheit bei der Implementierung lässt.

Beispiel

Der Quick-Sort-Algorithmus ist ein rekursiver und unter gewissen Umständen relativ schneller Sortieralgorithmus, der nach dem Teile-und-herrsche-Prinzip arbeitet. In Pseudo-Code formuliert, sieht der Algorithmus (hier nur die äußere Funktion) so aus:

```
FUNKTION Sortiere(links, rechts)
   IF links < rechts DANN
      teiler = Partitioniere(links, rechts)
      Sortiere (links, teiler-1)
      Sortiere (teiler+1, rechts)
   ENDE IF
ENDE FUNKTION
```

Es ist leicht einzusehen, dass dieser Algorithmus relativ einfach in eine Zielsprache übersetzt werden kann. Die Herausforderung liegt hier eher in der konkreten Implementierung, also z. B. in der Frage, wie die Speicherverwaltung möglichst effizient gestaltet werden kann.

Vergleichen Sie damit nun ein beliebiges Muster, und Sie werden erkennen, dass Muster ein gutes Stück abstrakter sind.

Fertige Lösungen

Entwurfsmuster sind konkret genug, damit Sie sie in Code direkt auf Ihr Problem abbilden können, aber andererseits ist jedes Problem im Detail verschieden, und so können Sie ein Muster nur selten völlig unverändert übernehmen. Das beginnt schon mit der Benennung der Akteure, die dem fachlichen Kontext entsprechen sollte, und es endet mit den Details der Implementierung.

Manche Muster sind nur wenig mehr als ein Gerüst. Das Model-View-Controller-Muster (MVC-Muster) ist so ein Beispiel, das – naturgemäß – weder über das Modell noch über die View oder den Controller irgendwelche Annahmen machen kann.

Ein Programm nur aus einer Reihe von Mustern zusammenzusetzen wird also nicht funktionieren, allerdings kann eine Anwendung mehrere Muster gewinnbringend einsetzen.

Snippets oder Kopiervorlagen

Snippets, Kopierschablonen oder Kopiervorlagen sind in der jeweiligen Zielsprache erstellte und in unveränderter Form anwendbare Codebestandteile. Manchmal müssen natürlich auch solche Snippets noch angepasst bzw. erweitert werden, aber sie sind schon deshalb keine Muster, weil sie eben an die Zielsprache gebunden sind.

Muster hingegen lassen sich prinzipiell in jeder Hochsprache umsetzen (in manchen freilich ein wenig eleganter als in anderen), und einige Muster schreien förmlich nach einer OO-Sprache, aber darauf kommt es in Grunde nicht an.

Beispiel

Die Fehlerbehandlung in Java könnte mit diesem Snippet ein wenig schneller von der Hand gehen:

```
try {
   |
} catch(@ExceptionType e) {
   |
}
```

Die Pipe-Symbole kennzeichnen dabei die Positionen des Cursors nach dem Einfügen, und das @-Symbol kennzeichnet zu ersetzende Bestandteile des Snippets.

Best Practices

Entwurfsmuster können schon Best Practices sein, aber dieser Begriff ist viel zu unspezifisch dafür. Eine Best Practice ist, ganz allgemein gesagt, eine optimale oder in der Praxis besonders bewährte Vorgehensweise – wofür auch immer.

Das gilt nun für das Design von Benutzeroberflächen genauso wie für die Abwicklung eines Betriebs nach der Insolvenz und ist schon daher mit dem Begriff *Entwurfsmuster* nicht gleichzusetzen.

Eine Lösung für alle Probleme

Die Nützlichkeit von Entwurfsmustern steht im unmittelbaren Zusammenhang mit dem Wiedererkennungswert, und dieser wiederum hängt direkt von der Anzahl der Anwendungsfälle ab. Schon aus diesem Grund haben sich lange Zeit die ursprünglichen 23 Muster der GoF gehalten, weil sie eben in Büchern und Veranstaltungen immer wieder gelehrt und in der Praxis vieler Entwickler tagtäglich genutzt werden.

Das bedeutet auch: Für viele Probleme gibt es kein Muster oder jedenfalls kein bekanntes Muster – entweder weil das Problem zu selten auftritt oder weil es jeweils ganz spezifisch im jeweiligen Kontext neu gelöst werden muss. Sicherheitsprobleme in verteilten Anwendungen gehören in die zweite Kategorie.

APIs

Eine API, d. h. eine Programmierschnittstelle, stellt Methoden, Klassen und andere Elemente und Konstrukte bereit, die ein Entwickler in seinen eigenen Anwendungen nutzen kann. APIs sind in den meisten Fällen entweder sprachgebunden (z. B. Java APIs), an eine Technologie gebunden (z. B. .NET-Klassen) oder an eine bestimmte Ausführungsumgebung gebunden (z. B. Windows WinRT-API).

Alle diese Einschränkungen grenzen APIs von Entwurfsmustern ab, die von alledem unabhängig sind. Allerdings können natürlich Entwurfsmuster in der Implementierung der APIs von großem Nutzen sein, und so nimmt es nicht Wunder, dass Entwurfsmuster in APIs eigentlich überall anzutreffen sind.

Beispiel

In der Klasse `javax.xml.xpath.XPathFactory` ist das Entwurfsmuster *Abstrakte Fabrik* umgesetzt. Sie enthält also eine statische Methode, die ein Objekt vom Typ `XPathFactory` erzeugt und zurückgibt:

```
public static final XPathFactory newInstance()
```

In anderen Fällen setzt eine gewisse Technologie wiederum das Verwenden eines Musters voraus oder begünstigt dies wenigstens. Das *MVVM*-Muster zum Beispiel wäre ohne gewisse Microsoft-Technologien (wie WPF oder Silverlight) und ohne HTML5 bestimmt nur halb so bekannt.

1.3 Der OO-Werkzeugkasten

Bevor wir ein kleines Beispiel zur Illustration betrachten, möchte ich an dieser Stelle einige Missverständnisse ausräumen, die für das Verständnis der Entwurfsmuster hinderlich wären. Die folgenden Abschnitte helfen Ihnen auch zu verstehen, warum manche Muster so aussehen, wie sie nun einmal aussehen. Es geht hier um ganz alltägliche Werkzeuge objektorientierter Programmiersprachen aus Sicht der Entwurfsmuster.

1.3.1 Schnittstellen-Implementierung vs. Klassen-Vererbung

In meiner Praxis erlebe ich sehr häufig, dass Entwickler (ganz besonders übrigens Dienstleister) glauben, in Schnittstellen den Stein des Weisen gefunden zu haben. Diese Entwickler gehen von folgender Denkweise aus:

Die Schnittstellen-Halbwahrheit

Wer gegen Schnittstellen programmiert, kann seinen Code später leicht erweitern und macht ihn unabhängig gegenüber einer konkreten Implementierung.

Manchmal liest (und sieht) man daher, dass für jede Klasse eine Schnittstelle zu erstellen sei. Sprachlich wird diese Praxis auch dadurch unterstützt, dass man landläufig von *Schnittstellenvererbung* spricht, wie auch von *Klassenvererbung* die Rede ist. Während Schnittstellen zwar von anderen Schnittstellen erben können, also deren Signaturen übernehmen, können Klassen von Schnittstellen rein gar nichts erben. Die Entwickler von Java haben das schön erkannt und das elegante Schlüsselwort `implements` verwendet, um klarzustellen, dass Klassen von Schnittstellen eben nicht *erben* können, sondern diese lediglich *implementieren*, während Klassen über das Schlüsselwort `extends` auch wirklich von anderen Klassen erben können.

Kurz gesagt:

Schnittstelle vs. Klasse

Eine *implementierte Schnittstelle* erzwingt eine Menge an Operationen, die die Schnittstelle definiert. Wer die Schnittstelle implementiert, ist also kompatibel zu einem Client, der nicht gegen die konkrete Klasse, sondern gegen die Schnittstelle programmiert, egal aus welcher Vererbungshierarchie die Klasse kommen mag. Darin besteht der große Vorteil: Man kann einem alten Hund, pardon einer alten »Klasse«, also neue Tricks beibringen.

Eine *vererbte Klasse* erbt nicht nur die Schnittstelle, also den Typ, der Basisklasse, sondern auch dessen Implementierung. Ein Subtyp kann die Klasse also erweitern (daher das Schlüsselwort `extends` in Java) oder die Implementierung überschreiben.

Dabei ist es nun egal, ob eine Variable den Typ der Basisklasse oder den Typ der Unterklasse hat. Entscheidend für die Wahl, welcher Code ausgeführt wird, ist der Typ des konkret instanziierten Objekts. Man nennt dieses Verhalten *Polymorphie* und die Methoden, die in abgeleiteten Klassen überschrieben werden können, *virtuelle Methoden*.

Die meisten Entwurfsmuster arbeiten mit der klassischen Vererbung, häufig sogar mit abstrakten Basisklassen, also Basisklassen ohne konkrete Implementierung. Das hat seinen Sinn, und Sie sollten dies nicht ohne Grund ändern. Häufig sieht man zusätzlich zu der Vererbung auch noch eine implementierte Schnittstelle, zum Beispiel dann, wenn eine Kommandoklasse (des Befehl-Entwurfsmusters) nicht nur von der Basisklasse `Command` erbt, sondern zusätzlich noch die Schnittstelle `ICommand` implementiert. Das ist im besten Fall unnötig, schafft neue Komplexität und eröffnet einem Client eine weitere, ebenfalls unnötige Wahlmöglichkeit: Er kann nun auf die Basisklasse casten oder auf die Schnittstelle.

Beachten Sie daher Folgendes:

Leitsatz zum Umgang mit Schnittstellen

Es gilt: Vererbung *vor* Schnittstellenimplementierung. Verwenden Sie Schnittstellen immer dann, wenn Sie

- Objekten, die aus verschiedenen Klassenhierarchien stammen, ein gemeinsames Verhalten beibringen müssen, oder
- Querschnittsfunktionen implementieren (zum Beispiel für verschiedene Listen die Fähigkeit, sortierbar zu sein) oder
- eine Technologie einsetzen, die Schnittstellen für ihre korrekte Funktionsweise erforderlich machen.

In den meisten anderen Fällen sollten Sie der Vererbung den Vorzug geben.

Warum Sie das tun sollten, das wird klar, wenn wir uns die Nachteile ansehen, die aus der Verwendung von Schnittstellen entstehen können:

- **Redundanz**: Da Schnittstellen keine Funktionalität implementieren, wird dieselbe Funktionalität häufig mehrfach entwickelt.
- **Inkompatibilität**: Das größte Problem besteht aber darin, dass Sie Schnittstellen nicht erweitern oder verändern können, ohne dass Sie mit allen Klassen brechen, die diese Schnittstellen bereits in ihrer alten Version implementieren. Das betrifft natürlich ganz besonders Entwickler von Bibliotheken, die gar nicht wissen können, wer ihre Schnittstellen in eigenen Klassen implementiert.

Nun gibt es aber häufiger den Fall, dass eine Basisklasse zwar eine Schnittstelle bereitstellen soll, aber selbst noch nichts Vernünftiges implementieren kann. Dann ist eine abstrakte Basisklasse vielleicht der richtige Weg. Sie werden sehen, dass Entwurfsmuster sehr häufig abstrakte Basisklassen einsetzen, vermutlich jedenfalls häufiger als die meisten Entwickler.

Auch hier spricht einiges für abstrakte Basisklassen und gegen Schnittstellen:

- Es ist nicht unwahrscheinlich, dass Sie später aus der abstrakten Basisklasse eine gewöhnliche Basisklasse machen wollen, weil Sie eine neue Funktion hinzufügen möchten, die bereits in der Basisklasse eine vernünftige Implementierung zulässt. Eine solche Änderung können Sie vornehmen, ohne dass Sie die Klassen, die von dieser Basisklasse abgeleitet wurden, groß ändern müssen.
- Viele Programmiersprachen erlauben es, dass Sie nur Teile (zum Beispiel einzelne Methoden) einer Klasse abstrakt machen, für andere Methoden aber wiederum Implementierungen in der Basisklasse anbieten.

