Bernhard Lahres, Gregor Raýman

Objektorientierte Programmierung

Das umfassende Handbuch

Galileo Press

Auf einen Blick

Inhalt

In diesem Kapitel stellen die Autoren sich und dieses Buch vor. Sie erklären, was Objektorientierung im Bereich der Softwareentwicklung bedeutet. Und am Ende des Kapitels geraten sie auch schon in die erste Diskussion miteinander.

1 Einleitung

1.1 Was ist Objektorientierung?

Objektorientierung ist eine mittlerweile bewährte Methode, um die Komplexität von Softwaresystemen in den Griff zu bekommen. Die Entwicklung der Objektorientierung als Konzept der Softwaretechnik ist sehr eng mit der Entwicklung von objektorientierten Programmiersprachen verbunden.

Von Alan Kay, dem Erfinder der Sprache Smalltalk, die als die erste konsequent objektorientierte Programmiersprache gilt, wird die folgende Geschichte erzählt:

Bei einer Präsentation bei Apple Mitte der 80er-Jahre hielt ein Mitarbeiter einen Vortrag über die erste Version der neu entwickelten Programmiersprache Oberon. Diese sollte der nächste große Schritt in der Welt der objektorientierten Sprachen sein.

Da meldete sich Alan Kay zu Wort und hakte nach:

»Diese Sprache unterstützt also keine Vererbung?«
»Das ist korrekt.«
»Und sie unterstützt keine Polymorphie?«
»Das ist korrekt.«
»Und sie unterstützt auch keine Datenkapselung?«
»Das ist ebenfalls korrekt.«
»Dann scheint mir das keine objektorientierte Sprache zu sein.«

Der Vortragende meinte darauf: »Nun, wer kann schon genau sagen, was nun objektorientiert ist und was nicht.«

Woraufhin Alan Kay zurückgab: »Ich kann das. Ich bin Alan Kay, und ich habe den Begriff geprägt.«

Der geschilderte Dialog enthält genau die drei Grundelemente, die als Basis von objektorientierten Programmiersprachen gelten:

- Unterstützung von Vererbungsmechanismen
- Unterstützung von Datenkapselung
- Unterstützung von Polymorphie

Auf alle drei Punkte werden wir eingehen. Objektorientierung geht allerdings mittlerweile weit über diese Grunddefinition hinaus. Die genannten Punkte bilden lediglich den kleinsten gemeinsamen Nenner.

1.2 Hallo liebe Zielgruppe

Die Frage »Für wen schreiben wir dieses Buch?« haben wir uns beim Schreiben selbst immer wieder gestellt.

Die Antwort darauf hängt eng mit der Frage zusammen, warum wir eigentlich dieses Buch geschrieben haben. Nun, neben dem ganz offensichtlichen Grund, dass die Veröffentlichung die beiden Autoren reich und berühmt machen wird, was ja ein ganz angenehmer Nebeneffekt ist, gibt es noch einen ganz zentralen weiteren Grund: Wir wollten das Buch schreiben, das wir uns selbst an bestimmten Punkten unseres Ausbildungswegs und unserer Berufslaufbahn gewünscht hätten.

Wir haben uns beide von Anfang an im Umfeld der objektorientierten Softwareentwicklung bewegt. Allerdings unterschied sich das, was wir in der Praxis an Anforderungen vorfanden, dann doch stark von dem, was Universität und Lehrbücher vorbereitet hatten.

Theorie und Praxis

Aufgrund dieser Erfahrung hätten wir uns ein Buch gewünscht, das den Brückenschlag zwischen Theorie und Praxis bewerkstelligt. Nicht mal so sehr auf der Ebene von praktischen Programmen, sondern eher: Wie werden die theoretischen Ansätze (die wir Prinzipien nennen) denn nun umgesetzt? Was hat es mit dem Thema Garbage Collection auf sich? Was ist denn wirklich mit Model-View-Controller gemeint? Warum müssen wir uns denn überhaupt mit tiefen Kopien, flachen Kopien, Identität, Gleichheit beschäftigen?

Wir glauben, dass wir es ein Stück in diese Richtung geschafft haben und Ihnen eine interessante Verbindung aus Theorie und Praxis präsentieren.

Wer sollte unser Buch lesen?

Wir wollen Studenten der Informatik eine praktische, interessante und hoffentlich auch unterhaltsame Einführung und Vertiefung zum Thema

Objektorientierung bieten. Wir wollen Berufseinsteigern im Bereich Softwareentwicklung einen leichteren Start ermöglichen, indem wir Begriffe praktisch klären, mit denen sie regelmäßig konfrontiert werden. Wir wollen im Beruf aktiven Softwareentwicklern die Chance geben, einfach noch mal darüber nachzudenken, was sie eigentlich in der täglichen Arbeit so machen und ob nicht vielleicht noch die eine oder andere Verbesserung möglich ist.

Wir wenden uns also an Softwareentwickler in Ausbildung oder im aktiven Einsatz, an Menschen, die Programme schreiben. Natürlich würden wir uns auch freuen, wenn Projektleiter und Projektmanager unser Buch in die Hand nehmen. Geschrieben haben wir es allerdings als Softwareentwickler für andere Softwareentwickler.

Was wir voraussetzen

Als Voraussetzung gehen wir davon aus, dass unsere Leser grundsätzliche Erfahrung mit der Programmierung haben. Erfahrungen mit der Programmierung in objektorientierten Sprachen sind hilfreich, obwohl nicht unbedingt notwendig. Unser Buch wendet sich damit auch explizit an Menschen, die bereits mit Programmiersprachen arbeiten, die objektorientierte Mechanismen anbieten, wie dies zum Beispiel bei Java der Fall ist. Da aber nur durch die Verwendung von objektorientierten Sprachen noch lange keine objektorientierte Software entsteht, wollen wir mit unserem Buch genau diesen Schritt ermöglichen.

Wir werden zu den verwendeten Programmiersprachen keine detaillierte Einführung geben. Stattdessen findet sich im Anhang jeweils eine Kurzbeschreibung der jeweiligen Programmiersprache. Diese erläutert die grundlegenden Sprachkonstrukte mit Bezug zur objektorientierten Programmierung. Außerdem geben wir Verweise auf weitere Informationsquellen zur Programmiersprache, sofern möglich auch mit Hinweisen auf verfügbare freie Versionen von Compilern, Interpretern oder Entwicklungsumgebungen.

1.3 Was bietet dieses Buch (und was nicht)?

Objektorientierung ist eine Vorgehensweise, die den Fokus auf modulare Systeme legt. Dieses Modulprinzip legen wir in zweierlei Hinsicht auch diesem Buch zugrunde.

Zum einen sehen wir das Buch als einen Baustein, der beim Verständnis des Themas Objektorientierung eine zentrale Rolle spielt. Wir beschreiben, wie Sie Objektorientierung einsetzen können, um auf der Grund-

lage von zentralen Prinzipien beherrschbare und änderbare Software erstellen können. Wenn wir auf diesem Weg an Entwurfsmustern vorbeikommen, werden wir diese vorstellen und ihren Nutzen für den Entwurf oder die aktuelle Problemstellung erläutern.

Zum anderen bauen die Kapitel des Buchs modular aufeinander auf. Ausgehend von abstrakten Prinzipien gehen wir immer weiter in die konkrete Anwendung der Objektorientierung. Diesen modularen Aufbau wollen wir in einer Übersicht kurz vorstellen.

1.3.1 Bausteine des Buchs

Abbildung 1.1 stellt die einzelnen Kapitel als Bausteine in der Übersicht dar.

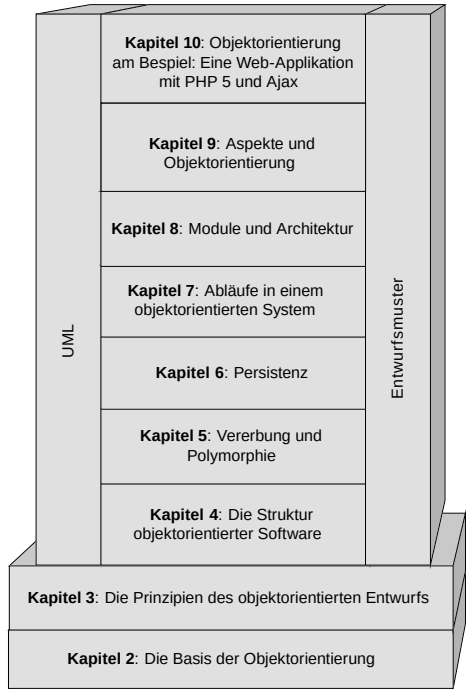


Abbildung 1.1 Modularer Aufbau der Kapitel

Kapitel 2: Basis

In Kapitel 2, »Die Basis der Objektorientierung«, stellen wir zunächst die grundlegenden Unterschiede der objektorientierten Vorgehensweise im Vergleich zur strukturierten Programmierung vor. Wir beschreiben dort auch im Überblick die Basismechanismen der Objektorientierung: Datenkapselung, Polymorphie und Vererbung.

Kapitel 3: Prinzipien

Danach schließt sich Kapitel 3, »Die Prinzipien des objektorientierten Entwurfs«, an. Das Kapitel stellt die grundlegenden Prinzipien vor, die für den objektorientierten Ansatz entscheidend sind. In Abschnitt 1.1 haben wir einen kleinen Dialog vorgestellt, in dem die grundlegenden Eigenschaften von objektorientierten Sprachen angesprochen werden. Genau wie die dort genannten Eigenschaften geben auch die Prinzipien der Objektorientierung einen Rahmen für eine bestimmte Art der Softwareentwicklung vor. Die Prinzipien sind zentral für das Vorgehen bei der Entwicklung von objektorientierter Software. Das Kapitel legt also die grundsätzliche Basis, ohne bereits detailliert auf Objekte, Klassen oder Ähnliches einzugehen.

Kapitel 4: Struktur

Anschließend erläutern wir in Kapitel 4, »Die Struktur objektorientierter Software«, das Konzept von Objekten, Klassen und der darauf aufbauenden Möglichkeiten. Wir erläutern das Konzept der Klassifizierung und betrachten die Rolle der Klassen als Module. Den verschiedenen Arten, in denen Objekte untereinander in Beziehung stehen können, ist ebenfalls ein Teilkapitel gewidmet.

Kapitel 5: Vererbung und Polymorphie

In Kapitel 5, »Vererbung und Polymorphie«, gehen wir darauf ein, wie die Vererbung der Spezifikation – im Zusammenspiel mit der Fähigkeit der Polymorphie – Programme flexibel und erweiterbar halten kann. Wir gehen dabei auf die verschiedenen Varianten der Vererbung im Detail ein und stellen deren Möglichkeiten und Probleme vor. An Beispielen erläutern wir dabei auch, wie die Möglichkeiten der Polymorphie am besten genutzt werden können.

Kapitel 6: Persistenz

Kapitel 6, »Persistenz«, beschreibt, wie Objekte in relationalen Datenbanken gespeichert werden können. In fast allen Anwendungen taucht die Notwendigkeit auf, dass Objekte persistent gespeichert werden müssen. Wir stellen die Abbildungsregeln für Vererbungsbeziehungen und andere Beziehungen zwischen Objekten und Klassen auf eine relationale Datenbank vor. Schließlich setzen wir diese Abbildungsregeln in Beziehung zu den verschiedenen Stufen der Normalisierung in einer relationalen Datenbank.

Kapitel 7: Abläufe

In Kapitel 7, »Abläufe in einem objektorientierten System«, beschreiben wir die Vorgänge innerhalb des Lebenszyklus von Objekten. Wir gehen detailliert auf die Erzeugung von Objekten ein und erläutern, wie Sie ein System erweiterbar halten, indem Sie möglichst flexible Methoden der Objekterzeugung einsetzen. Außerdem enthält das Kapitel Beschreibungen von Interaktionsszenarien, die sich häufig in objektorientierten Systemen finden. Der Abschluss des Objektlebenszyklus, der meist über den

Mechanismus der Garbage Collection stattfindet, wird ebenfalls in diesem Kapitel beschrieben.

