Kapitel 2
Die Basis der Objektorientierung

In diesem Kapitel werfen wir vorab einen Blick auf die technischen Möglichkeiten, die uns die Objektorientierung bietet. Und wir stellen die Basiskonzepte Datenkapselung, Vererbung und Polymorphie kurz vor, ohne bereits ins Detail zu gehen.

Vor der Frage, wie objektorientierte Verfahren am besten eingesetzt werden, drängt sich die Frage auf, warum Sie denn solche Verfahren einsetzen sollten. Die Objektorientierung hat sich seit Anfang der Neunzigerjahre des letzten Jahrhunderts als Standardmethode der Softwareentwicklung etabliert. Aber nur weil etwas mittlerweile als Standard gilt, muss es noch lange nicht nützlich sein. Das allein reicht als Motivation für objektorientierte Verfahren nicht aus.

Objektorientierung löst einige Probleme, …

Die Techniken der objektorientierten Softwareentwicklung unterstützen uns dabei, Software *einfacher erweiterbar*, *besser testbar* und *besser wartbar* zu machen.

… aber nicht alle.

Allerdings dürfen Sie sich von der Objektorientierung nicht Antworten auf alle Probleme und Aufgabenstellungen der Softwareentwicklung erhoffen. Die Erwartungen an die Möglichkeiten dieser Vorgehensweise werden in vielen Projekten zu hochgesteckt.

Zum einen führt die Nutzung objektorientierter Basismechanismen und objektorientierter Programmiersprachen nicht automatisch zu guten Programmen. Zum anderen adressiert die Objektorientierung einige Problemstellungen gar nicht oder bietet nur unvollständige Lösungsstrategien dafür. Bei der Vorstellung des *Prinzips der Trennung von Anliegen* im nächsten Kapitel werden Sie zum Beispiel sehen, dass die Objektorientierung dieses Prinzip nur unvollständig unterstützt.

Grundelemente der Objektorientierung

Die Objektorientierung bietet aber einen soliden Werkzeugkasten an, der es uns erlaubt, die Zielsetzungen der Entwicklung von Software anzugehen. Die Basiswerkzeuge in diesem Werkzeugkasten sind die drei Grundelemente objektorientierter Software:

- Datenkapselung
- Polymorphie
- Vererbung

Wir geben im Folgenden einen kurzen Überblick über die drei Basistechniken. Dabei werden wir auf Begriffe vorgreifen müssen, die erst später im Buch eingeführt werden. Sie sollen aber hier bereits einen ersten Eindruck davon erhalten, welche Möglichkeiten die Objektorientierung bietet. Die Details und formalen Definitionen werden wir Ihnen im weiteren Verlauf des Buches nachreichen, versprochen.

Um Ihnen die Vorteile der objektorientierten Programmierung verdeutlichen zu können, beginnen wir aber zunächst mit einer Kurzzusammenfassung der Verfahren der strukturierten Programmierung, die ja der Vorläufer der objektorientierten Vorgehensweise ist.

2.1 Die strukturierte Programmierung als Vorläufer der Objektorientierung

Die objektorientierte Softwareentwicklung baut auf den Verfahren der strukturierten Programmierung auf. Um die Motivation für die Verwendung von objektorientierten Methoden zu verstehen, gehen wir einen Schritt zurück und werfen einen Blick auf die Mechanismen der strukturierten Programmierung und auch auf deren Grenzen.

Programmiersprachen, die dem Paradigma[1] des strukturierten Programmierens folgen, sind zum Beispiel PASCAL oder C.

Die Struktur von Programmen und Daten

Der Inhalt des benutzten Computerspeichers kann für die meisten Programme in zwei Kategorien unterteilt werden. Einerseits enthält der Speicher *Daten*, die bearbeitet werden, andererseits enthält er *Instruktionen*, die bestimmen, was das Programm macht.[2]

1 Als Paradigma wird in der Erkenntnistheorie ein System von wissenschaftlichen Leitlinien bezeichnet, das die Art von Fragen und die Methoden zu deren Beantwortung eingrenzt und leitet. Im Bereich der Programmierung bezieht sich der Begriff auf eine bestimmte Sichtweise, die die Abbildung zwischen Wirklichkeit und Programm bestimmt.

2 Die Daten eines Programms können Instruktionen eines anderen Programms sein. Zum Beispiel stellt der Java-Bytecode Instruktionen für ein Java-Programm dar, für die Java Virtual Machine (JVM) ist der Java-Bytecode jedoch eine Sammlung von Daten. Ein anderes Beispiel sind Compiler, deren Ausgabedaten Instruktionen für die kompilierten Programme sind.

Jetzt werden Sie auf einige Begriffe treffen, die Ihnen sehr wahrscheinlich bekannt sind, die aber unterschiedlich interpretiert werden können. Wir schicken deshalb einige kurze Definitionen vorweg.

Routine

Ein abgegrenzter, separat aufrufbarer Bestandteil eines Programms. Eine Routine kann entweder Parameter haben oder ein Ergebnis zurückgeben. Eine Routine wird auch als Unterprogramm bezeichnet.

Routinen sind das Basiskonstrukt der strukturierten Programmierung. Indem ein Programm in Unterprogramme zerlegt wird, erhält es seine grundsätzliche Struktur.

Funktion

Eine Funktion ist eine Routine, die einen speziellen Wert zurückliefert, ihren Rückgabewert.

Prozedur

Eine Prozedur ist eine Routine, die keinen Rückgabewert hat. Eine Übergabe von Ergebnissen an einen Aufrufer kann trotzdem über die Werte der Parameter erfolgen.

Die Unterscheidung zwischen Funktion und Prozedur, die wir hier getroffen haben, bewegt sich auf der Ebene von Programmen. Mathematische Funktionen lassen sich sowohl über Prozeduren als auch über Funktionen abbilden.

Strukturierung von Instruktionen und Daten

Ein Weg, der stets wachsenden Komplexität der erstellten Programme Herr zu werden, ist die Strukturierung der Instruktionen und der Daten. Statt einfach die Instruktionen als einen monolithischen Block mit Sprüngen zu implementieren, werden die Instruktionen in Strukturen wie Verzweigungen, Zyklen und Routinen unterteilt. In Abbildung 2.1 ist ein Ablaufdiagramm dargestellt, das die Berechnung von Primzahlen als einen solchen strukturierten Ablauf darstellt.

Außerdem werden Daten bei der strukturierten Programmierung nicht als ein homogener Speicherbereich betrachtet. Man benutzt globale und lokale, statisch und dynamisch allozierte Variablen, deren Inhalte definierte Strukturen wie einfache Typen, Zeiger, Records, Arrays, Listen, Bäume oder Mengen haben.

Typen von Daten

In den strukturierten Programmiersprachen definieren wir Typen der Daten, und wir weisen sie den Variablen, die sie enthalten können, zu. Auch die Parameter der Routinen, also der Prozeduren und der Funktionen, haben definierte Typen, und wir können sie nur mit entsprechend strukturierten Daten aufrufen. Eine Prozedur mit falsch strukturierten Parametern aufzurufen, ist ein Fehler, der im besten Fall von einem Compiler erkannt wird, im schlimmeren Fall zu einem Laufzeitfehler oder einem Fehlverhalten des Programms führt.