1.3.2 Is-A vs. Has-A

Oder, anders gesagt: IST EIN vs. HAT EIN. Die klassische Vererbung begründet ein *Is-A-Verhältnis*. Ein Objekt vom Typ `Hund` ist also auch ein Objekt vom Typ `Tier`. Die Klasse `Hund` hat nahezu unbeschränkten Zugriff auf die Klasse `Tier`, nicht aber umgekehrt. Das hat Vorteile, aber auch Nachteile, denn zum einen können Klassen meist nur von einer Basisklasse erben, und selbst wenn Mehrfachvererbung möglich ist, wie in C++ zum Beispiel, entsteht daraus nicht selten hoch komplexer und fehleranfälliger Code. Andererseits ist es vielleicht überhaupt nicht gewünscht, dass eine Klasse auf eine andere Klasse Zugriff erhält.

Einige der hier vorgestellten Muster – und ich finde, es sind die besonders eleganten – beziehen ihre Stärke daher nicht aus der Vererbung, sondern aus der *Komposition* von Objekten. Das Kompositum-Muster hat sogar seinen Namen davon. Diese Komposition begründet nun eine *Has-A*-Beziehung: Ein Objekt hat also Zugriff auf verschiedene andere Objekte, von denen es nur die Schnittstellen, nicht aber die interne Arbeitsweise kennt.

Die klassische Vererbung hat natürlich den Charme, die Implementierung in Unterklassen ändern zu können. Andererseits sind Ober- und Unterklasse natürlich aufs Engste miteinander verbunden, und nicht selten bedingen Änderungen an der Oberklasse zumindest einen intensiven Test aller davon abgeleiteten Unterklassen. Häufig müssen sogar Details der Implementierung verändert werden. Ober- und Unterklasse sind außerdem zur Compilezeit miteinander verwoben. Polymorphie sorgt zwar dafür, dass zur Laufzeit der richtige Code ausgeführt wird, also der Code, der zur Klasse des konkreten Objekts gehört, aber das ist dann auch schon die ganze Dynamik.

In der Realität geht es häufig darum, zur Laufzeit ein gewisses Verhalten aus verschiedenen Teilen zusammenzubauen, also ein Objekt zu kreieren, das wiederum aus Teilobjekten besteht, die alle ein bestimmtes Einzelverhalten implementieren. Was daraus entsteht, ist jeweils einzigartig, und keine Vererbungshierarchie ist dieser Komposition im Weg. Das *Erbauer*-Muster ist ein Beispiel für ein Entwurfsmuster, das genau das zum Ziel hat.

Manchmal ist die Bildung von Unterklassen logisch, manchmal scheint die Oberklasse aber auch nicht recht zu passen, weil sie beispielsweise viel Funktionalität implementiert, die die Unterklasse gar nicht benötigt, oder weil die Unterklasse eine ganz neue Funktionalität umsetzen möchte. Das ist, wie gesagt, auch der Fall, wenn das konkrete Verhalten eines Objekts erst zur Laufzeit bestimmt werden kann.

Objektkomposition ist daher, ganz allgemein gesprochen, ein weiteres Mittel, um die Abhängigkeit zwischen verschiedenen Klassen zu verringern, und das ist meist eine gute Idee in stark vernetzten Systemen. Oder, anders gesagt:

Leitsatz zur Objektkomposition

Wo immer es möglich ist, sollten Sie lieber Objekte komponieren, als Klassen zu vererben.

Plug-in-Systeme sind ein gutes Beispiel für Systeme, die sich zur Laufzeit aus verschiedenen Komponenten, also Objekten, zusammensetzen. Im Idealfall lassen sich dann einzelne Komponenten austauschen und auch zur Laufzeit neu dazukonfigurieren, ohne das bestehende System in seiner Funktion zu stören.

Aber auch hier gilt: Hüten Sie sich vor den Extremen!

1.4 Ein kleines Beispiel aus der Praxis

Nehmen wir einmal als nicht ganz fiktives Beispiel eine Webanwendung, die zur Vermittlung von Sprachreisen dient. Alles beginnt, wie so häufig, mit dem, was Verantwortliche gerne als »gesunden Optimismus« und Projektmanager als »agil« bezeichnen – kurz: Es wurde einfach mal drauflosentwickelt.

1.4.1 Die Anforderung

Es gibt eine Klasse pro Webanfrage, also zum Beispiel für

- Login
- Suchen von Sprachreisen
- Buchen
- Impressum usw.

Natürlich funktioniert die Webanwendung am Ende auch. Die Anwendung hat vielleicht 30 Klassen, zwei einfach strukturierte Datenbanken (für den Produktkatalog und die Kunden) und tut, was man von ihr verlangt. Sie sieht vielleicht so ähnlich wie im Schema von Abbildung 1.2 aus:

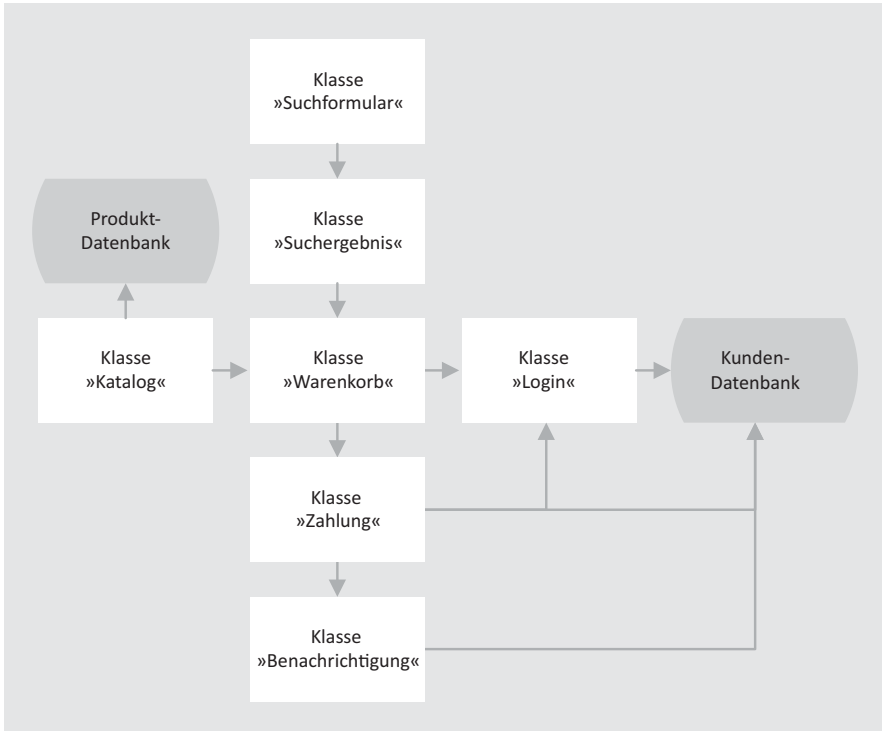

Abbildung 1.2 Eine einfache Webanwendung, die auf die »herkömmliche« Art geschrieben wurde. Die Pfeile geben den Fluss durch die Anwendung an.

1.4.2 Und dann kommt die Änderung der Anforderung

Irgendwann, in gar nicht allzu ferner Zeit hat ein Manager eine Idee: Man könnte die Anwendung doch auch auf Mobilgeräten zugänglich machen, wer hat schon auf Reisen einen Desktop-PC dabei?

Schnell wird klar: Die zu generierende Seite muss für mobile Clients völlig anders aussehen. Wird jetzt nicht auf Entwurfsmuster geachtet, entsteht schnell so etwas wie hier im Pseudocode dargestellt:

```
WENN Browser = MOBIL DANN
  GeneriereHtmlFuerMobil()
SONST
  GeneriereHtmlFuerDesktop()
ENDE
```


Praktisch alle Klassen bekommen auf diese Art eine Browserweiche verpasst, und das Problem – die Auswahl der geeigneten Darstellung – wird auf den gesamten Code verteilt.

Aber auch das ist vielleicht noch überschaubar. Der nächste Vorschlag lässt nicht auf sich warten: Neben dem eigenen Login soll nun auch ein Login über Google, Microsoft Live und Facebook möglich sein.

1.4.3 Der Versuch, das Ganze doch noch irgendwie hinzubekommen

Die Lösung könnte nun wiederum so aussehen:

- Ergänze die Datenbank um weitere Felder, wie sie Google, Facebook und Microsoft benötigen, während andere Felder – zum Beispiel das Passwortfeld – in einzelnen Fällen nun nicht mehr benötigt werden.
- Füge vier Radiobuttons auf der Login-Seite hinzu, um die Art des Logins für den Anwender auswählbar zu machen.
- Je nach Antwort, Sie ahnen es bestimmt schon, gibt es wieder einen ELSE-Zweig, in dem der Code dann zu finden ist.

Natürlich hilft die Objektorientierung mit ihren Möglichkeiten. Sie könnten zum Beispiel

- eine Schnittstelle, sagen wir `ILogin`, entwickeln und vier Klassen für alle vier Provider, die diese Schnittstellen implementieren, oder aber
- eine abstrakte Basisklasse entwerfen und vier konkrete Implementierungen dafür schreiben.

Der Code ist nun schon ein wenig übersichtlicher:

```
WENN LoginType = Facebook DANN
  loginForm = NEW FacebookLoginForm()
SONST WENN LoginType = Google DANN
  loginForm = NEW GoogleLoginForm()
…
```

Manche der angebotenen Alternativen, Microsoft zum Beispiel, verlangen, dass Sie statt einer eigenen Login-Seite ein Redirect auf deren Login-Seite einrichten. Sie sehen schon: noch mehr IF-ELSE-Anweisungen.

Und dennoch besteht ein Problem weiter: Was soll mit dem Passwort geschehen? Es wird nur für die eigene Implementierung gebraucht, nicht aber für die Login-Prozeduren der anderen Anbieter.

So oder so ähnlich präsentieren sich viele Probleme in der Praxis. Mit ein wenig Wissen und Fantasie ist jedes Problem schon lösbar, aber die Lösungen sind häufig

- unnötig komplex, weil der Code stark verschachtelt oder stark zerklüftet ist
- schwer zu warten
- noch schwerer zu erweitern
- nicht frei von redundantem Code
- widersprüchlich, weswegen immer wieder weitere Fallunterscheidungen im Code untergebracht werden

Freilich, bei den beiden Anforderungen wird es nicht bleiben. Vielleicht soll demnächst die Hosting-Plattform ausgetauscht werden, oder soll statt einer MS-SQL-Datenbank MySQL zum Einsatz kommen? Oder soll der Warenkorb in der Datenbank persistiert und dazu die Benutzerverwaltung aufgebohrt werden? Wie auch immer: Wenn es um Komplexität geht, dann gilt häufig: 1+1 > 2, denn jeder neue Code muss nicht nur für sich funktionieren, sondern auch im Zusammenspiel mit all den vielen anderen Klassen, Tabellen und weiteren Systemen einer Anwendung.

Solchen Problemen ist auch gemeinsam, dass wir mit der Objektorientierung allein nicht weit genug kommen, weil Objekte, Schnittstellen und Vererbung allein noch nicht ausreichen, um das Problem mit Butz und Stängel aus der Welt zu schaffen.

1.4.4 Entwurfsmuster als Lösung

Hier kommen nun die Entwurfsmuster ins Spiel. Die Webanwendung, aus Sicht des MVC-Entwurfsmusters, lässt sich dann so darstellen wie in Abbildung 1.3.

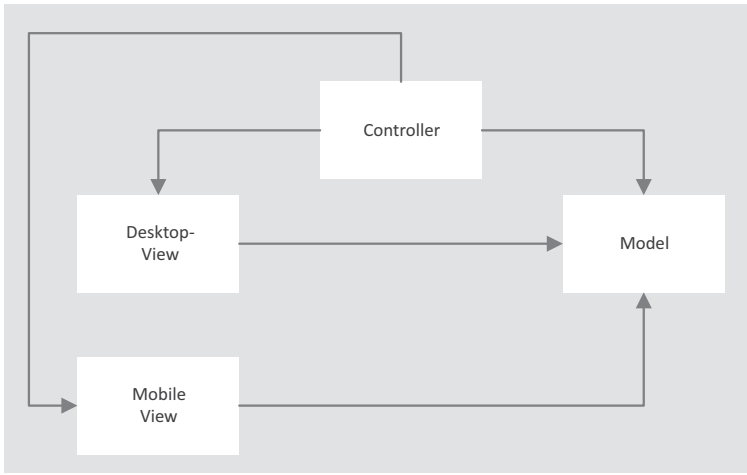

Abbildung 1.3 Die Webanwendung mit zwei Views

Entscheidend sind nun die Akteure, die Sie in einem späteren Kapitel noch kennenlernen werden. Hier sei nur so viel gesagt: Jeder Akteur hat seine spezifischen Aufgaben und Abhängigkeiten. Der Controller kann beispielsweise auf die View zugreifen

und diese steuern, die View kann aber nicht direkt auf den Controller zugreifen, sondern ist lediglich indirekt angebunden, und zwar über einen Beobachter-Mechanismus (der selbst wiederum einem Muster entspringt). Das Muster führt also nicht nur eine zweite View ein, was ziemlich trivial gewesen wäre, sondern geht weit darüber hinaus.