Kapitel 8: Architektur

In Kapitel 8, »Module und Architektur«, stellen wir Beispiele dafür vor, wie objektorientierte Entwürfe in reale Systeme integriert werden. Wir stellen das Konzept von in der Praxis verwendeten Ansätzen wie Frameworks und Anwendungscontainern vor. Am Beispiel der Präsentationsschicht einer Anwendung erläutern wir, wie objektorientierte Verfahren die Interaktionen in einem System strukturieren können. Dazu stellen wir das Architekturmuster Model-View-Controller (MVC) und dessen Einsatzszenarien vor.

Kapitel 9: Aspekte

In Kapitel 9, »Aspekte und Objektorientierung«, stellen wir dar, wie sich eine Reihe von Einschränkungen der objektorientierten Vorgehensweise durch Aspektorientierung aufheben lässt. Wir erläutern, welche Beschränkungen der objektorientierten Vorgehensweise überhaupt zur Notwendigkeit einer aspektorientierten Sichtweise führen. Die Verwaltung von sogenannten übergreifenden Anliegen (engl. *Crosscutting Concerns*) ist mit den Mitteln der klassischen objektorientierten Programmierung nur sehr aufwändig zu realisieren. Bei Crosscutting Concerns handelt es sich um Anforderungen, die mit Mitteln der Objektorientierung nur klassen- oder komponentenübergreifend realisiert werden können.

Deshalb zeigen wir in diesem Kapitel verschiedene Möglichkeiten auf, Lösungen dafür in objektorientierte Systeme zu integrieren. Wir erläutern den Weg zu den aspektorientierten Lösungen und stellen diese an praktischen Beispielen vor.

Kapitel 10: Objektorientierung am Beispiel einer Web-Applikation

Abgerundet wird unser Buch dann durch Kapitel 10, »Objektorientierung am Beispiel: Eine Web-Applikation mit PHP 5 und Ajax«. Dort greifen wir am Beispiel einer Web-Anwendung eine ganze Reihe von Konzepten auf, die in den vorhergehenden Kapiteln erläutert wurden. Im Kontext einer einfachen, aber vollständigen Web-Anwendung auf Basis von PHP 5 und Ajax stellen wir Schritt für Schritt vor, wie Geschäftslogik und Präsentationsschicht durch objektorientierte Vorgehensweisen klar voneinander entkoppelt werden. Dabei gehen wir auch darauf ein, durch welchen Aufbau ein Austausch von Teilen der Präsentationsschicht erleichtert wird.

Anhang: Programmiersprachen

Im Anhang werden wir die im Buch verwendeten Programmiersprachen jeweils mit einer Kurzreferenz vorstellen und Hinweise auf weitere Informationen geben.

1.3.2 Crosscutting Concerns: übergreifende Anliegen

Die beschriebenen Kapitel bauen in Form einer Modulstruktur aufeinander auf. Aber ähnlich wie bei der Strukturierung objektorientierter Software gibt es natürlich auch hier Themen, die übergreifend sind und sich nicht genau einem der Kapitel zuordnen lassen.

Im Folgenden gehen wir kurz darauf ein, welche Rolle diese weiteren Themengebiete spielen.

Unified Modeling Language

Die Unified Modeling Language (UML) hat sich mittlerweile als ein sinnvolles und weit verbreitetes Modellierungsmittel durchgesetzt. Wir werden die nach unserer Ansicht wichtigsten Darstellungselemente der UML anhand von Beispielen vorstellen. Wir werden diese dabei immer dann einführen, wenn die betreffende Modellierung für unser aktuell behandeltes Thema relevant wird. Wir verwenden die UML in der Regel auch zur Illustration von Strukturen und Abläufen.

Objektorientierte Analyse

Die objektorientierte Analyse betrachtet eine Domäne als System von kooperierenden Objekten. Obwohl objektorientierte Analysemethoden nicht unser zentrales Thema sind, werden wir diese immer dann heranziehen, wenn wir auf die Verbindung von Analyse und Design eingehen.

Entwurfsmuster

Entwurfsmuster (engl. *Design Patterns*) für den objektorientierten Entwurf wurden mit der Publikation *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software* als Begriff geprägt. Die Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides führen dabei eine standardisierte Beschreibungsform für wiederkehrende Vorgehensweisen beim Entwurf guter objektorientierter Software ein[1]. Aufgrund von Umständen, die mit der Sortierung unseres Alphabets zusammenhängen, wird die Publikation meist mit Erich Gamma assoziiert. Da das aber nicht korrekt ist, werden wir uns mit dem Namen [Entwurfsmuster 2004] darauf beziehen, da die aktuelle deutsche Ausgabe aus dem

1 Allerdings sind auch die Autoren des Entwurfsmuster-Buchs mittlerweile nicht mehr bei allen der vorgestellten Muster davon überzeugt, dass es sich wirklich um empfehlenswerte Vorgehensweisen handelt. So hat sich beispielsweise das Entwurfsmuster »Singleton« mittlerweile einen eher fragwürdigen Ruf erarbeitet.

Jahr 2004 stammt und eine Verwechslungsgefahr mit anderen Entwurfsmusterbüchern in diesem Jahr nicht besteht.

Grundsätzlich sind solche Entwurfsmuster unabhängig davon, ob wir eine objektorientierte Vorgehensweise gewählt haben, da sie nur eine Schablone für gleichartige Problemstellungen sind. Allerdings hat sich in der Praxis herausgestellt, dass bei Verwendung von objektorientierten Methoden diese Muster einfacher zu erkennen und besser zu beschreiben sind.

In der Folge ist eine ganze Reihe von unterschiedlichen Entwurfsmustern entstanden, die meisten mit Bezug auf objektorientierte Methoden. In der Kategorie Anti-Patterns werden häufige Fehler beim Systementwurf zusammengefasst.

Wir werden eine ganze Reihe von Entwurfsmustern verwenden. Allerdings werden wir diese dann vorstellen, wenn ein Muster ein bestimmtes Thema gut illustriert oder dafür zentral ist. In diesem Fall geben wir eine kurze Vorstellung des Musters und erläutern dessen Anwendung.

Deutsche und englische Begriffe

Wir werden außerdem weitgehend deutsche Begriffe verwenden. Wenn allerdings der englische Begriff der wesentlich gängigere ist, werden wir diesen verwenden. Dies gilt auch, wenn eine deutsche Übersetzung die Bedeutung verzerren würde oder zu umständlich wird. So werden wir zum Beispiel den Begriff *Multiple Dispatch* verwenden, weil die in Frage kommenden Übersetzungen *Multiple Verteilung* oder *Verteilung auf der Grundlage von mehreren Objekten* nicht wesentlich erhellender oder aber sehr umständlich sind.

Wir werden die englische Entsprechung bei der ersten Erwähnung meist mit aufführen. So werden wir zum Beispiel das Prinzip *Offen für Erweiterung, geschlossen für Änderung* bei der ersten Erwähnung auch als *Open Closed Principle* vorstellen.

1.3.3 Die Rolle von Programmiersprachen

Die meisten der von uns diskutierten Prinzipien der Objektorientierung finden sich in den objektorientierten Programmiersprachen wieder, entweder als direkte Sprachkonstrukte oder als Möglichkeiten bei der Programmierung.

Allerdings: Die Unterstützung variiert sehr stark zwischen den einzelnen objektorientierten Sprachen. Bestimmte Prinzipien werden nur von wenigen Sprachen unterstützt, andere in unterschiedlichem Ausmaß.

Wir haben deshalb eine überschaubare Anzahl von Programmiersprachen ausgewählt, um diese jeweils dann als Beispiel heranzuziehen, wenn ein bestimmtes Prinzip besonders gut (oder vielleicht auch besonders schlecht) unterstützt wird.

Objektorientierte Softwareentwicklung lässt sich nicht beschreiben, ohne auf die Entwicklung der objektorientierten Programmiersprachen einzugehen. Durch ihre jeweils spezifischen Möglichkeiten sind Programmiersprachen selbst sehr gute Beispiele dafür, wie Konzepte der Objektorientierung in der Praxis umgesetzt werden.

Alan Perlis hatte mit seiner Aussage völlig Recht, als er schrieb: *Eine Programmiersprache, die nicht die Art beeinflusst, in der du über das Programmieren nachdenkst, ist es nicht wert, dass man sie kennt.*[2]

Deshalb werden Sie in den folgenden Kapiteln auch einiges über die Besonderheiten und speziellen Möglichkeiten von mehreren objektorientierten Sprachen erfahren. Alleine schon durch die Betrachtung der Unterschiede zwischen einzelnen Sprachen lassen sich Konzepte gut illustrieren. Jede der Sprachen hat ihre eigenen Stärken, jede macht bestimmte Sachen einfach. Jede hat ihre eigenen Idiome und Muster.

Wenn man mehrere Programmiersprachen kennt, lernt man oft neue Vorgehensweisen. Auch wenn man in der Praxis die meiste Zeit nur mit einer Programmiersprache arbeitet, lohnt es sich, auch einen Blick auf andere zu werfen. Vielleicht nur um die ausgetretenen Pfade zu verlassen und zu erkennen, dass manche Aufgaben, die man für schwierig hielt, sich in Wirklichkeit ganz einfach lösen lassen.

Hier also die Liste der Programmiersprachen, die sich in den Code-Beispielen wiederfinden:

- **Java**
  Java ist natürlich dabei, weil es mittlerweile eine sehr verbreitete Sprache ist, die einen Fokus auf Objektorientierung setzt.

2 Das Originalzitat in Englisch lautet: »A language that doesn't affect the way you think about programming, is not worth knowing.« und ist enthalten in einem Artikel von Alan Perlis in den *SIGPLAN Notices Vol. 17, No. 9, September 1982*.

- **C++**
  Auch C++ hat immer noch einen hohen Verbreitungsgrad und unterstützt die Basisprinzipien der Objektorientierung zu großen Teilen.
- **JavaScript**
  JavaScript wird hauptsächlich als Beispiel für eine objektorientierte Sprache, die mit Prototypen arbeitet, angeführt.
- **Ruby**
  Ruby ist eine Skriptsprache, die wegen ihrer einfachen und intuitiven Handhabung immer beliebter wird. Ruby hat einen sehr starken Fokus auf Objektorientierung. Außerdem sind Klassenerweiterungen (Mixins) möglich.
- **C#**
  Die von Microsoft entwickelte Sprache aus der .NET-Familie fasst eine ganze Reihe von Konzepten der Objektorientierung gut zusammen.
- **Python**
  Python ist eine interaktive objektorientierte Skriptsprache, die nach der britischen Komikertruppe Monty Python benannt ist. Python betrachtet alle im Programm vorhandenen Daten als Objekte.

Wir werden Beispiele in den verschiedenen Sprachen immer dann einbringen, wenn sich eine Sprache gerade besonders gut zur Illustration eignet. Außerdem werden wir zur Erläuterung der aspektorientierten Erweiterungen zur Objektorientierung hauptsächlich AspectJ heranziehen.

1.4 Warum überhaupt Objektorientierung?

Wir wollen hier zunächst einmal eine Aussage vorausschicken, die trivial anmuten mag: Es ist nicht einfach, gute Software zu entwickeln. Speziell die Komplexität von mittleren und großen Softwaresystemen stellt oft ein großes Problem dar.

Objektorientierte Softwareentwicklung ist nicht die einzige Methode, um diese Komplexität in den Griff zu bekommen, aber sie hat sich in verschiedenen Anwendungskontexten bewährt. Außerdem liefert sie mittlerweile ein Instrumentarium für eine ganze Reihe von Problemfeldern.

Objektorientierte Vorgehensweisen ergänzen sich außerdem gut mit einigen anderen Ansätzen in der Softwareentwicklung. Deshalb ist das

letzte Kapitel auch dem Thema Aspektorientierung gewidmet, da dieser Ansatz einige der Defizite der objektorientierten Vorgehensweise ausbügeln kann.

1.4.1 Gute Software: Was ist das eigentlich?

Je nachdem, wen wir fragen, wird die Antwort auf die Frage »Was macht gute Software für Sie aus?« unterschiedlich ausfallen.