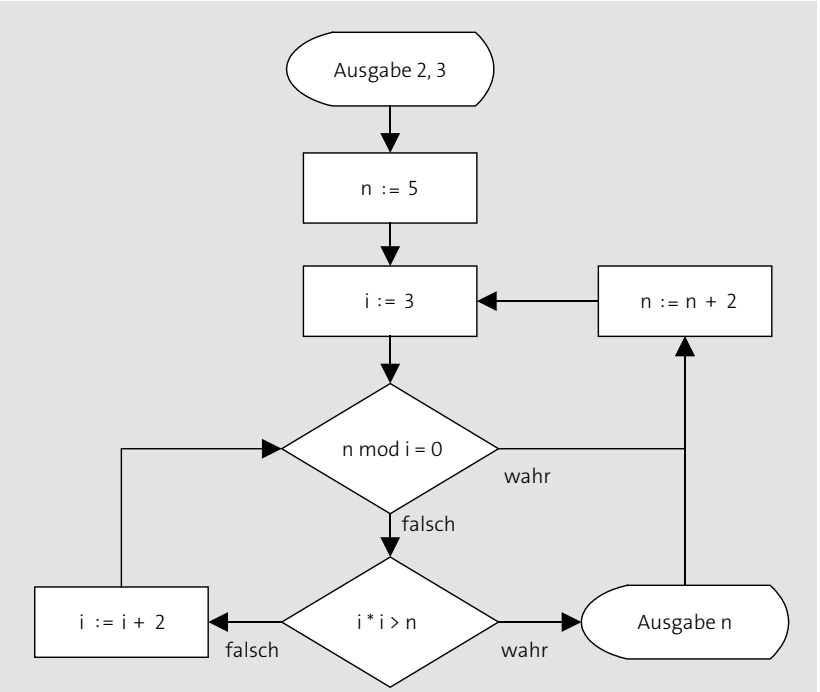


Abbildung 2.1 Ablaufdiagramm zur Berechnung von Primzahlen

Kontrolle und Verantwortung

Beim Programmieren haben wir die volle Kontrolle darüber, welche Routinen mit welchen Daten aufgerufen werden. Diese Macht ist jedoch mit der Verantwortung verbunden, dafür zu sorgen, dass die richtigen Routinen mit den richtigen Daten aufgerufen werden.

In Abbildung 2.2 ist eine andere Variante der Darstellung für den Ablauf bei der Berechnung von Primzahlen dargestellt. Die in den sogenannten Nassi-Shneiderman-Diagrammen gewählte Darstellung bildet besser als ein Ablaufdiagramm die zur Verfügung stehenden Kontrollstrukturen ab.

Wir sehen: Das strukturierte Programmieren war ein großer Schritt in die Richtung der Beherrschung der Komplexität.

Doch wie jede Vorgehensweise stößt auch diese bei bestimmten Problemen an ihre Grenzen. Eine Erweiterung der gewählten Vorgehensweise wird dadurch notwendig.

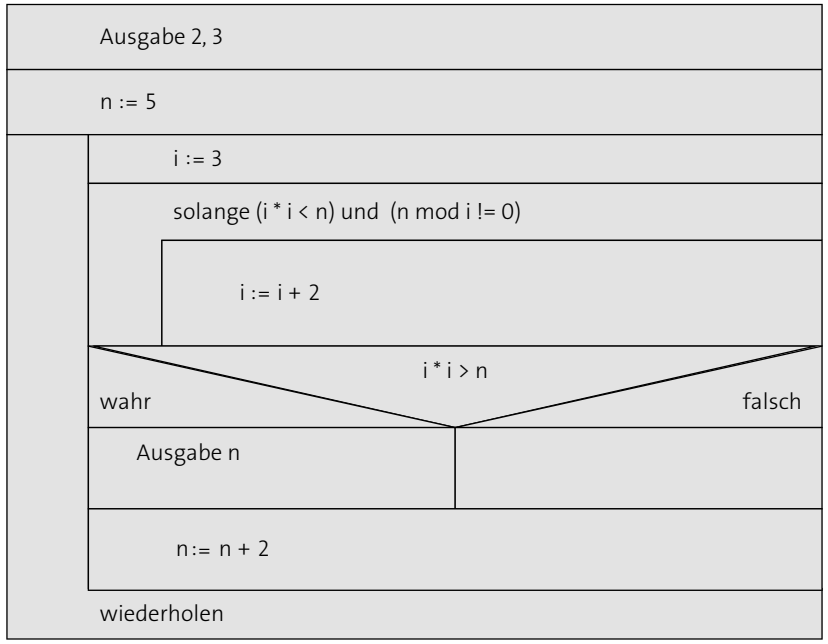


Abbildung 2.2 Nassi-Shneiderman-Diagramm: bessere Darstellung der Struktur

Die Objektorientierung stellt die Erweiterungen bereit. Im Folgenden lernen Sie die drei wichtigsten Erweiterungen in einem kurzen Überblick kennen.

2.2 Die Kapselung von Daten

In der strukturierten Programmierung sind Daten und Routinen voneinander getrennt.

Die Objektorientierung verändert diese Sicht durch die Einführung von Objekten. Daten gehören nun explizit einem Objekt, ein direkter Zugriff wie auf die Datenstrukturen in der strukturierten Programmierung ist nicht mehr erlaubt.

Objekte haben damit also das alleinige Recht, ihre Daten zu verändern oder auch lesend auf sie zuzugreifen. Möchte ein Aufrufer (zum Beispiel ein anderes Objekt) diese Daten lesen oder verändern, muss er sich über

eine klar definierte Schnittstelle an das Objekt wenden und eine Änderung der Daten anfordern.

Konsistenz der Daten

Der große Vorteil besteht nun darin, dass das Objekt selbst dafür sorgen kann, dass die Konsistenz der Daten gewahrt bleibt. Falls zum Beispiel zwei Dateneinträge immer nur gemeinsam geändert werden dürfen, kann das Objekt dies sicherstellen, indem eine Änderung eines einzelnen Werts gar nicht vorgesehen wird.

Ein weiterer Vorteil besteht darin, dass von einer Änderung der zugrunde liegenden Datenstruktur nur die Objekte betroffen sind, die diese Daten verwalten. Wenn jeder beliebig auf die Daten zugreifen könnte, wäre die Anzahl der Betroffenen in einem System möglicherweise sehr hoch, eine Anpassung entsprechend aufwendig. In Abbildung 2.3 ist dargestellt, wie direkte Zugriffe auf die Daten eines Objekts unterbunden werden und der Zugriff nur über definierte Routinen erlaubt wird, die dem Objekt direkt zugeordnet sind.

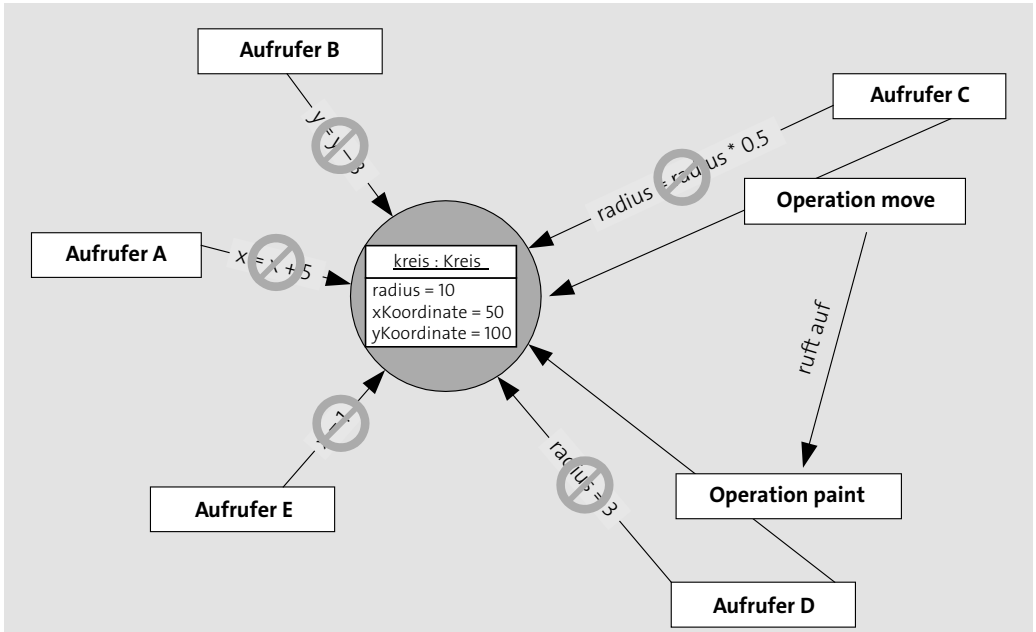


Abbildung 2.3 Zugriff auf ein Objekt nur über definierte Routinen

Durch das in der Objektorientierung gegebene *Prinzip der Datenkapselung* haben Sie also auf jeden Fall Vorteile, weil Sie die Konsistenz von Daten wesentlich einfacher sicherstellen können. Damit ist es auch einfacher, die Korrektheit Ihres Programms zu gewährleisten. Außerdem reduzieren Sie den Aufwand bei Änderungen. Die internen Daten und Vorgehensweisen eines Objekts können geändert werden, ohne dass andere Objekte ebenfalls angepasst werden müssten.