Man muss sich also auf ein Muster erst einmal im Ganzen einlassen. Es ist nicht einfach möglich, Teile eines Musters herauszugreifen, ohne dadurch das Muster – und dessen Vorteile – zu zerstören. Die meisten Muster bringen aber dennoch Varianten mit. Im obigen Beispiel könnte es mehrere Controller geben, einen Controller für jede View, wenn sich die Views in ihrer Darstellung und in ihrem Verhalten stark voneinander unterscheiden würden.

Das letzte Beispiel zeigt auch: Muster lassen sich häufig kombinieren. Denn Controller könnten auch von einer gemeinsamen Basisklasse erben, sodass es einfach ist, neue Controller hinzuzufügen, ohne jeweils die gesamte Steuerungslogik neu entwickeln zu müssen. Für die Instanziierung der Controller selbst gibt es ebenfalls wieder eine Reihe von Mustern, die Sie ebenfalls in diesem Buch finden.

1.5 Überlegungen zum Einsatz

So viel über das Wesen von Entwurfsmustern. Bevor wir nun richtig loslegen, folgen hier nun einige allgemeine, aber wichtige Hinweise zum Einsatz von Entwurfsmustern.

1.5.1 Erkenne das Muster

Jedes Muster besteht aus einem

- zu lösenden Problem,
- einem mehr oder minder abstrakten Kern,
- schmückendem Beiwerk drum herum und
- Implementierungsdetails.

Es kommt darauf an, ob Ihre Problemstellung zum Kern eines Musters passt – nur in diesem Fall ist die Verwendung wirklich von Vorteil.

Beispiel: Singleton-Muster

Problem

Der Kern des *Singleton*-Musters ist, dass das Muster sicherstellt, dass es genau nur ein Exemplar gibt, auf das über einen – meist global zugreifbaren – Bezeichner zugegriffen werden kann. Ihr Problem passt also zu diesem Muster, wenn Sie genau das

sicherstellen müssen, zum Beispiel wenn es lediglich eine zentral genutzte Ressource gibt.

Kern

Der zentrale Kern ist hier, dass die einzige Instanz durch ein statisches Feld sichergestellt wird. Außerdem soll ein privater Konstruktor verhindern, dass ein Anwender selbst Instanzen des Objekts anlegt.

Schmückendes Beiwerk

Vielleicht dient eine statische Initialisierungsmethode dazu, das Objekt bei der ersten Verwendung zu initialisieren. Das ist natürlich für die Lösung wichtig, für das Muster allerdings nicht, weil es ja nicht darum geht, wie genau das zu erstellende Objekt aussehen soll, sondern eben darum, dass es davon nur ein Exemplar gibt. Oder Sie entscheiden sich, das Objekt nicht erst beim ersten Zugriff zu erstellen, sondern deterministisch beim Programmstart.

Implementierungsdetails

In der Implementierung könnte eine wichtige Frage sein, ob in der Programmiersprache der Wahl die Erzeugung des Objekts threadsicher ist und überhaupt sein muss. Diese Details unterscheiden sich von Sprache zu Sprache.

Manche der Muster unterscheiden sich nur im Detail, dann sind sowohl die präzise Kenntnis der Muster gefragt als auch genaues Hinsehen. Bei den Erzeugungsmustern gibt es welche, mit denen verschiedene Ableitungen von Objekten erstellt werden können (die abstrakte Fabrik), und solche, bei denen ein Objekt einer einzigen Klasse erstellt wird, dieses aber in verschiedenen Konfigurationen (der Erbauer, zum Beispiel).

1.5.2 Dokumentation und Bezeichnung

Der Verständlichkeit Ihres Codes hilft es manchmal ungemein, wenn Sie die Verwendung eines Musters kommentieren, zum Beispiel durch einen Kommentar im Code:

```
//Command Pattern
//Client=Main Application Form, Command=Execute, Receiver=Canvas
```

Es gibt ziemlich viele Muster, und nicht immer kennen Entwickler das Muster oder erkennen es gar aus dem Code.

Die Benennung ist ebenfalls wichtig. Oft liest man, dass Objekte wie die Bausteine der Muster benannt sind. Und so gibt es eben ein Objekt mit dem Namen `Singleton`. Das ist selten eine gute Idee, denn in Ihren Code sollten die fachlich zu lösenden Probleme die erste Geige spielen, nicht die Muster selbst. Aber es spricht natürlich nichts

dagegen, den Namen um den entsprechenden UML-Akteur zu ergänzen, also zum Beispiel Namen zu wählen wie:

- `ConfigurationSingleton` (Singleton-Muster)
- `PageController` (MVC-Muster)
- `TaskFactory` (Abstract Factory Method)

Auch hier kann ein Kommentar oder ein eingestreutes UML-Diagramm es Dritten erleichtern, die Zusammenhänge in Ihrem Code zu verstehen.

1.5.3 Anti-Patterns

Gelegentlich kann es förderlich sein, sich sogenannte Anti-Patterns anzusehen, also das, was man möglichst nicht tun sollte, um daraus dann das optimale Verhalten – also das Gegenteil – abzuleiten. Der Vorteil dieses Verfahrens ist es, dass einem wieder bewusst wird, welche Probleme eigentlich im Detail zu lösen sind.

Beispiel

Ein bekanntes Anti-Pattern ist es, SQL-Statements im Code zusammenzusetzen:

```
String sql = "INSERT INTO table(field1) VALUES("+editField1.Text+")";
```

Dieses Beispiel verdeutlicht, wo die Gefahren liegen, nämlich in dem ungeprüften Hinzufügen von User-Control-Inputs zu einem SQL-Statement, was Angreifern Tür und Tor öffnet. Man nennt solche Angriffe auch *SQL Injection*.

Anti-Patterns (warum gibt es dafür eigentlich kein eingängiges deutsches Wort?) kommen in verschiedenen Ausprägungen daher. Sie verschlimmbessern also zum Beispiel:

- die Sicherheit einer Anwendung (wie im Beispiel oben)
- die Größe einer Anwendung durch überflüssigen oder redundanten Code
- die Stabilität einer Anwendung

Manchmal sind Muster in den Augen einiger zugleich auch Anti-Patterns. Das Singleton-Muster kann als Beispiel herhalten. Als Muster stellt es die Einzigartigkeit eines Objekts sicher, aber es schränkt auch ein, weil es zum Beispiel der Skalierbarkeit im Wege steht – was es im Einzelfall gleichzeitig zu einem Anti-Pattern macht.

1.5.4 Verhältnismäßigkeit

Ein Muster muss zum Problem passen, nicht nur fachlich, sondern auch im Hinblick auf die Komplexität des Problems und die Komplexität des Musters. Das soll heißen:

Einfache Probleme kann man ruhig auch einfach lösen. Nur weil man einen Hammer hat, wird nicht automatisch alles zum Nagel.

Obwohl es raffinierte Muster für die Objekterzeugung gibt, werden Sie dennoch an vielen Stellen einfach ein Objekt auf die gewöhnliche Art und Weise erzeugen wollen. Um die Verhältnismäßigkeit zu bewerten, können die folgenden Fragen dienen:

- Was ist die nächstbeste Alternative, die ohne Muster auskommt?
- Welchen Implementierungsaufwand kostet das Muster?
- Welchen Pflegeaufwand erwarte ich? Spart das Muster über die Zeit Aufwand und Kosten?
- Wird sich die Programmfunktion ändern, vor allem: Wird sie in nächster Zeit erweitert werden?
- Kann Code von anderen Programmteilen wiederverwendet werden, um das Problem vielleicht einfacher und schneller zu lösen?

1.6 Wegweiser

Was erwartet Sie in diesem Buch? Hier folgt nun, als Appetizer und der Übersichtlichkeit halber, eine kurze Darstellung der Kapitel und der Muster.

Die Muster können unabhängig voneinander gelesen werden. Sie können sich also gerade das Muster herauspicken, das für Ihre Praxis nützlich ist oder Ihrem Interesse entspricht. Querverbindungen zwischen den Mustern können (und sollten) Sie folgen, denn dadurch treten vor allem die Unterschiede zutage, und Sie können leichter entscheiden, welches Muster für den konkreten Einsatzzweck wirklich das beste ist.

Wenn Sie wollen, können Sie diesem (ganz und gar subjektiven) »Leseplan« folgen, der die wichtigsten Dinge an den Anfang Ihrer Lektüre stellt. In der »ersten Welle« könnten das die folgenden Kapitel sein:

1. Designprinzipien (dort vor allem: *Open-Closed-Principle*, *Dependency Inversion Principle* und *Single Responsibility Principle*)
2. Abstrakte Fabrik
3. Fabrikmethode
4. Singleton
5. Fassade
6. Proxy
7. Befehl
8. Iterator
9. Beobachter

10. Strategie
11. Datentransferobjekt
12. Spätes (verzögertes) Laden
13. Konkurrierender Zugriff auf Daten
14. Dependency Injection
15. Model View Controller

Die zweite Welle:

1. Adapter
2. Dekorierer
3. Schablonenmethode
4. Kompositum
5. Datenzugriffsobjekt
6. Unit of Work
7. Model View ViewModel (MVVM)

Die dritte Welle umfasst dann den Rest der Kapitel.

1.6.1 Erzeugungsmuster (Creational Design Patterns)

Los geht es in Kapitel 2 mit einigen »klassischen« Entwurfsmustern, nämlich den Mustern, die Objekte erzeugen:

Fabrikmethode (Factory Method)

Dieses beliebte Muster beschreibt eine Methode, die ein Objekt erzeugt und zurückgibt, wobei Unterklassen über den konkreten Typ des zu erzeugenden Objekts entscheiden.

Einzelstück (Singleton)

Ein Singleton stellt sicher, dass es von einer Klasse nur ein einziges Objekt geben kann, und eignet sich damit für zentrale, nur einmal vorhandene Ressourcen.

Multiton

Dieses Muster erweitert das Singleton-Muster um einen festen Pool von Objekten, zum Beispiel um fünf offene Datenbankverbindungen, die von Clients angefordert und verwendet werden können.

Abstrakte Fabrik (Abstract Factory)

Dieses Muster kann Objekte beliebiger Ebenen in Vererbungshierarchien erzeugen, ohne dass ein Verwender diese kennen muss.

Erbauer (Builder)

Mit dem Erbauer-Muster lassen sich zum Beispiel Objekte aus verschiedenen Teilen konstruieren, indem die Erstellung eines komplexeren Objekts von seiner Darstellung getrennt wird.

Prototyp (Prototype)

Mithilfe dieses Musters lassen sich immer neue Instanzen auf Grundlage einer prototypischen Instanz erzeugen, durch Kopieren und Anpassen des Prototyps.

1.6.2 Strukturmuster (Structural Design Patterns)

Die nächste Gruppe von Entwurfsmustern, die in Kapitel 3 beschrieben werden, ist ein wenig komplexer und beschreibt Strukturen, zum Beispiel durch Komposition von kleineren Objekten zu einem größeren Objekt.

Adapter

Der Adapter bringt Klassen zusammen, die eigentlich aufgrund von nicht kompatiblen Schnittstellen nicht zusammenarbeiten könnten, indem er eine Schnittstelle in eine andere übersetzt.

Brücke (Bridge)

Mit der Brücke lassen sich Implementierung und Schnittstelle voneinander trennen, sodass beide unabhängig voneinander verändert werden können.

Kompositum (Composite)

Mithilfe dieses Musters lassen sich komplexere Objekte aus einfacheren Objekten erzeugen.