Fragen wir doch zunächst den *Anwender* von Software. Das sind wir ja praktisch alle. Hier werden wir häufig die folgenden Anforderungen hören:

- Software soll das machen, was ich von ihr erwarte: Software muss korrekt sein.
- Software soll es mir möglichst einfach machen, meine Aufgabe zu erledigen: Software muss benutzerfreundlich sein.
- Ich möchte meine Arbeit möglichst schnell und effizient erledigen, ich möchte keine Wartezeiten haben: Software muss effizient und performant sein.

Fragen wir nun denjenigen, der für Software und Hardware *bezahlt*, kommen weitere Anforderungen hinzu:

- Ich möchte keine neue Hardware kaufen müssen, wenn ich die Software einsetze: Software muss effizient mit Ressourcen umgehen.
- Ich möchte, dass die Software möglichst günstig erstellt werden kann.
- Mein Schulungsaufwand sollte gering sein.

Fragen wir den *Projektmanager*, der die Entwicklung der Software vorantreiben soll, dann kommt noch mehr zutage:

- Meine Auftraggeber kommen oft mit neuen Ideen. Es darf nicht aufwändig sein, diese neuen Ideen auch später noch umzusetzen: Software muss sich anpassen lassen.
- Ich habe feste Zieltermine, das Management sitzt mir im Nacken. Deshalb muss ich diese Software in möglichst kurzer Zeit fertigstellen.
- Das Programm soll über Jahre hinweg verwendet werden. Ich muss Änderungen auch dann noch mit überschaubarem Aufwand umsetzen können.

Trotz der unterschiedlichen Sichtweisen kristallisieren sich hier einige zentrale Kriterien für gute Software heraus[3]:

- **Korrektheit**
  Software soll genau das tun, was von ihr erwartet wird.
- **Benutzerfreundlichkeit**
  Software soll einfach und intuitiv zu benutzen sein.
- **Effizienz**
  Software soll mit wenigen Ressourcen auskommen und gute Antwortzeiten für Anwender haben.
- **Wartbarkeit**
  Software soll mit wenig Aufwand erweiterbar und änderbar sein.

Ein Hauptfeind dieser Kriterien ist die Komplexität, die sich in der Regel bei Softwaresystemen mit zunehmender Größe aufbaut.

1.4.2 Die Rolle von Prinzipien

Wenn wir Sie motivieren wollen, dass Sie objektorientierte Methoden einsetzen sollten, müssen wir drei verschiedene Arten von Fragen stellen:

1. Welche Probleme wollen Sie eigentlich angehen? Häufig wird die Aufgabenstellung sein, die bereits genannten Kriterien wie Korrektheit, Benutzerfreundlichkeit, Effizienz und Wartbarkeit einzuhalten.
2. Welche abstrakten Prinzipien helfen Ihnen dabei? Beispiele sind hier Kapselung von Information oder klare Zuordnung von Veranwortlichkeiten zu Modulen.
3. Wie können Sie diese Prinzipien in einem Softwaresystem sinnvoll umsetzen?

Da Objektorientierung eben nur eine Methode ist, um diese Ziele umzusetzen, ist es beim Entwurf von Systemen immer wichtig, dass Sie sich bewusst sind, warum Sie einen bestimmten Entwurf gewählt haben.

Am Ende kommt es darauf an, den ursprünglichen Zielen möglichst nahe zu kommen. Sie werden dabei oft Kompromisse finden müssen. Die mit

3 Neben den aufgelisteten Kriterien gibt es noch weitere allgemeine Anforderungen an Software. Es ist von der jeweiligen Anwendung abhängig, wie zentral die jeweilige Anforderung ist. So ist für Software zur Steuerung eines Atomkraftwerks Korrektheit und Fehlertoleranz wichtiger als der effiziente Umgang mit Ressourcen.

der objektorientierten Vorgehensweise verbundenen Prinzipien sind dabei Richtlinien, diese müssen aber oft gegeneinander abgewogen werden.

Deshalb beginnen wir in Kapitel 3, »Die Prinzipien des objektorientierten Entwurfs«, auch mit der Vorstellung dieser grundlegenden Prinzipien, die der objektorientierten Softwareentwicklung zugrunde liegen. Zum überwiegenden Teil können diese völlig unabhängig von Begriffen wie Klasse oder Objekt vorgestellt werden. Erst nach der Einführung der Prinzipien werden wir also erklären, wie diese mittels Klassen und Objekten umgesetzt werden können.

1.4.3 Viele mögliche Lösungen für ein Problem

Auch wenn Sie sich einmal entschieden haben, nach dem objektorientierten Paradigma vorzugehen, werden für eine Problemstellung häufig verschiedene Lösungen möglich sein.

Obwohl sich die beiden Autoren grundsätzlich darüber einig sind, dass objektorientierte Techniken zu sinnvollen Problemlösungen führen, ergeben sich hinsichtlich der in einem konkreten Fall zu wählenden Lösung doch häufig Differenzen.

Diskussionen können hilfreich sein.

In so einem Fall werden wir die resultierende Diskussion einfach ganz offen vor unserer Leserschaft austragen und diese dabei mit Beispielen aus unseren praktischen Erfahrungen mit Objekttechnologie unterfüttern. Das wird ein ganz schön lebhaftes Buch werden, das können wir an dieser Stelle schon versprechen.

Diskussion: Was gibt's denn hier zu diskutieren?

Gregor: *Findest du das denn eine gute Idee, einfach vor unseren Leserinnen und Lesern zu diskutieren? Bestimmt erwarten sie doch, dass wir uns auf unserem Fachgebiet einig sind und auch klare Lösungen vorstellen. Ich weiß natürlich, dass wir uns wirklich nicht immer einig sind, aber wir könnten doch zumindest so tun.*

Bernhard: *Nun, wenn wir ständig diskutieren würden, wäre das sicherlich etwas irritierend. Und wir sind uns ja auch größtenteils einig, in vielen Fällen gibt es einfach auch generell anerkannte Vorgehensweisen. Aber spannend wird das Ganze erst an den Punkten, an denen das Vorgehen eben nicht mehr völlig klar ist und wir uns für eine von mehreren möglichen Vorgehensweisen entscheiden müssen.*

Gregor: *Stimmt schon, hin und wieder so eine kleine Diskussion zwischendurch war ja auch beim Schreiben des Buchs durchaus nützlich. Dann lass uns jetzt aber loslegen. Die nächste Diskussion kommt bestimmt bald.*

Icons Um Ihnen die Orientierung im Buch zu erleichtern, haben wir zwei Icons verwendet:

[»] Hier finden Sie hilfreiche Definitionen, die die wesentlichen Aspekte des Themas zusammenfassen.

[zB] Konkrete Beispiele erleichtern Ihnen das Verstehen.

Weiterführende Informationen finden Sie auf der Webseite zum Buch unter *www.objektorientierte-programmierung.de*.

Prinzipien helfen uns, das Richtige zu tun. Dies gilt in der Softwareentwicklung genauso wie in unserem täglichen Leben. Wir müssen uns allerdings auch vor Prinzipienreiterei hüten und hin und wieder ein Prinzip beiseite legen können. In diesem Kapitel beschreiben wir die grundlegenden Prinzipien, die einem guten objektorientierten Softwareentwurf zugrunde liegen und warum ihre Einhaltung meistens eine gute Idee ist.

3 Die Prinzipien des objektorientierten Entwurfs

Software ist eine komplizierte Angelegenheit. Es ist nicht einfach, menschliche Sprache zu erkennen, einen Cache effektiv zu organisieren oder eine Flugbahn in Echtzeit zu berechnen. Mit dieser Art der Komplexität, Komplexität der verwendeten Algorithmen, beschäftigen wir uns in diesem Buch jedoch nicht.

Einfache Aufgaben – komplexe Programme

Einen Text auszugeben, auf das Drücken einer Taste zu reagieren, Kundendaten aus einer Datenbank herauszulesen und sie zu bearbeiten – das sind einfache Programmieraufgaben. Und aus diesen einfachen Funktionen entstehen komplexe Programme. Unser Teufel steckt nicht im Detail, sondern in der großen Anzahl der einfachen Funktionen und der Notwendigkeit, diese zu organisieren.

Vorbereitung auf Änderungen

Außerdem ändern sich die Anforderungen an Software in der Praxis sehr häufig. Auch das macht Softwareentwicklung zu einer komplizierten Angelegenheit, da wir immer auf diese Änderungen vorbereitet sein müssen.

Es ist unsere Aufgabe – Aufgabe der Softwarearchitekten, Softwaredesigner und Softwareentwickler –, die Programme so zu organisieren, dass sie nicht nur für die Anwender, sondern auch für uns, als Beteiligte bei der Entwicklung von Software, beherrschbar werden und auch bleiben.

In diesem Kapitel stellen wir Prinzipien vor, die bei der Beherrschung der Komplexität helfen. In den darauf folgenden Kapiteln werden wir zeigen, ob und wie diese Prinzipien in der objektorientierten Program-

mierung eingehalten werden können. Allerdings: Prinzipien kann man ja nie genug haben. Die Auflistung ist deshalb nicht vollständig, sie enthält aber die wichtigsten Prinzipien.

3.1 Prinzip 1: Prinzip einer einzigen Verantwortung

Das grundsätzliche Prinzip der Komplexitätsbeherrschung und Organisation lautet: *Teile und herrsche*. Denn Software besteht aus Teilen.

In diesem Kapitel wollen wir uns nicht mit spezifisch objektorientierten Methoden beschäftigen. Deswegen werden wir hier meistens nicht spezifisch über Objekte, Klassen und Typen schreiben, sondern verwenden stattdessen den Begriff *Modul*.

[»] **Module**

Unter einem Modul versteht man einen überschaubaren und eigenständigen Teil einer Anwendung – eine Quelltextdatei, eine Gruppe von Quelltextdateien oder einen Abschnitt in einer Quelltextdatei. Etwas, was ein Programmierer als eine Einheit betrachtet, die als ein Ganzes bearbeitet und verwendet wird. Solch ein Modul hat nun innerhalb einer Anwendung eine ganz bestimmte Aufgabe, für die es die Verantwortung trägt.

[»] **Verantwortung (Responsibility) eines Moduls**

Ein Modul hat innerhalb eines Softwaresystems eine oder mehrere Aufgaben. Damit hat das Modul die Verantwortung, diese Aufgaben zu erfüllen. Wir sprechen deshalb von einer Verantwortung oder auch mehreren Verantwortungen, die das Modul übernimmt.

Module und Untermodule

Module selbst können aus weiteren Modulen zusammengesetzt sein, den *Untermodulen*. Ist ein Modul zu kompliziert, sollte es unterteilt werden. Ein mögliches Indiz dafür ist, dass der Entwickler das Modul nicht mehr gut verstehen und anpassen kann.

Besteht ein Modul aus zu vielen Untermodulen, sind also die Abhängigkeiten zwischen den Untermodulen zu komplex und nicht mehr überschaubar, sollten Sie über die Hierarchie der Teilung nachdenken. Sie können dann zusammengehörige Untermodule in einem Modul zusammenfassen oder ein neues Modul erstellen, das die Abhängigkeiten zwischen den zusammenhängenden Untermodulen koordiniert und nach außen kapselt.

Bevor wir uns die Beziehungen unter den Modulen genauer anschauen, betrachten wir zuerst die Module selbst und formulieren das Prinzip, das uns bei der Frage unterstützt, was wir denn in ein Modul aufnehmen sollen.

Prinzip einer einzigen Verantwortung (Single Responsibility Principle) [«]

Jedes Modul soll genau eine Verantwortung übernehmen, und jede Verantwortung soll genau einem Modul zugeordnet werden. Die Verantwortung bezieht sich auf die Verpflichtung des Moduls, bestimmte Anforderungen umzusetzen. Als Konsequenz gibt es dann auch nur einen einzigen Grund, warum ein Modul angepasst werden muss: Die Anforderungen, für die es verantwortlich ist, haben sich geändert. Damit lässt sich das Prinzip auch alternativ so formulieren: Ein Modul sollte nur einen einzigen, klar definierten Grund haben, aus dem es geändert werden muss.