2.3 Polymorphie

Wir werden Polymorphie formal in Kapitel 5, »Vererbung und Polymorphie«, definieren und ihre Anwendungen dort erklären.

Der Nutzen der Polymorphie

Um aber einen ungefähren Eindruck davon zu geben, was der Nutzen der Polymorphie ist, stellen wir zur Erläuterung ein Beispiel aus dem Alltag vor: den Austausch eines Leuchtmittels.

Wörtlich übersetzt bedeutet Polymorphie »Vielgestaltigkeit«. Mit Bezug auf Leuchtmittel können wir diesen Begriff definitiv anwenden: Leuchtmittel gibt es in den verschiedensten Formen und Gestalten. Da reicht das Repertoire vom 150-Watt-Halogenstab bis zu LED-Leuchtmitteln mit wenigen Watt Leistung. Die verschiedenen Arten von Leuchtmitteln können wir aber alle an einer ganz definierten Stelle anbringen: in einer dafür vorgesehenen Fassung.

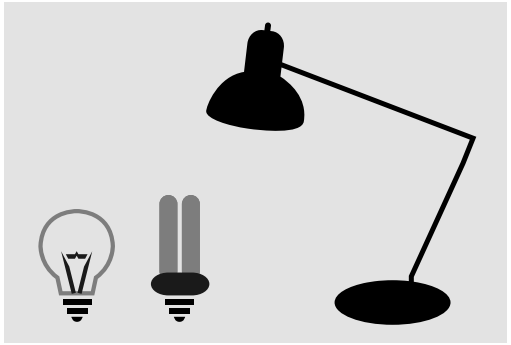


Abbildung 2.4 Verschiedene Leuchtmittel passen in dieselbe Fassung.

Nun haben sich die Hersteller von Fassungen und die Hersteller von Leuchtmitteln bis zu einem gewissen Grad auf einen gemeinsamen Standard geeinigt. Unabhängig vom Stromverbrauch können dieselben Fassungen verwendet werden, neue LED-Leuchtmittel passen oft in Fassungen, die früher die klassischen Glühlampen aufgenommen haben. Die Voraussetzung dafür ist lediglich, dass alle Leuchtmittel der Spezifikation der gemeinsamen Normierungsorganisation für Leuchtmittel und deren Fassungen folgen.

Dass sich die einzelnen Leuchtmittel dann ganz unterschiedlich verhalten, ist nicht mehr relevant. Das eine schummert vor sich hin, das andere

leuchtet hell und verbraucht massig Strom, und das dritte Leuchtmittel wiederum hat nur eine Lebensdauer von zwei Wochen: Für das Einsetzen in die Fassung ist das nicht relevant.

Wir haben durch die Konzentration auf das Zusammenspiel mit der Fassung eine Abstraktion geschaffen: Alle Leuchtmittel, die bestimmte Eigenschaften aufweisen, können mit der genormten Fassung zusammenarbeiten. Andere Eigenschaften der Leuchtmittel sind dafür nicht relevant und können sich durchaus unterscheiden.

Möglichkeiten im objektorientierten System

Genau diese Möglichkeiten, die es für den (auf den ersten Blick banalen) Bereich der Leuchtmittel gibt, möchten wir auch in unseren objektorientierten Systemen nutzen. Wir möchten Stellen in unserem Code haben, die es uns erlauben, einzelne Elemente auszutauschen – wie eine Fassung für Leuchtmittel: Ein Berechnungsverfahren für eine bestimmte Aufgabe ist nicht mehr schnell genug für Ihre Ansprüche? Tauschen Sie es doch einfach gegen ein effizienteres Verfahren aus und klinken Sie es in die dafür vorgesehene Fassung ein. Ein neues Übertragungsprotokoll für Daten muss unterstützt werden? Schrauben Sie es in die Fassung, in die schon alle anderen Übertragungsprotokolle eingesetzt werden konnten.

Die Polymorphie im Zusammenspiel mit einem cleveren Systementwurf ermöglicht uns ein solches Vorgehen. Gegenstand des Entwurfs ist es, die Stellen zu finden, an denen Fassungen vorgesehen werden müssen, und die Objekte zu bestimmen, die in diese Fassungen geschraubt werden. An den richtigen Punkten eingesetzt, kann die Nutzung der Polymorphie dadurch zu wesentlich flexibleren Programmen führen. Sie steigert damit die Wartbarkeit und Änderbarkeit unserer Software.

2.4 Die Vererbung

Vererbung spielt in objektorientierten Systemen in zwei unterschiedlichen Formen eine Rolle. In den folgenden beiden Abschnitten stellen wir anhand von Analogien aus dem Alltag kurz vor, wie diese beiden Arten der Vererbung funktionieren.

2.4.1 Vererbung der Spezifikation

In diesem Abschnitt wollen wir kurz das Beispiel der Leuchtmittel wieder aufgreifen, um zu erläutern, wie denn Vererbung eingesetzt werden kann, um Polymorphie in unseren Programmen zu ermöglichen.

Die Leuchtmittel in unserem Beispiel können ganz unterschiedliche Formen aufweisen, der Energieverbrauch und die Leuchtkraft können sich erheblich unterscheiden. Trotzdem können wir sie alle in die gleiche Fassung schrauben, und, sofern sie nicht defekt sind, werden sie das tun, was wir von einem Leuchtmittel erwarten: Sie leuchten.

Wir stellen also fest, dass diese verschiedenen Leuchtmittel eine grundlegende Gemeinsamkeit haben: Sie entsprechen einer Spezifikation, die es erlaubt, sie in eine genormte Fassung einzusetzen und mit der in Deutschland vorgesehenen Netzspannung von 230 Volt zu betreiben.

Erben von Spezifikationen

In einer objektorientierten Anwendung könnten wir diese Gemeinsamkeiten der verschiedenen Leuchtmittelarten modellieren, indem wir die sogenannte *Vererbung der Spezifikation* verwenden. Eine abstrakte Spezifikation, wie sie für Leuchtmittel von einer Normungsorganisation kommt, gibt dabei vor, welche Eigenschaften die Objekte haben müssen, um die Spezifikation zu erfüllen. In objektorientierten Anwendungen würden wir in diesem Fall davon sprechen, dass die verschiedenen Arten von Leuchtmitteln ihre Spezifikation von der abstrakten Spezifikation *erben*, die als Norm ausgegeben worden ist.

Ein weiteres Beispiel für die Vererbung der Spezifikation in unserer täglichen Lebenswelt sind verschiedene Elektrogeräte, die alle an dieselbe Steckdose angeschlossen werden können.