Dekorierer (Decorator)

Anstatt Unterklassen zu bilden, können Sie eine Klasse mit dem Decorator-Muster um neue Funktionalitäten dynamisch erweitern.

Fassade (Facade)

Eine Fassade ist eine einheitliche und vereinfachte Schnittstelle, die einen einfachen und konsistenten Zugang zu einer Menge von Schnittstellen eines Subsystems ermöglicht.

Fliegengewicht (Flyweight)

Das Fliegengewicht (welch schöner Name) macht die Verwendung vieler Objekte einfacher und performanter, die zudem variable Informationen teilen.

Proxy

In diesem besonders bekannten Entwurfsmuster dient ein Proxy-Objekt dazu, auf ein anderes, nachgelagertes Objekt zuzugreifen, und zwar in einer kontrollierten Art und Weise.

1.6.3 Verhaltensmuster (Behavioral Design Patterns)

Bei den Verhaltensmustern aus Kapitel 4 geht es um das Verhalten von Objekten zueinander, also um die Interaktion zwischen Objekten und die Aufteilung von Zuständigkeiten zwischen Objekten. Verhaltensmuster sind besonders zahlreich und wichtig, weil sie unser Augenmerk auf das Muster – die kontrollierte Interaktion – richten und nicht von uns verlangen, dass wir den Kontrollfluss selbst aus dem Programm herauslesen.

Zuständigkeitskette (Chain of Responsibility)

Die Zuständigkeitskette erlaubt es, dass mehrere Objekte miteinander verkettet werden, damit sie gemeinsam eine eingehende Anforderung verarbeiten. Diese Anforderung wird so lange entlang der Kette weitergereicht, bis eines der Objekte in der Kette die Anforderung bearbeiten kann.

Befehl (Command)

Das Befehlsmuster kapselt Befehle in Objekten, die dann parametrisiert oder auch weitergeleitet, verzögert oder für ein späteres Undo gespeichert werden können.

Interceptor

Mithilfe einer Umleitung wird die ursprüngliche Kette der Verarbeitung um ein neues Glied erweitert, sodass die Funktionalität bequem erweitert werden kann.

Interpreter

Mithilfe eines Interpreters lassen sich Probleme in einer eigenen Sprache beschreiben. Der Interpreter interpretiert dann die Grammatik dieser Sprache und löst das Problem.

Iterator

Das Iterator-Muster ist ebenfalls häufig anzutreffen. Es ermöglicht das sequenzielle Durchlaufen einer Menge von Objekten oder der Elemente eines zusammengesetzten Objekts.

Vermittler (Mediator)

Ein Vermittler-Muster vermittelt zwischen verschiedenen Objekten, indem es das Zusammenwirken dieser Objekte in sich kapselt.

Memento

Das Memento-Muster erstellt eine Momentaufnahme des inneren Zustands eines Objekts, sodass später das Objekt wieder in diesen Zustand versetzt werden kann, zum Beispiel um ein Undo zu ermöglichen.

Beobachter (Observer)

Mithilfe eines Observers lassen sich beliebig viele Beobachter-Objekte benachrichtigen, sobald sich am beobachteten Objekt etwas verändert, ohne dass das zu beobachtende Objekt seine Beobachter vorher kennen muss.

Zustand (State)

Dieses Chamäleon unter den Mustern erlaubt es, dass ein Objekt sein Verhalten ändert, sobald sich sein innerer Zustand ändert. Von außen sieht es also so aus, als ob eine andere Klasse verwendet würde.

Stratregie (Strategy)

Eine Strategie ist im Kontext dieses Musters ein bestimmtes Verhalten (oder ein bestimmter Algorithmus). Das Strategie-Muster gestattet es nun, dass dieses Verhalten zur Laufzeit ausgetauscht werden kann, und zwar in Abhängigkeit vom Verwender.

Schablonenmethode (Template Method)

Eine abstrakte Klasse definiert ein »Skelett« eines Algorithmus, dessen konkrete Teile dann an Unterklassen delegiert werden.

Besucher (Visitor)

Dieses Muster trennt eine Klassenhierarchie von einer zweiten Klassenhierarchie, die die Operationen enthält.

1.6.4 Architekturmuster (Architectural Design Patterns)

In Kapitel 5 geht es weniger um Code als in den anderen Kapiteln, weil die hier beschriebenen Muster weniger Design oder Implementierung unterstützen, sondern die Architektur einer Anwendung oder eines Systems von Anwendungen.

Kleine Architekturmusterkunde

Am Anfang steht ein Überblick über typische Architekturstile: was sie ausmacht und worin sie sich unterscheiden.

The (8) Fallacies of Distributed Computing

Dieser Abschnitt beschäftigt sich mit verteilten Anwendungen und dem, was dort in der Praxis häufig zu Problemen führt.

Serviceorientierte Architektur (Service-oriented Architecture – SOA)

SOA beschreibt ein Paradigma, wie verteilte Komponenten, also Komponenten die auf verschiedenen Rechnern in verschiedenen Prozessen laufen, miteinander interagieren können.

Event Sourcing

Beim Event Sourcing geht es darum, die Ereignisse zu speichern, um auf diese Weise die Zeit zurückdrehen zu können oder um eine vollständige Dokumentation allen Geschehens zu erhalten.

Command Query Responsibility Segregation (CQRS)

Im Kern geht es um die Tatsache, dass das Lesen von Daten und das Verändern von Daten zwei völlig verschiedene Vorgänge sind, die auch verschiedene Vorgehensweisen beim Design von Informationsverarbeitungssystemen ratsam machen.

1.6.5 Datenmuster (Data Design Patterns)

In Kapitel 6 geht es um Entwurfsmuster, die im Zusammenhang mit der Verarbeitung von Daten stehen.

Unit of Work

Dieses Muster passt eigentlich an verschiedene Stellen dieses Buchs. Es beschreibt, wie eine Sammlung von Objekten – üblicherweise Geschäftstransaktionen – gemeinsam und koordiniert in die Datenquelle geschrieben werden.

Transaktionen

Dieser Abschnitt beschäftigt sich mit lokalen und verteilten Transaktionen, sowie ihren Eigenschaften und Besonderheiten – ein Thema, das in der Praxis häufig zu den verschiedensten Problemen führt.

Datentransferobjekt (Data Transfer Object)

Ein Datentransferobjekt dient dazu, Daten zu bündeln, sodass sie in einem Aufruf vom Aufrufer an seine Gegenstelle übertragen werden können.

Table Data Gateway

Das Table Data Gateway ist ein Objekt, das die Verbindung zu einer Datenbanktabelle kapselt.

Row Data Gateway

Im Gegensatz dazu verwaltet dieses Muster die Verbindung zu einem Datensatz, es gibt also für jeden Datensatz eine eigene Instanz eines Row Data Gateways.

Identity Map / Identity Function

Mithilfe dieses Musters lässt sich ein Cache realisieren. Es kann prima mit einem Row Data Gateway kombiniert werden.

Optimistisches Sperren

Dieser und der nächste Abschnitt beschreiben die beiden grundlegenden Sperrmechanismen, die in Mehrbenutzersystemen ungemein wichtig sind. Zunächst betrachten wir das optimistische Sperren, das Locks vermeidet.

Pessimistisches Sperren

Das pessimistische Sperren beruht im Kern auf Locking, also auf dem Sperren von Ressourcen für die gleichzeitige Verwendung.

Vererbung

Mit der Frage, wie sich die Vererbung der OO-Welt in der Welt der relationalen Datenbanken abbilden lässt, befasst sich der letzte Abschnitt von Kapitel 6.

1.6.6 GUI-Muster

In Kapitel 7 beschreibe ich einige Muster, die (vor allem) für Benutzeroberflächen und zur Interaktion mit dem Anwender eingesetzt werden.

Model View Controller (MVC)

Hier wird ein Muster beschrieben, das aus einem Modell (den darzustellenden Daten), einer Präsentationsschicht und einem Controller (für die Interaktion) besteht. Das Muster wird vor allem in Webanwendungen gern und häufig eingesetzt, und auch einige Technologien, wie ASP.NET MVC, erfordern oder begünstigen seine Verwendung.

Model View Presenter (MVP)

Auch dieses Muster kennt wieder drei Akteure: das Modell (die Logik) und die Ansicht der Daten (für Ein- und Ausgaben) sowie den Präsentator, das verbindende Element zwischen Modell und Ansicht, der die Abläufe steuert und die Funktionalität implementiert.

Model View ViewModel (MVVM)

MVVM ist eine Variante von MVC. Es ist vor allem bekannt geworden durch Microsoft-UI-Technologien wie WPF und Silverlight, aber auch durch HTML5. Im Kern geht es dabei um die Trennung der Verantwortlichkeiten zwischen den Entwicklern und den Designern einer Anwendung.

1.6.7 Design- und Entwicklungsprinzipien (Design Principles)

In Kapitel 8 beschreibe ich wichtige Designprinzipien und allgemeine Prinzipien der Softwareentwicklung. Vielleicht erkennen Sie einige oder alle davon aus Ihrem Informatikstudium wieder. Dann betrachten Sie es einfach als willkommene Wiederholung; ansonsten helfen diese Prinzipien vor allem bei der Diskussion mit Kollegen und bei dem Design von Lösungen.

Merkmale schlechten Designs

Den Anfang machen einige Überlegungen zu gutem und schlechtem Design, sozusagen als Motivation für die im Folgenden vorgestellten Prinzipien. Es folgen die *SOLID-Prinzipien*:

Eine-Verantwortlichkeit-Prinzip (Single Responsibility Principle)

Das vielleicht wichtigste Muster aus diesem Kapitel besagt, dass jede Klasse (aber auch jede Methode) einem einzigen Zweck dienen soll, was die Komplexität redu-

ziert, die Wartbarkeit und die Testbarkeit verbessert – neben vielen anderen Vorteilen.

Offen-Geschlossen-Prinzip (Open-Closed-Principle)
Softwarebausteine, wie Klassen und Funktionen, sollen offen für Erweiterungen sein (*Open*), aber nicht modifiziert werden können (*Closed*).

Liskovsches Substitutionsprinzip (Liskov's Substitution Principle)
Bei diesem Prinzip geht es um die Forderung, dass ein Programm statt mit Objekten einer Basisklasse auch mit Objekten abgeleiteter Klassen zurechtkommen muss.

Schnittstellenaufteilungsprinzip (Interface Segregation Principle)
In diesem Muster geht es um die Frage, wie umfangreich Schnittstellen sein dürfen, damit die implementierenden Klassen nicht unnötig umfangreich werden und vor allem keine unnötigen Member implementieren müssen.

Dependency-Inversion-Prinzip (Dependency Inversion Principle)
Weiter geht es mit der Umkehrung der Verantwortung (*Inversion*) und den Abhängigkeiten von Modulen (*Dependency*). Häufig wird dieses Muster auch im Zusammenhang mit Dependency-Injection-Containern genannt.

Das agile Manifest (Agile Manifesto)

Der Abschnitt, der das agile Manifest und seine Werte und Prinzipien beschreibt, entfernt sich ein Stück weit vom Kern dieses Buchs, den Mustern. Ich habe ihn aufgenommen, weil so viele Publikationen auf das agile Manifest verweisen und weil es das Fundament der agilen Softwareentwicklung darstellt.

Designprinzipien (Design Principles)

Wieder eine Spur konkreter sind die Designprinzipien, die einem immer und immer wieder begegnen und die oft ein guter Ratgeber sind, ganz unabhängig von Technologie und Programmiersprache. Man sollte sie einfach kennen.

Design Smells und Anti-Patterns

Schlechtes Softwaredesign verrät sich meist durch Design Smells und die Verwendung von Anti-Patterns. Die wichtigsten davon finden Sie in drei Abschnitten: zu Abstraktionen, zur Kapselung und zu Hierarchien.

Kapitel 2
Erzeugungsmuster

Hätte man bei der Erschaffung der Welt eine Kommission eingesetzt, dann wäre sie heute noch nicht fertig.
– George Bernard Shaw

Erzeugungsmuster haben an Bedeutung zugenommen, und zwar in dem Maße, wie Software flexibler, konfigurierbarer, modulorientierter und orchestrierbarer wurde. Die Erzeugungsmuster sind für die Erzeugung von Objekten in OO-Sprachen da, das verrät bereits der Name.