Jedes Modul dient also einem Zweck: Es erfüllt bestimmte Anforderungen, die an die Software gestellt werden.

Code mit unklaren Aufgaben

Zumindest sollte es so sein. Viel zu oft findet man in alten, gewachsenen Anwendungen Teile von totem, nicht mehr genutztem Code, die nur deswegen noch existieren, weil einfach niemand bemerkt hat, dass sie gar keine sinnvolle Aufgabe mehr erfüllen. Noch problematischer ist es, wenn nicht erkennbar ist, welchen Zweck ein Anwendungsteil erfüllt. Es wird damit riskant und aufwändig, den entsprechenden Teil der Anwendung zu entfernen.

Vielleicht kennen Sie eine verdächtig aussehende Verpackung im Kühlschrank der Kaffeeküche? Eigentlich kann der Inhalt nicht mehr genießbar sein. Aber warum sollten Sie es wegwerfen? Sie sind doch nicht der Kühlschrankbeauftragte, und außerdem haben Sie gehört, Ihr Boss solle den schwedischen Surströmming[1] mögen – warum sollten Sie das Risiko eingehen, den Boss zu verärgern?

Sie sollten von Anfang an darauf abzielen, eine solche Situation gar nicht erst entstehen zu lassen, weder im Kühlschrank noch in der Software. Es sollte immer leicht erkennbar sein, welchem Zweck ein Softwaremodul dient und wem die Packung verdorbener Fisch im Bürokühlschrank gehört.

1 Für diejenigen, die sich im Bereich skandinavischer Spezialitäten nicht auskennen: Surströmming ist eine besondere Konservierungsmethode für Heringe. Diese werden dabei in Holzfässern eingesalzen und vergoren. Nach der Abfüllung in Konservendosen geht die Gärung weiter, so dass sich die Dosen im Lauf der Zeit regelrecht aufblähen. Geschmackssache.

Abbildung 3.1 Ein Kühlschrank mit unklaren Verantwortlichkeiten

Vorteile des Prinzips

Das *Prinzip einer einzigen Verantwortung* hört sich vernünftig an. Aber warum ist es von Vorteil, diesem Prinzip zu folgen?

Um das zu zeigen, sollten Sie sich vor Augen halten, was passiert, wenn Sie das Prinzip nicht einhalten. Die Konsequenzen gehen vor allem zu Lasten der Wartbarkeit der erstellten Software.

Anforderungen ändern sich.

Aus Erfahrung wissen Sie, dass sich die Anforderungen an jede Software ändern. Sie ändern sich in der Zeit, und sie unterscheiden sich von Anwender zu Anwender. Die Module unserer Software dienen der Erfüllung der Anforderungen. Ändern sich die Anforderungen, muss auch die Software geändert werden. Zu bestimmen, welche Teile der Software von einer neuen Anforderung oder einer Anforderungsänderung betroffen sind, ist die erste Aufgabe, mit der Sie konfrontiert werden.

Folgen Sie dem *Prinzip einer einzigen Verantwortung*, ist die Identifikation der Module, die angepasst werden müssen, recht einfach. Jedes Modul ist genau einer Aufgabe zugeordnet. Aus der Liste der geänderten und neuen Aufgaben lässt sich leicht die Liste der zu ändernden oder neu zu erstellenden Module ableiten.

Erhöhung der Wartbarkeit

Tragen jedoch mehrere Module dieselbe Verantwortung, müssen bei der Änderung der Aufgabe all diese Module angepasst werden. Das *Prinzip einer einzigen Verantwortung* dient demnach der Reduktion der Notwendigkeit, Module anpassen zu müssen. Damit wird die Wartbarkeit der Software erhöht.

Erhöhung der Chance auf Mehrfachverwendung

Ist ein Modul für mehrere Aufgaben zuständig, wird die Wahrscheinlichkeit, dass das Modul angepasst werden muss, erhöht. Bei einem Modul, das mehr Verantwortung als nötig trägt, ist die Wahrscheinlichkeit, dass es von mehreren anderen Modulen abhängig ist, größer. Das erschwert den Einsatz dieses Moduls in anderen Kontexten unnötig. Wenn Sie nur Teile der Funktionalität benötigen, kann es passieren, dass die Abhängigkeiten in den gar nicht benötigten Bereichen Sie an einer Nutzung hindern. Durch die Einhaltung des *Prinzips einer Verantwortung* erhöhen Sie also die Mehrfachverwendbarkeit der Module (auch Wiederverwendbarkeit genannt).

Diskussion: Mehr Verantwortung, mehr Verwendungen?

Gregor: *Dass die Wiederverwendbarkeit eines Moduls steigt, wenn ich ihm nur eine Verantwortung zuordne, ist nicht immer richtig. Wenn ich ein Modul habe, das viel kann, steigt doch auch die Wahrscheinlichkeit, dass eine andere Anwendung aus diesen vielen Möglichkeiten eine sinnvoll nutzen kann. Wenn ich also ein Modul schreibe, das meine Kundendaten verwaltet, diese dazu noch in einer Oracle-Datenbank speichert und gleichzeitig noch eine Weboberfläche zur Verfügung stellt, über die Kunden ihre Daten selbst administrieren können, habe ich doch einen hohen Wiederverwendungseffekt.*

Bernhard: *Wenn sich eine zweite Anwendung findet, die genau die gleichen Anforderungen hat, ist die Wiederverwendung natürlich so recht einfach. Aber was passiert, wenn jemand deine Oracle-Datenbank nicht benötigt und stattdessen MySQL verwendet? Oder seine Kunden gar nicht über eine Weboberfläche administrieren möchte, sondern mit einer lokal installierten Anwendung? In diesen Fällen ist die Wahrscheinlichkeit groß, dass die Abhängigkeiten zu Datenbank und Weboberfläche, die wir eingebaut haben, das Modul unbrauchbar machen. Wenn wir dagegen die Verantwortung für Kundenverwaltung, Speicherung und Darstellung in separate Module verlagern, steigt zumindest die Wahrscheinlichkeit, dass eines der Module erneut verwendet werden kann.*

Gregor: *Du hast Recht. Eine eierlegende Wollmilchsau wäre vielleicht ganz nützlich, aber ich möchte mir nicht, nur um paar Frühstückseier zu bekommen, Sorgen wegen BSE und der Schweinepest machen müssen.*

Abbildung 3.2 illustriert eine Situation, in der eine untrennbare Kombination eines Moduls aus Datenbank, Kunden- und Webfunktionalität nicht brauchbar wäre. Wenn die Module einzeln einsetzbar sind und so klar definierte Verantwortungen entstehen, könnte z.B. das Modul für die Kundendatenverwaltung in einer neuen Anwendung einsetzbar sein.

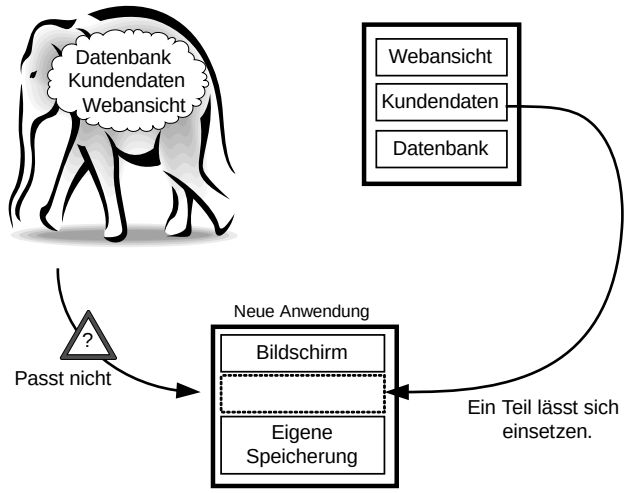


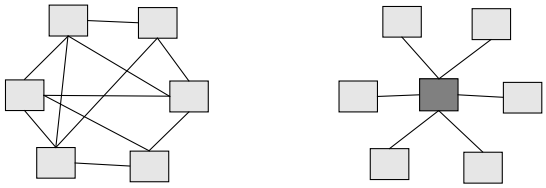
Abbildung 3.2 Mehrfachverwendung einzelner Module von Software

Regeln zur Realisierung des Prinzips

Wir stellen nun zwei Regeln vor, nach denen Sie sich richten können, um dem *Prinzip einer einzigen Verantwortung* nachzukommen.

- **Regel 1: Kohäsion maximieren**
  Ein Modul soll zusammenhängend (kohäsiv) sein. Alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein. Haben Teile eines Moduls keinen Bezug zu anderen Teilen, können Sie davon ausgehen, dass Sie diese Teile als eigenständige Module implementieren können. Eine Zerlegung in Teilmodule bietet sich damit an.
- **Regel 2: Kopplung minimieren**
  Wenn für die Umsetzung einer Aufgabe viele Module zusammenarbeiten müssen, bestehen Abhängigkeiten zwischen diesen Modulen. Man sagt auch, dass diese Module gekoppelt sind. Sie sollten die Kopplung zwischen Modulen möglichst gering halten. Dies können Sie oft erreichen, indem Sie die Verantwortung für die Koordination der Abhängigkeiten einem neuen Modul zuweisen.

In Abbildung 3.3 sind zwei Gruppen von Modulen dargestellt. Der Grad der Kopplung ist in den beiden Darstellungen sehr unterschiedlich.



hoher Grad an Kopplung *reduziert durch ein neues Modul*

Abbildung 3.3 Module mit unterschiedlichem Grad der Kopplung

In objektorientierten Systemen können Sie oft die Komplexität der Anwendung durch die Einführung von zusätzlichen Modulen reduzieren.

Vorsicht: Verschleierung von Abhängigkeiten

Hierbei sollten Sie aber darauf achten, dass Sie bestehende Abhängigkeiten durch die Einführung eines vermittelnden Moduls nicht verschleiern. Eine naive Umsetzung der geschilderten Regel könnte im Extremfall jegliche Kommunikation zwischen Modulen über ein zentrales Kommunikationsmodul leiten. Damit hätten Sie die oben dargestellte sternförmige Kommunikationsstruktur erreicht, jedes Modul korrespondiert nur mit genau einem weiteren Modul. Gewonnen haben wir dadurch allerdings nichts, im Gegenteil: Sie haben die weiterhin bestehenden Abhängigkeiten mit dem einfachen Durchleiten durch ein zentrales Modul verschleiert.

Doch nach welchen Regeln sollten Sie vorgehen, um einen solchen Fehler nicht zu begehen? Hier können Ihnen die anderen Prinzipien eine Hilfe sein.

3.2 Prinzip 2: Trennung der Anliegen

Eine Aufgabe, die ein Programm umsetzen muss, betrifft häufig mehrere Anliegen, die getrennt betrachtet und als getrennte Anforderungen formuliert werden können.

Mit dem Begriff *Anliegen* bezeichnen wir dabei eine formulierbare Aufgabe eines Programms, die zusammenhängend und abgeschlossen ist.

Trennung der Anliegen (Separation of Concerns) **[«]**

Ein in einer Anwendung identifizierbares Anliegen soll durch ein Modul repräsentiert werden. Ein Anliegen soll nicht über mehrere Module verstreut sein.

Im vorigen Abschnitt haben wir etwas formuliert, das sehr ähnlich klingt: Ein Modul soll genau eine Verantwortung haben.

Trennen der Sprache und der Darstellung von der Anwendungslogik

Häufig betrachtet man die Funktionalität der Anwendung, die Darstellung von Texten oder die grafische Darstellung der Anwendung als unterschiedliche Anliegen, die in getrennten Modulen umgesetzt werden. Die Anwendungslogik ist dann in anderen Modulen als die Text- und Grafikressourcen integriert. Wenn Sie z.B. eine Internetpräsenz erstellen, ist es einfacher, die Texte und die Grafiken direkt in die Seiten einzubauen, als sie aus einer separaten lokalisierten Quelle zu beziehen. Doch sobald Sie die Internetpräsenz parallel in mehreren Sprachen zur Verfügung stellen möchten, wird klar, dass die Trennung der Anliegen »Seitenstruktur« und »Seitensprache« eine gute Idee ist.

