mitp

Erich Gamma

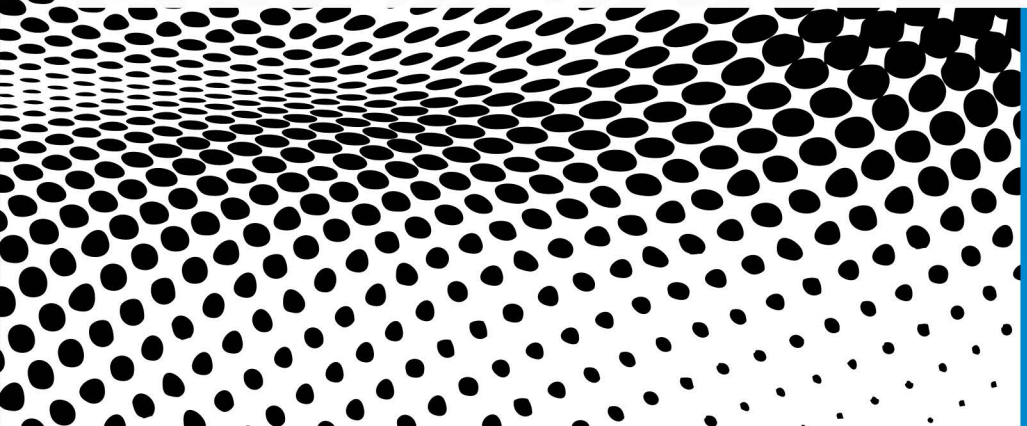
Richard Helm

Ralph Johnson

John Vlissides

Design Patterns

Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software



Kapitel 1

Einführung

Das Entwickeln bzw. Entwerfen objektorientierter Software ist zweifellos kein leichtes Unterfangen – und das Designen *wiederverwendbarer* objektorientierter Software gestaltet sich sogar noch anspruchsvoller und komplexer: Neben der Bestimmung der relevanten Objekte und deren Abstrahierung zu Klassen in geeigneter Granularität müssen außerdem passende Schnittstellen und Vererbungshierarchien definiert sowie die zentralen Beziehungen zwischen den einzelnen Klassen bestimmt werden. Darüber hinaus sollte sich das Softwaredesign natürlich einerseits speziell an den jeweiligen Erfordernissen orientieren, andererseits aber gleichermaßen eine hinreichende Universalität aufweisen, um auch für zukünftige Problemstellungen und Anforderungen gewappnet zu sein. Und Designrevisionen sollten nach Möglichkeit ebenfalls vermieden oder zumindest auf ein Minimum reduziert werden.

Erfahrene objektorientierte Programmierer sind sich im Allgemeinen darüber im Klaren, dass es sehr schwierig, wenn nicht gar unmöglich ist, ein wiederverwendbares, flexibles Design gleich von Anfang an »richtig« hinzubekommen. Aus diesem Grund greifen sie häufig mehrfach auf ihre früheren Entwürfe zurück und versuchen zunächst, weitere Modifikationen zu implementieren, bevor sie ein Softwaredesign endgültig als »abgeschlossen« betrachten.

Das Erlernen und Verinnerlichen der maßgeblichen Aspekte, die ein gutes objektorientiertes Design ausmachen, erfordert viel Zeit und Geduld. Folglich fällt es erfahrenen objektorientierten Programmierern in der Regel leichter, gute Designs zu entwerfen, als weniger geübten Entwicklern, die sich nicht selten von den zahlreichen Designoptionen überfordert fühlen und daher oft doch lieber die nicht objektorientierten Techniken anwenden, die sie früher schon eingesetzt haben. Erfahrenen Programmierern ist also offenbar spezielles Wissen zu eigen, über das unerfahrene Entwicklern (noch) nicht verfügen. Doch worum genau handelt es sich dabei?

Nun, versierten Softwaredesignern ist beispielsweise bewusst, dass sie *nicht* ständig versuchen müssen, für jedes Problem eine völlig neue Lösung zu entwickeln. Stattdessen machen sie sich Lösungsmöglichkeiten zunutze, die sich nach ihrer persönlichen Erfahrung bereits bewährt haben – mit anderen Worten: Haben sie erst einmal gute Lösungsansätze gefunden, dann »recyceln« sie diese und wenden sie immer wieder an. Dieser Erfahrungsvorsprung zeichnet die Experten im Bereich des Softwaredesigns aus – und erklärt auch, warum viele objektorientierte

Systeme häufig wiederkehrende Klassenmuster und kommunizierende Objekte enthalten. Solche Muster (engl. *Patterns*) beheben bestimmte Designprobleme und gestalten objektorientierte Softwareentwürfe nicht nur flexibler und eleganter, sondern gestatten letztlich überhaupt erst ihre Wiederverwendbarkeit. Sie ermöglichen den Entwicklern, neue Designs auf gelungenen Entwürfen aufzusetzen und darauf aufzubauen. Kurz gesagt: Softwaredesigner, die mit Patterns vertraut sind, können diese unmittelbar auf die jeweils vorliegenden Problemstellungen anwenden, ohne erst »das Rad neu erfinden« zu müssen.

Ein vergleichbares Prinzip findet auch in anderen kreativen Prozessen Anwendung. So erarbeiten beispielsweise Roman- und Bühnenautoren die Handlungen ihrer Geschichten und Theaterstücke nur höchst selten von Grund auf – vielmehr bedienen sie sich in den meisten Fällen ebenfalls eines bewährten »Musters« bzw. Leitmotivs wie etwa »tragische Heldenfigur« (Macbeth, Hamlet etc.) oder »romantische Erzählung« (zahllose Liebesromane). In ähnlicher Weise setzen auch die objektorientierten Programmierer Entwurfsmuster (engl. *Design Patterns*) wie etwa »Zustände werden durch Objekte repräsentiert« oder »Objekte werden zwecks einfacher Ergänzung/Entfernung von Eigenschaften dekoriert« ein. Und steht das Muster erst einmal fest, ergeben sich viele weitere Designentscheidungen ganz automatisch.

Als Entwickler wissen wir alle den Wert der praktischen Designerfahrung zu schätzen. Haben Sie beim Betrachten eines Softwaredesigns nicht auch schon mal ein *Déjà-vu*-Erlebnis gehabt – das Gefühl, ein bestimmtes Problem in der Vergangenheit bereits bewältigt zu haben, ohne dass Sie sich daran erinnern könnten, wie genau oder in welchem Zusammenhang? Wenn Ihnen die exakte Problemstellung bzw. der seinerzeit eingeschlagene Lösungsweg wieder einfallen würde, könnten Sie auf diese Erfahrung zurückgreifen, statt ganz von vorn anfangen zu müssen. Leider werden die im Laufe eines objektorientierten Designprozesses gewonnenen Erkenntnisse in der Regel jedoch nicht so umfassend protokolliert, dass auch Dritte davon profitieren könnten.

Deshalb lautet die Zielsetzung dieses Buches, besagten Erkenntnisgewinn anhand von sogenannten **Design Patterns** (auch **Entwurfsmuster** oder kurz **Patterns** genannt) zu dokumentieren, damit er für jedermann zugänglich und effizient nutzbar wird. Zu diesem Zweck haben wir die wichtigsten Patterns in Katalogform zusammengefasst, wobei jedes Muster systematisch je ein bedeutsames wiederkehrendes Design benennt, beschreibt und evaluiert.

Design Patterns erleichtern nicht nur die Wiederverwendung erfolgreicher Softwaredesigns und -architekturen, sondern bieten Entwicklern zudem auch einen besseren, ungehinderten Zugang zu bewährten Techniken. Außerdem dienen sie als Entscheidungshilfe bei der Wahl möglicher Designalternativen zwecks Gewährleistung der Wiederverwendbarkeit eines Systems und verhindern gleichzeitig Alternativen, die einer Wiederverwendung entgegenstehen. Im Übrigen können Patterns ebenfalls dazu beitragen, die Dokumentation und Wartung bereits existenter Systeme zu verbessern, indem sie explizite Spezifikationen für

die Klassen- und Objektinteraktionen sowie deren Intention bereitstellen. Unterm Strich heißt das: Design Patterns unterstützen Programmierer dabei, ihre Softwaredesigns schneller »richtig« hinzubekommen.

Keins der in diesem Buch beschriebenen Patterns ist vollkommen neu oder unerprobt. Im Gegenteil: Es werden ausschließlich solche Design Patterns verwendet, die schon mehrfach in verschiedenen Systemen eingesetzt wurden. Die meisten von ihnen wurden allerdings noch nie zuvor dokumentiert, sondern entstammen entweder dem gemeinschaftlichen Fundus der objektorientierten Community oder erfolgreichen objektorientierten Systemen – also zwei Bereichen, die sich unerfahrenen Programmierern nicht so leicht erschließen. Aber auch wenn die vorgestellten Patterns nicht brandneu sind, werden sie doch in einer neuartigen, leicht verständlichen Art und Weise vorgestellt: in Form eines Design-Pattern-Katalogs, der einer einheitlichen Präsentationskonvention folgt.

Aufgrund des naturgemäß begrenzten Platzangebots in diesem Buch repräsentieren die hier erläuterten Muster lediglich einen Bruchteil des Pattern-Fundus, der echten Programmierexperten zur Verfügung steht. So werden beispielsweise keine Design Patterns für die nebenläufige, die verteilte oder die Echtzeitprogrammierung oder für spezifische Anwendungsdomänen verwendet. Ebenso wenig werden Sie in diesem Buch Informationen zur Entwicklung von Benutzeroberflächen, zum Schreiben von Gerätetreibern oder zum Einsatz objektorientierter Datenbanken vorfinden. Für all diese Aufgabenbereiche existieren wiederum spezielle Design Patterns – die es jedoch sicherlich ebenfalls wert wären, eigens katalogisiert zu werden.

1.1 Was ist ein Design Pattern?

Der US-amerikanische Architekturtheoretiker Christopher Alexander erklärt in seinem Buch *»Eine Muster-Sprache«* [Löcker Verlag, Wien, 1995, Seite *x*]: »Jedes Muster beschreibt zunächst ein in unserer Umwelt immer wieder auftretendes Problem, beschreibt dann den Kern der Lösung dieses Problems, und zwar so daß man diese Lösung millionenfach anwenden kann, ohne sich je zu wiederholen.« Natürlich bezieht sich Alexander in diesem Fall auf Muster, die sich in Gebäuden und Stadtplanungsentwürfen wiederfinden, dennoch trifft seine Definition auch auf objektorientierte Design Patterns zu – mit dem Unterschied, dass die Lösungen hier nicht in Form von Wänden und Türen, sondern von Objekten und Schnittstellen ausgedrückt werden. Prinzipiell repräsentieren jedoch beide Musterspezies Problemlösungen für bestimmte Situationen in bestimmten Kontexten.

Ein Design Pattern umfasst im Wesentlichen vier maßgebliche Elemente:

1. Der kurze, meist aus ein oder zwei Wörtern bestehende **Pattern-Name** dient zur Referenzierung des jeweiligen Designproblems sowie der zugehörigen Lösungen und deren Auswirkungen. Durch die Benennung eines Design Patterns erweitern wir unser designbezogenes Vokabular in der Art, dass ein

Arbeiten auf einer höheren Abstraktionsebene möglich wird – denn dieses Vokabular erleichtert die Dokumentation des Softwaredesigns und vor allem auch die Kommunikation mit Kollegen und Dritten, z. B. zur Erörterung der Vor- und Nachteile neuer Ideen und Vorschläge, in erheblichem Maße. Wirklich geeignete Namen zu finden, ist allerdings keine leichte Aufgabe – das zeigt auch die Tatsache, dass die Vergabe passender Bezeichnungen (im Englischen wie im Deutschen) für die in diesem Buch vorgestellten Design Patterns bei der Entwicklung unseres Katalogs eine der größten Herausforderungen überhaupt darstellte.

2. Die **Problemstellung** beschreibt die Situation, in der das Design Pattern anzuwenden ist. Sie erläutert die jeweils vorliegende Problematik sowie deren Kontext. Dabei kann es sich um spezifische Designprobleme handeln, wie z. B. in welcher Form Algorithmen als Objekte abgebildet werden sollten, aber auch um für unflexible Designs symptomatische Klassen- oder Objektstrukturen. Mitunter werden im Rahmen der Problembeschreibung außerdem bestimmte Bedingungen aufgeführt, die erfüllt sein müssen, damit der Einsatz des Design Patterns überhaupt sinnvoll ist.
3. Die **Lösung** berücksichtigt neben den konkreten designbildenden Elementen auch deren Beziehungen zueinander, ihre Zuständigkeiten sowie ihre Interaktionen. Sie definiert allerdings kein bestimmtes Design oder eine konkrete Implementierung, denn ein Pattern entspricht eher einer Art Schablone, die in vielen verschiedenen Situationen anwendbar ist: Es skizziert eine abstrakte Beschreibung eines Designproblems und zeigt auf, wie dieses durch eine generelle Anordnung von Elementen (in unserem Fall Klassen und Objekten) bewältigt werden kann.
4. Die **Konsequenzen** bilden eine Übersicht der Auswirkungen und Kompromisse ab, die sich aus der Anwendung des Patterns ergeben. Leider bleiben sie, obwohl sie für die Bewertung möglicher Designalternativen und das Abwägen ihrer Vor- und Nachteile von entscheidender Bedeutung sind, in der Argumentation für eine Designentscheidung häufig unerwähnt.

   In der Softwareentwicklung stehen die Konsequenzen eines Pattern-Einsatzes oftmals in direktem Zusammenhang mit dem Speicherplatzbedarf und den Ausführungszeiten, sie können aber ebenso gut auch Sprach- und andere Implementierungsaspekte betreffen. Und da die Wiederverwendbarkeit in der objektorientierten Programmierung eine wichtige Rolle spielt, schließt dies selbstverständlich auch den Einfluss des infrage stehenden Design Patterns auf die Flexibilität, Erweiterbarkeit und Portabilität des Systems mit ein. Generell gilt also: Die ausdrückliche, sachliche Erwägung der zu erwartenden Konsequenzen ist für die lückenlose Evaluierung eines Patterns unverzichtbar.

Die Interpretation des tatsächlichen Nutzwertes eines Design Patterns hängt im Endeffekt immer auch von der Perspektive des Betrachters ab: Während ein Pat-

tern für den einen Entwickler besonders hilfreich erscheinen mag, kann es für einen anderen Programmierer bloß einen primitiven Baustein darstellen. Bei der Zusammenstellung des Design-Pattern-Katalogs haben wir uns vordergründig auf eine bestimmte Abstraktionsebene konzentriert. Die Patterns verkörpern keine Entwürfe für verkettete Listen oder Hashtabellen, die als separate Klassen programmiert und dann im Ist-Zustand wiederverwendet werden können. Ebenso wenig stellen sie komplexe, domänenspezifische Designs für eine komplette Anwendung oder ein Teilsystem dar. Vielmehr handelt es sich bei den hier beschriebenen Design Patterns um *Darstellungen kommunizierender Objekte und Klassen, die auf die Lösung eines allgemeinen Designproblems in einem speziellen Kontext zugeschnitten sind.*

Ein Design Pattern benennt, abstrahiert und identifiziert die Schlüsselaspekte einer allgemeinen Designstruktur, durch die es sich für die Entwicklung eines wiederverwendbaren objektorientierten Designs auszeichnet. Es identifiziert die beteiligten Klassen und Instanzen, deren Rollen und Interaktionen sowie die ihnen zugedachte Aufgabenverteilung. Jedes Pattern eignet sich für eine ganz bestimmte objektorientierte Designproblematik und besitzt spezielle Merkmale, die Aufschluss darüber geben, wann seine Anwendung zweckmäßig ist, ob es trotz möglicher anderer designbedingten Einschränkungen überhaupt eingesetzt werden kann und welche Konsequenzen und Vor- und Nachteile es mit sich bringt. Und weil die Design Patterns schlussendlich natürlich implementiert werden müssen, ist zu jedem von ihnen auch ein Codebeispiel in C++ und (mitunter) Smalltalk angegeben, das eine mögliche Implementierungsvariante aufzeigt.