Früher war die Sache einfach. Objekte wurden statisch im Code erzeugt, so zum Beispiel:

```
MwstCalculator calculator = new MwstCalculator ();
float vat = calculator.berechne(4500);
```

Natürlich bietet das keine ausreichende Flexibilität, denn wenn die Berechnung der Mehrwertsteuer für verschiedene Länder durchgeführt werden soll, braucht man bereits eine weitere Fallunterscheidung, entweder über einen mitgeführten Parameter wie

```
MwstCalculator calculator = new MwstCalculator("DE");
```

oder über eine klassische Vererbungshierarchie:

```
MwstCalculator calculator = new MwstCalculatorDE();
```

Schon bald kam der Gedanke auf, weitere Funktionalitäten einer bestehenden Installation hinzuzufügen, meist indem weitere Dateien kopiert und diese in einer Konfiguration bekannt gemacht werden, sagen wir die Mehrwertsteuerberechnung für Kasachstan – für ein zusätzliches Salär versteht sich. Der Markt der Add-ins war geboren und die Softwareentwicklung um ein weiteres Problem reicher: Wie lässt sich die Erzeugung von Objekten zur Laufzeit flexibel und konfigurierbar steuern?

Noch mehr Flexibilität erfordern Softwaresysteme, deren Objekte nicht mehr von festen Vererbungshierarchien abhängen, sondern deren Komponenten nur lose gekoppelt sind. Man spricht dann auch häufig davon, dass diese Komponenten »komponiert« oder »orchestriert« werden. Besonders in Java wurden (und werden)

dafür gern *Dependency Injection Container* eingesetzt, also Softwarekomponenten, die die Abhängigkeiten eines Objekts zur Laufzeit auflösen – und dafür Muster einsetzen, wie sie in diesem Kapitel beschrieben werden.

Natürlich gibt es viele weitere Gründe, Objekte nicht einfach mittels `new` im Code zu instanziieren, aber die Grundidee, die hinter allen Erzeugungsmustern steckt, ist immer dieselbe: Der Teil des Codes, der ein Objekt eines bestimmten Typs benötigt, weiß nichts über dessen Erzeugung. Stattdessen beauftragt er eine (oder mehrere) Erzeugerklassen, die dieses Wissen kapseln und damit steuern können,

- *welches* konkrete Objekt erzeugt werden soll,
- *wie* das geschieht,
- welche weiteren Klassen an der Erzeugung beteiligt sind (*wer* das Objekt also wirklich erzeugt) und
- mitunter *wann* es erzeugt wird.

Wenn das konkrete Objekt schon nicht bekannt ist (nur die Erzeugerklasse kennt es), braucht der Code, der das Objekt schließlich verwenden möchte, natürlich einen flexibleren Zugang zum Objekt. Daher wird sehr häufig gegen Schnittstellen programmiert oder – seltener – eine abstrakte Basisklasse verwendet, der »kleinste gemeinsame Nenner aller infrage kommenden Objekte«:

```
MwstInterface calculator = MwstFactory.create("DE");
```

Sie werden in diesem Kapitel alle gebräuchlichen Erzeugungsmuster kennenlernen. Manche davon ergänzen sich, andere wiederum sind Alternativen, aus denen Sie wählen können – ich gehe jeweils an Ort und Stelle auf die Querverbindungen der Muster untereinander ein.

Das ist ein Praxisbuch, und daher gehe ich in jedem Kapitel auch auf Beispiele ein, wie sie in der Praxis eben anzutreffen sind. Denn schließlich ist das Wichtigste an Entwurfsmustern, ihre Einsatzmöglichkeiten zu (er)kennen, den Rest kann man schließlich im Bedarfsfall nachlesen.

Meist gibt es für ein Erzeugungsproblem gleich mehrere Muster, die infrage kommen. Um das »richtige« Muster auszuwählen, oder sagen wir besser: das geeignetste Muster, können Sie einige Fragen an die hier vorgestellten Muster stellen:

- Kann das Muster nur einen Objekttyp erzeugen oder eine ganze Hierarchie an Objekten?
- Geht es im Kern darum, Objekte aus einer Hierarchie zu erzeugen, oder darum, Objekte aus verschiedenen Komponenten zu bauen?
- Wie groß ist die Lösung, in die das Muster passen soll?

- Gibt es ganz spezielle Anforderungen, zum Beispiel die, dass es nur genau ein Objekt geben darf?

Auch dann gibt es noch Fälle, in denen mehrere Muster passen. Ich empfehle Ihnen dann, nicht gleich auf das einfachste Muster, die Fabrikmethode, zu verfallen, sondern auch den anderen, komplexeren Mustern eine Implementierungschance zu geben.

Aber genug der Vorrede, beginnen wir mit dem Inbegriff des Erzeugungsmusters, der Fabrikmethode.

2.1 Fabrikmethode

Der Begriff *Factory* (Fabrik) wird ein wenig inflationär gebraucht. In Frameworks wimmelt es nur von Factorys, und nicht immer wird dieser Begriff einheitlich verwendet. Das Fabrikmethode-Entwurfsmuster definiert zuallererst eine Methode, die ein Objekt erzeugt. Welche Klasse instanziiert wird, entscheidet die konkrete Klasse, die diese Methode implementiert. Es gibt also zwei Vererbungshierarchien: eine mit den Factory-Klassen und eine zweite mit den zu erzeugenden Objekten, den »Produkten«.

2.1.1 Steckbrief

Deutscher Name: Fabrikmethode

Englischer Name: Factory Method

Gruppe: Erzeugungsmuster

2.1.2 Beschreibung

UML

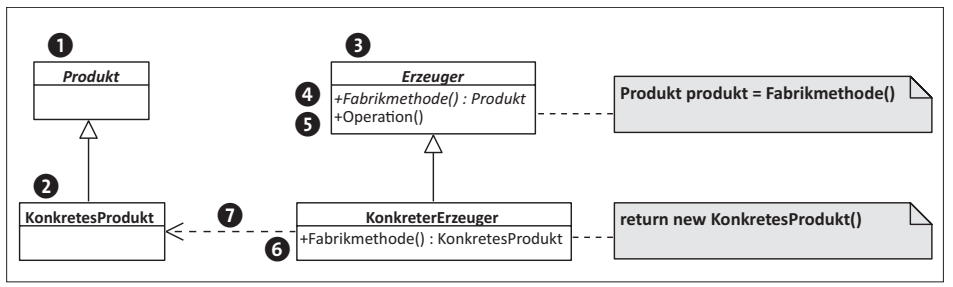

Abbildung 2.1 Das Fabrikmethode-Muster in UML

Nr.	Erläuterung
❶	Die abstrakte Basisklasse des zu erstellenden Objekts
❷	Die Klasse des konkret zu erstellenden Objekts
❸	Die abstrakte Basisklasse, deren wichtigste Aufgabe die Bereitstellung der Fabrikmethode ist
❹	Die Fabrikmethode. Diese kann entweder abstrakt sein oder selbst schon ein »Basisobjekt« erzeugen.
❺	Eine beliebige Operation, die ein Objekt vom Typ `Produkt` bzw. dessen Ableitungen benötigt
❻	Die in der Unterklasse überschriebene Fabrikmethode, die nun das konkrete Objekt instanziiert und zurückgibt
❼	Die konkrete Klasse zur Erstellung eines Objekts muss natürlich einen Verweis auf die Klasse mit dem konkreten Produkt haben, die Klasse also kennen – daher diese Abhängigkeit. Und auch der Erzeuger kennt die Produktklasse.

Tabelle 2.1 Akteure des Fabrikmethode-Musters

Erläuterungen

Wie in allen Erzeugungsmustern geht es um die Erstellung eines Objekts, und zwar eines Objekts, das von der abstrakten Basisklasse `Produkt` abgeleitet ist. Häufig wird die Basisklasse abstrakt sein, sie muss es aber nicht grundsätzlich sein. Nehmen wir einmal ein System, in dem Geschäftsvorfälle bearbeitet werden, ein ERP-System (*Enterprise Resource Planning*). Dann könnte die konkrete Vererbungshierachie so wie in Abbildung 2.2 aussehen.

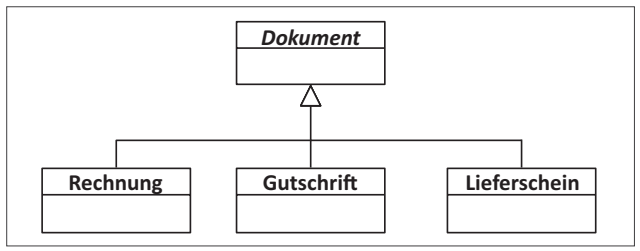

Abbildung 2.2 Ein konkretes Beispiel für die zu erstellenden Objekte

Die Klasse *Dokument* ist hier abstrakt, weil es in der Praxis eben immer nur konkrete Dokumente, also zum Beispiel Gutschriften, gibt, die reale Geschäftsvorfälle abbilden; dennoch wird sie Methoden und Felder enthalten, zum Beispiel ein Feld zur Aufnahme der Belegnummer.

Die nächste Klasse im Bunde ist die `Erzeuger`-Klasse. In unserem Beispiel wird das eine Klasse sein, die zur Faktur von Belegen dient. Nennen wir sie einmal `Fakturierer`. Auch sie kann, muss aber nicht abstrakt sein, kann also auch schon eine Standardimplementierung enthalten. Auf jeden Fall wird sie an irgendeiner Stelle ein Objekt benötigen, im Beispiel also ein Dokument. Im UML-Diagramm aus Abbildung 2.1 ist das durch die Methode `Operation()` angedeutet. Für die Objekterstellung ist die Factory-Methode zuständig, also `Fabrikmethode()`. Sie gibt ein Objekt vom Typ `Dokument` zurück. Wenn die `Dokument`-Klasse abstrakt ist, dann muss es auch die `Fakturierer`-Klasse sein, denn eine abstrakte Klasse kann eben nicht instanziiert werden. Diese Aufgabe wird den Ableitungen von `Fakturierer` zuteil, also zum Beispiel der Klasse `GutschriftFakturierer`. Diese konkreten Implementierungsklassen erzeugen jeweils ein Dokument vom gewünschten Typ.

2.1.3 Anwendungsfälle

An vielen Stellen gibt es abstrakte Klassen mit jeweils einer variablen Anzahl von Implementierungen. Vor allem Bibliotheken und Frameworks arbeiten ausgiebig damit.

Einige Beispiele, diesmal aus dem .NET-Framework:

- die abstrakte Basisklasse `System.IO.Stream` und deren Ableitungen `GZipStream`, `MemoryStream` oder `SqlFileStream`
- die ebenfalls abstrakte Basisklasse `DbConnection` und die konkreten Klassen `SqlConnection` oder `OracleConnection`

Als Entwickler programmieren wir nun gern gegen die Basisklasse und überlassen »die Details« den untergeordneten Klassen, also beispielsweise die Implementierung einer Stored Procedure für den SQL-Server und Oracle.

Nun wird die Entscheidung über die konkret zu implementierende Klasse erst sehr spät, zur Laufzeit, getroffen, zum Beispiel durch die Konfiguration in der Anwendungskonfigurationsdatei, ob als SQL-Server *Oracle* oder der *Microsoft SQL-Server* zum Einsatz kommt.

Betrachten wir das Beispiel von oben weiter: die Belegverarbeitung. Nehmen wir einmal an, fleißige Menschen im Kundenservice erfassen Tag für Tag Geschäftsvorfälle: Kündigungen, Bestellungen, Gutschriften, Auslieferungen und vieles mehr. Die `fakturiereDokumente`-Methode kann nun unmöglich wissen, welche konkreten Objekte benötigt werden (soll heißen, welche Belege fakturiert werden sollen), aber sie weiß, dass für jeden Geschäftsvorfall ein konkretes Belegobjekt benötigt wird und wann es im Prozess zu erzeugen ist. Daher wird lediglich die Fabrikmethode aufgerufen; die eigentliche Objekterzeugung findet aber erst in den konkreten, abgeleiteten Klassen statt, wie in der Klasse `GutschriftFakturierer`.