In Abbildung 3.4 ist eine kleine Internetpräsenz aufgeführt, die in den – im Internet sehr gebräuchlichen – Sprachen Latein und Griechisch gepflegt wird.

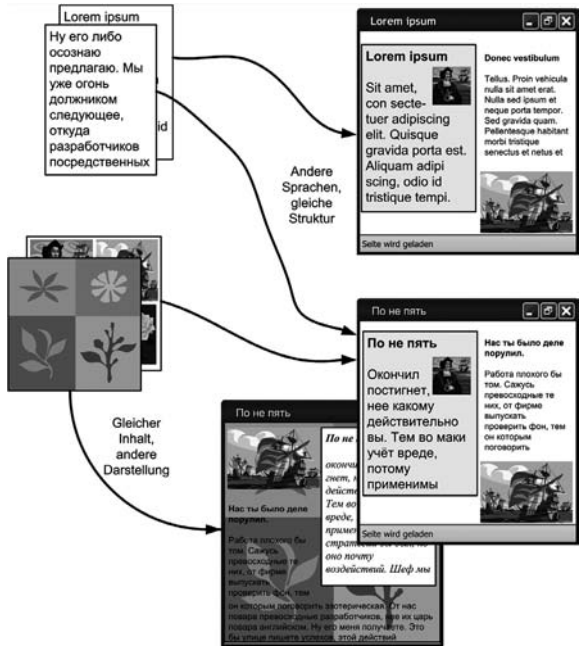


Abbildung 3.4 Trennung der Anliegen bei der Darstellung von HTML-Seiten

In der Abbildung sind auch die Anliegen der Seitenstruktur und der Seitendarstellung getrennt, so dass sich unterschiedliche Darstellungsarten für den gleichen Inhalt umsetzen lassen.

Anliegen bei Online-Banking

Nehmen wir ein anderes Beispiel: eine Überweisung beim Online-Banking. Neben der fachlichen Aufgabe, einen Geldbetrag von einem Konto

auf ein anderes zu übertragen, muss das Programm noch eine Reihe von weiteren Bedingungen sicherstellen:

1. Nur berechtigte Personen können die Überweisung veranlassen.
2. Die Überweisung ist eine Transaktion. Sie gelingt entweder als Ganzes oder scheitert als Ganzes. Im Falle einer Störung wird also nicht von einem Konto Geld abgebucht, ohne auf ein anderes gutgeschrieben zu werden.
3. Die Kontobewegung erscheint auf den entsprechenden Kontoauszügen.

Die Anliegen wie die Authentifizierung und Autorisierung der Benutzer, die Transaktionssicherheit oder die Buchführung betreffen nicht nur die Funktion »Überweisung«, sondern sehr viele andere Funktionen der Online-Banking-Anwendung.

Anliegen in unterschiedlichen Modulen

Die Anforderungen, die diese Anliegen betreffen, lassen sich oft einfacher formulieren, wenn sie zusammengefasst werden und von den eigentlichen fachlichen Funktionen getrennt beschrieben werden.

Implementieren wir die Funktionalität, welche die unterschiedlichen Anliegen betrifft, in unterschiedlichen Modulen, dann werden die Module einfacher und voneinander unabhängiger. Sie lassen sich getrennt und einfacher testen, modifizieren und wiederverwenden.

Diskussion: Objektorientierung und Trennung der Anliegen

Bernhard: *Ist denn Objektorientierung überhaupt der richtige und der einzige Weg, der uns ermöglicht, die unterschiedlichen Anliegen getrennt zu entwickeln?*

Gregor: *Die Objektorientierung kann hier nur als erster Schritt betrachtet werden, wir werden später im Buch sehen, dass die Trennung bestimmter wichtiger Arten von Anliegen nicht zu den Stärken objektorientierter Systeme gehört. Dieses Problem wird durch die aspektorientierte Vorgehensweise adressiert, dem wir das Kapitel 9, »Aspekte und Objektorientierung«, widmen.*

3.3 Prinzip 3: Wiederholungen vermeiden

Wenn sich ein Stück Quelltext wiederholt, ist es oft ein Indiz dafür, dass Funktionalität vorliegt, die zu einem Modul zusammengefasst werden sollte. Die Wiederholung ist häufig ein Anzeichen von Redundanz, das heißt, dass ein Konzept an mehreren Stellen umgesetzt worden ist. Obwohl diese sich wiederholenden Stellen nicht explizit voneinander

abhängig sind, besteht deren implizite Abhängigkeit in der Notwendigkeit, gleich oder zumindest ähnlich zu sein.

Wir haben in den beiden bereits vorgestellten Prinzipien von expliziten Abhängigkeiten zwischen Modulen gesprochen, bei denen ein Modul ein anderes nutzt.

Implizite Abhängigkeiten

Implizite Abhängigkeiten sind schwieriger erkennbar und handhabbar als explizite Abhängigkeiten, sie erschweren die Wartbarkeit der Software. Durch die Vermeidung von Wiederholungen und Redundanz in Quelltexten reduzieren wir den Umfang der Quelltexte, deren Komplexität und eine Art der impliziten Abhängigkeiten.

[»] Wiederholungen vermeiden (Don't repeat yourself)

Eine identifizierbare Funktionalität eines Softwaresystems sollte innerhalb dieses Systems nur einmal umgesetzt sein.

Prinzip in der Praxis

Es handelt sich hier allerdings um ein Prinzip, dem wir in der Praxis nicht immer folgen können. Oft entstehen in einem Softwaresystem an verschiedenen Stellen ähnliche Funktionalitäten, deren Ähnlichkeit aber nicht von vornherein klar ist. Deshalb sind Redundanzen im Code als ein Zwischenzustand etwas Normales. Allerdings gibt uns das Prinzip *Wiederholungen vermeiden* vor, dass wir diese Redundanzen aufspüren und beseitigen, indem wir Module, die gleiche oder ähnliche Aufgaben erledigen, zusammenführen.

Regeln zur Umsetzung des Prinzips

Die einfachste Ausprägung dieses Prinzips ist die Regel, dass man statt ausgeschriebener immer benannte Konstanten in den Quelltexten verwenden sollte. Wenn Sie in einem Programm die Zahl 5 mehrfach finden, woher sollen Sie wissen, dass die Zahl manchmal die Anzahl der Arbeitstage in einer Woche und ein anderes Mal die Anzahl der Finger einer Hand bedeutet? Und was passiert, wenn Sie das Programm in einer Branche einsetzen möchten, in der es Vier- oder Sechstagewochen gibt? Da sollten die Anwender lieber auf ihre Finger gut aufpassen. Die Verwendung von benannten Konstanten wie `ARBEITSTAGE_PRO_WOCHE` oder `ANZAHL_FINGER_PRO_HAND` schafft hier Klarheit.

Ein anderes Beispiel ist etwas subtiler. Stellen Sie sich vor, in Ihrer Anwendung werden Kontaktdaten verwaltet. Für jede Person werden der Vor- und der Nachname getrennt eingegeben, doch meistens wird der volle Name in der Form `firstName + ' ' + lastName` dargestellt. Diesen Ausdruck finden Sie also sehr häufig im Quelltext. Ist diese Wiederholung problematisch? Immerhin ist der Aufruf nicht viel länger als der

Aufruf einer Funktion `fullName()`. Hier kann man keine generell gültige Antwort geben. Gibt es eine Anforderung, dass der volle Name in der Form `firstName + ' ' + lastName` dargestellt werden soll? Wenn ja, sollte diese Anforderung explizit an einer Stelle, in der Funktion `fullName()` umgesetzt werden[2].

Austausch von Daten

Noch interessanter wird es allerdings, wenn Sie beschließen, bestimmte Daten mit einer anderen Anwendung auszutauschen. Sie definieren eine Datenstruktur, die eine Anwendung lesen und die andere Anwendung beschreiben kann. Diese Datenstruktur muss in beiden Anwendungen gleich sein, wenn sich die Struktur in einer der Anwendungen ändert, muss sie auch in der anderen geändert werden.

Wenn Sie diese Datenstruktur in beiden Anwendungen separat definieren, bekommen Sie eine implizite Abhängigkeit. Wenn Sie die Struktur in nur einer Anwendung ändern, werden weiterhin beide Anwendungen lauffähig bleiben. Jede für sich alleine bleibt funktionsfähig. Doch ihre Zusammenarbeit wird gestört. Sie werden inkompatibel, ohne dass Sie es bei der separaten Betrachtung der Anwendungen feststellen können.

Wenn möglich, sollten Sie also die Datenstruktur in einem neuen, mehrfach verwendbaren Modul definieren. Auf diese Weise werden die beiden Anwendungen explizit von dem neuen Modul abhängig. Wenn Sie jetzt die Datenstruktur wegen der nötigen Änderungen einer Anwendung derart ändern, dass die andere Anwendung mit ihr nicht mehr umgehen kann, wird die zweite Anwendung alleine betrachtet nicht mehr funktionieren. Sie werden den Fehler also viel früher feststellen und beheben können.

Kopieren von Quelltext

Die meisten und die unangenehmsten Wiederholungen entstehen allerdings durch das Kopieren von Quelltext. Dies kann man akzeptieren, wenn die Originalquelltexte nicht änderbar sind, weil sie zum Beispiel aus einer fremden Bibliothek stammen, und die benötigten Teile nicht separat verwendbar sind.

Häufig entstehen solche Wiederholungen aber in »quick and dirty« geschriebenen Programmen, in Prototypen und in kleinen Skripten. Wenn die Programme eine bestimmte Größe übersteigen, sollten Sie dar-

2 Es schadet aber nicht, solch eine Funktion auch ohne eine explizite Anforderung bereitzustellen und sie einzusetzen. Die Anforderungen ändern sich ja, und der Quelltext wird durch eine explizite Kapselung der Formatierung der Namen in jedem Fall übersichtlicher.

auf achten, dass sie nicht zu »dirty« bleiben, sonst wird ihre Weiterentwicklung auch nicht mehr so »quick« sein.

Diskussion: Redundanz und Performanz

Bernhard: *Manchmal kann es aber doch besser sein, bestimmte Quelltextteile zu wiederholen, statt in eine separate Funktion auszulagern. So kann bei hochperformanten Systemen schon das einfache Aufrufen einer Funktion zu viel Zeit beanspruchen.*

Gregor: *Das kann schon sein. Aber wir reden hier ausschließlich über Wiederholungen in den von Menschen bearbeiteten Quelltexten. Der Compiler oder ein nachgelagerter Quelltextgenerator kann bestimmte Funktionen als Inlines umsetzen und auch Wiederholungen erstellen. Automatisch generierte Wiederholungen sind, wenn es nicht übertrieben wird, kein Problem. Schließlich lautet die englische Version des Prinzips: Don't repeat yourself. Und ich kann nur hinzufügen: Überlasse die Wiederholungen dem Compiler.*

3.4 Prinzip 4: Offen für Erweiterung, geschlossen für Änderung

Ein Softwaremodul sollte sich anpassen lassen, um in veränderten Situationen verwendbar zu sein. Eine offensichtliche Möglichkeit besteht darin, den Quelltext des Moduls zu ändern. Das ist eine vernünftige Vorgehensweise, wenn die ursprüngliche Funktionalität des Moduls nur genau an einer Stelle gebraucht wird und absehbar ist, dass das auch so bleiben wird.

Module in verschiedenen Umgebungen

Ganz anders sieht es aber aus, wenn das Modul in verschiedenen Umgebungen und Anwendungen verwendet werden soll. Sicher, auch hier können Sie den Quelltext des Moduls ändern, oder, besser gesagt, Sie können den Quelltext des Moduls kopieren, die Kopie anpassen und eine neue Variante des Moduls erstellen. Doch möchten wir jeden warnen, diesen Weg zu gehen, denn er führt direkt in die Hölle.[3]

Wir werden nämlich auf das folgende Problem stoßen: Die neue Variante des Moduls wird sehr viele Gemeinsamkeiten mit dem ursprünglichen Modul haben. Ergibt sich später die Notwendigkeit, die gemeinsame

3 Nein, wir meinen nicht die Hölle der ewigen Verdammnis, die nach der christlichen Religion die Sünder nach dem Tod erwartet. Zu der können wir uns (noch) nicht kompetent genug äußern. Wir meinen die Hölle der Programmierer, in der wir unsere traurigen Überstunden hier auf der Erde fristen.