Auch wenn Patterns prinzipiell objektorientierte Designs beschreiben, basieren sie dennoch auf praktischen Lösungen, die in etablierten objektorientierten Programmiersprachen wie Smalltalk und C++ statt in prozeduralen (Pascal, C, Ada) oder dynamischen objektorientierten Sprachen (CLOS, Dylan, Self) implementiert sind. In diesem Buch haben wir uns aus rein pragmatischen Gründen für die Verwendung von Smalltalk und C++ entschieden, weil wir selbst tagtäglich mit diesen Programmiersprachen arbeiten.

Die Wahl der Programmiersprache spielt insofern eine gewichtige Rolle, als sie den eigenen Blickwinkel beeinflusst. Die in diesem Buch vorgestellten Patterns setzen Sprachfunktionen auf Smalltalk- bzw. C++-Niveau voraus, die somit auch bestimmen, was leicht bzw. was weniger leicht implementiert werden kann. Würden dagegen prozedurale Sprachen zugrunde gelegt, müsste der Katalog ggf. um weitere Design Patterns, z. B. der Art »Inheritance« (Vererbung), »Encapsulation« (Kapselung) oder »Polymorphism« (Polymorphie), ergänzt werden. Außerdem gilt für manche der weniger populären objektorientierten Sprachen, dass diese die Funktionalität einiger unserer Patterns bereits von Haus aus unterstützen. So sind in CLOS beispielsweise Multimethoden verfügbar, die den Bedarf für ein Design Pattern wie *Visitor* (*Besucher*, siehe Abschnitt 5.11) verringern. Aber auch Smalltalk

und C++ unterscheiden sich in vielerlei Hinsicht so weit voneinander, dass sich manche Design Patterns in der einen Sprache einfacher ausdrücken lassen als in der anderen (siehe zum Beispiel *Iterator* (*Iterator*, Abschnitt 5.4)).

1.2 Design Patterns im Smalltalk MVC

In Smalltalk-80 wird zur Entwicklung von Benutzeroberflächen das aus den drei Komponenten *Model* (Datenmodell), *View* (Präsentation) und *Controller* (Programmsteuerung) bestehende *MVC-Architekturschema* eingesetzt. Zur Verdeutlichung des Konzepts der »Patterns« sollen die in MVC verwendeten Design Patterns an dieser Stelle einmal etwas eingehender betrachtet werden.

Das MVC-Architekturschema besteht aus drei Objektarten bzw. -schichten: Das `Model`-Objekt entspricht dem Anwendungsobjekt, das `View`-Objekt repräsentiert dessen Bildschirmdarstellung und das `Controller`-Objekt steuert die Reaktionen der Benutzeroberfläche auf userseitige Eingaben. Vor der Einführung des MVC-Schemas wurden diese Objekte in Benutzeroberflächendesigns tendenziell einfach in einem einzigen Objekt zusammengefasst. In MVC werden sie dagegen separat verwendet, wodurch eine bessere Flexibilität und Wiederverwendbarkeit gewährleistet ist.

Das MVC-Architekturmuster nutzt ein *Subscribe/Notify-Benachrichtigungsprotokoll*, um die `View`- und `Model`-Objekte zu entkoppeln und sicherzustellen, dass die `View`-Objekte stets den aktuellen Zustand ihres zugehörigen `Model`-Objekts abbilden. Zu diesem Zweck werden Änderungen an den Daten im `Model`-Objekt an die jeweils abhängigen `View`-Objekte kommuniziert, damit sie sich entsprechend aktualisieren können. Dieser Ansatz ermöglicht die Anbindung mehrerer `View`-Objekte an ein `Model`-Objekt, um verschiedene Präsentationen anzubieten, und gestattet zudem die Erzeugung neuer `View`-Objekte, ohne das `Model`-Objekt umschreiben zu müssen.

Abbildung 1.1 zeigt ein `Model`-Objekt mit drei zugehörigen `View`-Objekten. (Der Einfachheit halber wurden die `Controller`-Objekte in diesem Beispiel weggelassen.) Das `Model`-Objekt enthält einige Daten, die mittels der `View`-Objekte in verschiedenen Ausgabeformaten wiedergegeben werden: als Tabelle, als Histogramm und als Tortendiagramm. Im Fall einer Datenänderung benachrichtigt das `Model`-Objekt die `View`-Objekte entsprechend, woraufhin diese auf die geänderten Daten zugreifen und die nötigen Anpassungen vornehmen.

Oberflächlich betrachtet, demonstriert dieses Pattern-Beispiel lediglich ein Design, bei dem die `View`-Objekte und das `Model`-Objekt entkoppelt werden. Auf den zweiten Blick lässt es jedoch ein weitaus umfassenderes Konzept erkennen: die Separierung von Objekten in der Form, dass sich Änderungen an einem Objekt auf eine beliebige Anzahl anderer Objekte auswirken können, ohne dass das geänderte Objekt die genaue Beschaffenheit der anderen Objekte berücksich-

tigen muss. Dieses allgemeinere Design wird durch das Design Pattern *Observer* (*Beobachter*, siehe Abschnitt 5.7) beschrieben.

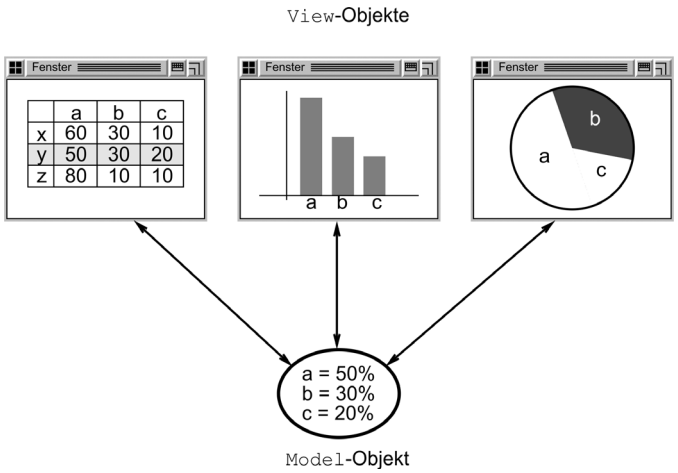


Abb. 1.1: MVC-Design-Pattern-Beispiel

Ein weiteres Merkmal des MVC-Architekturschemas ist außerdem die Schachtelung von `View`-Objekten. So könnte beispielsweise ein mit Buttons bestückter Dialog als ein komplexes `View`-Objekt implementiert werden, das wiederum aus geschachtelten Button-`View`-Objekten besteht. Oder die Oberfläche eines Objektinspektors könnte aus geschachtelten `View`-Objekten bestehen, die sich im Debugger wiederverwenden lassen. Das MVC-Architekturschema unterstützt geschachtelte `View`-Objekte durch die Klasse `CompositeView`, eine Unterklasse von `View`. `CompositeView`-Objekte verhalten sich genauso wie `View`-Objekte und können überall dort verwendet werden, wo auch Views einsetzbar sind – darüber hinaus enthalten und verwalten sie aber auch geschachtelte `View`-Objekte.

Nun könnte man sich darunter einfach ein Design vorstellen, das es ermöglicht, einen `CompositeView` genauso zu behandeln wie eine seiner Komponenten. Dieses Design begegnet jedoch zusätzlich noch einer allgemeineren Problematik, die eintritt, wenn man Objekte zu einer Gruppe zusammenfassen und diese Gruppe dann wie ein individuelles Objekt behandeln möchte. Ein derartiges universeller einsetzbares Design wird durch das Design Pattern *Composite* (*Kompositum*, siehe Abschnitt 4.3) beschrieben: Es ermöglicht die Erstellung einer Klassenhierarchie, in der einige Unterklassen primitive Objekte (wie z. B. Schaltflächen) definieren und andere Klassen wiederum die `Composite`-Objekte (`CompositeView`), die die primitiven Objekte zu komplexeren Objekten zusammenfügen.

Zudem gestattet das MVC-Architekturmuster auch die Steuerung der Reaktion eines `View`-Objekts auf eine Benutzereingabe, ohne dessen visuelle Präsentation zu modifizieren. So ließe sich beispielsweise die Reaktion des `View`-Objekts auf

Tastatureingaben verändern oder statt Tastenbefehlen ein entsprechendes Popup-Menü nutzen. MVC kapselt den Reaktionsmechanismus in sogenannten `Controller`-Objekten, die einer Klassenhierarchie unterliegen, welche die Erstellung neuer Variationen eines bereits vorhandenen `Controller`-Objekts erleichtert.

Zur Implementierung einer bestimmten Reaktionsstrategie nutzen `View`-Objekte eine Instanz einer `Controller`-Unterklasse. Soll also eine andere Strategie implementiert werden, wird die Instanz einfach durch ein anderes `Controller`-Objekt ersetzt. Es ist sogar möglich, den Controller eines `View`-Objekts zur Laufzeit zu ändern, um die Reaktion des Views auf User-Eingaben zu modifizieren. Wollte man beispielsweise erreichen, dass ein `View`-Objekt nicht mehr auf Eingaben reagiert, sprich deaktiviert wird, müsste man ihm lediglich ein `Controller`-Objekt zuweisen, das Eingabeereignisse ignoriert.

Die Beziehung zwischen `View`- und `Controller`-Objekt ist ein Beispiel für das Design Pattern *Strategy* (*Strategie*, siehe Abschnitt 5.9). Ein Strategieobjekt ist ein Objekt, das einen Algorithmus repräsentiert. Es ist insbesondere dann nützlich, wenn ein Algorithmus strategisch oder dynamisch ersetzt werden soll, zahlreiche Varianten des Algorithmus vorliegen oder der Algorithmus komplexe Datenstrukturen enthält, die gekapselt werden sollen.

Darüber hinaus setzt das MVC-Architekturschema auch andere Design Patterns ein, wie z. B. *Factory Method* (*Fabrikmethode*, siehe Abschnitt 3.3) zur Spezifikation der Standard-`Controller`-Klasse eines `View`-Objekts oder auch *Decorator* (*Dekorierer*, siehe Abschnitt 4.4), um ein `View`-Objekt mit Scrollfähigkeit auszustatten – die wichtigsten Beziehungen im MVC-Architekturschema werden im Allgemeinen jedoch durch die Design Patterns *Observer* (*Beobachter*, siehe Abschnitt 5.7), *Composite* (*Kompositum*, siehe Abschnitt 4.3) und *Strategy* (*Strategie*, siehe Abschnitt 5.9) definiert.

1.3 Beschreibung der Design Patterns

Wie lassen sich Design Patterns beschreiben? Grafische Notationen, die zwar ebenso wichtig wie hilfreich sind, reichen hier nicht aus, denn sie erfassen das Endprodukt des Designprozesses lediglich als Beziehungen zwischen Klassen und Objekten. Zur Gewährleistung der Wiederverwendbarkeit des Softwaredesigns müssen jedoch auch die Entscheidungen, Alternativen und Erwägungen, die letztlich zu ihm geführt haben, aufgezeichnet werden. Hilfreich sind in diesem Zusammenhang außerdem konkrete Beispiele, die die Arbeitsweise des Designs verdeutlichen.

Die Beschreibung der Design Patterns erfolgt in diesem Buch nach einem einheitlichen Schema, das jedes Pattern in bestimmte Eigenschaften gliedert. Auf diese Weise ist eine gleichmäßige Informationsstruktur gewährleistet, die das Erlernen, Vergleichen und Anwenden der Design Patterns erleichtert.

Pattern-Name und -Klassifizierung

Die Bezeichnung des Design Patterns charakterisiert zugleich dessen Hauptfunktion bzw. -eigenschaft. Ein aussagekräftiger, prägnanter Name spielt für den täglichen Sprachgebrauch in der Designarbeit eine ganz entscheidende Rolle, um eindeutigen Bezug auf das Pattern nehmen zu können. Die Klassifizierung des Design Patterns erfolgt nach dem in Abschnitt 1.5 beschriebenen System.

Zweck

Die Zweckbeschreibung des Patterns fasst die Antworten auf folgende Fragen in knapper Form zusammen: Was bewirkt das Design Pattern? Welchem Grundprinzip folgt es und welche Intention steht dahinter? Auf welche spezifischen Designfragestellungen oder -probleme ist es ausgerichtet?

Auch bekannt als

Hier sind weitere Bezeichnungen angegeben, unter denen das Pattern ggf. auch bekannt ist.

Motivation

Zum besseren Verständnis der im weiteren Verlauf dieses Buches beschriebenen abstrakteren Eigenschaften des Design Patterns wird je ein Beispielszenario für eine zugrunde liegende Designproblematik angeführt, das aufzeigt, inwiefern die Klassen- und Objektstrukturen des Patterns das Problem lösen.

Anwendbarkeit

In welchen Situationen lässt sich das Design Pattern nutzbringend einsetzen? Welche Designprobleme lassen sich damit beheben? Wie erkennt man solche Situationen?

Struktur

Die grafische Darstellung der in dem Design Pattern enthaltenen Klassen basiert auf der *Object-Modeling Technique* (OMT)-Notation [RBP+91]. Darüber hinaus werden zur Veranschaulichung von Abfrage- und Interaktionsabfolgen zwischen Objekten auch diverse Interaktionsdiagramme [JCJO92, Boo94] dargestellt. Diese Notationen sind in Anhang B beschrieben.

Teilnehmer

Hier werden die an einem Design Pattern beteiligten Klassen und/oder Objekte sowie deren Zuständigkeiten erläutert.

Interaktionen

An dieser Stelle wird beschrieben, in welcher Weise die Teilnehmer zwecks Ausübung ihrer Zuständigkeiten interagieren.

Konsequenzen

In welcher Weise wird die Zielsetzung des Design Patterns unterstützt? Welche Vor- und Nachteile bzw. Resultate ergeben sich aus der Anwendung des Patterns? Welche Bereiche der Systemstruktur lassen sich unabhängig voneinander variieren?

Implementierung

Welche Stolperfallen, Tipps oder Techniken sollten bei der Implementierung des Design Patterns beachtet werden? Könnten sprachspezifische Probleme auftreten?

Codebeispiele

Die Codefragmente demonstrieren, wie sich das Design Pattern in C++ oder Smalltalk implementieren lässt.

Praxisbeispiele

Zur Demonstration des praktischen Einsatzes des Design Patterns in realen Systemen werden mindestens zwei Beispiele aus unterschiedlichen Anwendungsbereichen angeführt.

Verwandte Patterns

Welche anderen Design Patterns stehen in einem engeren Bezug zu dem aktuellen Pattern? Worin bestehen ihre wichtigsten Unterschiede? Welche Patterns lassen sich mit dem aktuellen Pattern kombinieren?

Weiterführende Hintergrundinformationen, die zum besseren Verständnis der Design Patterns beitragen, finden Sie in den Anhängen dieses Buches: Anhang A hält ein Glossar mit der gängigen Designterminologie bereit. Im bereits erwähnten Anhang B werden die verschiedenen Notationen erläutert, deren Kernaspekte in den nachfolgenden Kapiteln beschrieben werden. Und im Anhang C finden Sie den Quelltext für die Basisklassen, die in den Codebeispielen verwendet werden.

1.4 Der Design-Patterns-Katalog

Der in diesem Buch präsentierte Katalog umfasst insgesamt 23 Design Patterns, die im Folgenden in einer Kurzübersicht unter Angabe ihrer jeweiligen Namen

sowie ihrer Zweckbestimmung vorgestellt werden. Eine ausführliche Beschreibung der einzelnen Patterns folgt ab Kapitel 3.

Hinweis

Hinter dem englischen Pattern-Namen ist stets auch die deutsche Entsprechung sowie das Kapitel bzw. der Abschnitt angegeben, in dem es detailliert besprochen wird. Diese Konvention wird im gesamten Buch durchgängig eingehalten.