Diese Implementierung ist nun also ausgelagert, vermutlich in ganz eigenen Bibliotheken. Sie ist auch erweiterbar, und in der Praxis werden Implementierungen häufig sogar zur Laufzeit bei Bedarf dazu geladen. Wir sind damit in der Lage, das Framework, also die Klassen zur Steuerung des Prozesses (der Belegverarbeitung), von der Implementierung zu trennen.

Damit ergeben sich die wichtigsten Anwendungsfälle:

- Die konkret zu erzeugenden Objekte sollen bequem erweiterbar sein.
- Sie sind im Framework zur Entwicklungszeit nicht bekannt, und dennoch soll damit gearbeitet werden.

Auf der anderen Seite muss es auch möglich sein, im Framework-Code etwas Sinnvolles zu leisten, also das Objekt zu erzeugen und damit zu arbeiten, egal welches konkrete Objekt zur Laufzeit dann tatsächlich verwendet wird. Hätten Rechnungen und Gutschriften wenig oder nichts miteinander zu tun, dann wäre dieses Muster nicht von Nutzen.

2.1.4 Implementierung

Führen wir unser Beispiel zur Belegverarbeitung aus. Zunächst betrachten wir die »Produktklassen«, also das Dokument und die konkreten Belegtypen.

Produkt / Konkretes Produkt

```
public abstract class Dokument
{
  private int belegnummer;

  public int getBelegnummer()
  {
    return belegnummer;
  }
  public void setBelegnummer (int belegnummer)
  {
    if (belegnummer > 0)
      this.belegnummer = belegnummer;
  }
}
public class Rechnung extends Dokument
{
}
```



```
public class Gutschrift extends Dokument
{
  //Besser: java.math.BigDecimal, wenn exakte Beträge im Spiel sind
  private float Gutschriftbetrag;

  //getter & setter
}
public class Lieferschein extends Dokument
{
}
```

Listing 2.1 Die abstrakte Dokument-Klasse und deren Ableitungen

Das Fabrikmethode-Muster stellt an diese Klassen keine besonderen Anforderungen. Sie benötigen keine speziellen Methoden oder Felder und auch keine besondere Implementierung. Das ist in der Praxis wichtig, denn diese Vererbungshierarchie ist ja Teil der Geschäftsdomäne, und sie bestimmt auch, wie diese Klassen aussehen sollen.

Erzeuger / Konkreter Erzeuger

Die Erzeugerklasse enthält nun die Fabrikmethode und eine weitere Methode, die ein Objekt vom Typ `Dokument` benötigt:

```
public abstract class Fakturierer
{
  public abstract Dokument erzeugeDokument();

  public void fakturiereDokument()
  {
    Dokument dokument = erzeugeDokument();
    //Arbeite mit Dokument
  }
}
```

Listing 2.2 Die abstrakte Erzeugerklasse »Fakturierer«

Die Fabrikmethode ist abstrakt, sie gibt ein Objekt vom ebenfalls abstrakten Typ `Dokument` zurück. Dennoch enthält sie auch eine Methode, in der sie mit dem erzeugten Objekt etwas Sinnvolles anstellen kann, nämlich die Methode `fakturiereDokument`. Hier wird das Dokument beispielsweise

- berechnet
- gebucht

- gedruckt
- archiviert

Diese »Arbeitermethode« kann in der Praxis natürlich auf mehrere Methoden aufgeteilt sein, von denen alle oder einige virtuell sind, sich also in den abgeleiteten Klassen in ihrem Verhalten überschreiben lassen. Weiten wir dazu die Klasse ein wenig aus, um sie praxisnäher zu gestalten:

```
public abstract class Fakturierer
{
  public abstract Dokument erzeugeDokument();

  public void fakturiereDokument()
  {
    Dokument dokument = erzeugeDokument();
    if (!isValid(dokument))
      throw new IllegalArgumentException(
        "Das zu verarbeitende Dokument ist nicht gültig");
  }

  public boolean isValid(Dokument dokument)
  {
    if (dokument.getBelegnummer() == 0)
      return false;
    return true;
  }
}
```

Listing 2.3 Die um eine virtuelle Methode erweiterte Erzeugerklasse

Besonders praktisch ist das Fabrikmethode-Muster natürlich, wenn ein großer Teil der Geschäftslogik bereits in der Erzeugerklasse enthalten ist und abgeleitete Klassen nur noch einzelne Verhaltensweisen ändern, wie die Validierung in der nun abgeleiteten Klasse `GutschriftFakturierer`:

```
public class GutschriftFakturierer extends Fakturierer
{
  @Override
  public Dokument erzeugeDokument()
  {
    return new Gutschrift();
  }
```



```
  @Override
  public boolean isValid(Dokument dokument)
  {
    return super.isValid(dokument) &&
     ((Gutschrift)dokument).getGutschriftbetrag() > 0;
  }
}
```

Listing 2.4 Die konkrete Erzeugerklasse »GutschriftFakturierer«

Die konkrete Erzeugerklasse kennt nun den Typ des konkret zu erzeugenden Dokuments, im Beispiel die Gutschrift, daher erzeugt nun die überschriebene Fabrikmethode ein Objekt vom Typ `Gutschrift`.

Im Beispiel wurde nicht nur die Fabrikmethode überschrieben, sondern auch die Validierung erweitert, sodass die Basisklasse nur das Vorhandensein einer Belegnummer prüft und die abgeleiteten Erzeugerklassen die Besonderheiten des jeweils zu verarbeitenden Dokuments in die Validierung einbringen.

2.1.5 Weitere Überlegungen und Alternativen

Der Grundgedanke dieses Musters ist es, Spezifika – also zum Beispiel die Details einer Gutschrift – aus dem Framework herauszuhalten, damit die Klasse `Fakturierer` weitgehend unabhängig davon entwickelt werden kann.

Allerdings hängt an diesem Muster ein Preisschild: Es werden zwei Vererbungshierarchien benötigt, eine für die zu erzeugenden Produkte und eine weitere für die Erzeuger, die Hand in Hand entwickelt werden müssen. Ein Client, also der Code, der die Erzeugerklassen verwendet, muss sich darauf einstellen, indem er immer die richtige Erzeugerklasse verwendet. Man könnte argumentieren, das Ableiten der Erzeugerklasse sei eine Doppelung der ohnehin schon vorhandenen Produkthierarchie – und hätte damit recht.

Weitere Hierarchieebenen, noch mehr Erzeuger oder die Alternative: Fabrikmethoden mit Parametern

Besonders deutlich wird das, wenn wir eine oder mehrere weitere Ebenen einziehen, also zum Beispiel die Rechnung weiter unterteilen in:

- Nachnahmerechnung
- Vorausrechnung
- Teilzahlungsrechnung

Andererseits ist es eine Stärke dieses Musters, dass genau das möglich ist – und zwar in vielen Fällen auch noch zu einem späteren Zeitpunkt.

Dennoch: Manchmal ist ein Muster des Musters wegen schlicht zu viel des Guten. Daher sollten Sie grundsätzlich überlegen, die Auswahl des konkreten Produkts nicht allein durch Ableitung zu bewerkstelligen, sondern indem Sie der Fabrikmethode einen Parameter mitgeben. Fügen wir nun die dritte Ebene ein, also die verschiedenen Rechnungstypen:

```
public class Nachnahmerechnung extends Rechnung {
}
public class Vorausrechnung extends Rechnung {
}
public class Teilzahlungsrechnung extends Rechnung {
}

public enum Rechnungstyp
{
  NACHNAHMERECHNUNG,
  VORAUSRECHNUNG,
  TEILZAHLUNGSRECHNUNG
}
```

Der Aufzählungstyp `Rechnungstyp` ist reiner Komfort zur Gewährleistung der Typsicherheit. In vielen realen Projekten findet man einfach einen Integerwert zur Differenzierung der verschiedenen Typen. Wobei viele Sprachen, nicht zuletzt Java, eine Aufzählung letztendlich auch wieder zu einem Integer machen.

```
public class RechnungFakturierer extends Fakturierer
{
  @Override
  public Dokument erzeugeDokument()
  {
    return new Rechnung();
  }

  public Rechnung erzeugeRechnung(Rechnungstyp rechnungstyp)
  {
    switch (rechnungstyp)
    {
      case NACHNAHMERECHNUNG:
        return new Nachnahmerechnung();

      case TEILZAHLUNGSRECHNUNG:
        return new Teilzahlungsrechnung();
```



```
      case VORAUSRECHNUNG:
        return new Vorausrechnung();

      default:
        return new Rechnung();
    }
  }
}
```

Listing 2.5 Fabrikmethode mit Parameter

Die Klasse `Rechnungfakturierer` bietet dieselbe Fabrikmethode wie auch die beiden anderen Fakturierer, damit wird eine »gewöhnliche« Rechnung erzeugt, also ein Objekt der Klasse `Rechnung`.

Zudem enthält sie eine zweite Fabrikmethode, `erzeugeRechnung`, die den Typ der zu erzeugenden Rechnung entgegennimmt und im Code selbst die Fallunterscheidung vornimmt. Gibt es diesen Typ nicht, wird wiederum eine gewöhnliche Rechnung erzeugt.

In diesem Beispiel sieht man auch: Beide Verfahren lassen sich auch in einer Erzeugerhierarchie mischen.

Allerdings könnte man nun zum Extrem greifen und eine einzige, potenziell riesige Fabrikmethode schreiben, die alle Objekte erzeugen kann. Dagegen sprechen einige Dinge:

- Die Erzeugerklasse muss alle Produktklassen kennen, weil sie diese ja alle instanzieren können muss.
- Die Erweiterbarkeit ist eingeschränkt, weil nicht einfach neue Erzeugerklassen der bestehenden Ableitungshierarchie hinzugefügt werden können.
- Im ursprünglichen Ansatz können die konkreten Erzeugerklassen auch in anderen Bibliotheken residieren und somit auch eigenständig ausgeliefert und gepflegt werden. Im neuen Ansatz geht das nicht mehr.
- Die Übersichtlichkeit leidet doch stark und nimmt mit der Anzahl der Produktklassen schnell ab.
- Es geht die Fähigkeit verloren, in den abgeleiteten Erzeugerklassen den Code weiter zu strukturieren, ohne dass die übergeordneten Klassen davon betroffen wären. Beispielsweise könnte die Erzeugung eines Produkts weiter unten im Ableitungsbaum ja viel komplexer sein, und diese Komplexität wäre dann in der großen Fabrikmethode zu finden (oder in einer von ihr aufgerufenen Methode) – wo sie nicht hingehört.

Abstrakte und konkrete Erzeugerklassen und Fabrikmethoden

Das Beispiel von eben mischt zwei verschiedene Arten von Erzeugerklassen:

- Die Basisklasse `Fakturierer` ist abstrakt. Das gilt ebenso für die Fabrikmethode `erzeugeDokument` und die Produktklasse `Dokument`. Jedem Client ist schnell klar: Objekte vom Typ `Dokument` lassen sich nicht erzeugen, und die Compiler verhindern dies zuverlässig.
- Die abgeleitete Klasse `RechnungFakturierer` erzeugt Objekte der Klasse `Rechnung`, aber auch Objekte von davon abgeleiteten Klassen, also Gutschriften, Teilzahlungsrechnungen und Vorausrechnungen. Die Klasse `Rechnung` ist also selbst eine verwendbare Klasse und wiederum Basis für weitere Ableitungen.

Aus der Kombination von Erzeugerklasse und Fabrikmethode ergeben sich drei Möglichkeiten, die alle im Zusammenhang mit dem Entwurfsmuster *Fabrikmethode* anwendbar sind:

- **Abstrakte Erzeugerklasse enthält eine abstrakte Fabrikmethode**: Ableitung notwendig, da sonst kein Objekt erzeugt werden kann.
- **Abstrakte Erzeugerklasse, enthält aber konkrete Fabrikmethode**: Es wird ein »Defaultobjekt« erzeugt.
- **Konkrete Erzeugerklasse und konkrete Fabrikmethode**: Es wird ebenfalls ein »Defaultobjekt« erzeugt.

Die fachliche Aufgabenstellung bestimmt, welche Variante zum Einsatz kommt. Vor allem müssen Sie bei der Entscheidung die Frage betrachten, ob es überhaupt ein »Defaultobjekt« geben kann, das eine Erzeugerklasse ohne Ableitung erzeugen kann.