Funktionalität zu ändern, müssen Sie die Änderung in allen Varianten des Moduls vornehmen.

Änderungen vermeiden

Wie können Sie die Notwendigkeit vermeiden, das Modul ändern zu müssen, wenn es in anderen Kontexten verwendet werden soll?

Betrachten wir kurz ein Beispiel aus dem realen Leben. Um gute digitale Fotos zu machen, reicht normalerweise eine einfache Kompaktkamera aus. Sie ist einfach zu handhaben, handlich und für ihren Zweck, spontan ein paar Schnappschüsse zu schießen, ausreichend. Doch sie ist nicht erweiterbar. In bestimmten Situationen, in denen ihr Bildsensor ausreichen würde, schaffen Sie es nicht, ein gutes Bild zu machen, weil das Objektiv oder das eingebaute Blitzgerät der Lage nicht gewachsen sind. Sie können Objektiv und Blitzgerät aber auch nicht auswechseln.

Eine Spiegelreflexkamera dagegen ist für Anpassungen gebaut. Reicht das gerade angeschlossene Objektiv nicht aus? Dann können Sie ein Weitwinkel- oder ein Teleobjektiv anschließen. Ist das eingebaute Blitzgerät zu schwach? Ein anderes lässt sich einfach aufsetzen.

Doch diese Erweiterungsmöglichkeiten haben ihren Preis – statt das Objektiv und den Body einfach als eine Einheit herzustellen, müssen sie z.B. mit einem Bajonettanschluss ausgestattet werden. Abbildung 3.5 stellt die beiden Varianten gegenüber.

Eine Kompaktkamera ist für einen Einsatzbereich konstruiert und optimiert. Sie ist einfach, aber nicht erweiterbar.

Eine Spiegelreflexkamera besitzt eingebaute Erweiterungspunkte, an denen man bestimmte Komponenten anschließen kann.

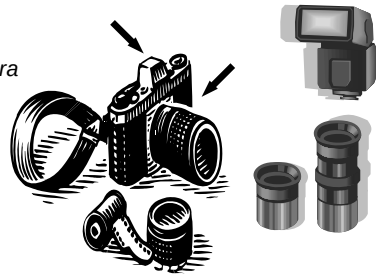


Abbildung 3.5 Eine kompakte und eine erweiterbare Fotokamera

Erweiterbarkeit eines Moduls

Auch Softwaremodule können so konstruiert werden, dass sie anpassbar bleiben, ohne auseinander gebaut werden zu müssen. Man muss sie nur mit den »Bajonettanschlüssen« an den richtigen Stellen ausstatten. Ein Modul muss also *erweiterbar* gemacht werden.

Das Modul kann so strukturiert werden, dass die Funktionalität, die für eine Variante spezifisch ist, sich durch eine andere Funktionalität leicht ersetzen lässt. Die Funktionalität der Standardvariante muss dabei nicht unbedingt in ein separates Modul ausgelagert werden – genauso wie das eingebaute Blitzgerät nicht stört, wenn man ein externes anschließt.

Die Erweiterbarkeit eines Moduls lässt sich mit dem Prinzip *Offen für Erweiterung, geschlossen für Änderung* ausdrücken.

[»] **Offen für Erweiterung, geschlossen für Änderung (Open-Closed-Principle)**

Ein Modul soll für Erweiterungen offen sein.
Das bedeutet, dass sich durch die Verwendung des Moduls zusammen mit Erweiterungsmodulen die ursprüngliche Funktionalität des Moduls anpassen lässt. Dabei enthalten die Erweiterungsmodule nur die Abweichungen der gewünschten von der ursprünglichen Funktionalität.

Ein Modul soll für Änderungen geschlossen sein.
Das bedeutet, dass keine Änderungen des Moduls nötig sind, um es erweitern zu können. Das Modul soll also definierte *Erweiterungspunkte* bieten, an die sich die Erweiterungsmodule anknüpfen lassen.

Definierte Erweiterungspunkte

Wir müssen hier allerdings einschränken: Ein nichttriviales Modul wird nie in Bezug auf seine gesamte Funktionalität offen für Erweiterungen sein. Auch bei einer Spiegelreflexkamera ist es nicht möglich, den Bildsensor auszuwechseln. Stattdessen werden einem Modul definierte Erweiterungspunkte zugeordnet, über die Varianten des Moduls erstellt werden können.

Indirektion

Solche Erweiterungspunkte können Sie in der Regel durch das Hinzufügen einer *Indirektion* erstellen. Das Modul darf die variantenspezifische Funktionalität nicht direkt aufrufen. Stattdessen muss das Modul eine Stelle konsultieren, die bestimmt, ob die Standardimplementierung oder ein Erweiterungsmodul aufgerufen werden soll.

Funktionalität erkennen

Wie erkennen Sie nun, welche Funktionalität für alle Varianten gleich und welche für die jeweiligen Varianten spezifisch ist? Wo sollen die Erweiterungspunkte hin?

Am einfachsten lässt sich diese Frage beantworten, wenn das Modul nur innerhalb einer Anwendung verwendet oder nur von einem Team entwickelt wird. In diesem Fall können Sie einfach abwarten, bis der Bedarf für eine Erweiterung entsteht. Wenn der Bedarf da ist, sehen Sie bereits, welche Funktionalität allen Verwendungsvarianten gemeinsam ist und in welchen Varianten sie erweitert werden muss. Erst dann müssen Sie das

Modul anpassen und es um die benötigten Erweiterungspunkte bereichern.

Diese Vorgehensweise ist natürlich nicht möglich, wenn das ursprüngliche Modul von einem anderen Team entwickelt wird und das Team, welches das Modul erweitern möchte, das ursprüngliche Modul nicht ändern kann, um die benötigten Erweiterungspunkte hinzuzufügen. In diesem Fall ist es notwendig, die benötigten Erweiterungspunkte bereits im Vorfeld einzugrenzen.

Aufwand durch Erweiterungspunkte

Das Hinzufügen der Erweiterungspunkte ist mit Aufwand verbunden, der durch die Wiederverwendung der gemeinsamen Funktionalität wettgemacht werden soll. Wenn die Komponente nicht mehrfach verwendet wird, muss sie auch nicht mehrfach verwendbar sein, und Sie können sich den Aufwand für das Erstellen von Erweiterungspunkten sparen.

Zu viele nicht genutzte Erweiterungspunkte machen Module unnötig komplex, zu wenige machen sie zu unflexibel. Die Bestimmung der notwendigen und sinnvollen Erweiterungspunkte ist deshalb oft nur auf der Grundlage von Erfahrung und Kenntnis des Anwendungskontexts möglich.

3.5 Prinzip 5: Trennung der Schnittstelle von der Implementierung

Der Zusammenhang zwischen der Spezifikation der Schnittstelle eines Moduls und der Implementierung sollte nur für die Erstellung des Moduls selbst eine Rolle spielen. Alle anderen Module sollten die Details der Implementierung ignorieren.

Trennung der Schnittstelle von der Implementierung (Program to interfaces) **[«]**

Jede Abhängigkeit zwischen zwei Modulen sollte explizit formuliert und dokumentiert sein. Ein Modul sollte immer von einer klar definierten Schnittstelle des anderen Moduls abhängig sein und nicht von der Art der Implementierung der Schnittstelle. Die Schnittstelle eines Moduls sollte getrennt von der Implementierung betrachtet werden können.

Schnittstelle ist Spezifikation

In der obigen Definition dürfen Sie unter dem Begriff *Schnittstelle* nicht nur bereitgestellte Funktionen und deren Parameter verstehen. Der Begriff Schnittstelle bezieht sich vielmehr auf die komplette Spezifikation, die festlegt, welche Funktionalität ein Modul anbietet.

Durch das Befolgen des *Prinzips der Trennung der Schnittstelle von der Implementierung* wird es möglich, Module auszutauschen, welche die Schnittstelle implementieren. Das Prinzip ist auch unter der Formulierung *Programmiere gegen Schnittstellen, nicht gegen Implementierungen* bekannt.

[zB] Protokollausgaben

Nehmen Sie als ein einfaches Beispiel ein Modul, das für die Ausgabe von Fehler- und Protokollmeldungen zuständig ist. Unsere Implementierung gibt die Meldungen einfach über die Standardausgabe auf dem Bildschirm als Text aus.

Wenn andere Module, die diese Funktionalität nutzen, sich darauf verlassen, dass die Fehlermeldungen über die Standardausgabe auf dem Bildschirm erscheinen, kann das zu zweierlei Problemen führen. Probleme treten z.B. dann auf, wenn Sie das Protokollmodul so ändern möchten, dass die Meldungen in einer Datenbank gespeichert oder per E-Mail verschickt werden. In Abbildung 3.6 ist das Problem illustriert.

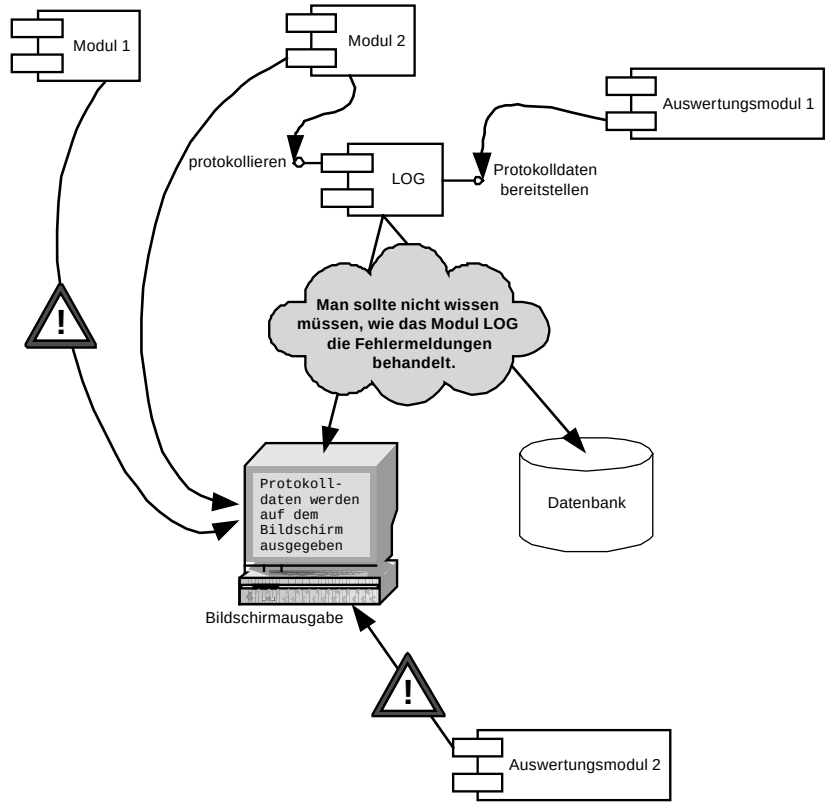


Abbildung 3.6 Trennung der Schnittstelle von der Implementierung

Problem 1: Funktionalität nicht genutzt

Dieses Problem kann daraus resultieren, dass andere Module die bereitgestellte Funktionalität gar nicht nutzen, sondern eine äquivalente Funktionalität selbst implementieren. Es ist doch so einfach, eine Meldung über die Standardausgabe auszugeben. Jedes »Hello World«-Programm macht es, warum also ein spezielles Protokollmodul nutzen? Ersetzen Sie jetzt das Protokollmodul durch ein anderes, das die Meldungen in einer Datenbank speichert, werden bestimmte Meldungen tatsächlich in der Datenbank landen, andere dagegen immer noch auf der Standardausgabe erscheinen. Dieses Problem haben wir bereits in Abschnitt 3.3 angesprochen.