***Abstract Factory* (*Abstrakte Fabrik*, siehe Abschnitt 3.1)**

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

***Adapter* (*Adapter*, siehe Abschnitt 4.1)**

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter (Adapter)* ermöglicht die Zusammenarbeit von Klassen, die ansonsten aufgrund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

***Bridge* (*Brücke*, siehe Abschnitt 4.2)**

Entkopplung einer Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

***Builder* (*Erbauer*, siehe Abschnitt 3.2)**

Getrennte Handhabung der Erzeugungs- und Darstellungsmechanismen komplexer Objekte zwecks Generierung verschiedener Repräsentationen in einem einzigen Erzeugungsprozess.

***Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1)**

Vermeidung der Kopplung eines Request-Auslösers mit seinem Empfänger, indem mehr als ein Objekt in die Lage versetzt wird, den Request zu bearbeiten. Die empfangenden Objekte werden miteinander verkettet und der Request wird dann so lange entlang dieser Kette weitergeleitet, bis er von einem Objekt angenommen und bearbeitet wird.

***Command* (*Befehl*, siehe Abschnitt 5.2)**

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen.

***Composite* (*Kompositum*, siehe Abschnitt 4.3)**

Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien. Das Design Pattern *Composite (Kompositum)* gestattet den Clients einen einheitlichen Umgang sowohl mit individuellen Objekten als auch mit Objektkompositionen.

***Decorator* (*Dekorierer*, siehe Abschnitt 4.4)**

Dynamische Erweiterung der Funktionalität eines Objekts. `Decorator`-Objekte stellen hinsichtlich der Ergänzung einer Klasse um weitere Zuständigkeiten eine flexible Alternative zur Unterklassenbildung dar.

***Facade* (*Fassade*, siehe Abschnitt 4.5)**

Bereitstellung einer einheitlichen Schnittstelle zu einem Schnittstellensatz in einem Subsystem. Das Design Pattern *Facade (Fassade)* definiert eine Schnittstelle höherer Ebene, die die Nutzung des Subsystems vereinfacht.

***Factory Method* (*Fabrikmethode*, siehe Abschnitt 3.3)**

Definition einer Schnittstelle zur Objekterzeugung, wobei die Bestimmung der zu instanziierenden Klasse den Unterklassen überlassen bleibt. Das Design Pattern *Factory Method (Fabrikmethode)* gestattet einer Klasse, die Instanziierung an Unterklassen zu delegieren.

***Flyweight* (*Fliegengewicht*, siehe Abschnitt 4.6)**

Gemeinsame Nutzung feingranularer Objekte, um sie auch in großer Anzahl effizient nutzen zu können.

***Interpreter* (*Interpreter*, siehe Abschnitt 5.3)**

Definition einer Repräsentation der Grammatik einer gegebenen Sprache sowie Bereitstellung eines Interpreters, der diese Grammatik nutzt, um in der betreffenden Sprache verfasste Sätze zu interpretieren.

***Iterator* (*Iterator*, siehe Abschnitt 5.4)**

Bereitstellung eines sequenziellen Zugriffs auf die Elemente eines aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen.

***Mediator* (*Vermittler*, siehe Abschnitt 5.5)**

Definition eines Objekts, das die Interaktionsweise eines Objektsatzes in sich kapselt. Das Design Pattern *Mediator (Vermittler)* begünstigt lose Kopplungen, indem es die explizite Referenzierung der Objekte untereinander unterbindet und so eine individuelle Steuerung ihrer Interaktionen ermöglicht.

***Memento* (*Memento*, siehe Abschnitt 5.6)**

Erfassung und Externalisierung des internen Zustands eines Objekts, ohne dessen Kapselung zu beeinträchtigen, so dass es später wieder in diesen Zustand zurückversetzt werden kann.

***Observer* (*Beobachter*, siehe Abschnitt 5.7)**

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

***Prototype* (*Prototyp*, siehe Abschnitt 3.4)**

Spezifikation der zu erzeugenden Objekttypen mittels einer prototypischen Instanz und Erzeugung neuer Objekte durch Kopieren dieses Prototyps.

***Proxy* (*Proxy*, siehe Abschnitt 4.7)**

Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. eines Platzhalters zwecks Zugriffssteuerung eines Objekts.

***Singleton* (*Singleton*, siehe Abschnitt 3.5)**

Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.

***State* (*Zustand*, siehe Abschnitt 5.8)**

Anpassung der Verhaltensweise eines Objekts im Fall einer internen Zustandsänderung, so dass es den Anschein hat, als hätte es seine Klasse gewechselt.

***Strategy* (*Strategie*, siehe Abschnitt 5.9)**

Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen. Das Design Pattern *Strategy (Strategie)* ermöglicht eine variable und von den Clients unabhängige Nutzung des Algorithmus.

***Template Method* (*Schablonenmethode*, siehe Abschnitt 5.10)**

Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen. Das Design Pattern *Template Method (Schablonenmethode)* ermöglicht den Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen grundlegende Struktur zu verändern.

***Visitor* (*Besucher*, siehe Abschnitt 5.11)**

Darstellung einer auf die Elemente einer Objektstruktur anzuwendenden Operation. Das Design Pattern *Visitor (Besucher)* ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

1.5 Aufbau des Katalogs

Design Patterns unterscheiden sich sowohl in ihrer Granularität als auch in ihrem Abstraktionsgrad. Und weil es eine ganze Reihe von Patterns gibt, müssen sie irgendwie organisiert werden.

Bei der im Folgenden dargestellten Klassifizierungsmethode werden die Design Patterns des hier vorgestellten Katalogs nach verwandtschaftlichen Beziehungsgraden gruppiert. Diese Art der Gliederung fördert nicht nur ein besseres und schnelleres Verständnis der einzelnen Pattern-Konzepte, sondern erleichtert auch das Auffinden neuer Patterns.

		Zweck		
		Erzeugungsmuster (Creational Patterns)	**Strukturmuster (Structural Patterns)**	**Verhaltensmuster (Behavioral Patterns)**
Gültigkeitsbereich	**Klasse**	*Factory Method* (*Fabrikmethode*, Abschnitt 3.3)	*Adapter* (*Adapter*, (klassenbasiert), Abschnitt 4.1)	*Interpreter* (*Interpreter*, Abschnitt 5.3) *Template Method* (*Schablonenmethode*, Abschnitt 5.10)
	Objekt	*Abstract Factory* (*Abstrakte Fabrik*, Abschnitt 3.1) *Builder* (*Erbauer*, Abschnitt 3.2) *Prototype* (*Prototyp*, Abschnitt 3.4) *Singleton* (*Singleton*, Abschnitt 3.5)	*Adapter* (*Adapter*, (objektbasiert), Abschnitt 4.1) *Bridge* (*Brücke*, Abschnitt 4.2) *Composite* (*Kompositum*, Abschnitt 4.3) *Decorator* (*Dekorierer*, Abschnitt 4.4) *Facade* (*Fassade*, Abschnitt 4.5) *Flyweight* (*Fliegengewicht*, Abschnitt 4.6) *Proxy* (*Proxy*, Abschnitt 4.7)	*Chain of Responsibility* (*Zuständigkeitskette*, Abschnitt 5.1) *Command* (*Befehl*, Abschnitt 5.2) *Iterator* (*Iterator*, Abschnitt 5.4) *Mediator* (*Vermittler*, Abschnitt 5.5) *Memento* (*Memento*, Abschnitt 5.6) *Observer* (*Beobachter*, Abschnitt 5.7) *State* (*Zustand*, Abschnitt 5.8) *Strategy* (*Strategie*, Abschnitt 5.9) *Visitor* (*Besucher*, Abschnitt 5.11)

Tabelle 1.1: Klassifizierung der Design Patterns

In diesem Buch werden die Design Patterns nach zwei übergeordneten Kriterien klassifiziert (siehe Tabelle 1.1). Das erste Kriterium gibt den **Zweck** des Patterns an,

also was es bewirkt. Design Patterns dienen entweder der **Objekterzeugung** oder sie sind **struktur-** bzw. **verhaltensorientiert**:

- *Erzeugungsmuster* beziehen sich auf den Erstellungsprozess von Objekten.
- *Strukturmuster* wirken sich auf die Zusammensetzung von Klassen und Objekten aus.
- *Verhaltensmuster* charakterisieren die Art und Weise der Interaktion von Klassen und Objekten sowie die Verteilung der Zuständigkeiten.

Das zweite Kriterium, der **Gültigkeitsbereich**, spezifiziert, ob das Design Pattern vorwiegend auf Klassen oder auf Objekte angewendet wird.

- *Klassenbasierte* Design Patterns beeinflussen die Beziehungen der Klassen zu ihren Unterklassen, die durch Vererbung erstellt werden und damit statisch sind – d. h., sie werden schon beim Kompilieren festgelegt.
- *Objektbasierte* Design Patterns haben dagegen Auswirkungen auf die Beziehungen zwischen Objekten, die sich zur Laufzeit ändern können und somit dynamischer sind.

Weil fast alle Design Patterns ein gewisses Maß an Vererbung nutzen, werden prinzipiell nur diejenigen als »klassenbasiert« bezeichnet, die sich auch wirklich auf die Klassenbeziehungen konzentrieren. Im Allgemeinen sind die meisten Design Patterns jedoch objektbasiert.

Klassenbasierte Erzeugungsmuster delegieren die Objekterstellung teilweise an Unterklassen, *objektbasierte Erzeugungsmuster* dagegen an ein anderes Objekt. *Klassenbasierten Strukturmuster* stützen sich bei der Zusammenführung auf das Konzept der Vererbung, *objektbasierte Strukturmuster* bilden dagegen Wege ab, um Objekte zusammenzufügen. Und *klassenbasierte Verhaltensmuster* greifen wiederum ebenfalls auf die Vererbung zurück, um sowohl Algorithmen als auch den Programmablauf zu bestimmen, während *objektbasierte Verhaltensmuster* beschreiben, in welcher Form eine Gruppe von Objekten interagiert, um einen Task auszuführen, der nicht von einem einzelnen Objekt bewerkstelligt werden kann.

Es gibt aber noch andere Möglichkeiten, die Design Patterns zu klassifizieren. Manche Patterns werden beispielsweise häufig gemeinsam verwendet, wie etwa *Composite* (*Kompositum*) mit *Iterator* (*Iterator*) oder *Visitor* (*Besucher*). Andere Patterns stellen hingegen Alternativen zueinander dar, wie z. B. im Fall von *Prototype* (*Prototyp*) und *Abstract Factory* (*Abstrakte Fabrik*), die oftmals alternativ eingesetzt werden. Und schließlich gibt es auch noch solche Patterns, die im Endeffekt zu gleichartigen Designs führen, obwohl sie jeweils unterschiedliche Zwecke erfüllen. Ein Beispiel hierfür sind *Composite* (*Kompositum*) und *Decorator* (*Dekorierer*), die sehr ähnliche Strukturdiagramme aufweisen.

Eine weitere Klassifizierungsvariante für Design Patterns basiert auf ihren verwandtschaftlichen Beziehungen zueinander, die in Abbildung 1.2 in einer Übersicht dargestellt sind.

Insgesamt gibt es eine Vielzahl von hilfreichen Klassifizierungsarten für Design Patterns – und sie alle haben durchaus ihre Daseinsberechtigung, denn: Unterschiedliche Denkansätze im Hinblick auf die Klassifizierung führen in jedem Fall zu einem weitreichenderen, besseren Verständnis der Funktionsweise der einzelnen Design Patterns, bieten aufschlussreiche Vergleichsmöglichkeiten und zeigen auf, wann ihr Einsatz sinnvoll ist.

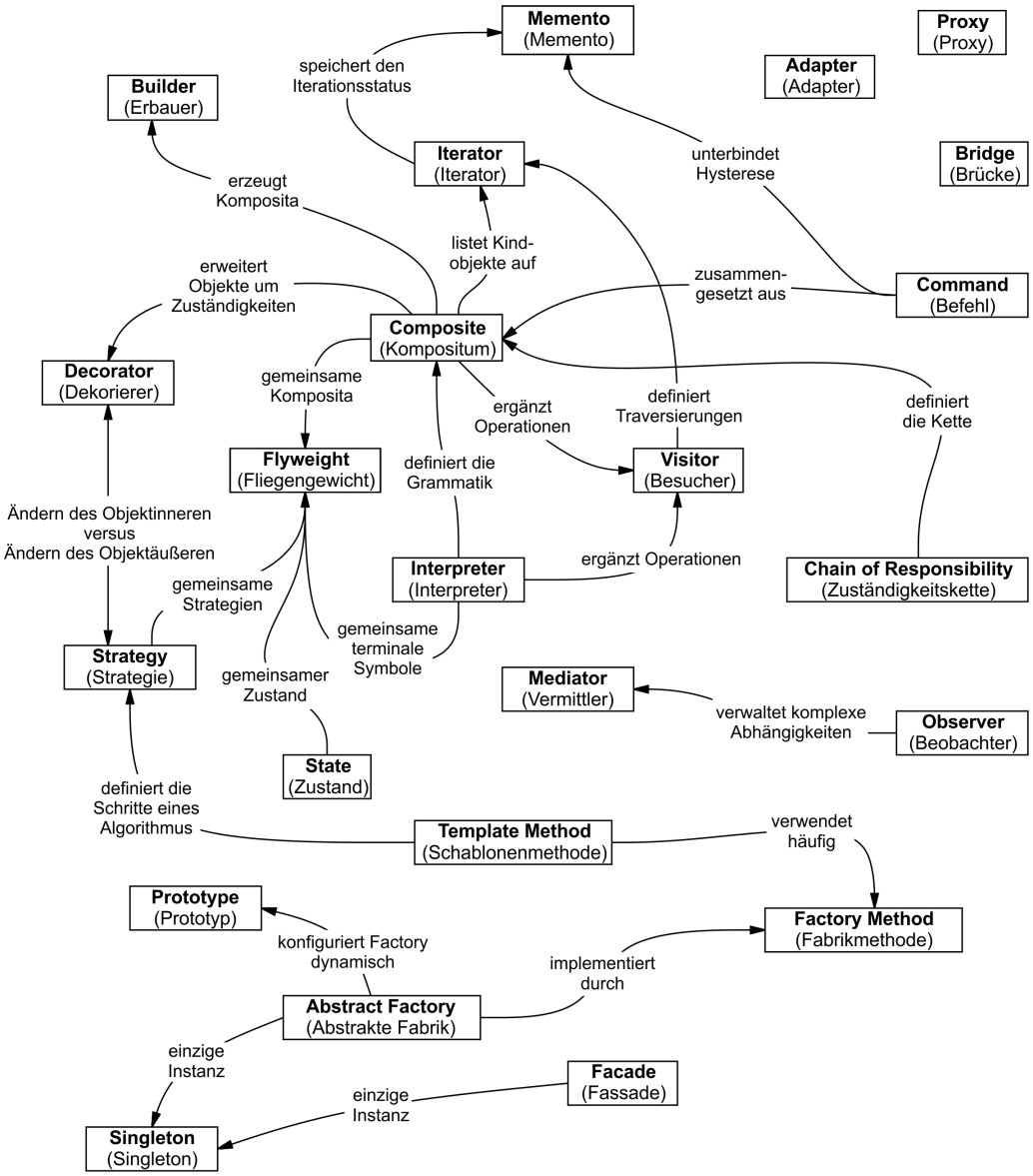


Abb. 1.2: Verwandtschaftliche Beziehungen der Design Patterns

1.6 Die Anwendung von Design Patterns zur Behebung von Designproblemen

Design Patterns bieten objektorientierten Softwareentwicklern die Gelegenheit, zahlreichen Problemen, denen sie tagtäglich gegenüberstehen, auf sehr unterschiedliche Art und Weise zu begegnen. Die nachfolgenden Beispiele veranschaulichen einige solcher Problemfälle und demonstrieren, wie sie sich durch den Einsatz von Design Patterns lösen lassen.

1.6.1 Passende Objekte finden

Objektorientierte Programme setzen sich – wie der Name schon vermuten lässt – aus **Objekten** zusammen. Jedes Objekt umfasst neben dem reinen Datenbestand auch die prozeduralen Routinen – die sogenannten **Methoden** oder **Operationen** –, die auf diesen Daten basieren. Sobald ein Objekt einen **Request** (also eine **Anfrage** oder eine **Nachricht**) von einem **Client** erhält, führt es eine Operation aus.