Erzeugerklasse vs. Produktklasse

In vielen Fällen ist es sinnvoller, die Geschäftslogik nicht in die Erzeugerklasse zu packen, sondern diese lediglich für die Objekterzeugung zu nutzen, weil die Geschäftslogik sicherlich auch außerhalb der Erzeugerklasse verwendet wird. Die Geschäftslogik in die Businessobjekte zu integrieren liegt dann näher. Im Beispiel betrifft das die Validierung, die wir besser so implementieren sollten:

```
public abstract class Dokument
{
  private int belegnummer;

  public int getBelegnummer()
  {
    return belegnummer;
  }
  public void setBelegnummer (int belegnummer)
  {
    if (belegnummer > 0)
      this.belegnummer = belegnummer;
  }

  public boolean isValid()
  {
    return belegnummer > 0;
  }
}

public class Gutschrift extends Dokument
{
  …
  @Override
  public boolean isValid()
  {
    return super.isValid() && (gutschriftbetrag > 0);
  }
}

… (class Fakturierer)
public void fakturiereDokument()
{
  Dokument dokument = erzeugeDokument();
  if (!dokument.isValid())
    throw new IllegalArgumentException(
      "Das zu verarbeitende Dokument ist nicht gültig");
}
```

Listing 2.6 Alternative mit Geschäftslogik (Validierung) in der Produktklasse

Sie sehen schon, dass diese Alternative eleganter ist, weil sie kürzer ist und Ihnen zudem das Casten auf den konkreten Produkttyp erspart. Und was eleganter ist, ist meist auch besser.

Statische Fabrikmethoden

Ich habe es schon erwähnt: Heutige Bibliotheken und Framework wimmeln nur so von Factorys. Häufig sind das allerdings nicht Factorys im Sinne des Erfinders, also so, wie sie hier beschrieben sind, sondern statische Methoden, die ein Objekt erzeugen und zurückgeben.

Als Beispiel kann die `Task`-Klasse aus dem .NET-Framework dienen, stellvertretend für viele weitere Beispiele. Diese Klasse ist eine »gewöhnliche« Klasse, kann also instanziiert werden und stellt dann eine abzuarbeitende Aufgabe dar. Allerdings birgt sie auch eine Besonderheit, und zwar eine Fabrik im Inneren:

```
public static TaskFactory Factory
```

Die Eigenschaft gibt also ein Objekt vom Typ `TaskFactory` zurück, eine Fabrik also. Und diese Fabrik definiert eine statische Methode, `StartNew`, die es zudem noch in vielen Überladungen gibt, zum Beispiel diese:

```
public Task StartNew(Action action, CancellationToken cancellationToken)
```

Erstellt wird also ein Objekt der Klasse `Task`. Damit kann die `Task`-Klasse selbst verwendet werden, um über eine Fabrik ein Objekt von sich selbst zu erzeugen, also:

```
Task myTask = Task.Factory.StartNew(…)
```

In Methoden wie dieser wird also das Objekt erzeugt und manchmal auch vorkonfiguriert und damit als »Defaultobjekt« zurückgegeben. Das ist komfortabel, weil vom Entwickler leicht benutzbar.

In anderen Fällen wird die Factory selbst wie ein gewöhnliches Objekt erstellt, und zwar mithilfe einer der Konstruktoren, die die Fabrik auf einen bestimmten Typ von Objekten einstellen. Die Fabrikmethode erzeugt dann diese immer gleichen Objekte, wenn Sie wollen wie am Fließband. Die `TaskFactory`-Klasse selbst ist dafür ein gutes Beispiel, denn ein Objekt davon lässt sich z. B. so erzeugen:

```
TaskFactory tf =
 new TaskFactory(TaskScheduler.FromCurrentSynchronizationContext());
```

Alle damit erstellten Tasks werden also im aktuellen Synchronisationskontext ausgeführt, was nichts anderes bedeutet, als dass sie im Thread des Aufrufers laufen.

```
Task myTask = Tf.StartNew(…)
```

Sie sehen schon: Mit dem hier beschriebenen Entwurfsmuster *Fabrikmethode* hat das wenig zu tun, schon allein deshalb, weil keine Ableitungen im Spiel sind und auch andere Akteure des Musters fehlen. Dennoch wird es häufig synonym verwendet und kann in der Praxis ebenso gewinnbringend eingesetzt werden wie die »richtige« Fabrikmethode.

Zum Schluss zeige ich Ihnen noch ein Beispiel für die Implementierung:

```
public class MyFactory
{
  private String defaultEigenschaft;
```



```
  public MyObject create(String eineEigenschaft)
  {
    return new MyObject(eineEigenschaft);
  }

  public MyFactory(String defaultEigenschaft)
  {
    this.defaultEigenschaft = defaultEigenschaft;
  }

  public MyFactory()
  {
  }

  public MyObject create()
  {
    return new MyObject(defaultEigenschaft);
  }
}
```

Listing 2.7 Die Klasse »MyFactory«

Diese Klasse `MyFactory` ist nicht statisch, und es gibt einen Konstruktor, der einen Standardparameter für alle neu erzeugten Objekte vom Typ `MyObject` entgegennimmt. Auf diese Weise ließen sich jetzt freilich beliebig viele Objekte erzeugen, die aber alle über den Konstruktor parametrisiert werden müssten. Hier kommt nun die nächste Klasse ins Spiel:

```
public class MyObject
{
  private static MyFactory factory = new MyFactory();
  private String eineEigenschaft;

  public MyObject(String eineEigenschaft)
  {
    factory = new MyFactory();
    this.eineEigenschaft = eineEigenschaft;
  }

  public static MyFactory getFactory()
  {
```

```
    return factory;
  }
}
```

Listing 2.8 Die Klasse »MyObject«

Ihre wichtigste Eigenschaft ist das statische Feld `factory`, die ein Objekt vom Typ `MyFactory` zurückgibt, ohne Standardparameter.

Die Verwendung ist nun höchst komfortabel. Zunächst betrachten wir die Varianten ohne explizite Erstellung der Factory:

```
MyObject parameterlosesObjekt = MyObject.getFactory().create();
MyObject objektMitParameter = MyObject.getFactory().create("einParameter");
```

Man sieht, es lassen sich damit Objekte mit und ohne Parameter erzeugen. Nun folgt die Variante, in der ein Client die Factory selbst erzeugt und mit einem Defaultparameter vorkonfiguriert:

```
MyFactory factory = new MyFactory("eine Default-Eigenschaft");
MyObject objektMitDefaultParameter = factory.Create();
```

Alle Aufrufe von `create()` erzeugen nun Objekte mit dem bei der Erzeugung der Factory angegebenen Parameter.

2.2 Singleton

Das Singleton-Muster hat vielleicht nicht den besten Ruf, was daran liegt, dass in Zeiten von Multicore-Systemen Parallelisierung der Trend ist, während das Singleton-Muster sicherstellt, dass es von einer Klasse jeweils nur genau ein Objekt gibt. Das ist das genaue Gegenteil, denn wenn es nur ein Objekt gibt, kann auch zu einer Zeit nur jeweils ein Thread Änderungen am Objekt vornehmen.

2.2.1 Steckbrief

Deutscher Name: Singleton

Alternativer deutscher Name: Einzelstück (eher selten verwendet)

Englischer Name: Singleton

Gruppe: Erzeugungsmuster

2.2.2 Beschreibung

UML

Das Singleton-Muster zählt zu den einfachen Mustern. Entsprechend übersichtlich fällt das UML-Diagramm in Abbildung 2.3 aus.

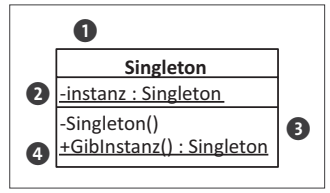

Abbildung 2.3 Das Singleton-Muster in UML

Nr.	Erläuterung
❶	Der Name der Klasse, hier exemplarisch »Singleton« benannt
❷	Der eigentliche Speicher für das Objekt ist ein Klassenattribut. Es ist privat, weil nur die *Singleton*-Klasse selbst es zuweisen darf.
❸	Der private Konstruktor verhindert, dass die *Singleton*-Klasse außerhalb der *Singleton*-Klasse selbst instanziiert werden kann.
❹	Clients greifen auf das *Singleton*-Objekt über diese öffentliche Klassenmethode zu.

Tabelle 2.2 Akteuere des Singleton-Musters

Erläuterungen

Beim Singleton-Muster geht es um Einzigartigkeit, also darum, dass ein Objekt nur einmal im Speicher vorhanden ist. Das ist unter allen Umständen zu gewährleisten, und das technische Mittel, um das zu gewährleisten, sind Klassenmethoden und Klassenattribute.

Daher ist zuerst einmal zu verhindern, dass irgendjemand die Singleton-Klasse selbst instanziiert. Daher wird der Konstruktor kurzerhand als `private` (oder `protected`) deklariert. Und wo kein Konstruktor vorhanden (oder zugreifbar) ist, kann kein Objekt erzeugt werden.

Die *Singleton*-Klasse selbst erhält nun eine Klassenmethode, `GibInstanz`, die das Objekt erzeugt und in der ebenfalls statischen Klassenvariable `instanz` speichert. Sie tut das aber nur beim ersten Mal. Zukünftig wird dann einfach das Objekt über diese Variable zurückgegeben.

2.2.3 Anwendungsfälle

Das Singleton-Muster ist überall dort sinnvoll anzuwenden, wo es naturgemäß nur eine einzige Ressource gibt oder wo wir aus anderen Gründen den Zugriff zentralisieren wollen. Einige Beispiele:

- **Zugriff auf zentrale (Hardware)-Ressourcen:** So kann beispielsweise nur ein Softwaresystem zu einer bestimmten Zeit auf einen installierten Scanner zugreifen.
- **Bereitstellung eines Verteilmechanismus, zum Beispiel eines Druckerpools:** Während die Abarbeitung selbst durch mehrere Objekte stattfinden kann (häufig außerdem multithreaded), ist die Queue selbst als Singleton implementiert.
- **Zur fachlichen Serialisierung:** Denken Sie an ein System zur Faktur von Rechnungen. Dort könnte die Klasse zur Vergabe der Belegnummer als Singleton ausgeführt sein. Damit ist sichergestellt, dass zwei (gleichzeitig) anfragende Clients dennoch zwei aufeinander folgende Belegnummern erhalten.
- **Bei Klassen, deren Instanziierung großen Aufwand kostet:** Nehmen wir eine Konfigurationsklasse, die selbst eine große Konfigurationsdatei lädt. Sie wollen vermutlich nicht, dass bei jedem der häufigen Zugriffe auf die Konfiguration stets ein neues Objekt erzeugt wird, das dann immer wieder diese Datei laden muss.

Wie gesagt: Das Singleton-Muster implementiert diese Anforderungen dadurch, dass es von einer (Singleton-)Klasse immer nur ein konkretes Objekt gibt. Das ist keine bloße Konvention, sondern wird durch die Implementierung technisch sichergestellt. Allerdings muss diese Klasse dann auch folgerichtig einen globalen Zugriff ermöglichen.

In der Praxis gibt es viel mehr zentrale Ressourcen, die jeweils nur von einem Client genutzt werden können, als es den Anschein hat, wenn man die Bibliotheken von Java, .NET, C++ & Co. sichtet. So kann eine Festplatte nur eine einzige Schreib- oder Leseoperation gleichzeitig bedienen, wenn die Anfrage nicht vom internen Cache beantwortet werden kann, und es gibt auch ungleich mehr gleichzeitig laufende Threads, als physikalische Prozessoren (bzw. Kerne) vorhanden sind. Meist werden diese Ressourcen »weiter hinten« im gesamten Prozess fair aufgeteilt. Üblicherweise geschieht dies durch Warteschlangen, in die die Anfragen in ihrer Reihenfolge und Priorität eingereiht werden. Häufig kommen zudem Zeitscheibenverfahren zum Einsatz, wie bei der Zuteilung von Rechenzeit an laufende Threads, um eine Gleichzeitigkeit zu simulieren, wo in Wirklichkeit ein echter »Singleton« am Werke ist.