Abbildung 2.5 Vererbung der Spezifikation im Alltag

Austauschbarkeit

Die Vererbung der Spezifikation hängt sehr eng mit der Polymorphie zusammen. Dass unsere Elektrogeräte nämlich alle die Normen für den Stromanschluss erfüllen, macht sie austauschbar. Jedes von ihnen kann an die gleiche Steckdose angeschlossen werden und wird dann seine Arbeit verrichten.

Die Konzepte der Vererbung werden wir in Kapitel 5, »Vererbung und Polymorphie«, noch ausführlich erläutern.

2.4.2 Vererbung von Umsetzungen (Implementierungen)

Neben der Vererbung der Spezifikation gibt es im Alltag und auch in der Objektorientierung noch eine andere Art von Vererbung: die Vererbung von umgesetzten Verfahren.

Erben gesetzlicher Regelungen

Ein Beispiel für eine solche Form von Vererbung sind gesetzliche Regelungen, die auf verschiedenen Ebenen vorgenommen werden. Wenn Sie in Bonn leben und Steuern zahlen, greifen verschiedene rechtliche Regelungen für Sie:

- Die Europäische Union legt rechtliche Rahmenbedingungen fest.
- Die Bundesrepublik Deutschland hat gesetzliche Regelungen für das Steuerrecht.
- Das Land Nordrhein-Westfalen hat wiederum eigene spezielle Regelungen.
- Schließlich legt die Stadt Bonn noch eigene Regelungen fest, zum Beispiel den sogenannten Hebesatz für die Gewerbesteuer.

Wenn es nun um die Festlegung der von Ihnen zu zahlenden Steuern geht, greifen diese verschiedenen Ebenen der Regelung ganz automatisch für Sie. In einer etwas freien Formulierung könnte man sagen, dass Sie diese ganzen Regeln selbst erben. Wichtiger ist aber, dass die Regelungen der Stadt Bonn die Regeln des Landes erben. Diese wiederum erben die Regeln des Bundes, und der Bund muss die Regeln der Europäischen Union akzeptieren. Der Effekt ist also, dass eine Änderung an den bundesdeutschen Steuergesetzen für alle Bundesbürger sichtbar wird, egal ob sie nun in Bonn oder Hamburg wohnen. Obwohl es spezielle Regelungen für die Kommunen gibt, wird der Großteil der Regeln einfach von oben nach unten durchgereicht.

Die Vererbung in diesem Beispiel hat mit der Vererbung von Implementierung in objektorientierter Software vieles gemein und funktioniert nach diesen Grundsätzen:

- Die Umsetzung einer Aufgabenstellung, in diesem Fall einer steuerrechtlichen Regelung, wird für den speziellen Fall aus dem allgemeinen Fall übernommen. In unserem Beispiel wird der allgemeine Einkommensteuersatz auch für die Bonner direkt aus der bundesweiten Regelung übernommen.
- Eine Änderung in der Umsetzung des allgemeinen Falls führt dazu, dass sich die Situation für die spezielleren Fälle ändert. Wenn sich der Einkommensteuersatz bundesweit ändert, wird das auch für die Bonner spürbar.

- In einem bestimmten Rahmen können eigene Umsetzungen in den speziellen Fällen erfolgen. Für die Gewerbesteuer gibt es spezifisch für Bonn eine eigene Umsetzung.

Hierarchische Regeln

Die Regelungen sind hierarchisch organisiert, wobei die weiter oben liegenden Regeln jeweils weiter unten liegende überschreiben. Das sollte man sich etwa so vorstellen wie bei dem Stapel Bücher aus Abbildung 2.6: Man prüft zunächst im obersten Buch im Stapel, ob eine Regelung für einen Bereich vorliegt. Findet man sie dort nicht, geht man zum nächsten Buch im Stapel weiter – und so fort, bis eine Regelung gefunden wird.

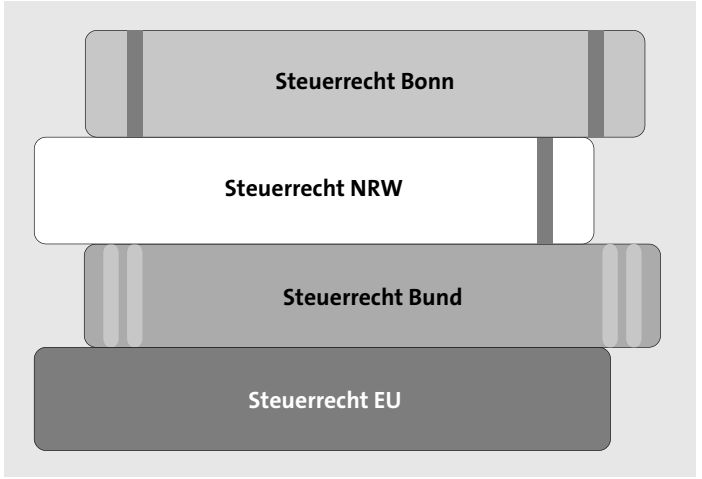


Abbildung 2.6 Hierarchie von Regelungen im Steuerrecht

Durch dieses Vorgehen wird viel bedrucktes Papier (und damit auch Redundanz) vermieden, denn wenn alle Kommunen die bereits existierenden Regeln erneut selbst auflegen müssten, würde extrem viel Papier unnütz bedruckt.

In Kapitel 5, »Vererbung und Polymorphie«, werden wir diese Art der Vererbung im Detail vorstellen.

Zunächst werden wir uns aber im folgenden Kapitel die grundlegenden Prinzipien des objektorientierten Entwurfs näher ansehen.

Kapitel 3

Die Prinzipien des objektorientierten Entwurfs

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 beiseitelegen 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.

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 Softwarearchitektinnen und -architekten, -entwicklerinnen und -entwickler –, die Programme so zu organisieren, dass sie nicht nur für die, die sie anwenden, 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 darauffolgenden Kapiteln werden wir zeigen,

ob und wie diese Prinzipien in der objektorientierten Programmierung 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, das wir beim Programmieren als eine Einheit betrachte, 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 Sie das Modul nicht mehr gut verstehen und anpassen können.

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 aufwendig, 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 oder die Kühlschrankbeauftragte, und außerdem haben Sie gehört, Ihre Chefin solle den schwedischen Surströmming[1] mögen – warum also das Risiko eingehen, die Chefin zu verärgern?

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, sodass sich die Dosen im Lauf der Zeit regelrecht aufblähen. Geschmackssache.

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.

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 auch 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 zulasten 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 die gleiche 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 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 das 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 ein paar Frühstückseier zu bekommen, Sorgen um BSE und 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.

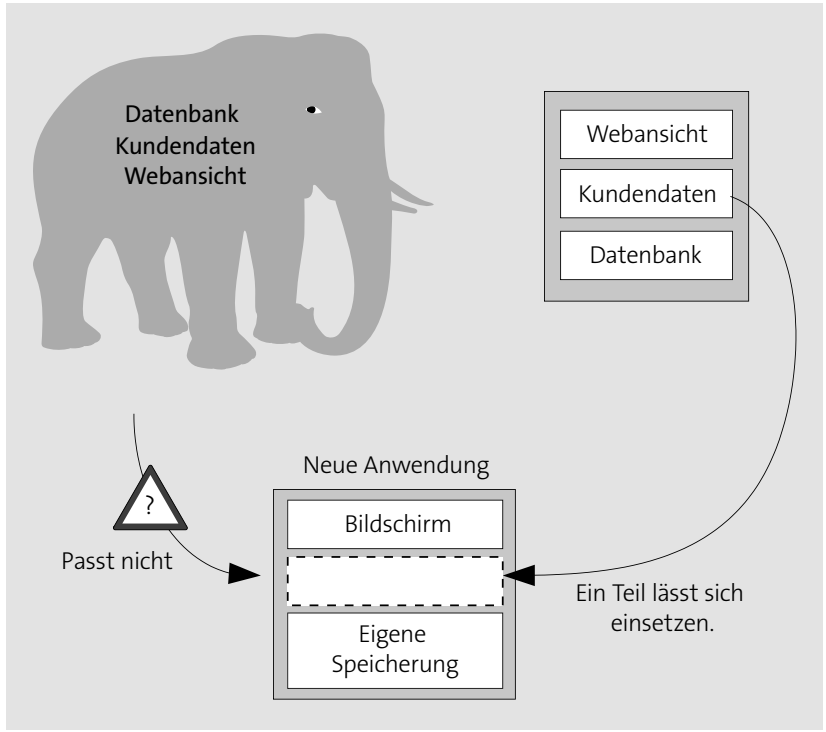


Abbildung 3.2 Mehrfachverwendung einzelner Module von Software

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. Das 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.