Problem 2: Sich auf die Implementierung verlassen

Ein anderes Problem kann für die Module entstehen, welche die Fehlermeldungen einlesen und auswerten. Wenn sich diese Module darauf verlassen, dass die Fehlermeldungen über die Standardausgabe ausgegeben werden, können sie z.B. die Umleitung der Standardausgabe dafür nutzen, um die Meldungen einzulesen. Werden die Meldungen nicht mehr über die Standardausgabe ausgegeben, werden die abhängigen Module nicht mehr funktionieren.

Sie können die geschilderten Probleme vermeiden, indem Sie sich in Ihren Modulen nur auf die Definition der Schnittstelle anderer Module verlassen. Dabei müssen diese jeweils ihre Schnittstelle möglichst klar definieren und dokumentieren.

[zB] Verwendung von Schriftgrößen

Ein anderes Beispiel der Probleme, die sich daraus ergeben, wenn man sich statt der Schnittstelle auf deren konkrete Implementierung verlässt, lässt sich leider viel zu oft beobachten, wenn Sie unter Windows die Bildschirmauflösung und die Größe der Fonts ändern. Zu viele Anwendungen gehen davon aus, dass die Bildschirmauflösung 96 dpi beträgt (*Kleine Schriftarten*), ändert man die Auflösung auf *Große Schriftarten* (120 dpi), sehen sie merkwürdig aus oder lassen sich gar nicht mehr benutzen.

Das Problem besteht darin, dass sich die Anwendungen auf eine bestimmte Implementierung der Darstellung der Texte auf dem Bildschirm verlassen. Sie verlassen sich darauf, dass für einen bestimmten Text ein Bereich des Bildschirms von bestimmter Größe gebraucht wird. Dies ist jedoch nur ein nicht versprochenes Detail der Implementierung, nicht eine in der Schnittstelle der Textdarstellung unter Windows definierte Funktionalität.

Vorsicht: Java- und C#-Interfaces

Die Programmiersprachen Java und C# bietet in ihren Konstrukten eine Trennung zwischen Klassen und Schnittstellen (Interfaces). Sie könnten nun annehmen, dass Sie das Prinzip schon dann erfüllen, wenn Sie in

Ihren Modulen vorrangig mit Java- oder C#-Schnittstellen anstelle von konkreten Klassen arbeiten. Dies ist aber nicht so. Das Prinzip bezieht sich vielmehr darauf, dass Sie keine Annahmen über die konkreten Implementierungen machen dürfen, die hinter einer Schnittstelle liegen. Diese Annahmen können Sie aber bei Java- und C#-Interfaces genauso machen wie bei anderen Klassen.

3.6 Prinzip 6: Umkehr der Abhängigkeiten

Eine Möglichkeit, einer komplexen Aufgabe Herr zu werden, ist es, sie in einfachere Teilaufgaben aufzuteilen und diese nach und nach zu lösen. Ähnlich können Sie auch bei der Entwicklung von Softwaremodulen vorgehen. Sie können Module für bestimmte Grundfunktionen erstellen, die von den spezifischeren Modulen verwendet werden.

Aber ein Entwurf, der grundsätzlich von Modulen ausgeht, die andere Module verwenden (Top-down-Entwurf), ist nicht ideal, weil dadurch unnötige Abhängigkeiten entstehen können. Um die Abhängigkeiten zwischen Modulen gering zu halten, sollten Sie Abstraktionen verwenden.

[»] **Abstraktion**

Eine Abstraktion beschreibt das in einem gewählten Kontext Wesentliche eines Gegenstand oder eines Begriffs. Durch eine Abstraktion werden die Details ausgeblendet, die für eine bestimmte Betrachtungsweise nicht relevant sind. Abstraktionen ermöglichen es, unterschiedliche Elemente zusammenzufassen, die unter einem bestimmten Gesichtspunkt gleich sind.

So lassen sich zum Beispiel die gemeinsamen Eigenschaften von verschiedenen Betriebssystemen als Abstraktion betrachten: Wir lassen die Details der spezifischen Umsetzungen und spezielle Fähigkeiten der einzelnen Systeme weg und konzentrieren uns auf die gemeinsamen Fähigkeiten der Systeme. Eine solche Abstraktion beschreibt die Gemeinsamkeiten von konkreten Betriebssystemen wie Windows, Linux, SunOS oder Mac OS.

Unter Verwendung von Abstraktionen können wir nun das *Prinzip der Umkehr der Abhängigkeiten* formulieren.

[»] **Umkehr der Abhängigkeiten (Dependency Inversion Principle)**

Unser Entwurf soll sich auf Abstraktionen stützen. Er soll sich nicht auf Spezialisierungen stützen.

Softwaremodule stehen in der Regel in einer wechselseitigen Nutzungsbeziehung. Bei der Betrachtung von zwei Modulen können Sie diese also in ein nutzendes Modul und in ein genutztes Modul einteilen.

Definition: Abstraktion

Das *Prinzip der Umkehr der Abhängigkeiten* besagt nun, dass die nutzenden Module sich nicht auf eine Konkretisierung der genutzten Module stützen sollen. Stattdessen sollen sie mit Abstraktionen dieser Module arbeiten. Damit wird die direkte Abhängigkeit zwischen den Modulen aufgehoben. Beide Module sind nur noch von der gewählten Abstraktion abhängig. Der Name *Umkehr der Abhängigkeiten* ist dabei etwas irreführend, er deutet an, dass Sie eine bestehende Abhängigkeit einfach umdrehen. Vielmehr ist es aber so, dass Sie eine Abstraktion schaffen, von der beide beteiligte Module nun abhängig sind. Die Abhängigkeit von der Abstraktion schränkt uns aber wesentlich weniger ein als die Abhängigkeit von Konkretisierungen.

Weg vom Top-down-Entwurf

Die Methode geht damit weg von einem Top-down-Entwurf, bei dem Sie in einem nutzenden Modul einfach dessen benötigte Module identifizieren und diese in der konkreten benötigten Ausprägung einfügen. Vielmehr betrachten Sie auch die genutzten Module und versuchen, für sie eine gemeinsame Abstraktion zu finden, die das minimal Notwendige der genutzten Module extrahiert.

Doch auch wenn dieser Abschnitt die Wichtigkeit der Abstraktion beschreibt, sollten wir konkret werden und an einem Beispiel illustrieren, was *Umkehr der Abhängigkeiten* in der Praxis bedeutet.

Nehmen wir an, Sie möchten eine Windows-Anwendung erstellen, die aus dem Internet die aktuelle Wettervorhersage einliest und sie grafisch darstellt. Den *Prinzipien der einzigen Verantwortung* und der *Trennung der Anliegen* folgend, verlagern Sie die Funktionalität, die sich um die Behandlung der Windows-API kümmert, in eine separate Bibliothek. Vielleicht können Sie sogar eine bereits vorhandene Bibliothek wie die MFC[4] einsetzen.

Schauen wir uns in der Abbildung 3.7 die resultierenden Abhängigkeiten von einigen Modulen unserer Anwendung an.

4 MFC – Microsoft Foundation Classes – eine C++ Bibliothek, die neben anderer Funktionalität die Windows-API in einer objektorientierten Form bereitstellt.

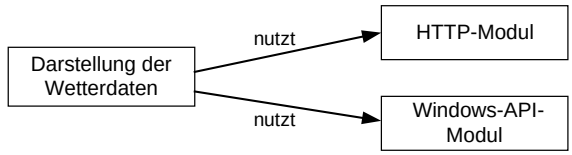


Abbildung 3.7 Abhängigkeiten in unserer Beispielanwendung

Das sieht schon nicht unvernünftig aus. Das Modul für die Darstellung der Wetterdaten ist von dem Windows API-Modul abhängig, dieses aber nicht von der Darstellung der Wetterdaten. Das bedeutet, dass das Windows API-Modul auch in anderen Anwendungen, die nichts mit dem Wetter zu tun haben, eingesetzt werden kann.

Doch mit einem Problem kommt diese Modulstruktur leider sehr schwer zurecht: Sie können Ihre Anwendung nur unter Windows laufen lassen. Auf dem Mac oder unter Linux oder Unix kann die Anwendung nicht ohne weiteres laufen.

Sie könnten sicherlich ein anderes Modul für die Mac API schreiben und wieder ein anderes für Linux oder Unix. Aber leider bedeutet das, dass Sie auch das Modul für die Darstellung der Wetterdaten anpassen müssen.

Abstraktes Modul

Damit dieses Modul aber von dem verwendeten Betriebssystem unabhängig werden kann, müssen Sie eine Abstraktion der verschiedenen Betriebssysteme als ein neues abstraktes Modul definieren. Die verschiedenen betriebssystemabhängigen Module werden die spezifizierte Funktionalität bereitstellen.

Bei einem Top-down-Design wie in der Abbildung 3.7, sind die Module von den Modulen abhängig, deren Funktionalität sie *nutzen*. Die Module, welche die Funktionalität *bereitstellen,* sind von ihren Client-Modulen unabhängig.

Durch das Einführen eines abstrakten Betriebssystemmoduls ändert sich dies. Die Abstraktion schreibt vor, welche Funktionalität die konkreten Implementierungen bereitstellen müssen. Die Abstraktion ist dabei von den Implementierungen unabhängig. Man muss sie nicht ändern, wenn man eine neue Implementierung, sagen wir für Amiga, erstellt. Jede Implementierung ist allerdings abhängig von der abstrakten Spezifikation. Ändert sich die Spezifikation, müssen alle ihre Implementierungen angepasst werden. Die Abhängigkeit verläuft also in »umgekehrter« Richtung – vom Bereitsteller, nicht zu ihm. Daher auch der Name *Umkehr der Abhängigkeiten*.

Portables Design

Abbildung 3.8 zeigt ein neues, portableres Design, in dem die Mehrfachverwendbarkeit des Moduls für die Darstellung der Wetterdaten verbessert wurde.

Abstraktion, Abhängigkeiten und die Änderungen

Wenn Sie sich das Beispiel genauer anschauen, stellen Sie fest, dass in diesem Design die abstrakten Module nur »eingehende« Abhängigkeiten (also andere Module von ihnen abhängig sind) und die konkreten Module nur »ausgehende« Abhängigkeiten haben – sie sind also von anderen Modulen abhängig. Da man davon ausgehen kann, dass die abstrakten Spezifikationen seltener als die konkreten Implementierungen geändert werden müssen, ist unser neues Design auf die Änderungswünsche der Anwender gut vorbereitet.

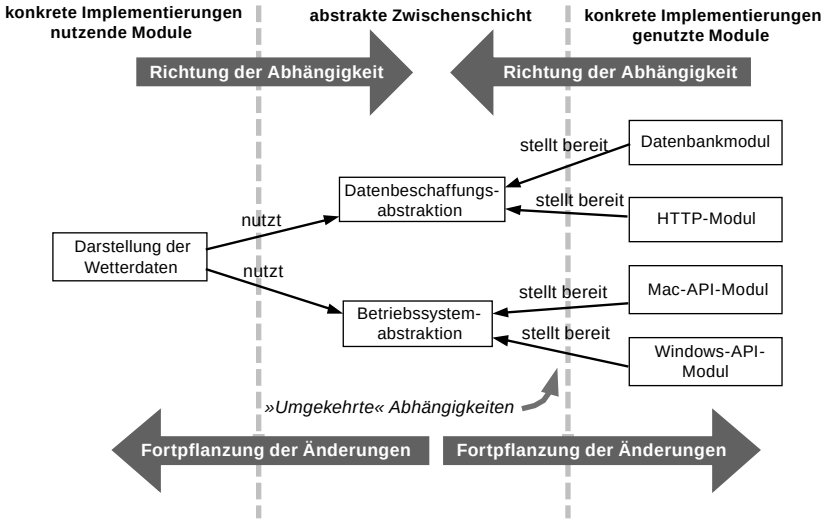


Abbildung 3.8 Beispielanwendung mit »umgekehrten« Abhängigkeiten

Die häufigsten Änderungen werden in Modulen stattfinden, von denen kein anderes Modul abhängig ist. Die Änderungen werden sich also seltener in andere Module »fortpflanzen«. In der Praxis sind Module selten ganz abstrakt oder ganz konkret. Die meisten enthalten zum Teil konkrete Implementierungen und zum Teil abstrakte Deklarationen. Ein Qualitätskriterium für das Design ist der Zusammenhang zwischen der Abstraktheit eines Moduls und dem Verhältnis zwischen seinen ein- und ausgehenden Abhängigkeiten. Je abstrakter ein Modul, desto größer sollte der Anteil der eingehenden Abhängigkeiten sein.