Requests stellen die *einzige* Möglichkeit dar, ein Objekt zur Durchführung einer Operation zu veranlassen. Und Operationen bieten ihrerseits die *einzige* Möglichkeit, die internen Daten eines Objekts zu modifizieren. Dieses Konzept wird als **Kapselung** bezeichnet: Es ist kein direkter Zugriff auf den internen Zustand des Objekts möglich und er ist auch nicht nach außen hin sichtbar.

Eine besondere Herausforderung beim objektorientierten Design besteht in der Aufgliederung eines Systems in einzelne Objekte – denn dabei müssen zahlreiche Faktoren berücksichtigt werden, die sich nicht selten noch dazu in widersprüchlicher Art und Weise auswirken: die Kapselung, die Granularität, die Abhängigkeiten, die Flexibilität, das Laufzeitverhalten, die Evolution, die Wiederverwendbarkeit und vieles mehr.

Objektorientierte Designmethoden begünstigen in dieser Hinsicht viele verschiedene Ansätze. So könnte man beispielsweise die beherrschenden Substantive und Verben zur Beschreibung der vorliegenden Problemstellung herausgreifen und dazu passende Klassen und Operationen erstellen. Oder man richtet sein Augenmerk mehr auf die Interaktionen und Zuständigkeiten in dem System. Oder man fertigt ein realitätsnahes Modell an, analysiert es und überträgt die dabei ermittelten Objekte in das Design. Welcher Ansatz letztlich der beste ist, ist immer auch ein wenig Geschmackssache.

In vielen Fällen entstammen die Objekte dem realitätsnahen Analysemodell, darüber hinaus werden in objektorientierten Designs häufig aber auch Klassen benutzt, für die es keine realen Entsprechungen gibt. Einige davon, wie z. B. Arrays, besitzen einen niedrigen Abstraktionsgrad, andere hingegen weisen einen deutlich höheren auf. So nutzt beispielsweise das Design Pattern *Composite* (*Kompositum*, siehe Abschnitt 4.3) eine Abstraktion zur einheitlichen Behandlung von

Objekten, für die es kein physisches Gegenstück gibt. Eine strikt an der realen Welt orientierte Modellbildung hat ein System zur Folge, das zwar die gegenwärtigen Realitäten widerspiegelt, nicht notwendigerweise aber auch zukünftige – der Schlüssel für eine flexible Gestaltung des Designs sind die Abstraktionen, die aus dem Designprozess hervorgehen.

Patterns erleichtern die Identifizierung der weniger deutlich erkennbaren Abstraktionen sowie der Objekte, die diese erfassen können. So finden sich für Objekte, die einen Prozess oder einen Algorithmus repräsentieren, normalerweise keine natürlichen Entsprechungen, trotzdem sind sie ein wichtiger Bestandteil flexibler Designs. Das Design Pattern *Strategy* (*Strategie*, siehe Abschnitt 5.9) beschreibt, wie sich austauschbare Algorithmusfamilien implementieren lassen. Und das Pattern *State* (*Zustand*, siehe Abschnitt 5.8) bildet jeden Zustand einer Entität in Form eines Objekts ab. Solche Objekte sind im Rahmen einer Analyse oder in einer frühen Phase des Designprozesses nur selten ausfindig zu machen, sondern treten meist erst später in Erscheinung, wenn es darum geht, das Design flexibler und wiederverwendbar auszugestalten.

1.6.2 Objektgranularität bestimmen

Sowohl die Größe als auch die Anzahl der verwendeten Objekte kann erheblich variieren. Außerdem können sie von der Hardware bis zur vollständigen Anwendung so ziemlich alles repräsentieren. Wie also lässt sich bestimmen, was genau als Objekt genutzt werden sollte?

Auch auf diese Frage liefern Design Patterns eine Antwort. Das Pattern *Facade* (*Fassade*, siehe Abschnitt 4.5) beschreibt beispielsweise, wie sich komplette Subsysteme als Objekte abbilden lassen, und das Pattern *Flyweight* (*Fliegengewicht*, siehe Abschnitt 4.6) definiert die Unterstützung einer großen Zahl von Objekten geringster Granularität. Andere Design Patterns zeigen wiederum bestimmte Arten der Zergliederung eines Objekts in mehrere kleinere Objekte auf. *Abstract Factory* (*Abstrakte Fabrik*, siehe Abschnitt 3.1) und *Builder* (*Erbauer*, siehe Abschnitt 3.2) befassen sich mit Objekten, die ausschließlich für die Erzeugung anderer Objekte zuständig sind. *Visitor* (*Besucher*, siehe Abschnitt 5.11) und *Command* (*Befehl*, siehe Abschnitt 5.2) zielen dagegen auf Objekte ab, deren einzige Aufgabe in der Implementierung eines Requests an ein anderes Objekt bzw. eine Objektgruppe besteht.

1.6.3 Objektschnittstellen spezifizieren

Jede von einem Objekt deklarierte Operation weist eine sogenannte **Signatur** auf, die den Namen, die als Parameter enthaltenen Objekte sowie den Rückgabewert der Operation spezifiziert. Die Menge aller durch die Operationen eines Objekts definierten Signaturen wird als **Schnittstelle** des Objekts bezeichnet. Sie gibt vor, welche Requests an das Objekt übermittelt werden können – d.h. jeder Request, der einer in der Objektschnittstelle erfassten Signatur entspricht.

Der **Typ** eines Objekts bezeichnet eine bestimmte Schnittstelle. Wenn ein Objekt beispielsweise alle Requests für die in seiner Schnittstelle `Window` definierten Operationen akzeptiert, dann spricht man in diesem Fall davon, dass es vom Typ `Window` ist. Grundsätzlich kann ein Objekt mehrere Typen haben und umgekehrt können sehr verschiedene Objekte einen gemeinsamen Typ besitzen. Ebenso können einzelne Teile einer Objektschnittstelle unterschiedlichen Typs sein, d.h., zwei Objekte desselben Typs haben gegebenenfalls lediglich individuelle Schnittstellenbestandteile gemeinsam. Darüber hinaus können Schnittstellen auch andere Schnittstellen als Untermengen enthalten. Wenn die Schnittstelle eines Typs die Schnittstelle seines übergeordneten **Supertyps** enthält, wird dieser Typ als **Subtyp** bezeichnet. Häufig spricht man in diesem Zusammenhang auch davon, dass ein Subtyp die Schnittstelle seines Supertyps *erbt*.

Schnittstellen sind ein grundlegender Bestandteil objektorientierter Systeme. Die Objekte sind ausschließlich über ihre Schnittstellen bekannt. Grundsätzlich gibt es keine andere Möglichkeit, etwas über ein Objekt in Erfahrung zu bringen oder es zu etwas zu veranlassen, als über die Schnittstelle. Sie sagt jedoch nichts über die Implementierung des Objekts aus: Verschiedene Objekte können Requests auf ganz unterschiedliche Art und Weise implementieren – d.h., zwei Objekte mit völlig verschiedenen Implementierungen können dennoch identische Schnittstellen besitzen.

Bei der Übermittlung eines Requests an ein Objekt ist die daraufhin auszuführende Operation *sowohl* von dem Request *als auch* von dem empfangenden Objekt abhängig. Unterschiedliche Objekte, die identische Requests unterstützen, können wiederum verschiedene Implementierungen der Operationen nutzen, die diese Requests ausführen. Die zur Laufzeit erfolgende Zuweisung eines Requests zu einem Objekt und einer seiner Operationen wird als **dynamische Bindung** bezeichnet.

Dank der dynamischen Bindung ist beim Absetzen eines Requests bis zum Zeitpunkt der Ausführung keine Festlegung auf eine spezifische Implementierung erforderlich. Somit ist es problemlos möglich, Programme zu entwickeln, die Objekte mit bestimmten Schnittstellen erwarten, weil jedes Objekt, das die passende Schnittstelle aufweist, diesen Request akzeptieren wird. Zudem gestattet die dynamische Bindung auch die Substitution, sprich den Austausch von Objekten mit identischen Schnittstellen zur Laufzeit – dieses Schlüsselkonzept objektorientierter Systeme wird als **Polymorphie** bezeichnet. Dadurch braucht ein Client-Objekt außer in Bezug auf deren Unterstützung einer bestimmten Schnittstelle keine weiteren Annahmen über andere Objekte zu treffen. Die Polymorphie vereinfacht also die Client-Definition, entkoppelt Objekte voneinander und gestattet es ihnen außerdem, ihre wechselseitigen Beziehungen zur Laufzeit zu variieren.

Design Patterns erleichtern die Schnittstellendefinition, indem sie ihre Schlüsselelemente sowie die Datentypen, die über eine Schnittstelle geleitet werden,

identifizieren. Mitunter geben sie auch vor, was *nicht* in einer Schnittstelle enthalten sein sollte. So beschreibt zum Beispiel das Pattern *Memento* (*Memento*, siehe Abschnitt 5.6), wie der interne Zustand eines Objekts gekapselt und gespeichert sein muss, damit das Objekt zu einem späteren Zeitpunkt wieder in diesen Zustand zurückversetzt werden kann. Hier lautet die Vorgabe, dass `Memento`-Objekte zwei Schnittstellen definieren müssen: eine *eingeschränkte* Schnittstelle, die den Clients zum Verwahren und Kopieren der Mementos dient, und eine *privilegierte* Schnittstelle, die ausschließlich vom ursprünglichen Objekt genutzt werden kann, um seinen Zustand im Memento zu speichern und wiederherzustellen.

Darüber hinaus spezifizieren Design Patterns auch die Beziehungen zwischen den Schnittstellen. Insbesondere setzen sie häufig voraus, dass einige Klassen ähnliche Schnittstellen aufweisen oder sehen Einschränkungen für bestimmte Klassenschnittstellen vor. So erfordern sowohl *Decorator* (*Dekorierer*, siehe Abschnitt 4.4) als auch *Proxy* (*Proxy*, siehe Abschnitt 4.7), dass die Schnittstellen der `Decorator`- und `Proxy`-Objekte mit denen der dekorierten bzw. stellvertretenen Objekten identisch sind. Und beim Pattern *Visitor* (*Besucher*, siehe Abschnitt 5.11) muss die `Visitor`-Schnittstelle die Klassen aller Objekte abbilden, die der `Visitor` besuchen kann.

1.6.4 Objektimplementierungen spezifizieren

Nachdem sie bislang lediglich am Rande erwähnt wurde, soll die konkrete Definition eines Objekts an dieser Stelle einmal ausführlicher betrachtet werden. Die Implementierung eines Objekts wird durch seine **Klasse** definiert. Sie spezifiziert die internen Daten sowie die Repräsentation des Objekts und bestimmt darüber hinaus auch, welche Operationen es ausführen kann.

In der auf der *Object-Modeling Technique* (*OMT*, Objekt-Modellierungstechnik) basierenden Notation (siehe Anhang B) wird eine Klasse in einem Objektdiagramm wie folgt dargestellt: Zuoberst ist der Klassenname in Fettschrift angegeben, das nächste Segment enthält eine Auflistung der Operationen in Normalschrift, und zum Schluss sind die von der Klasse definierten Daten aufgeführt (siehe Abbildung 1.3).

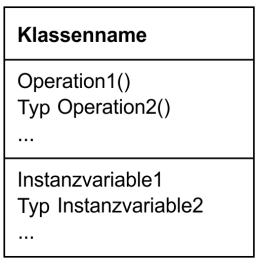


Abb. 1.3: Objektdiagramm einer Klasse

Da hier nicht von einer statisch typisierten Implementierungssprache ausgegangen wird, sind die Typen der Rückgabewerte und Instanzvariablen optional.

Objekte werden durch die **Instanziierung** einer Klasse erzeugt und dementsprechend als **Instanzen** der Klasse bezeichnet. Während des Instanziierungsvorgangs wird zum einen der Speicherbereich für die internen (aus den **Instanzvariablen** bestehenden) Daten des Objekts angelegt, und zum anderen werden die Operationen mit diesen Daten verknüpft. Durch die Instanziierung einer Klasse können viele ähnliche Instanzen eines Objekts erzeugt werden.

Eine Klasse, die Objekte einer anderen Klasse instanziiert, wird durch einen gestrichelten Pfeil gekennzeichnet. Die Pfeilspitze verweist dabei auf die Klasse der instanziierten Objekte:

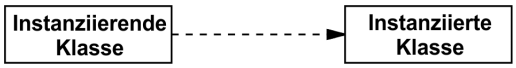


Abb. 1.4: Instanziierung von Objekten anderer Klassen

Die **Klassenvererbung** ermöglicht die Erstellung neuer Klassen, die auf bereits existierenden Klassen basieren. Und wenn eine **Unterklasse** (auch **abgeleitete** oder **Subklasse** genannt) von einer **Basisklasse** (auch als **Eltern-** oder **Superklasse** bezeichnet) erbt, beinhaltet sie auch die Definitionen aller zugehörigen Daten und Operationen, die in der Basisklasse definiert sind. Somit enthalten die Objekte, die Instanzen der Unterklasse sind, ebenfalls alle in der Unterklasse und ihren Basisklassen definierten Daten und sind in der Lage, sämtliche dort definierten Operationen auszuführen. Die Beziehung von Unterklasse und Basisklasse ist in Abbildung 1.5 durch eine vertikale Linie mit aufgesetztem Dreiecksymbol gekennzeichnet:

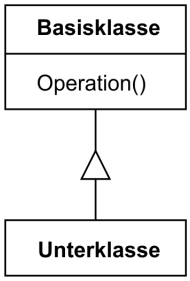


Abb. 1.5: Beziehung Basisklasse – Unterklasse

Der Hauptzweck einer **abstrakten Klasse** besteht in der Definition einer gemeinsamen Schnittstelle für alle ihre Unterklassen. Da solche Klassen ihre Implementierungen teilweise oder auch vollumfänglich an die in den Unterklassen definierten

Operationen delegieren, können sie nicht instanziiert werden. Die von einer abstrakten Klasse deklarierten, aber nicht implementierten Operationen werden als **abstrakte Operationen** bezeichnet. Nicht abstrakte Klassen werden **konkrete Klassen** genannt.

Unterklassen können das Verhalten ihrer Basisklassen weiter verfeinern und auch umdefinieren. Genauer gesagt: Eine Klasse ist in der Lage, eine von ihrer Basisklasse definierte Operation zu **überschreiben**. Dadurch können Unterklassen stellvertretend für ihre Basisklassen die Bearbeitung von Requests übernehmen. Das Konzept der Klassenvererbung gestattet also nicht nur die Definition neuer Klassen durch die schlichte Erweiterung anderer Klassen, sondern erleichtert auch die Definition von Objektfamilien mit verwandter Funktionalität.