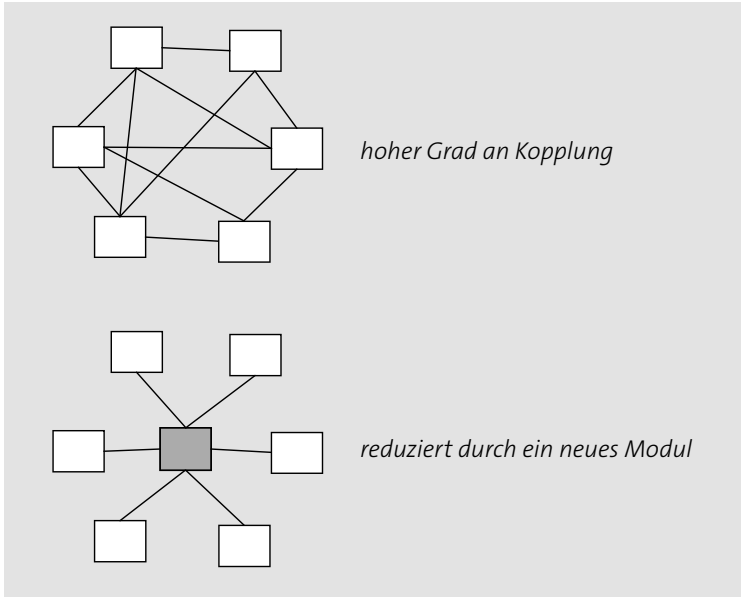


Abbildung 3.3 Module mit unterschiedlichem Grad der Kopplung

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 und 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 beispielsweise 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.

In der Abbildung sind auch die Anliegen der Seitenstruktur und der Seitendarstellung getrennt, sodass sich unterschiedliche Darstellungsarten für denselben Inhalt umsetzen lassen.

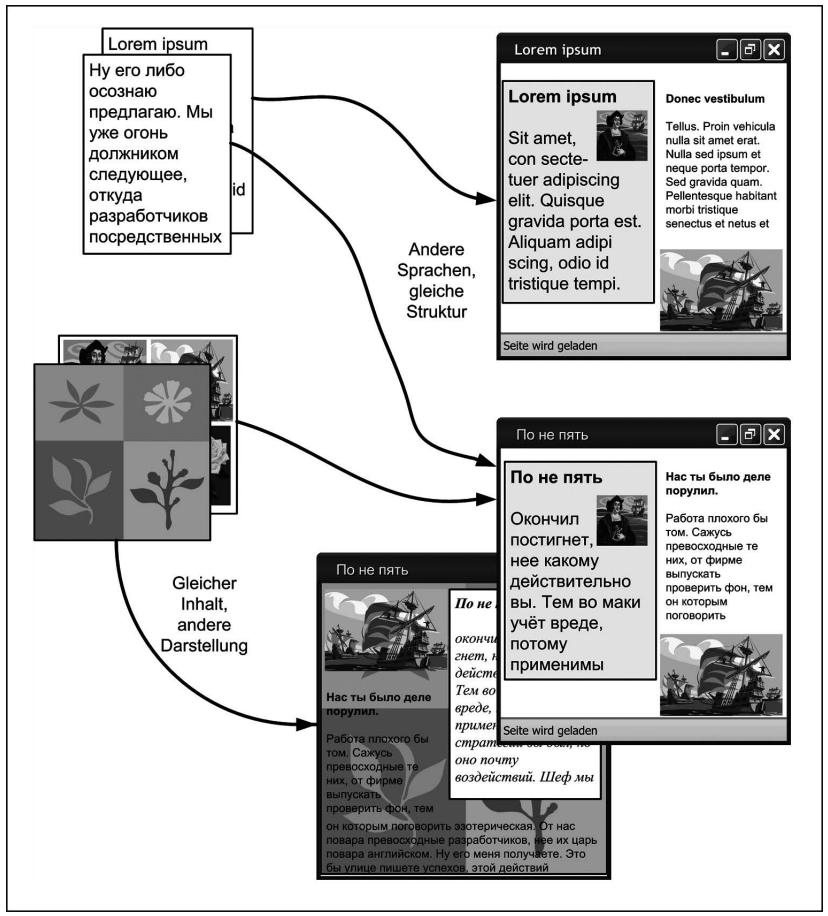


Abbildung 3.4 Trennung der Anliegen bei der Darstellung von HTML-Seiten

Anliegen beim Onlinebanking

Nehmen wir ein anderes Beispiel: eine Überweisung beim Onlinebanking. 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 Fall einer Störung wird also nicht von einem Konto Geld abgebucht, ohne dass es auf ein anderes gutgeschrieben wird.
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 Onlinebanking-Anwendung.

Anliegen in unterschiedlichen Modulen

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

Implementieren wir die Funktionalität, die die unterschiedlichen Anliegen betrifft, in unterschiedlichen Modulen, werden die Module einfacher und voneinander unabhängiger. Sie lassen sich getrennt 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, der wir Kapitel 9, »Aspekte und Objektorientierung«, widmen.*

3.3 Prinzip 3: Wiederholungen vermeiden

Wenn sich ein Stück Quelltext wiederholt, ist das 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, denn 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 Zwischenzustand etwas Normales. Allerdings gibt uns das Prinzip *Wiederholungen vermeiden* vor, dass wir diese Redundanzen aufspüren und beseitigen, indem wir Module zusammenführen, die gleiche oder ähnliche Aufgaben erledigen.

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 und Anwenderinnen lieber gut auf ihre Finger 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()`. Hierauf 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 be-

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

schreiben kann. Diese Datenstruktur muss in beiden Anwendungen gleich sein. Ändert sich die Struktur in einer der Anwendungen, 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, sind weiterhin beide Anwendungen lauffähig. Jede für sich allein bleibt funktionsfähig. Doch ihre Zusammenarbeit wird gestört. Sie werden inkompatibel, ohne dass Sie das bei der separaten Betrachtung der Anwendungen feststellen können.

Sofern 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 allein 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. Das 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 darauf 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 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 ist. 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.

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 eines aus dem Ruder gelaufenen Entwicklungsprojekts.

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

Abbildung 3.5 Eine kompakte und eine erweiterbare Fotokamera

Erweiterbarkeit eines Moduls

Auch Softwaremodule können so konstruiert werden, dass sie anpassbar bleiben, ohne auseinandergebaut werden zu müssen. Man muss sie nur mit »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 neu 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 nicht triviales 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, das 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 oftmals 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, die 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.