Wie auch immer: Hier geht es um das Singleton-Muster, und dessen Reichweite ist begrenzt. In .NET und in Java wird ein Singleton üblicherweise über statische Methoden implementiert, und die stellen eine Eindeutigkeit, vereinfacht gesagt, nur in dem aktuell laufenden Prozess sicher.

2.2.4 Implementierung

Die wichtigste Aufgabe des Codes ist sicherzustellen, dass es auch wirklich nur ein Objekt der Klasse gibt. Dafür gibt es in Programmiersprachen das Konzept der Klassenattribute, die sich daher für die Implementierung gut eignen.

Nehmen wir als Beispiel eine Konfigurationsklasse, sodass sich die Konfiguration von beliebigen Stellen im Code lesen und schreiben lässt.

Die Klasse selbst ist öffentlich und bietet Methoden zum Lesen und Schreiben von Konfigurationsdaten, wie wir das auch tun würden, ohne das Singleton-Muster anzuwenden.

```
public class Konfiguration
{
  private HashMap<String, String> keyValuePaare;

  public Konfiguration()
  {
    keyValuePaare = new HashMap<String, String>();
    //Lade die Konfiguration aus einer Datei
  }

  public HashMap<String, String> getWerte()
  {
    return keyValuePaare;
  }

  public String getWert(String key)
  {
    if (keyValuePaare.containsKey(key))
      return keyValuePaare.get(key);
    else
      return null;
  }

  public void setWert(String key, String value)
  {
    if (keyValuePaare.containsKey(key))
      keyValuePaare.replace(key, value);
    else
      keyValuePaare.put(key, value);
    schreibeKonfiguration();

  }
```

```
  private void schreibeKonfiguration()
  {
    //Konfiguration zurück auf Datenträger schreiben
  }
}
```

Listing 2.9 Die Klasse »Konfiguration« noch ohne Singleton-Eigenschaften

Die Implementierung der Konfiguration ist nicht weiter wichtig. In diesem sehr einfachen Beispiel wird die Konfiguration auf sehr einfache Weise in einem assoziativen Speicher verwaltet.

Um nun das Singleton-Muster anzuwenden, sind drei Schritte notwendig:

1. Zunächst müssen Sie verhindern, dass Anwender die Klasse selbst im Code instanziieren. Das geht am einfachsten, indem der Konstruktor `private` wird:

   ```
   private Konfiguration()
   {
     keyValuePaare = new HashMap<String, String>();
     // Lade die Konfiguration aus einer Datei
   }
   ```

2. Als Nächstes wird ein Feld benötigt, in dem das `Konfiguration`-Objekt gespeichert wird. Üblicherweise wird das Objekt `Instanz` (`Instance`) genannt, aber das ist reine Konvention.

   ```
   private static Konfiguration instanz;
   ```

   Man könnte bereits jetzt den statischen Initialisierer verwenden und das Objekt an Ort und Stelle erzeugen:

   ```
   private static Konfiguration _instanz = new Konfiguration();
   ```

   In den meisten Programmiersprachen ist diese Vorgehensweise threadsicher. Oder Sie wählen die gebräuchlichere Vorgehensweise aus dem letzten Schritt.

3. Dieser letzte Schritt besteht darin, einem Client einen Zugriff auf das `Konfiguration`-Objekt zu bieten, denn er kann ja selbst kein Objekt mehr erstellen; der private Konstruktur verhindert dies ja.

   ```
   private static Konfiguration gibInstanz()
   {
     if (instanz == null)
       instanz = new Konfiguration();
     return instanz;
   }
   ```


   Das Objekt wird in diesem Beispiel erzeugt, sobald zum ersten Mal auf die `gibInstanz`-Methode und damit auf das `Konfiguration`-Objekt selbst zugegriffen wird (man nennt das auch *Lazy Loading*). Der Vorteil liegt auf der Hand: Wird auf das Objekt überhaupt nie zugegriffen, sparen wir uns das Erstellen von vornherein.

Ein Client kann ohnehin nur über den Getter `gibInstanz()` Zugriff erhalten. Der Client kann daher in der Verwendung überhaupt keinen Fehler machen.

```
String einWert = Konfiguration.gibInstanz().getWert("einKey");
```

bzw.:

```
Konfiguration.gibInstanz().setWert("einKey", "einKey");
```

2.2.5 Weitere Überlegungen und Alternativen

Das Singleton-Muster hat eine lange Geschichte und zählt daher zu den bekanntesten und am besten verstandenen Mustern überhaupt. Dennoch (oder gerade deshalb) gibt es einige Variationen.

Globale Variablen

In der Praxis sieht man anstelle des Singleton-Musters häufig globale Variablen. Das Singleton-Muster hat allerdings einige Vorteile, denn der Charme des Singletons besteht darin, dass der Zugriff strikt kontrolliert ist, während eine globale Variable von überall aus zugreifbar ist. Außerdem kann das `Konfiguration`-Objekt in einem eigenen Namensraum leben und bequem in ein eigenes Package bzw. eine eigene DLL ausgelagert werden. Und die `Konfiguration`-Klasse wird erst dann instanziiert, wenn sie auch wirklich zum ersten Mal benötigt wird. Nicht umsonst hat es sich inzwischen herumgesprochen, dass globale Variablen, in aller Regel jedenfalls, vermieden werden sollten.

Statische Klasse

Manchmal wird die `Konfiguration`-Klasse auch vollständig zur statischen Klasse umfunktioniert. Allerdings sind Sie dann hinsichtlich der Implementierung eingeschränkt, denn eine gewöhnliche Klasse kann ihren eigenen Zustand einfach viel bequemer in privaten Variablen speichern und auch nichtstatische Methoden für den Client anbieten.

Vererbung

Grundsätzlich verträgt sich das Singleton-Muster auch mit Vererbung, allerdings darf die Sichtbarkeit des Konstruktors dann nicht mehr `private` sein, sondern `protected`, sodass die abgeleitete Klasse auf den Konstruktor der Singleton-Basisklasse

zugreifen kann. Dennoch rate ich davon ab, schon allein deshalb, weil ein Anwender der Singleton-Klasse dies im Regelfall nicht erwarten würde. Um diesen Gedanken explizit auszudrücken, bieten die meisten Sprachen eine Versiegelung der Klasse an. In C# lässt sich eine Klasse so gegen Vererbung absichern:

```
public sealed class Konfiguration
```

In Java geschieht das auf diese Weise:

```
public final class Konfiguration
```

Einzigartig, aber was?

Die Reichweite des Musters ist eingeschränkt. Im .NET-Framework beispielsweise wird die Eindeutigkeit einer statischen Variable lediglich für eine Application Domain gewährleistet. Nehmen wir einmal an, die Klasse zur Erstellung der nächsten Belegnummer wäre nicht mehr lokal vorhanden, sondern in einem Webservice implementiert. Das ist eine realistische Annahme, weil die meisten Systeme, die Belege verarbeiten, natürlich mehrere Benutzer gleichzeitig zulassen.

Weitere Beispiele für Einzigartigkeit sind:

- transaktionsverarbeitende Systeme, die einzelne Transaktionen voneinander isolieren, indem sie diese serialisieren
- Queuing-Systeme, die über einen zentralen Service Elemente entgegennehmen, die sie dann in die Warteschlange einreihen
- Betriebssystem-APIs, die nur von jeweils einem Prozess verwendet werden können

Das geht natürlich über dieses Kapitel hinaus, und jedes der oben genannten Systeme bringt einen eigenen Mechanismus mit, um die Singleton-Eigenschaft sicherzustellen. In WCF, einer Bibliothek zur Erstellung von Services, geht das zum Beispiel über Attribute:

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class MyService:IMyService
```

Als Anwender des Singleton-Musters sollten Sie darauf achten, dass auch wirklich die gesamte Aufrufkette – eventuell auch über den Prozess oder gar über den eigenen Rechner hinaus – das Singleton-Prinzip richtig und vollständig umsetzt.

Threadsicherheit

Machen wir uns nichts vor: Threadsicherheit verkompliziert die Dinge. Im Falle des Singleton-Musters sind natürlich die Methoden selbst threadsicher zu gestalten, wie bei jeder anderen Klasse auch. Außerdem sollte die Erzeugung des Objekts selbst nur

von einem Thread aus erfolgen. Da dies allerdings nur beim ersten Zugriff geschieht ist, das Problem häufig nicht sehr groß.

Die genaue Implementierung (will man es denn perfekt machen) hängt von der verwendeten Sprache und manchmal sogar von deren spezifischer Version ab.

Für Java bieten sich gleich fünf Lösungsmöglichkeiten an:

- Möglichkeit 1 besteht darin, den ersten Zugriff kontrolliert von einem Thread durchzuführen, was natürlich die Vorteile des Lazy Loading zunichtemacht. Für wenig aufwendige Objekte, auf die aber sehr häufig zugegriffen wird, ist das eine absolut akzeptable Vorgehensweise.
- Wie schon oben erwähnt, kann die Initialisierung des Singleton-Objekts auch schon über einen statischen Initialisierer stattfinden:

  ```
  private static Konfiguration instanz = new Konfiguration();
  ```

  Die garantiert threadsichere Initialisierung wird dann beim Laden der Klasse durch die JVM ausgeführt.
- Eine sehr einfache Möglichkeit wäre es auch, die Methode zum Abruf und zur Erzeugung der Instanz, `gibInstanz()`, als `synchronized` zu markieren:

  ```
  public static synchronized Konfiguration gibInstanz()
  {
    if (instanz == null)
      instanz = new Konfiguration();
    return instanz;
  }
  ```

  Der Schönheitsfehler dabei ist, dass ja das Objekt nur einmal erzeugt werden muss, der Zugriff darauf – über dieselbe Methode – aber sehr oft geschieht und somit auch die lesenden Zugriffe auf das Singleton-Objekt synchronisiert werden.
- Die vierte Möglichkeit sperrt daher nur das Erzeugen des Singleton vor dem gleichzeitigen Zugriff mehrerer Threads, durch »doppelt überprüfte Sperrung«:

  ```
  private volatile static Konfiguration instanz;
  …
  public static synchronized Konfiguration gibInstanz()
  {
    if (instanz == null)
    {
      synchronized (Konfiguration.class)
      {
        if (instanz == null)
          instanz = new Konfiguration();
      }
  ```

```
  }
  return instanz;
}
```

Hier wird der synchronisierte Code auch wirklich nur noch dann ausgeführt, wenn das Singleton-Objekt noch nicht initialisiert wurde.

- Die fünfte Lösung ist ein wenig exotischer, aber spannend. Sie funktioniert mit Enums:

```
public enum Konfiguration
{
  INSTANCE;
… Methoden und andere Member
```

Die anderen Methoden und Member übernehmen Sie einfach von oben, außer natürlich der `gibInstanz()`-Methode, die hier nicht mehr erforderlich ist. Der Zugriff ist einfach und elegant über das Enum-Member möglich, die Java VM garantiert die threadsichere Initialisierung des Singletons.

```
String einWert = Konfiguration.INSTANCE.getWert("einKey");
```

Auch bei dieser Variante wird das Objekt erst beim ersten Zugriff erstellt. Der Nachteil könnte natürlich sein, dass Sie von Ihren Kollegen als Nerd angesehen werden.

Multiton

Auch wenn das Singleton-Muster die Einzahl schon im Namen trägt: Grundsätzlich kann das Muster auch verwendet werden, um eine endliche Zahl von Objekten zu erzeugen. Das entsprechende Muster heißt dann folgerichtig *Multiton*, und Sie finden dessen Beschreibung im nächsten Abschnitt.

Gefahren

Singletons sind eine prima Sache, wenn man sie denn sparsam einsetzt und nicht exzessiv und aus reiner Bequemlichkeit. Auch wenn Singletons meist besser sind als globale Variablen, teilen sie dennoch einige ihrer Schwachstellen. Ein Client, der sein eigenes Objekt besitzt, kann auch dessen Lebensdauer kontrollieren, während die Lebensdauer des Singletons vom Laufzeitsystem kontrolliert wird. Außerdem wird die Testbarkeit erschwert, weil sich das Objekt nicht so ohne Weiteres z. B. durch ein Mock-Objekt ersetzen lässt. Und es besteht auch immer die Gefahr, dass durch das Singleton Ausführungsstränge serialisiert werden, die eigentlich parallel ausgeführt werden könnten.