Zur besseren Unterscheidung von den konkreten Klassen sind die Namen von abstrakten Klassen – ebenso wie die abstrakten Operationen – in den nachfolgenden Diagrammen in Kursivschrift angegeben. Außerdem können die Diagramme auch Pseudocode für die Implementierung einer Operation ausweisen, der dann durch ein Eselsohr gekennzeichnet und über eine gestrichelte Linie mit der implementierten Operation verbunden ist (siehe Abbildung 1.6):

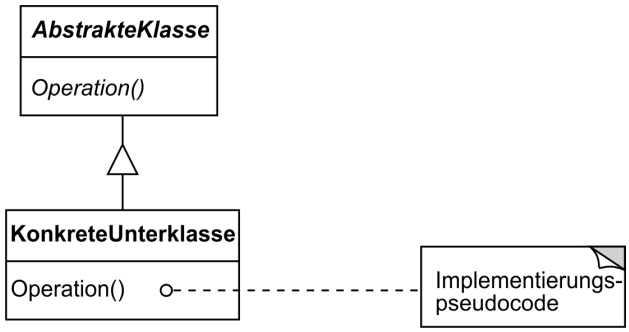


Abb. 1.6: Abstrakte und konkrete Klassen

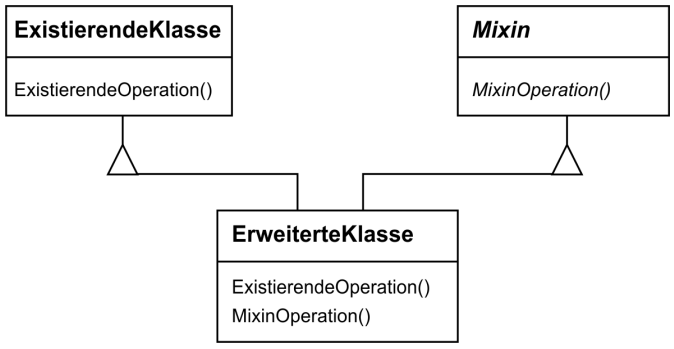


Abb. 1.7: Mehrfachvererbung der `Mixin`-Klasse

Eine **Mixin-Klasse** ist eine Klasse, die eine optionale Schnittstelle oder Funktionalität für andere Klassen bereitstellt. Mit einer abstrakten Klasse hat sie gemein, dass sie ebenfalls nicht instanziiert wird. `Mixin`-Klassen bedingen eine Mehrfachvererbung.

Klassenvererbung kontra Schnittstellenvererbung

Zwischen der **Klasse** und dem **Typ** eines Objekts besteht ein wesentlicher Unterschied: Die *Klasse* definiert, wie das Objekt implementiert ist. Sie beschreibt seinen internen Zustand und die Implementierung seiner Operationen. Im Gegensatz dazu bezieht sich der *Typ* eines Objekts ausschließlich auf seine Schnittstelle – die Gesamtheit der Requests, auf die es antworten kann. Ein Objekt kann viele Typen haben, andererseits können Objekte verschiedener Klassen aber auch denselben Typ besitzen.

Selbstverständlich stehen Klasse und Typ in einer engen Beziehung zueinander: Weil eine Klasse die Operationen definiert, die ein Objekt ausführen kann, bestimmt sie somit auch den Typ des Objekts. Die Feststellung, dass ein Objekt eine Instanz einer Klasse ist, bedeutet also im Grunde genommen, dass das Objekt die von der Klasse vorgegebene Schnittstelle unterstützt.

In Programmiersprachen wie C++ und Eiffel werden Klassen zur Spezifikation sowohl des Typs als auch der Implementierungen eines Objekts verwendet. Smalltalk-Programme deklarieren hingegen keine Typen für Variablen, und dementsprechend überprüft der Compiler auch nicht, ob die einer Variablen zugewiesenen Objekttypen Subtypen des Variablentyps sind. Das Versenden eines Requests erfordert zwar eine dahingehende Überprüfung, dass die Klasse des Empfängers den Request auch implementiert – ob der Empfänger eine Instanz einer bestimmten Klasse ist, muss jedoch nicht geprüft werden.

Ein weiterer wichtiger Aspekt, den es in diesem Zusammenhang zu beachten gilt, ist der Unterschied zwischen der **Klassenvererbung** und der **Schnittstellenvererbung** (auch als **Subtyping** bezeichnet). Bei der Klassenvererbung wird die Implementierung eines Objekts in Form der Implementierung eines anderen Objekts definiert. Es handelt sich also um einen Mechanismus zur Wiederverwertung bzw. Wiederverwendung von Code und Repräsentation. Bei der Schnittstellenvererbung geht es im Gegensatz dazu darum, wann ein Objekt *anstelle* eines anderen Objekts verwendet werden kann.

Diese beiden Konzepte sind leicht zu verwechseln, weil sie in vielen Sprachen nicht explizit unterschieden werden. In Programmiersprachen wie C++ und Eiffel bezieht sich das Konzept der Vererbung sowohl auf die Schnittstellen- als auch auf die Implementierungsvererbung. So wird die Vererbung einer Schnittstelle in C++ z. B. standardmäßig durch das öffentliche Erben von einer Klasse realisiert, die (rein) virtuelle Memberfunktionen aufweist. Eine Annäherung an eine reine Schnittstellenvererbung lässt sich in C++ durch das öffentliche Erben von rein

abstrakten Klassen erreichen. In Smalltalk erfolgt eine Vererbung grundsätzlich als Implementierungsvererbung. Hier können Variablen generell Instanzen beliebiger Klassen zugewiesen werden, solange diese die auf dem Wert der Variablen ausgeführte Operation unterstützen.

Auch wenn die meisten Programmiersprachen keine Unterscheidung zwischen Schnittstellen- und Implementierungsvererbung vornehmen, ist dies aufseiten der Entwickler durchaus gängige Praxis: Smalltalk-Programmierer behandeln Unterklassen in der Regel wie Subtypen (abgesehen von einigen bekannten Ausnahmen [Coo92]), und C++-Programmierer manipulieren Objekte mithilfe der von abstrakten Klassen definierten Typen.

Viele der in diesem Buch vorgestellten Design Patterns sind von dieser Unterscheidung abhängig. So müssen beispielsweise die Objekte im Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1) einen gemeinsamen Typ haben, weisen aber üblicherweise keine gemeinsame Implementierung auf. Im Pattern *Composite* (*Kompositum*, siehe Abschnitt 4.3) definiert die Komponente zwar eine gemeinsame Schnittstelle, das Kompositum definiert jedoch häufig eine gemeinsame Implementierung. Und die Design Patterns *Command* (*Befehl*, siehe Abschnitt 5.2), *Observer* (*Beobachter*, siehe Abschnitt 5.7), *State* (*Zustand*, siehe Abschnitt 5.8) und *Strategy* (*Strategie*, siehe Abschnitt 5.9) werden oftmals mit abstrakten Klassen implementiert, die reine Schnittstellen darstellen.

Gegen Schnittstellen statt gegen Implementierungen programmieren

Die Klassenvererbung stellt im Wesentlichen einen Mechanismus zur Erweiterung der Funktionalität einer Anwendung durch Wiederverwendung der in den Basisklassen enthaltenen Funktionen dar. Sie ermöglicht eine sehr schnelle, auf einem bereits existierenden Objekt basierende Definition eines neuen Objekts. Auf diese Weise lassen sich neue Implementierungen praktisch ohne Aufwand realisieren, weil ein Großteil dessen, was dafür nötig ist, von existierenden Klassen geerbt wird.

Die Wiederverwendung einer Implementierung ist allerdings erst »die halbe Miete« – denn auch die Fähigkeit, bei der Vererbung Objektfamilien mit *identischen* Schnittstellen definieren zu können (in der Regel durch das Erben von einer abstrakten Klasse), spielt hierbei eine bedeutende Rolle. Sie fragen sich warum? Weil die Polymorphie davon abhängig ist.

Eine sorgfältige (bzw. *korrekte*) Anwendung des Vererbungskonzepts bedingt, dass alle von einer abstrakten Klasse abgeleiteten Klassen auch deren Schnittstelle nutzen. Und das bedeutet wiederum, dass eine Unterklasse Operationen lediglich ergänzt oder überschreibt, nicht aber die Operationen der Basisklasse verbirgt. So können *alle* Unterklassen auf die Requests in der Schnittstelle der betreffenden abstrakten Klasse reagieren und werden somit auch allesamt zu Subtypen der abstrakten Klasse.

Die einzig auf der durch die abstrakten Klassen definierten Schnittstelle basierende Manipulation von Objekten bringt zwei Vorteile mit sich:

1. Solange die Objekte die clientseitig erwartete Schnittstelle ansprechen, bleiben den Clients die spezifischen verwendeten Objekttypen verborgen.
2. Die Klassen, die diese Objekte implementieren, bleiben den Clients ebenfalls verborgen. Sie erkennen lediglich die abstrakten Klassen, die die Schnittstelle definieren.

Durch diese Verfahrensweise werden die Implementierungsabhängigkeiten zwischen den Subsystemen derart reduziert, dass folgender Grundsatz des wiederverwendbaren objektorientierten Designs greift:

Programmiere gegen Schnittstellen, nicht gegen Implementierungen.

Variablen dürfen also nicht als Instanzen bestimmter konkreter Klassen deklariert, sondern ausschließlich auf eine von einer abstrakten Klasse definierte Schnittstelle ausgerichtet sein. Und genau dieses Prinzip ist auch eins der Hauptmotive der Design Patterns in diesem Buch.

Natürlich müssen aber irgendwo im System auch konkrete Klassen instanziiert bzw. eine bestimmte Implementierung spezifiziert werden – und eben dazu sind die **Erzeugungsmuster** (*Abstract Factory* (*Abstrakte Fabrik*, siehe Abschnitt 3.1), *Builder* (*Erbauer*, siehe Abschnitt 3.2), *Factory Method* (*Fabrikmethode*, siehe Abschnitt 3.3), *Prototype* (*Prototyp*, siehe Abschnitt 3.4) und *Singleton* (*Singleton*, siehe Abschnitt 3.5)) gedacht. Durch die Abstrahierung des Objekterstellungsprozesses bieten diese Patterns verschiedene Möglichkeiten, eine Schnittstelle bei der Instanziierung transparent mit ihrer Implementierung zu verknüpfen. Erzeugungsmuster gewährleisten, dass das System mit Blick auf die Schnittstellen und nicht auf die Implementierungen geschrieben wird.

1.6.5 Wiederverwendungsmechanismen einsetzen

Die meisten Entwickler sind sich über Konzepte wie Objekte, Schnittstellen, Klassen und Vererbung im Klaren. Die eigentliche Herausforderung besteht jedoch darin, sie bei der Entwicklung flexibler, wiederverwendbarer Software auch möglichst sinnvoll einzusetzen – und die Design Patterns zeigen auf, wie das funktioniert.

Vererbung kontra Komposition

Die zwei geläufigsten Techniken für die Wiederverwendung von Funktionalität in objektorientierten Systemen sind die **Klassenvererbung** und die **Objektkomposition**. Wie zuvor bereits erwähnt, ermöglicht die Klassenvererbung die Definition der Implementierung einer Klasse auf der Grundlage einer anderen. Das Konzept der Wiederverwendung durch Unterklassenbildung wird häufig auch als **White-Box-Wiederverwendung** bezeichnet. Der Begriff »White-Box« bezieht sich dabei

auf die Sichtbarkeit: Bei der Vererbung ist die interne Struktur der Basisklassen für die Unterklassen meist sichtbar.

Die Alternative zur Klassenvererbung ist die Objektkomposition. Bei dieser Technik wird eine neue, komplexere Funktionalität durch die Zusammensetzung, sprich die *Komposition* von Objekten erreicht. Voraussetzung ist hierbei, dass die zusammengeführten Objekte wohldefinierte Schnittstellen aufweisen. Diese Art der Wiederverwendung wird auch **Black-Box-Wiederverwendung** genannt, weil die interne Struktur der Objekte in diesem Fall nicht sichtbar ist – die Objekte treten lediglich als »Black-Boxes« in Erscheinung.

Sowohl die Vererbung als auch die Komposition haben jeweils ihre Vor- und Nachteile. Die Klassenvererbung wird schon beim Kompilieren statisch definiert und lässt sich unkompliziert anwenden, weil sie unmittelbar von der Programmiersprache unterstützt wird. Außerdem erleichtert sie die Modifikation der wiederverwendeten Implementierung. Wenn eine Unterklasse einige, aber nicht alle Operationen überschreibt, kann sie damit auch die geerbten Operationen beeinflussen, sofern diese die überschriebenen Operationen aufrufen.

Aber auch die Klassenvererbung hat einige Nachteile. Erstens können die von der Basisklasse geerbten Implementierungen nicht zur Laufzeit geändert werden, weil die Vererbung bereits beim Kompilieren definiert wird. Und zweitens, was meist schwerer wiegt, definieren die Basisklassen oftmals zumindest Teile der physischen Repräsentation ihrer Unterklassen. Da eine Unterklasse durch den Vererbungsprozess der detaillierten Implementierungsbeschaffenheit ihrer Basisklasse ausgesetzt wird, spricht man davon, dass die »Vererbung die Kapselung aufbricht« [Sny86]: Die Implementierung der Unterklasse wird in so hohem Maße mit der Implementierung ihrer Basisklasse verknüpft, dass jede Modifikation an der Basisklassenimplementierung auch eine Änderung der Unterklasse erforderlich macht.

Solche Implementierungsabhängigkeiten können bei dem Versuch, eine Unterklasse wiederzuverwenden, zu Problemen führen: Sollte irgendein Aspekt der geerbten Implementierung nicht für die neuen Anwendungen geeignet sein, muss die Basisklasse umgeschrieben oder durch eine passendere Klasse ersetzt werden. Zudem schränken solche Abhängigkeiten nicht nur die Flexibilität, sondern letztlich auch die Wiederverwendbarkeit ein. Eine Möglichkeit, diesem Problem zu begegnen, ist das ausschließliche Erben von abstrakten Klassen, da diese in der Regel nur wenig bis gar keine Implementierung vorsehen.

Die Objektkomposition wird durch die Zuweisung von Referenzen auf andere Objekte dynamisch zur Laufzeit definiert. Hierfür ist es erforderlich, dass die Objekte ihre jeweiligen Schnittstellen anerkennen – was wiederum sorgfältig ausgestaltete Schnittstellen voraussetzt, die die Verwendung eines Objekts im Zusammenspiel mit vielen anderen Objekten nicht be- bzw. verhindern. Die Sache hat aber auch ihr Positives: Weil ausschließlich über ihre Schnittstellen auf die

Objekte zugegriffen wird, wird die Kapselung nicht aufgebrochen. Jedes beliebige Objekt kann zur Laufzeit durch ein anderes ersetzt werden, solange dieses vom selben Typ ist. Und darüber hinaus ergeben sich bei der schnittstellenbasierten Implementierung eines Objekts auch erheblich weniger Implementierungsabhängigkeiten.

Zudem hat die Objektkomposition noch eine weitere Auswirkung auf das Systemdesign: Im Gegensatz zur Klassenvererbung erleichtert sie die Beibehaltung der Kapselung der Klassen sowie deren Ausrichtung auf eine Aufgabe. Die Klassen und Klassenhierarchien bleiben überschaubar und es ist unwahrscheinlicher, dass sie sich zu unkontrollierbaren Monstrositäten auswachsen. Andererseits weist ein auf der Objektkomposition basierendes Design mehr Objekte (wenn auch weniger Klassen) auf und das Systemverhalten ist von deren Wechselbeziehungen abhängig, statt in einer einzelnen Klasse definiert zu sein.

Und damit kommt nun auch der zweite Grundsatz des objektorientierten Designs zum Tragen:

Die Objektkomposition ist der Klassenvererbung vorzuziehen.

Idealerweise sollten zur Wiederverwendung vorhandener Klassen keine neuen Komponenten erzeugt werden müssen, sondern die gesamte benötigte Funktionalität einfach durch Zusammenführen der verfügbaren Komponenten mittels Objektkomposition erzielt werden können. Allerdings gelingt das nur selten, weil die Menge der vorhandenen Komponenten in der Praxis meist schlicht nicht umfassend genug ist. Die Wiederverwendung durch Vererbung gestaltet es einfacher, neue Komponenten zu erzeugen, die mit älteren zusammengeführt werden können. Vererbung und Objektkomposition arbeiten also Hand in Hand.

Dennoch machen viele Entwickler erfahrungsgemäß zu ausgiebigen Gebrauch von der Vererbung als Wiederverwendungstechnik. Häufig lassen sich Designs durch eine verstärkte Anwendung der Objektkomposition einfacher und in höherem Maße wiederverwendbar gestalten – und dementsprechend kommt sie in den Design Patterns auch immer wieder zum Einsatz.