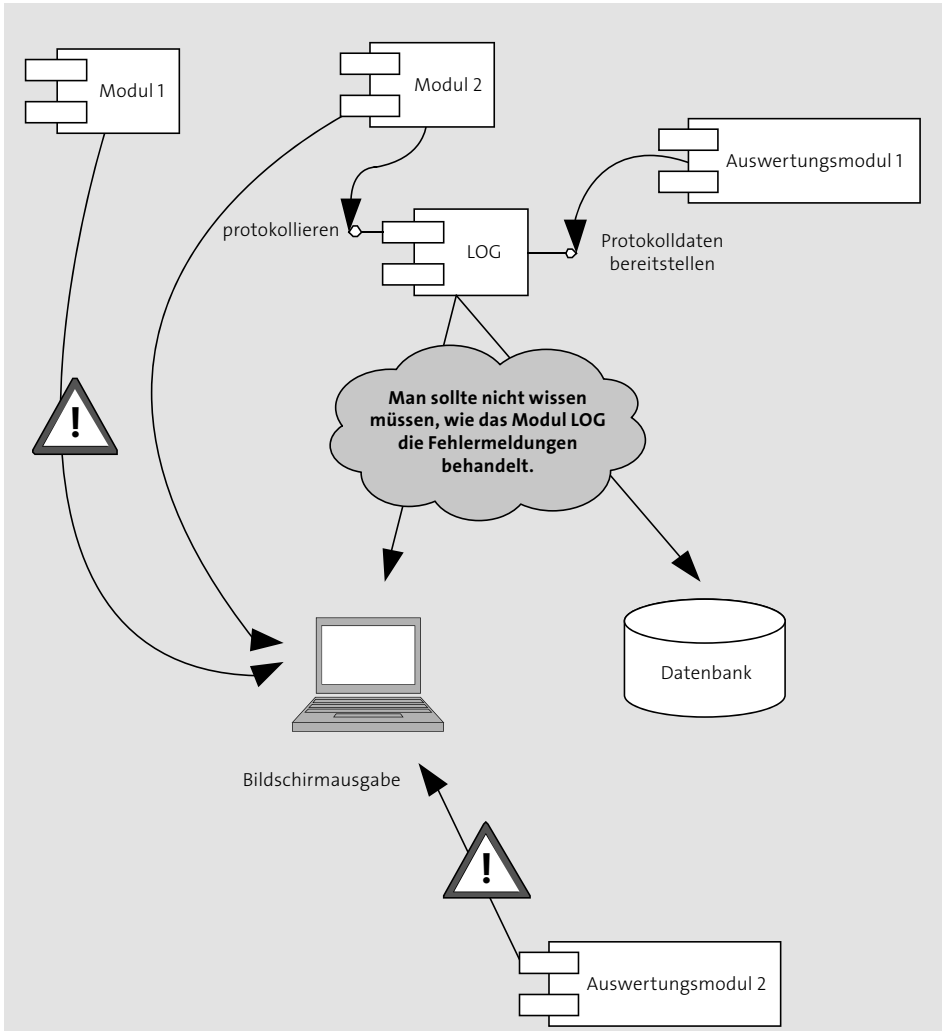


Abbildung 3.6 Trennung der Schnittstelle von der Implementierung

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 das, 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 Daten-

bank landen, andere dagegen immer noch auf der Standardausgabe erscheinen. Dieses Problem haben wir bereits in Abschnitt 3.3, »Prinzip 3: Wiederholungen vermeiden«, angesprochen.

Problem 2: Sich auf die Implementierung verlassen

Ein anderes Problem kann für die Module entstehen, die 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, 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 weiteres Beispiel für Probleme, die sich daraus ergeben, dass man sich statt auf die 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# bieten 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. Das 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 zu zerlegen 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 Gegenstands 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 oder macOS.

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 – nicht auf Spezialisierungen.

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 ar-

beiten. 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 beteiligten Module nun abhängig sind. Die Abhängigkeit von der Abstraktion schränkt uns jedoch 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 Abbildung 3.7 die resultierenden Abhängigkeiten von einigen Modulen unserer Anwendung an.

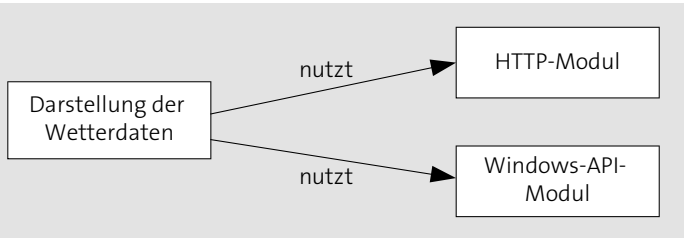


Abbildung 3.7 Abhängigkeiten in unserer Beispielanwendung

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

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 bzw. 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 Abbildung 3.7 sind die Module von den Modulen abhängig, deren Funktionalität sie *nutzen*. Die Module, die die Funktionalität *bereitstellen*, sind von ihren Clientmodulen unabhängig.

Das ändert sich mit der Einführung eines abstrakten Betriebssystemmoduls. 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 hin. 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 haben (also andere Module von ihnen abhängig sind) und die konkreten Module nur »ausgehende« Abhängigkeiten – 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 Änderungswünsche gut vorbereitet.

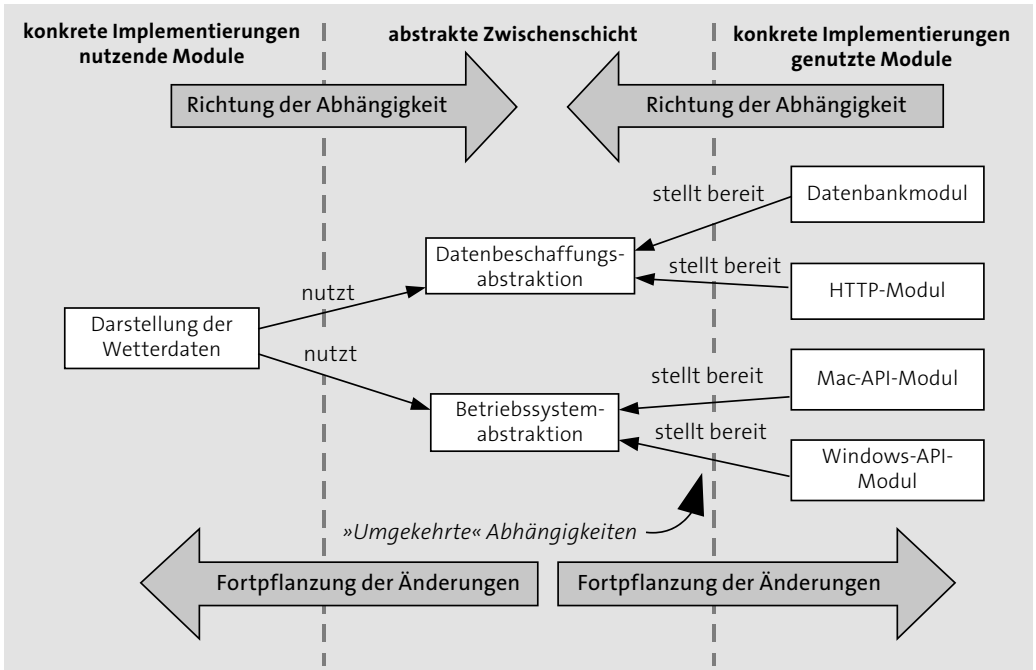


Abbildung 3.8 Beispielanwendung mit »umgekehrten« Abhängigkeiten

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

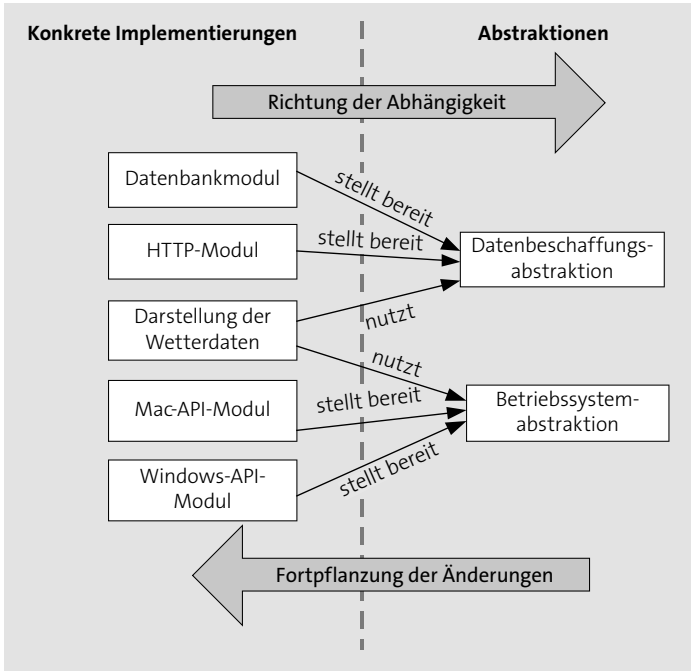


Abbildung 3.9 Konkrete Module sollen von abstrakten Modulen abhängig sein

3.6.1 Umkehrung des Kontrollflusses

Achtung: Das 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 (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.