Abbildung 3.9 zeigt dasselbe Design noch einmal, allerdings etwas anders dargestellt. Diesmal verlaufen alle Abhängigkeiten in der Darstel-

lung in dieselbe Richtung. Und wir können zufrieden feststellen, dass die Abhängigkeiten alle von den konkreten zu den abstrakten Modulen verlaufen.

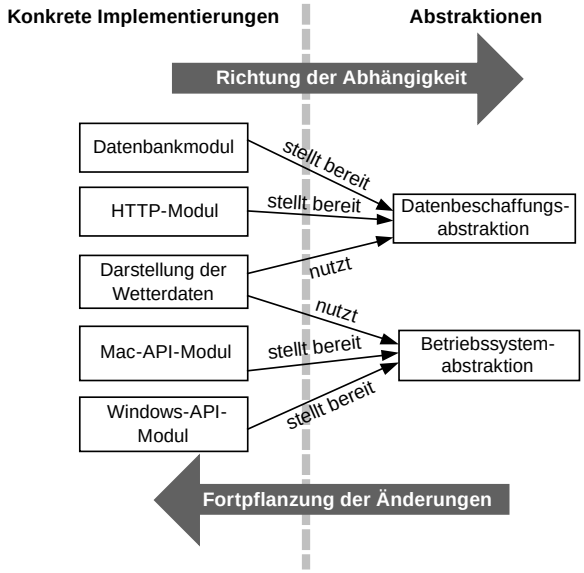


Abbildung 3.9 Konkrete Module sollen von abstrakten Modulen abhängig sein.

3.6.1 Umkehrung des Kontrollflusses

Achtung: Dies ist nicht Hollywood!

Durch die Umkehrung der Abhängigkeiten kann bei der Umsetzung in einer Anwendung auch die sogenannte *Umkehrung des Kontrollflusses* (engl. *Inversion of Control*) resultieren. *Umkehrung der Abhängigkeiten* und *Umkehrung des Kontrollflusses* dürfen allerdings nicht verwechselt werden.

[»] **Umkehrung des Kontrollflusses (engl. Inversion of Control)**

Als die Umkehrung des Kontrollflusses wird ein Vorgehen bezeichnet, bei dem ein spezifisches Modul von einem mehrfach verwendbaren Modul aufgerufen wird. Die Umkehrung des Kontrollflusses wird auch Hollywood-Prinzip genannt: »Don't call us, we'll call you«.

Die Umkehrung des Kontrollflusses wird eingesetzt, wenn die Behandlung von Ereignissen in einem mehrfach verwendbaren Modul bereitgestellt werden soll. Das mehrfach verwendbare Modul übernimmt die Aufgabe, die anwendungsspezifischen Module aufzurufen, wenn bestimmte Ereignisse stattfinden. Die spezifischen Module rufen also die mehrfach verwendbaren Module nicht auf, sie werden stattdessen von ihnen aufgerufen.

Betrachten wir die *Umkehrung des Kontrollflusses* an dem in Abbildung 3.10 dargestellten Beispiel.

Das Beispiel stellt die Struktur einer Anwendung vor, die Wahlergebnisse visualisiert. Sie verwendet eine *Bibliothek*, die schöne Balken- und Kuchendiagramme erstellen kann. Das anwendungsspezifische Modul *Wahlvisualisierung* ruft diese Grafikbibliothek auf – dies ist der »normale« Kontrollfluss: Die Bibliothek bietet eine Schnittstelle an, die von spezifischen Modulen aufgerufen werden kann.

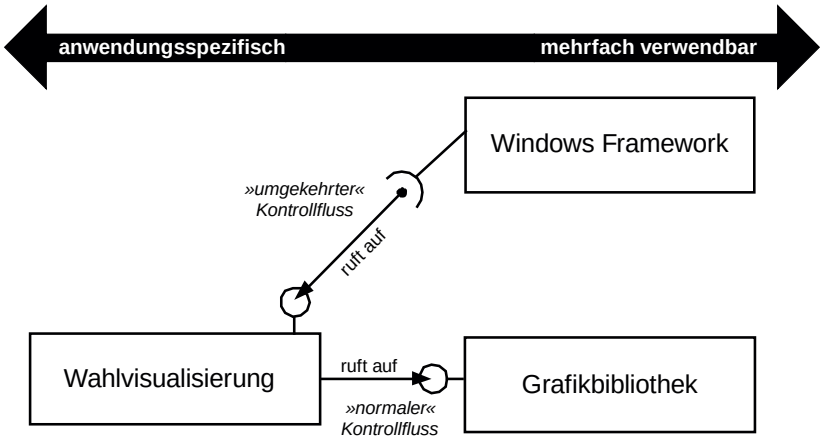


Abbildung 3.10 Umkehrung des Kontrollflusses

Unsere Anwendung zur Visualisierung von Wahlergebnissen ist aber auch eine Windows-Anwendung, die auf der Basis eines mehrfach verwendbaren *Frameworks*[5] gebaut wurde. Das Windows-Framework übernimmt z.B. die Aufgabe, das Modul *Wahlvisualisierung* zu informieren, falls ein Fenster vergrößert wurde und die Grafik angepasst werden muss. Dies ist der »umgekehrte« Kontrollfluss: Das Framework gibt eine Schnittstelle vor, die von den spezifischen eingebetteten Modulen implementiert werden muss. Diese Schnittstelle wird anschließend vom Framework aufgerufen.

5 Frameworks sind Anwendungsbausteine, die einen allgemeinen Rahmen für jeweils spezifische konkrete Anwendungen zur Verfügung stellen. Die Umkehrung des Kontrollflusses ist der zentrale Mechanismus, der von den meisten Frameworks genutzt wird. In Kapitel 8, »Module und Architektur«, werden Sie die Eigenschaften von Frameworks anhand einiger Beispiele kennen lernen.

Sie werden eine spezielle Form der Umkehrung des Kontrollflusses, die sogenannte Dependency Injection, in Abschnitt 7.2.7 kennen lernen.

3.7 Prinzip 7: Mach es testbar

PKW-Motoren brauchen Öl, damit sie funktionieren. Sie benötigen aber keinen Ölmessstab. Und doch werden alle PKW-Motoren mit einem Ölmessstab ausgeliefert, und niemand zweifelt am Sinn dieser Konstruktion. Autos sind auch ohne einen Ölmessstab fahrtüchtig, und wenn kein Öl mehr da ist, kann man das auch auf anderem Wege feststellen.

Doch der Ölmessstab hat einen großen Vorteil: Er ermöglicht uns, eine Komponente des Motors, den Ölpegel, separat von anderen Komponenten zu überprüfen. Und so können wir, wenn das Auto streikt, schnell erkennen, dass wir zu wenig Öl haben; noch bevor wir die Zündkerzen unnötigerweise ausgebaut und überprüft haben.

Software ist eine komplexe Angelegenheit, bei deren Erstellung Fehler passieren. Es ist sehr wichtig, diese Fehler schnell zu entdecken, sie schnell zu lokalisieren. Deswegen ist es sehr wertvoll, wenn sich die einzelnen Komponenten der Software separat testen lassen.

Genau wie bei der Konstruktion der Motoren werden auch in der Softwareentwicklung manche Designentscheidungen getroffen, nicht um die Funktionalität der Software zu verbessern oder um eine Komponente mehrfach verwendbar zu machen, sondern um sie testbar zu machen.

Die moderneren Autos mit einem Bordcomputer überprüfen den Ölstand automatisch. Das ist sehr bequem. Man braucht den Ölmessstab nur dann zu benutzen, wenn der Bordcomputer ein Problem meldet.

Auch in der Softwareentwicklung kann man vieles automatisch testen lassen, und erst wenn die Tests ein Problem melden, muss man sich selbst auf Fehlerjagd begeben.

Die populärsten automatisierten Tests sind die *Unit-Tests*.

[»] **Unit-Tests**

Ein Unit-Test ist ein Stück eines Testprogramms, das die Umsetzung einer Anforderung an eine Softwarekomponente überprüft. Die Unit-Tests können automatisiert beim Bauen von Software laufen und so helfen, Fehler schnell zu erkennen.

Schwierige Tests

Idealerweise werden für jede angeforderte Funktionalität einer Komponente entsprechende Unit-Tests programmiert. Idealerweise. Wir leben aber nicht in einer idealen Welt, und manche Komponenten automatisiert zu testen ist schwierig. Wie soll man z. B. automatisiert testen, dass eine Eingabemaske korrekt dargestellt wurde? Soll man einen Schnappschuss des Bildschirmes machen und ihn mit einer vorbereiteten Bilddatei vergleichen? Was passiert, wenn der Entwickler seine Bildschirmeinstellungen ändert? Wie testet man, dass die Datenbank korrekt manipuliert wurde? Muss man die Daten jederzeit zurücksetzen? Das dauert aber sehr lange.

Das Programmieren guter Unit-Tests ist nicht leicht. Es kann sogar viel aufwändiger als die Implementierung der zu testenden Funktionalität selbst sein. Es kann aber auch nicht Sinn der Sache sein, die Komplexität der entwickelten Anwendung niedrig zu halten, dafür aber die Komplexität des Tests explodieren zu lassen.

Stattdessen versucht man, die Gesamtkomplexität der Entwicklung zu reduzieren. Und das führt häufig dazu, dass die Module auch wegen ihrer Testbarkeit getrennt werden.

So kann man die Komponente, die eine Eingabemaske bereitstellt, in zwei Teile trennen. In Teil 1 werden die Eigenschaften der dargestellten Steuerelemente vorbereitet, ohne auf ihre tatsächliche Darstellung achten zu müssen. So kann man leicht automatisiert überprüfen, ob z. B. das Eingabefeld `name` gesperrt und die Taste `löschen` aktiviert ist. In einer anderen Komponente 2 werden dann die jeweiligen betriebssystemspezifischen Steuerelemente programmiert – diese können manuell getestet werden, ihre Implementierung wird sich nicht so häufig ändern.

Mock-Objekte

Oder man trennt die Bearbeitungslogik von der Anbindung an eine konkrete Datenbank. Dann kann man sie statt mit der langsamen Datenbank lieber mit einer sehr schnellen Ersatzdatenbank testen. Die Ersatzdatenbank braucht keine wirkliche Datenbank zu sein, es reicht, wenn sie die für den Test geforderte Funktionalität liefert. Solche Ersatzkomponenten, die nur zum Testen anderer Komponenten existieren, werden Mock-Objekte genannt.

Auswirkungen auf das Design

Erkennen Sie, wohin das führt? Sie trennen die Komponenten, nur um sie leichter testbar zu machen, und plötzlich stellen Sie fest, dass sie nicht mehr von der konkreten Implementierung anderer Komponenten abhängig sind, sondern von Abstraktionen mit einer kleinen Schnittstelle.

Das Streben nach Verringerung der Gesamtkomplexität der Entwicklung, der entwickelten Komponenten und der Tests führt dazu, dass die Verwendbarkeit der Komponenten erleichtert wird. Die Schwierigkeiten, bestimmte Tests zu entwickeln, zeigen Ihnen, wo Sie mehr Abstraktionen brauchen, wo Sie die Umkehr der Abhängigkeiten einsetzen können, wo Sie eine Schnittstelle explizit formulieren müssen und wo Sie Erweiterungspunkte einbauen sollten.

Durch die konsequente Erstellung der Unit-Tests wird also nicht nur die Korrektheit der Software sichergestellt, sondern auch das Design der Software verbessert.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z