Delegation

Mithilfe der **Delegation** lässt sich die Objektkomposition als ebenso leistungsstarke Technik für die Wiederverwendung einsetzen wie die Vererbung [Lie86, JZ91]. Bei dieser Methode sind *zwei* Objekte an der Bearbeitung eines Requests beteiligt: Ein empfangendes Objekt delegiert Operationen an sein Delegationsobjekt, den sogenannten **Delegate**. Dieses Verfahren ist analog der Request-Weiterleitung der Unterklassen an die Basisklassen zu sehen, allerdings kann sich eine geerbte Operation bei der Vererbungstechnik immer auf das empfangende Objekt beziehen – in C++ über die `this`-Membervariable und in Smalltalk mittels `self`. Um mit der Delegation denselben Effekt zu erzielen, leitet der Empfänger eine

Referenz auf sich selbst an den Delegate weiter, damit die weitergeleitete Operation auf den Empfänger verweist.

Statt beispielsweise die Klasse `Window` zu einer Unterklasse von `Rectangle` zu machen (weil Fenster nun mal rechteckig sind), könnte sie das Verhalten von `Rectangle` wiederverwenden, indem sie eine Instanzvariable `Rectangle` nutzt und das rechteckspezifische Verhalten an sie *delegiert*. Mit anderen Worten: Das Fenster *wäre* kein Rechteck, sondern würde ein Rechteck *enthalten*. In diesem Fall müsste die `Window`-Klasse Requests fortan explizit an ihre `Rectangle`-Instanz weiterleiten – im anderen Fall hätte sie diese Operationen geerbt.

Abbildung 1.8 zeigt, wie die Klasse `Window` ihre Operation `Area()` an eine `Rectangle`-Instanz delegiert.

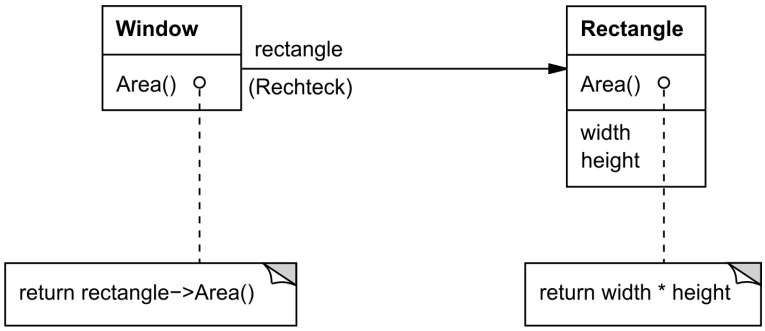


Abb. 1.8: Beispieldiagramm Delegation

Der Pfeil mit der durchgezogenen Linie symbolisiert, dass die Klasse eine Referenz auf eine Instanz einer anderen Klasse enthält. Diese Referenz kann optional benannt werden, hier beispielsweise mit `rectangle`.

Der wichtigste Vorteil der Delegation besteht darin, dass sie sowohl die Zusammenführung von Verhalten zur Laufzeit als auch die Änderung der Kompositionsstruktur erleichtert. So könnte man das Fenster in diesem Beispiel ohne Weiteres zur Laufzeit kreisförmig gestalten, indem man die `Rectangle`-Instanz durch eine `Circle`-Instanz ersetzt – Voraussetzung dafür wäre allerdings, dass `Rectangle` und `Circle` vom selben Typ sind.

Ebenso wie andere Techniken für eine flexiblere Softwaregestaltung, die sich die Objektkomposition zunutze machen, bringt aber auch die Delegation einen Nachteil mit sich: Dynamische, stark parametrisierte Software ist insgesamt schwieriger zu überblicken und zu verstehen als eher statisch orientierte Software. Abgesehen davon ist auch mit einigen Ineffizienzen hinsichtlich der Laufzeitperformance zu rechnen, wenngleich sich Ineffizienzen aufseiten des agierenden Menschen auf lange Sicht immer noch folgenschwerer auswirken. Die Anwendung des Delegationsprinzips ist immer dann eine gute Wahl, wenn dies zu einer

Vereinfachung des Designs beiträgt, statt die Dinge zu verkomplizieren. Die Formulierung fester Regelvorgaben für den gezielten Einsatz der Delegation ist allerdings schwierig, denn wie effizient sie sein wird, hängt einerseits vom jeweiligen Kontext und andererseits von der Erfahrung des Entwicklers ab. Generell gilt jedoch, dass sie am besten funktioniert, wenn sie in ausgesprochen stilisierter Art und Weise angewendet wird – d.h. im Rahmen von Standard-Patterns.

Es gibt mehrere Design Patterns, die die Delegation standardmäßig nutzen. *State* (*Zustand*, siehe Abschnitt 5.8), *Strategy* (*Strategie*, siehe Abschnitt 5.9) und *Visitor* (*Besucher*, siehe Abschnitt 5.11) basieren sogar vollständig auf diesem Konzept: Im Design Pattern *State* (*Zustand*) leitet ein Objekt Requests an ein `State`-Objekt weiter, das den aktuellen Zustand des ursprünglichen Objekts repräsentiert. Beim Pattern *Strategy* (*Strategie*) delegiert ein Objekt einen spezifischen Request an ein anderes Objekt, das eine Strategie für die Ausführung des Requests bereithält – natürlich nimmt ein Objekt nur einen Zustand ein, trotzdem kann es mehrere Strategien für verschiedene Arten von Requests vorsehen. Der Zweck dieser beiden Design Patterns besteht darin, das Verhalten eines Objekts durch die Modifikation der Objekte zu manipulieren, an die es die Requests delegiert. Und im Fall des Patterns *Visitor* (*Besucher*) wird die Operation, die auf jedes Element der Objektstruktur angewendet wird, grundsätzlich an das `Visitor`-Objekt delegiert.

Im Gegensatz zu den vorgenannten Beispielen machen andere Design Patterns weniger Gebrauch von der Delegation. So präsentiert das Pattern *Mediator* (*Vermittler*, siehe Abschnitt 5.5) ein Objekt zur Vermittlung der Kommunikation zwischen anderen Objekten. In manchen Fällen implementiert dieses `Mediator`-Objekt Operationen durch einfaches Weiterleiten an die übrigen Objekte, in anderen Fällen übergibt es allerdings zusätzlich noch eine Referenz auf sich selbst und wendet somit eine echte Form der Delegation an. Beim Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1) wird die Bearbeitung von Requests durch deren Weiterleitung innerhalb einer Objektkette von einem Objekt zum nächsten realisiert. Mitunter enthält ein solcher Request auch eine Referenz auf das Ausgangsobjekt, an das er ursprünglich gerichtet war – in diesem Fall nutzt das Pattern die Technik der Delegation. Und das Design Pattern *Bridge* (*Brücke*, siehe Abschnitt 4.2) entkoppelt schließlich eine Abstraktion von seiner Implementierung. Wenn die Abstraktion und eine bestimmte Implementierung eng aufeinander abgestimmt sind, kann die Abstraktion die Operationen einfach an diese Implementierung weiterleiten.

Die Delegation ist ein extremes Beispiel für die Objektkomposition, das zeigt, dass die Vererbung als Mechanismus für die Wiederverwendung von Code jederzeit durch die Objektkomposition ersetzt werden kann.

Vererbung kontra parametrisierte Typen

Eine weitere (nicht strikt objektorientierte) Technik zur Wiederverwendung von Funktionalität ist der Einsatz von **parametrisierten Typen**, auch als **Generics** (Ada,

Eiffel) und **Templates** (C++) bekannt. Diese Verfahrensweise ermöglicht die Definition eines Typs ohne vorherige Spezifizierung sämtlicher anderen von ihm verwendeten Typen. Die unspezifizierten Typen werden zum Nutzungszeitpunkt als *Parameter* bereitgestellt. So kann beispielsweise eine Klasse `List` durch den Typ der in ihr enthaltenen Elemente parametrisiert werden. Um eine Liste von Integerzahlen zu deklarieren, könnte dem parametrisierten Typ von `List` der Typ `Integer` als Parameter übergeben werden. Und zur Deklaration einer Liste mit `String`-Objekten würde der Typ `String` als Parameter verwendet. Die Sprachimplementierung erzeugt für jeden Elementtyp eine benutzerdefinierte Version des Templates der `List`-Klasse.

Parametrisierte Typen stellen (neben der Klassenvererbung und der Objektkomposition) also eine dritte Methode für die Verhaltenskomposition in objektorientierten Systemen zur Verfügung. Viele Designs lassen sich mithilfe jeder dieser drei Techniken implementieren. Zur Parametrisierung einer Sortierroutine anhand der Operation zum Abgleichen der Elemente könnte man den Vergleich

1. als eine durch Unterklassen implementierte Operation definieren (unter Anwendung des Design Patterns *Template Method* (*Schablonenmethode*, siehe Abschnitt 5.10)),
2. der Zuständigkeit eines Objekts zuweisen, das an die Sortierroutine übergeben wird (*Strategy* (*Strategie*, siehe Abschnitt 5.9)) oder
3. als Argument eines C++-Templates oder Ada-Generics deklarieren, das den Namen der Funktion zum Aufruf des Elementeabgleichs spezifiziert.

Diese Techniken unterscheiden sich in einigen wichtigen Punkten. Die Objektkomposition ermöglicht zwar die Änderung der Verhaltenskomposition zur Laufzeit, erfordert aber eine Dereferenzierung und kann weniger effizient sein. Die Vererbung gestattet neben der Bereitstellung von Standardimplementierungen für Operationen auch, dass diese von Unterklassen überschrieben werden. Und parametrisierte Typen erlauben die Modifizierung der von einer Klasse nutzbaren Typen. Aber weder die Vererbung noch die parametrisierten Typen können zur Laufzeit verändert werden. Welcher Ansatz am besten geeignet ist, hängt von dem jeweiligen Design und den Einschränkungen in Bezug auf die Implementierung ab.

Auch wenn keins der in diesem Buch vorgestellten Design Patterns unmittelbar mit parametrisierten Typen zu tun hat, werden sie dennoch gelegentlich für die individuelle Anpassung der C++-Implementierung eines Design Patterns genutzt. In einer Programmiersprache wie Smalltalk, die beim Kompilieren keine Typüberprüfung vornimmt, werden die parametrisierten Typen allerdings überhaupt nicht benötigt.

1.6.6 Strukturen der Laufzeit und beim Kompilieren abstimmen

Die **Laufzeitstruktur** eines objektorientierten Programms lässt häufig kaum Ähnlichkeiten mit seiner **Codestruktur** erkennen. Die Codestruktur wird beim Kompi-

lieren »eingefroren« – sie besteht aus Klassen in festen Vererbungsbeziehungen. Die Laufzeitstruktur eines Programms besteht hingegen aus einem Netzwerk miteinander kommunizierender Objekte, das sehr schnellen Änderungen unterworfen ist. Tatsächlich sind beide Strukturen größtenteils voneinander unabhängig. Der Versuch, die eine auf der Grundlage der anderen zu verstehen, kommt in etwa dem Versuch gleich, die Dynamik lebendiger Ökosysteme aus der statischen Pflanzen- und Tierartenbestimmung herzuleiten und umgekehrt.

Betrachten Sie einmal die Unterscheidung zwischen der **Aggregation** und der **Assoziation** von Objekten und wie verschieden sie sich beim Kompilieren bzw. zur Laufzeit darstellen. Aggregation bedeutet, dass ein Objekt ein anderes »besitzt« bzw. dafür verantwortlich ist. Im Allgemeinen spricht man davon, dass ein Objekt ein anderes Objekt *hat* bzw. Letzteres *Teil von* ihm ist (*»is-part-of«-Beziehung*). Dieses Prinzip bedingt auch, dass das aggregierte Objekt und dessen Besitzer identische Lebenszyklen haben.

Bei der Assoziation hat ein Objekt dagegen lediglich *Kenntnis von* einem anderen Objekt. Diese Beziehung wird oft auch als *Kennt-* oder *Benutzt-Beziehung* bezeichnet. Assoziierte Objekte können zwar wechselseitig Operationen abfragen, sind jedoch nicht füreinander verantwortlich. Die Assoziation repräsentiert gegenüber der Aggregation die schwächere Beziehung und beinhaltet eine wesentlich lockerere Verknüpfung zwischen den Objekten.

Assoziationen werden in den Diagrammen in diesem Buch durch einen einfachen Pfeil mit durchgezogener Linie dargestellt. Aggregationen werden durch denselben Pfeil symbolisiert, der jedoch zusätzlich am Ausgangspunkt durch eine liegende Raute gekennzeichnet ist, wie in Abbildung 1.9 zu sehen:

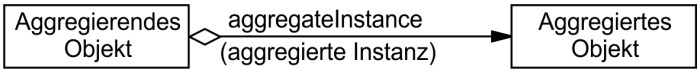


Abb. 1.9: Beispieldiagramm Aggregation

Aggregation und Assoziation sind leicht zu verwechseln, zumal sie häufig auf die gleiche Art und Weise implementiert werden. In Smalltalk referenzieren alle Variablen andere Objekte. Hier wird nicht zwischen Aggregation und Assoziation unterschieden. In C++ kann die Aggregation durch die Definition von Membervariablen implementiert werden, die echte Objekte repräsentieren, in der Praxis wird sie jedoch meist eher durch Pointer (Zeiger) oder Referenzen auf Instanzen definiert. Die Assoziation wird ebenfalls durch Pointer und Referenzen implementiert.

Im Endeffekt werden diese beiden Konzepte in erster Linie anhand ihrer Zweckentsprechung und nicht anhand expliziter Sprachmechanismen unterschieden. Aber auch wenn der Unterschied zwischen Aggregation und Assoziation in der Struktur beim Kompilieren nur schwer zu erkennen sein mag, so ist er doch

bedeutsam. Aggregationsbeziehungen treten in der Regel lediglich in geringer Zahl in Erscheinung und sind beständiger als Assoziationsbeziehungen. Diese werden hingegen häufiger erstellt, wieder aufgelöst und neu erstellt und existieren manchmal nur für die Dauer einer Operation. Außerdem sind Assoziationsbeziehungen dynamischer, wodurch sie im Quelltext schwieriger wahrzunehmen sind.

Angesichts dieser Ungleichheit zwischen den Strukturen zur Laufzeit und beim Kompilieren eines Programms wird deutlich, dass der Code keineswegs alles über die Funktions- bzw. Arbeitsweise eines Systems preisgibt. Die Laufzeitstruktur des Systems muss eher vom Entwickler statt von der Programmiersprache vorgegeben sein. Und auch die Beziehungen zwischen Objekten und deren Typen müssen sehr sorgfältig ausgestaltet werden, weil sie bestimmen, wie gut oder schlecht die Laufzeitstruktur ist.

Viele Design Patterns (insbesondere objektbasierte) treffen eine explizite Unterscheidung zwischen den Strukturen beim Kompilieren und zur Laufzeit. Die Patterns *Composite* (*Kompositum*, siehe Abschnitt 4.3) und *Decorator* (*Dekorierer*, siehe Abschnitt 4.4) dienen insbesondere der Errichtung komplexer Laufzeitstrukturen. Im *Observer* (*Beobachter*, siehe Abschnitt 5.7) finden sich Laufzeitstrukturen, die oftmals nur schwer zu verstehen sind, sofern man sich nicht ausreichend mit dem Pattern selbst auskennt. Und das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1) fördert Kommunikationsmuster zutage, die im Rahmen der Vererbung nicht in Erscheinung treten. Generell gilt also, dass die Laufzeitstrukturen ohne eine umfassende Kenntnis des verwendeten Patterns nicht aus dem Quelltext ersichtlich sind.

1.6.7 Designänderungen berücksichtigen

Ein Höchstmaß an Wiederverwendbarkeit lässt sich nur durch das vorausschauende Einkalkulieren zukünftiger und/oder wechselnder Anforderungen erzielen, so dass sich das Systemdesign bei Bedarf fortentwickeln kann.

Damit gewährleistet ist, dass ein System auch unter veränderten Bedingungen zuverlässig funktioniert, muss berücksichtigt werden, welche Anforderungen und Umstände im Laufe seines Lebenszyklus möglicherweise variieren könnten – denn ein Design, bei dem solche Überlegungen außer Acht gelassen werden, läuft stets Gefahr, relativ unvermittelt umfassend revidiert werden zu müssen. So könnten veränderte Rahmenbedingungen beispielsweise eine Neudefinition und Reimplementierung von Klassen, eine clientseitige Modifizierung oder auch erneute Testläufe erforderlich machen. Und die daraus resultierenden Designrevisionen betreffen im Allgemeinen viele Bestandteile eines Softwaresystems, so dass unerwartete Anpassungen an veränderte Gegebenheiten unweigerlich mit einem hohen (Kosten-)Aufwand verbunden sind.