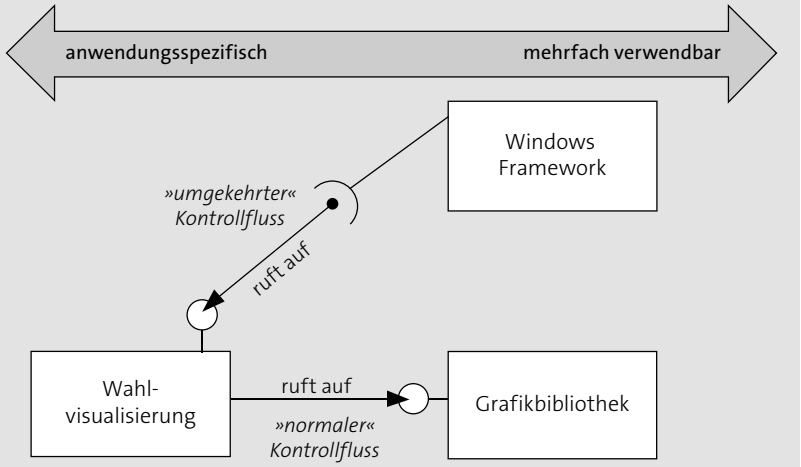


Abbildung 3.10 Umkehrung des Kontrollflusses

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. Das 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.

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

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 kennenlernen.

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, bevor das Auto streikt, bereits schnell erkennen, dass wir zu wenig Öl haben, ohne dass die Zündkerzen unnötigerweise ausgebaut und überprüft werden müssen.

Software ist eine komplexe Angelegenheit, bei deren Erstellung Fehler passieren. Es ist sehr wichtig, diese Fehler schnell zu entdecken und 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. 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 für manche Komponenten ist es schwierig, sie automatisiert zu testen. Wie soll man z. B. automatisiert testen, dass eine Eingabemaske korrekt dargestellt wurde? Soll man einen Schnappschuss des Bildschirms machen und ihn mit einer vorbereiteten

Bilddatei vergleichen? Was passiert, wenn ein 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 aufwendiger 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 die Komponente, die eine Eingabemaske bereitstellt, in zwei Teile getrennt werden. In Teil 1 werden die Eigenschaften der dargestellten Steuerelemente vorbereitet, ohne auf ihre tatsächliche Darstellung achten zu müssen. Damit 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.

Kapitel 8
Module und Architektur

Wir betrachten in diesem Kapitel anhand einer Reihe von Beispielen, wie objektorientierte Entwürfe in größere Kontexte eingebunden werden. Dabei diskutieren wir die verschiedenen Arten, mit denen Module in diesen Kontexten angepasst und erweitert werden können. Am Beispiel des Musters Model-View-Controller stellen wir vor, wie objektorientierte Verfahren Beiträge zu einer kompletten Systemarchitektur leisten können.

Objektorientierte Verfahren können uns beim Architekturentwurf unterstützen, indem sie uns vor allem bei einer klaren Aufgabenteilung helfen. In diesem Kapitel stellen wir Beispiele für den Einsatz von objektorientierten Verfahren beim Entwurf von Software-Architekturen vor.

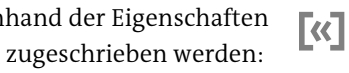
8.1 Module als konfigurierbare und änderbare Komponenten

Module sind die Bausteine, aus denen sich Software zusammensetzt. Die Mechanismen der Objektorientierung sollen uns helfen, diese Module zu definieren und interagieren zu lassen. Wir geben in diesem Abschnitt einen Überblick über die verschiedenen Möglichkeiten, Module erweiterbar zu halten und aus diesen Modulen ein funktionierendes System aufzubauen. Aber zunächst stellen wir die Frage: »Gibt es überhaupt so etwas wie eine objektorientierte Architektur?«

8.1.1 Relevanz der Objektorientierung für die Softwarearchitektur

Der Begriff *Softwarearchitektur* allein kann Grundlage für ganze Workshops sein. Eine einheitliche Definition werden Sie auch in der Literatur nicht finden.

Deshalb versuchen wir, den Begriff *Architektur* anhand der Eigenschaften zu beschreiben, die Softwarearchitekturen häufig zugeschrieben werden:



- Architektur ist das, was in einem Softwaresystem wichtig oder auch nur schwer zu ändern ist.[1]
- Architektur ist auch das, was wir jemandem, der nichts über unser System weiß, als Erstes vorstellen, damit er einen Überblick über das System erhält.
- Architektur hat Auswirkungen auf die Entwicklerinnen und Entwickler eines Systems, weil sie Vorgaben macht, wie bestimmte Dinge umzusetzen sind.

Objektorientierung ist nicht in erster Linie ein Verfahren zum Architekturentwurf. Wir sprechen deshalb in der Regel von *objektorientiertem Systemdesign*, nicht von einer objektorientierten Architektur.

Allerdings hat ein objektorientiertes Design Rückwirkungen auf die Systemarchitektur, und umgekehrt kann eine Systemarchitektur die Anwendung von objektorientierten Verfahren erleichtern oder erschweren.

Diskussion: Sind Architekturen schwer zu ändern?

Gregor: *Muss es denn wirklich immer so sein, dass eine Architektur schwer zu ändern ist? Ich denke, dass wir Architekturen auch so anlegen können, dass wir mögliche Änderungen schon mit in Betracht ziehen und die Architektur selbst änderbar halten.*

Bernhard: *Kannst du dafür ein Beispiel nennen? Nach meiner Erfahrung ändert man Architekturen nur mit größerem Aufwand. In der Praxis wirft man oft eine Architektur komplett weg und beschäftigt sich dann mit einer neuen Architektur.*

Gregor: *Da kann ich dir ein konkretes Beispiel nennen. In einem unserer Projekte haben wir eine Architektur auf Basis eines Applikationsservers verwendet. Dabei haben wir aber explizit mit Pojos, also reinen Java-Objekten, gearbeitet und eine Reihe von Möglichkeiten des Applikationsservers nicht genutzt. In der Folge konnten wir unsere Architektur für bestimmte Szenarien einfach auf eine reine Client-Server-Architektur umstellen, in der die komplette Logik inklusive des Datenzugriffs in den Client verlagert ist. Wir konnten praktisch durch Konfiguration unsere Architektur von Thin-Client auf einen Fat-Client umstellen.*

Bernhard: *Das ist sicherlich ein guter Ansatz. Allerdings sehe ich es eher so, dass eure Architektur beide Varianten umfasst. Damit sind praktisch beide Verteilungsverfahren Teil eurer Architektur. Das spricht zwar für deren Qualität, aber eine Änderung der Architektur liegt eben beim Umschalten des*

1 Als Konsequenz ist dann ein Softwarearchitekt bzw. eine Softwarearchitektin in einem Projekt eine Person, die wichtig ist oder deren Meinung schwer zu ändern ist.