Design Patterns tragen dazu bei, solche Situationen zu vermeiden, indem sie eine gewisse Anpassungsfähigkeit des Systems sicherstellen: Jedes Design Pattern ver-

schafft einem Bestandteil der Systemstruktur mehr Spielraum für Variabilität, ohne dass andere Aspekte des Systems in Mitleidenschaft gezogen würden, und sorgt damit zugleich dafür, dass das System beim Eintreten entsprechender Veränderungen dennoch stabil bleibt.

Die folgenden Beispiele zeigen ein paar allgemeine Szenarien auf, die normalerweise eine Designrevision erforderlich machen würden, denen jedoch mit den jeweils genannten Design Patterns entgegengewirkt werden kann:

1. *Erzeugung eines Objekts durch explizite Klassenspezifizierung.* Die Spezifizierung eines Klassennamens während der Erstellung eines Objekts geht immer mit der Festlegung auf eine spezifische Implementierung statt auf eine spezifische Schnittstelle einher. Solche Verbindlichkeiten können im Fall zukünftiger Änderungen allerdings Komplikationen hervorrufen. Um dies zu vermeiden, sollte die Objekterzeugung deshalb lieber von vornherein indirekt erfolgen.

   Design Patterns: *Abstract Factory* (*Abstrakte Fabrik*, s. Abschnitt 3.1), *Factory Method* (*Fabrikmethode*, s. Abschnitt 3.3), *Prototype* (*Prototyp*, s. Abschnitt 3.4).

2. *Abhängigkeit von spezifischen Operationen.* Bei der Spezifikation einer bestimmten Operation wird genau eine Möglichkeit festgelegt, einen Request zu beantworten. Die Vermeidung von hartkodierten Requests erleichtert hier die Beantwortung von Requests sowohl beim Kompilieren als auch zur Laufzeit.

   Design Patterns: *Chain of Responsibility* (*Zuständigkeitskette*, s. Abschnitt 5.1), *Command* (*Befehl*, s. Abschnitt 5.2).

3. *Abhängigkeit von Hardware- und Softwareplattform.* Externe Betriebssystem- und Anwendungsprogrammierschnittstellen (APIs) funktionieren auf verschiedenen Hard- und Softwareplattformen jeweils unterschiedlich. Von einer bestimmten Plattform abhängige Software lässt sich nur schwer auf andere Plattformen portieren – mitunter bereitet schon die Aktualisierung auf der nativen Plattform Probleme. Deshalb sollte bereits beim Designentwurf darauf geachtet werden, dass ein System möglichst wenige Plattformabhängigkeiten aufweist.

   Design Patterns: *Abstract Factory* (*Abstrakte Fabrik*, s. Abschnitt 3.1), *Bridge* (*Brücke*, s. Abschnitt 4.2).

4. *Abhängigkeit von Objektrepräsentationen oder -implementierungen.* Clients, denen die Art der Repräsentation, Speicherung, Lokalisierung und Implementierung eines Objekts bekannt ist, müssen im Fall einer Objektänderung gegebenenfalls entsprechend angepasst werden. Um eine kaskadenförmige Anhäufung der Änderungsmaßnahmen zu verhindern, sollten ihnen diese Informationen vorenthalten werden.

   Design Patterns: *Abstract Factory* (*Abstrakte Fabrik*, s. Abschnitt 3.1), *Bridge* (*Brücke*, s. Abschnitt 4.2), *Memento* (*Memento*, s. Abschnitt 5.6), *Proxy* (*Proxy*, s. Abschnitt 4.7).

5. *Abhängigkeit von Algorithmen.* Algorithmen werden im Rahmen der Entwicklung und Wiederverwendung häufig erweitert, optimiert oder ersetzt. Und jede Änderung an einem Algorithmus macht auch eine Anpassung der von ihm abhängigen Objekte erforderlich. Aus diesem Grund sollten Algorithmen, die aller Wahrscheinlichkeit nach öfter modifiziert werden müssen, isoliert werden.

   Design Patterns: *Builder* (*Erbauer*, s. Abschnitt 3.2), *Iterator* (*Iterator*, s. Abschnitt 5.4), *Strategy* (*Strategie*, s. Abschnitt 5.9), *Template Method* (*Schablonenmethode*, s. Abschnitt 5.10), *Visitor* (*Besucher*, s. Abschnitt 5.11).

6. *Enge Kopplung.* Eng miteinander verknüpfte Klassen lassen sich aufgrund ihrer gegenseitigen Abhängigkeit nur mit Mühe separat wiederverwenden. Solche engen Kopplungen haben monolithische Systeme zur Folge, in denen sich eine Klasse ohne genaue Kenntnis ihrer Beschaffenheit oder die gleichzeitige Anpassung vieler weiterer Klassen nicht modifizieren oder entfernen lässt. Das System wird zu einem dichten Geflecht, das nur schwer zu durchschauen, zu portieren und zu warten ist.

   Lose Kopplungen erhöhen dagegen die Wahrscheinlichkeit, dass eine Klasse eigenständig wiederverwendet und das System einfacher erlernt, portiert, modifiziert und erweitert werden kann. Design Patterns nutzen Techniken wie die abstrakte Kopplung und Schichtenmodelle, um lose verknüpfte Systeme zu fördern.

   Design Patterns: *Abstract Factory* (*Abstrakte Fabrik*, s. Abschnitt 3.1), *Bridge* (*Brücke*, s. Abschnitt 4.2), *Chain of Responsibility* (*Zuständigkeitskette*, s. Abschnitt 5.1), *Command* (*Befehl*, s. Abschnitt 5.2), *Facade* (*Fassade*, s. Abschnitt 4.5), *Mediator* (*Vermittler*, s. Abschnitt 5.5), *Observer* (*Beobachter*, s. Abschnitt 5.7).

7. *Funktionalitätserweiterung durch Unterklassenbildung.* Die individuelle Objektgestaltung mithilfe der Unterklassenbildung ist oft nicht einfach. Jede neue Klasse geht mit einem fest vorgegebenen Implementierungsaufwand (Initialisierung, Finalisierung usw.) einher, und die Definition einer Unterklasse bedingt ein umfassendes Verständnis von der Beschaffenheit der Basisklasse. So könnte das Überschreiben einer Operation beispielweise das Überschreiben einer anderen erfordern. Oder eine überschriebene Operation muss eine geerbte Operation aufrufen. Außerdem kann die Unterklassenbildung zu einer regelrechten Klassenexplosion führen, weil selbst eine einfache Erweiterung die Erstellung zahlreicher neuer Unterklassen nach sich zieht.

   Die Objektkomposition im Allgemeinen sowie die Delegation im Besonderen stellen in Bezug auf die Verhaltenskomposition flexible Alternativen zum Vererbungsprinzip dar. Um eine Anwendung mit neuer Funktionalität auszustatten, können statt der Definition neuer, auf bereits existierenden Klassen basierender Unterklassen auch neuartige Zusammenstellungen bereits vorhandener Objekte verwendet werden. Andererseits kann der intensive Einsatz der Objektkomposition jedoch dazu führen, dass das Design schwerer zu durchschauen ist. Viele Patterns begünstigen daher Softwaredesigns, die sich

schlicht durch die Definition einer einzigen Unterklasse und deren Instanzenbildung auf der Grundlage existierender Objekte mit individueller Funktionalität ausstatten lassen.

Design Patterns: *Bridge* (*Brücke*, s. Abschnitt 4.2), *Chain of Responsibility* (*Zuständigkeitskette*, s. Abschnitt 5.1), *Composite* (*Kompositum*, s. Abschnitt 4.3), *Decorator* (*Dekorierer*, s. Abschnitt 4.4), *Observer* (*Beobachter*, s. Abschnitt 5.7), *Strategy* (*Strategie*, s. Abschnitt 5.9).

8. *Unbequeme Klassenmodifikation*. Manche Klassen lassen sich nicht ganz so einfach modifizieren, etwa wenn dazu auf den Quelltext zurückgegriffen werden muss, der aber leider nicht verfügbar ist (was beispielsweise bei einer kommerziellen Klassenbibliothek der Fall sein kann). Oder wenn bei jeder Änderung an der Klasse gleichzeitig auch mehrere dazugehörige Unterklassen angepasst werden müssen. Design Patterns bieten hier komfortable Möglichkeiten, um die Modifikationen auch unter derartig widrigen Umständen bequem durchführen zu können.

   Design Patterns: *Adapter* (*Adapter*, s. Abschnitt 4.1), *Decorator* (*Dekorierer*, s. Abschnitt 4.4), *Visitor* (*Besucher*, s. Abschnitt 5.11).

Die vorgenannten Fälle demonstrieren, wie sich die Softwareentwicklung mithilfe von Design Patterns flexibler gestalten lässt. Welchen Stellenwert diese höhere Flexibilität hat, hängt dabei natürlich von der Art der entwickelten Software ab. Im Folgenden wird die Rolle der Patterns in der Entwicklungsarbeit am Beispiel dreier sehr umfassender Softwaregattungen veranschaulicht: *Anwendungsprogramme*, *Toolkits* und *Frameworks*.

Anwendungsprogramme

Bei der Entwicklung eines **Anwendungsprogramms**, beispielsweise eines Texteditors oder einer Tabellenkalkulation, haben die *interne* Wiederverwendbarkeit, die Wartbarkeit sowie die Erweiterbarkeit hohe Priorität. Die interne Wiederverwendbarkeit gewährleistet, dass nicht mehr neu entwickelt und implementiert werden muss als nötig. Design Patterns, die eine Reduzierung der Abhängigkeiten sicherstellen, können die interne Wiederverwendung begünstigen. Und losere Kopplungen erhöhen die Wahrscheinlichkeit, dass eine Objektklasse mit mehreren anderen zusammenarbeiten kann. Werden beispielsweise die Abhängigkeiten von bestimmten Operationen durch die Isolierung und Kapselung jeder einzelnen Operation eliminiert, erleichtert dies die Wiederverwendung der Operation in unterschiedlichen Kontexten. Das Gleiche gilt für die Entfernung algorithmischer und repräsentativer Abhängigkeiten.

In ähnlicher Weise können Design Patterns auch zur Wartbarkeit einer Anwendung beitragen, wenn sie zur Einschränkung von Plattformabhängigkeiten sowie zur Schichtenbildung im System genutzt werden. Ebenso verbessern sie die Erweiterbarkeit, indem sie aufzeigen, wie sich Klassenhierarchien ausdehnen las-

sen und in welcher Form die Objektkomposition instrumentalisiert werden kann. Und auch die reduzierte Kopplung ist der Erweiterbarkeit zuträglich, denn eine isolierte Klasse lässt sich einfacher erweitern, wenn sie nicht von zahlreichen anderen Klassen abhängig ist.

Toolkits

Oftmals werden in einer Anwendung Klassen verwendet, die aus einer oder mehreren Bibliotheken mit vordefinierten Klassen stammen, sogenannten **Toolkits**. Ein Toolkit besteht aus einem Satz verwandter und wiederverwendbarer Klassen, die nützliche, allgemein einsetzbare Funktionalität zur Verfügung stellen. Ein Beispiel für ein solches Toolkit wäre eine Reihe von Behälterklassen für Listen, assoziative Tabellen, Stacks und Ähnliches. Die **I/O-Streambibliothek für C++** ist ein weiteres Beispiel. Toolkits zwingen einer Anwendung kein bestimmtes Design auf, sondern stellen lediglich eine Funktionalität bereit, die der Anwendung bei der Erfüllung ihrer Aufgaben behilflich sein kann, so dass der Entwickler die grundsätzliche Funktionalität nicht jedes Mal erneut programmieren muss. Toolkits stellen die *Wiederverwendung von Code* in den Vordergrund. Sie sind das objektorientierte Gegenstück zu den Programmbibliotheken.

Das Toolkit-Design ist gegenüber dem Anwendungsdesign unter Umständen allerdings vergleichsweise schwieriger zu bewerkstelligen, weil Toolkits in vielen Anwendungen einsetzbar sein müssen, um tatsächlich von Nutzen zu sein. Erschwerend hinzu kommt auch, dass ein Toolkit-Autor nicht bereits im Voraus wissen kann, wie genau diese Anwendungen aussehen werden oder welche speziellen Anforderungen sie stellen. Deshalb ist es umso wichtiger, Annahmen und Abhängigkeiten, die die Flexibilität des Toolkits und somit auch dessen Anwendbarkeit und Effektivität einschränken könnten, zu vermeiden.

Frameworks

Ein **Framework** ist eine Sammlung kooperierender Klassen, die ein wiederverwendbares Design für eine spezifische Softwaregattung bilden [Deu89, JF88]. So kann ein Framework beispielsweise speziell für die Erstellung von Grafikeditoren für verschiedene Anwendungsbereiche wie das künstlerische Zeichnen, die Musikkomposition oder mechanisches CAD gedacht sein [VL90, Joh92]. Ein anderes Framework könnte dagegen die Entwicklung von Compilern für verschiedene Programmiersprachen und Zielsysteme stützen [JML92]. Und wieder ein anderes Framework könnte auf die Erzeugung von Anwendungen für Finanzmodelle abzielen [BE93]. Die individuelle Ausrichtung eines Frameworks auf eine bestimmte Anwendung erfolgt durch die Bildung anwendungsspezifischer Unterklassen der abstrakten Framework-Klassen.

Das Framework diktiert die Architektur der Anwendung. Es definiert ihre übergeordnete Struktur, die Klassen- und Objektunterteilung, deren Hauptzuständigkei-

ten, die Art und Weise, wie die Klassen und Objekte zusammenarbeiten, sowie den Programmablauf. Im Prinzip liefert es also eine Vorabdefinition dieser Designparameter, so dass sich der Entwickler auf die detaillierte Ausgestaltung der Anwendung konzentrieren kann. Frameworks erfassen sämtliche für den Einsatzbereich der jeweiligen Anwendung allgemein gültigen Designentscheidungen – und legen damit mehr Gewicht auf die *Wiederverwendung des Designs* als auf die Wiederverwendung von Code, wenngleich sie generell auch konkrete Unterklassen enthalten, die unmittelbar genutzt werden können.

Die Wiederverwendung auf dieser Ebene führt zu einer Kontrollumkehr zwischen der Anwendung und der Software, auf der sie basiert. Beim Einsatz eines *Toolkits* (oder auch einer herkömmlichen Programmbibliothek) wird der Hauptteil der Anwendung individuell geschrieben und der wiederzuverwendende Code aufgerufen. Beim Einsatz eines *Frameworks* wird hingegen der Hauptteil wiederverwendet und der Code, den das *Framework* aufruft, selbst geschrieben. Es müssen also zwar Operationen mit bestimmten Bezeichnungen und Aufrufkonventionen erstellt werden, dennoch sind insgesamt weniger Designentscheidungen zu treffen.

Dadurch lassen sich Anwendungen nicht nur schneller erstellen, sondern weisen auch stets ähnliche Strukturen auf – was wiederum zu einer leichteren Wartbarkeit und einer größeren Konsistenz führt. Ein wenig kreative Freiheit geht bei dieser Verfahrensweise aber natürlich schon verloren, weil ja viele Designentscheidungen bereits im Voraus getroffen wurden.

Während die Entwicklung von Anwendungen schon schwierig und die von Toolkits noch schwieriger ist, ist die Erstellung von Frameworks am schwierigsten zu bewerkstelligen. Ein Framework-Entwickler setzt darauf, dass eine einzige Architektur für alle Applikationen eines bestimmten Anwendungsbereichs geeignet ist. Daraus folgt aber auch, dass jede tiefgreifende Änderung an einem Framework-Design dessen Vorzüge erheblich mindert, weil der wichtigste Beitrag, den ein Framework für die Anwendung leistet, nun mal die in ihm definierte Architektur ist. Aus diesem Grund ist eine möglichst flexible und erweiterbare Gestaltung des Frameworks ein Muss.

Weil das Anwendungsdesign in so hohem Maße von dem Framework abhängig ist, sind Modifikationen an den Framework-Schnittstellen ganz besonders heikel. Anwendungen müssen sich der Weiterentwicklung eines Frameworks anpassen und sich entsprechend mit fortentwickeln. In dieser Hinsicht kommt den losen Kopplungen eine besondere Bedeutung zu, denn ohne sie hätten schon lediglich geringfügige Modifikationen am Framework erhebliche Auswirkungen zur Folge.

Die vorstehend beschriebenen Designprobleme spielen für das Framework-Design eine äußerst wichtige Rolle. Ein Framework, in dem diese Faktoren durch den Einsatz entsprechender Design Patterns gebührend berücksichtigt werden, wird in Bezug auf die Design- und Codewiederverwendung in aller Regel ein deutlich höheres Niveau erreichen als eins, das dies nicht tut. Ausgereifte Frameworks

machen sich normalerweise gleich mehrere Design Patterns zunutze, die die Eignung ihrer Architektur für möglichst viele verschiedene Anwendungen sicherstellen, ohne dass eine Designrevision erforderlich wird.

Ein zusätzlicher Vorteil kommt zum Tragen, wenn die verwendeten Design Patterns auch in der zum Framework gehörigen Dokumentation beschrieben sind [BJ94]. Dadurch erschließt sich die genaue Beschaffenheit des Frameworks deutlich leichter, so dass selbst Entwickler, denen die Patterns zuvor noch nicht geläufig waren, von der Struktur profitieren können, die sie der Framework-Dokumentation verleihen. Eine fortlaufende Verbesserung der zugehörigen Dokumentationen ist generell für alle Softwaregattungen von Bedeutung – ganz besonders gilt dies jedoch für Frameworks, da diese häufig mit einer recht steil ansteigenden Lernkurve einhergehen, die zunächst einmal bewältigt werden will, bevor eine sinnvolle Nutzung möglich ist. Natürlich sind auch Design Patterns nicht in der Lage, diese Lernkurve vollständig zu entschärfen, sie können sie aber durch die ausdrückliche Betonung der Schlüsselelemente des Framework-Designs erheblich abflachen.

Angesichts der Tatsache, dass Design Patterns und Frameworks viele Ähnlichkeiten aufweisen, stellen sich Entwickler nicht selten die Frage, worin bzw. ob sie sich überhaupt unterscheiden. Grundsätzlich gibt es drei wesentliche Unterscheidungsmerkmale:

1. *Design Patterns sind abstrakter als Frameworks.* Während sich Frameworks als gebrauchsfertiger Code ausdrücken lassen, können Patterns lediglich in Form von *Codebeispielen* dargestellt werden. Eine Stärke der Frameworks besteht darin, dass sie in den gewünschten Programmiersprachen geschrieben und somit nicht nur gelesen, sondern auch unmittelbar ausgeführt und wiederverwendet werden können. Die in diesem Buch vorgestellten Design Patterns haben im Gegensatz dazu hauptsächlich die Aufgabe, den Zweck, die Vor- und Nachteile sowie die Auswirkungen eines Designs zu veranschaulichen und müssen für jeden Einsatz individuell implementiert werden.
2. *Design Patterns sind kleinere architektonische Elemente als Frameworks.* Ein typisches Framework kann mehrere Design Patterns umfassen, der umgekehrte Fall ist jedoch nicht möglich.
3. *Design Patterns weisen eine geringere Spezialisierung auf als Frameworks.* Frameworks zielen immer auf einen bestimmten Anwendungsbereich ab. So könnte ein Framework für Grafikeditoren zwar *in* einer Fertigungssimulation eingesetzt werden, aber nicht *als* Simulationsframework. Im Gegensatz dazu können die Design Patterns aus diesem Katalog für nahezu alle Arten von Anwendungen genutzt werden. Natürlich gibt es darüber hinaus auch weitaus spezialisiertere Patterns (beispielsweise für verteilte Systeme oder parallele Programmierung), doch selbst diese schreiben die Anwendungsarchitektur nicht in dem Maße vor, wie es ein Framework tut.

Frameworks gewinnen mehr und mehr an Bedeutung und werden immer alltäglicher. Sie sind das Mittel der Wahl, wenn es darum geht, das größtmögliche Wiederverwendungspotenzial in objektorientierten Systemen auszuschöpfen. Umfassendere und aufwendigere objektorientierte Anwendungen werden zukünftig zunehmend aus mehreren miteinander kooperierenden Framework-Schichten bestehen – und auch deren Design und Code wird in weiten Teilen den verwendeten Frameworks entstammen oder zumindest von ihnen beeinflusst sein.

1.7 Auswahl eines Design Patterns

Da der in diesem Buch vorgestellte Katalog mehr als 20 Design Patterns enthält, kann es mitunter schwer fallen, genau das Pattern zu finden, das für ein bestimmtes Designproblem geeignet ist – dies gilt insbesondere für Entwickler, die den Katalog vorher noch nicht kannten und dementsprechend auch nicht mit ihm vertraut sind. Ein wenig Hilfestellung bieten diesbezüglich die nachfolgend beschriebenen Ansätze:

- *Erwägen Sie, in welcher Art und Weise Design Patterns Designprobleme lösen.* Abschnitt 1.6 erläutert, inwiefern die Design Patterns Ihnen behilflich sein können, passende Objekte zu finden, die Objektgranularität zu bestimmen und Objektschnittstellen zu spezifizieren. Darüber hinaus werden aber auch diverse andere Möglichkeiten aufgezeigt, wie sie sich zur Lösung von Designproblemen anwenden lassen. Diese Informationen können sich bei Ihrer Suche nach dem richtigen Pattern als sehr hilfreich erweisen.
- *Machen Sie sich mit dem Zweck der Design Patterns vertraut.* Abschnitt 1.4 enthält kompakte Beschreibungen der Zielsetzungen aller im Katalog enthaltenen Design Patterns, die Ihnen das Auffinden der für eine bestimmte Problemstellung relevanten Patterns erleichtern. Nutzen Sie das Klassifizierungsschema aus Tabelle 1.1, um Ihre Suche einzugrenzen.
- *Beachten Sie, in welcher Beziehung die Patterns zueinander stehen.* Die in Abbildung 1.2 dargestellte Grafik verdeutlicht die verwandtschaftlichen Beziehungen zwischen den einzelnen Design Patterns und bietet wertvolle Hilfestellung, um die richtigen Patterns bzw. den richtigen Patterns-Satz für die jeweils vorliegende Problematik zu finden.
- *Untersuchen Sie Design Patterns, die ähnliche Zwecke erfüllen.* Der Katalog (siehe Kapitel 3) ist in drei Kategorien unterteilt: die **Erzeugungsmuster**, die **Strukturmuster** und die **Verhaltensmuster**. Jeder Kategorieteil beginnt mit einer Einleitung zu den behandelten Design Patterns und schließt mit einer vergleichenden Übersicht, aus der sowohl die Ähnlichkeiten als auch die Unterschiede zwischen den einzelnen Patterns ersichtlich sind.
- *Ermitteln Sie die Gründe für eine Um- bzw. Neugestaltung des Designs.* Vergleichen Sie das vorliegende Designproblem mit den im Abschnitt 1.6.7 beschriebenen

häufigen Ursachen für Designrevisionen und prüfen Sie, ob die hier vorgeschlagenen Design Patterns Ihnen bei der Vermeidung einer Um- bzw. Neugestaltung des Designs von Nutzen sein können.

- *Prüfen Sie, welche Elemente in Ihrem Design variabel sein sollten*. Bei diesem Ansatz wird der entgegengesetzte Weg zur Ermittlung der Gründe für eine Designrevision eingeschlagen: Statt zu berücksichtigen, wodurch eine Designänderung *erzwungen* werden könnte, steht hierbei die Überlegung im Vordergrund, was geändert werden kann, *ohne* dass gleich das ganze Design revidiert werden muss. In diesem Fall liegt der Fokus auf der *Kapselung des variierenden Konzepts*, einem zahlreichen Design Patterns zugrunde liegenden Leitmotiv. Tabelle 1.2 enthält eine Auflistung der Designaspekte, die mithilfe von Design Patterns unabhängig voneinander variiert und somit modifiziert werden können, ohne dass dafür eine Um- bzw. Neugestaltung des Designs erforderlich ist.

Zweck	Design Pattern	Variierbare Aspekte
Erzeugungsmuster (Creational Patterns)	*Abstract Factory* (*Abstrakte Fabrik*, siehe Abschnitt 3.1)	Produktobjektfamilien
	Builder (*Erbauer*, siehe Abschnitt 3.2)	Erzeugung eines zusammengesetzten Objekts
	Factory Method (*Fabrikmethode*, siehe Abschnitt 3.3)	Unterklasse eines instanziierten Objekts
	Prototype (*Prototyp*, siehe Abschnitt 3.4)	Klasse eines instanziierten Objekts
	Singleton (*Singleton*, siehe Abschnitt 3.5)	Einzige Instanz einer Klasse
Strukturmuster (Structural Patterns)	*Adapter* (*Adapter*, siehe Abschnitt 4.1)	Schnittstelle zu einem Objekt
	Bridge (*Brücke*, siehe Abschnitt 4.2)	Implementierung eines Objekts
	Composite (*Kompositum*, siehe Abschnitt 4.3)	Struktur und Komposition eines Objekts
	Decorator (*Dekorierer*, siehe Abschnitt 4.4)	Zuständigkeiten eines Objekts ohne Unterklassenbildung
	Facade (*Fassade*, siehe Abschnitt 4.5)	Schnittstelle zu einem Subsystem
	Flyweight (*Fliegengewicht*, siehe Abschnitt 4.6)	Speicherkosten für Objekte
	Proxy (*Proxy*, siehe Abschnitt 4.7)	Zugriff auf ein Objekt sowie dessen Speicherort

Tabelle 1.2: Mithilfe von Design Patterns variierbare Designaspekte

Zweck	Design Pattern	Variierbare Aspekte
Verhaltensmuster (Behavioral Patterns)	*Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1)	Objekt, das einen Request bearbeiten kann
	Command (*Befehl*, siehe Abschnitt 5.2)	Zeitpunkt und Art der Ausführung eines Requests
	Interpreter (*Interpreter*, siehe Abschnitt 5.3)	Grammatik und Interpretation einer Sprache
	Iterator (*Iterator*, siehe Abschnitt 5.4)	Zugriff auf und Traversierung von Elementen eines Aggregats
	Mediator (*Vermittler*, siehe Abschnitt 5.5)	Interagierende Objekte sowie Art und Weise der Interaktion
	Memento (*Memento*, siehe Abschnitt 5.6)	Art und Zeitpunkt der Speicherung privater Informationen außerhalb eines Objekts
	Observer (*Beobachter*, siehe Abschnitt 5.7)	Anzahl der von einem anderen Objekt abhängigen Objekte sowie Aktualisierungsart der abhängigen Objekte
	State (*Zustand*, siehe Abschnitt 5.8)	Zustände eines Objekts
	Strategy (*Strategie*, siehe Abschnitt 5.9)	Ein Algorithmus
	Template Method (*Schablonenmethode*, siehe Abschnitt 5.10)	Schritte eines Algorithmus
	Visitor (*Besucher*, siehe Abschnitt 5.11)	Ohne Klassenänderung auf Objekte anwendbare Operationen

Tabelle 1.2: Mithilfe von Design Patterns variierbare Designaspekte (Forts.)

1.8 Anwendung eines Design Patterns

Nach der Auswahl eines Patterns stellt sich die Frage, wie es benutzt werden kann. Die nachfolgende Schritt-für-Schritt-Anleitung beschreibt, wie Sie ein Design Pattern effizient anwenden können:

1. *Lesen Sie die Beschreibung des Design Patterns, um sich einen Gesamtüberblick zu verschaffen*. Achten Sie dabei insbesondere auf die Ausführungen zur »Anwendbarkeit« und den »Konsequenzen«, damit gewährleistet ist, dass das Pattern auch tatsächlich für das Ihnen vorliegende Problem geeignet ist.
2. *Überprüfen Sie die Angaben zur »Struktur«, den »Teilnehmern« und den »Interaktionen«*. Machen Sie sich mit den Klassen und Objekten vertraut und ergründen Sie, in welcher Beziehung sie zueinander stehen.

3. *Machen Sie sich den »Beispielcode« zunutze.* Das unter dieser Überschrift präsentierte konkrete Beispiel leistet Ihnen Hilfestellung bei der Implementierung des gewählten Design Patterns im Quelltext.
4. *Geben Sie den Patterns-Teilnehmern aussagefähige Namen mit einem Bezug zum Anwendungskontext.* Die am Design Patterns beteiligten Objekte oder Klassen (Teilnehmer) sind normalerweise zu abstrakt benannt, als dass sie direkt in einer Anwendung verwendet werden könnten. Dennoch ist es sinnvoll, dass sich die Teilnehmernamen in den Bezeichnungen widerspiegeln, die auch in der Anwendung erscheinen, damit das Pattern in der Implementierung leichter zu erkennen ist. Wenn Sie also beispielsweise das Design Pattern *Strategy* (*Strategie*, siehe Abschnitt 5.9) für einen Textformatierungsalgorithmus nutzen, könnten Sie die zugehörigen Klassen mit `SimpleLayoutStrategy` oder `TeXLayoutStrategy` benennen.
5. *Definieren Sie die Klassen.* Deklarieren Sie ihre Schnittstellen, richten Sie ihre Vererbungsbeziehungen ein und definieren Sie die Instanzvariablen, die die Daten- und Objektreferenzen repräsentieren. Identifizieren Sie die in Ihrer Anwendung existierenden Klassen, auf die sich das Pattern auswirken wird, und modifizieren Sie sie entsprechend.
6. *Definieren Sie anwendungsspezifische Bezeichnungen für Operationen in dem Design Pattern.* Auch in diesem Fall ist die Namensgebung im Prinzip von der Anwendung abhängig. Orientieren Sie sich dabei an den mit den einzelnen Operationen verknüpften Zuständigkeiten und Interaktionen. Achten Sie außerdem darauf, Ihre Namenskonventionen konsequent und konsistent anzuwenden, indem Sie zum Beispiel durchgängig das Präfix `Create` nutzen, um das Pattern *Factory Method* (*Fabrikmethode*, siehe Abschnitt 3.3) anzuzeigen.
7. *Implementieren Sie die Operationen zur Umsetzung der Zuständigkeiten und Interaktionen im Design Pattern.* Die Ausführungen zur »Implementierung« der einzelnen Design Patterns liefern Ihnen ebenso wie der jeweils zugehörige »Beispielcode« wertvolle Hinweise zur Durchführung des Implementierungsvorgangs.

Die vorgenannten Richtlinien sollen Ihnen lediglich den Einstieg in die Arbeit mit den Design Patterns erleichtern, bis Sie im Laufe der Zeit Ihre eigene, individuelle Verfahrensweise entwickelt haben.

Aber natürlich wären unsere Erläuterungen zum Umgang mit den Design Patterns nicht vollständig, wenn wir Ihnen nicht auch ein paar warnende Worte dazu mit auf den Weg geben würden, wie Sie sie *nicht* einsetzen sollten. Generell gilt, dass Patterns nicht wahllos angewendet werden sollten. Ihre Flexibilität und Variabilität wird häufig erst durch die Ergänzung zusätzlicher Dereferenzierungen erreicht, die das Design um einiges komplexer machen oder zu Performanceeinbußen führen können. Deshalb empfiehlt es sich, ein Pattern wirklich nur dann anzuwenden, wenn die dadurch erreichte Flexibilität auch tatsächlich benötigt wird. Hilfestellung beim Abwägen der Vor- und Nachteile sowie zur Wirkweise eines Patterns finden Sie im jeweils zugehörigen »Konsequenzen«-Abschnitt.