Verteilungsverfahrens nicht vor. Änderungen an der Architektur selbst wären immer noch vergleichsweise schwer durchzuführen.

8.1.2 Erweiterung von Modulen

Beim Entwurf von Modulen ist die Fragestellung zentral, wie sich diese Module später erweitern lassen. Wir stellen eine Reihe von etablierten Verfahren vor, mit denen Erweiterbarkeit unterstützt wird.

Vererbung

Wenn wir eine Möglichkeit suchen, um existierende Module zu erweitern, fällt uns im Bereich der Objektorientierung natürlich die Möglichkeit der *Vererbung* ein. Der einfachste (aber eben auch naive) Ansatz für eine Modulerweiterung ist es, für eine Klasse, die uns zur Nutzung zur Verfügung gestellt wird, eine abgeleitete Klasse zu definieren und stattdessen diese zu nutzen. Eine denkbare Architektur eines Systems wäre also, während der Weiterentwicklung die existierende Hierarchie von Klassen immer weiter aufzufächern und neue Unterklassen einzuführen, sobald neue Anforderungen erkennbar werden. Das führt aber in der Regel auch dazu, dass existierende Hierarchien geändert werden müssen. In der Regel ist das nur dann möglich, wenn es sich um interne Module handelt und die explizite Hierarchie der Klassen nicht nach außen offengelegt ist.

In dieser Form ist Vererbung also als Basis für Erweiterungen eines Systems nicht gut geeignet. Sie haben auch bereits in Abschnitt 5.3.2, »Das Problem der instabilen Basisklassen«, gesehen, welche Probleme durch eine solche Nutzung von Vererbungsbeziehungen entstehen können.

Abgeleitete Klassen

Natürlich können Sie trotzdem Vererbung zur Erweiterung von Modulen einsetzen. Stellen Module eine dokumentierte Menge von Klassen und damit Operationen zur Verfügung, können Sie für diese Klassen abgeleitete Klassen erstellen, die begrenzte Änderungen einführen. Wenn Sie dann in Ihrer Applikation ein Exemplar der abgeleiteten Klasse erstellen, wird es das geänderte Verhalten zeigen.

Fabriken als Erweiterungspunkte

Damit dieser Ansatz sinnvoll funktionieren kann, müssen sich die Entwickler und Entwicklerinnen eines Moduls allerdings bereits intensiv Gedanken gemacht haben, was die über Ableitung von Klassen nutzbaren *Erweiterungspunkte* des Moduls sind.

In Abschnitt 7.2, »Fabriken als Abstraktionsebene für die Objekterzeugung«, haben wir gesehen, wie uns verschiedene Verfahren zur Objekterzeugung definierte Möglichkeiten bieten, Module nachträglich noch zu erweitern. Fabriken sind auch ein zentraler Mechanismus, mit dem sogenannte Container arbeiten. Durch diese Methodik lassen sich begrenzte und definierte Anpassungen an Modulen vornehmen.

Einen ähnlichen Zweck verfolgt auch das Entwurfsmuster »Strategie«, das Sie in Abschnitt 5.5.1 kennengelernt haben. Mithilfe des Entwurfsmusters lässt sich das Verhalten von existierenden Klassen anpassen.

Beide Mechanismen bieten uns Möglichkeiten, eigene Funktionalitäten in existierende Module einzubringen, ohne diese Module öffnen zu müssen. Sie sind also Beispiele für Vorgehensweisen, die uns helfen, das zentrale Prinzip *Offen für Erweiterung, geschlossen für Änderung* umzusetzen.

Frameworks

Eine eigene Sichtweise auf Erweiterbarkeit nehmen *Frameworks* ein. Diese haben als zentrales Konzept die *Umkehr des Kontrollflusses* (*Inversion of Control*).

Frameworks (Anwendungsrahmen)

Das Grundkonzept von Frameworks ist, einen Rahmen für eine Anwendung oder einen Anwendungsbereich zur Verfügung zu stellen. Damit legen Frameworks eine Art Schablone für diesen Bereich fest, die bei der Entwicklung einer konkreten Anwendung dann ausgeprägt wird. Die Entwicklung einer Anwendung auf Basis von Frameworks besteht darin, dass Klassen und Methoden umgesetzt werden, die aus dem bereits existierenden Framework heraus aufgerufen werden. Damit liegt die Steuerung des Kontrollflusses komplett bei den Framework-Klassen. Das Frameworks zugrunde liegende Prinzip wird deshalb auch *Umkehrung des Kontrollflusses* (*Inversion of Control*) oder *Hollywood-Prinzip* genannt.[2]

Ablaufsteuerung

Der intuitive Normalfall bei der Entwicklung eines Programms in einer prozeduralen Programmiersprache ist, dass der Programmablauf durch den von Ihnen geschriebenen Code bestimmt wird. Sie schreiben auf, was ausgeführt werden soll, und genau in dieser Reihenfolge passiert es dann auch.

2 Wir haben die Umkehrung des Kontrollflusses bereits in Abschnitt 3.6, »Prinzip 6: Umkehr der Abhängigkeiten«, an einem Beispiel vorgestellt.

Mit den Mitteln der objektorientierten Programmierung wird bereits ein relevanter Teil der Ablaufsteuerung nicht mehr explizit beschrieben, sondern an Mechanismen der Programmiersprache abgegeben. Bei Aufruf einer polymorphen Operation entscheidet der Typ eines Objekts zur Laufzeit darüber, welche Methode nun genau aufgerufen wird.

Wenn Sie aber die Kontrolle darüber, wann im Programmablauf unsere implementierte Funktionalität tatsächlich aufgerufen wird, an ein anderes Modul abgeben, sprechen wir von einer *Umkehrung des Kontrollflusses*. Frameworks sind Zusammenstellungen von Klassen, die den Rahmen für den Ablauf von solchen Anwendungen liefern. Eine Anwendung, die ein Framework nutzt, ist damit von der Aufgabe entbunden, den Kontrollfluss selbst zu steuern. Sie stellt nur noch im Rahmen dieses Kontrollflusses benötigte fachliche Implementierungen zur Verfügung, der Kontrollfluss wurde damit umgekehrt.

Schablonenmethoden

Ebenfalls häufig in Frameworks anzutreffen sind neben den bereits erwähnten Fabriken die Schablonenmethoden. Wir haben sie in Abschnitt 5.3.1, »Überschreiben von Methoden«, bereits vorgestellt. Schablonenmethoden geben einen Rahmen vor, in dem Operationen aufgerufen werden, die erst von abgeleiteten Klassen implementiert werden müssen. Dadurch können Frameworks über abstrakte Klassen eine Sequenz von Aktionen steuern.

In Abschnitt 8.2 werden Sie den Ansatz kennenlernen, das Model-View-Controller-Muster (MVC) für die Steuerung des Kontrollflusses in der Präsentationsschicht zu nutzen. In einem MVC-Framework können zum Beispiel die Klassen, die für konkrete Darstellungen auf dem Bildschirm zuständig sind, als Ableitungen von abstrakten Klassen umgesetzt werden. Diese müssen dann für ihre korrekte Aktualisierung sorgen. Die Erstellung dieser auch *Views* genannten Klassen und die Interaktion mit anderen Komponenten können jedoch vom Framework übernommen werden.

Swing unterstützt Aspekte von MVC

Typische Beispiele sind hierbei Frameworks, die den Ablauf für Interaktionen an einer Benutzeroberfläche steuern. So bringt zum Beispiel das GUI-Framework Swing von Java Bestandteile mit, die generell die Interaktion innerhalb eines MVC-Ansatzes regeln. Die vom Anwendungsprogrammierer eingebrachten Module fügen sich in den Ablauf des Frameworks ein. Die neu umgesetzte Funktionalität wird vom Framework innerhalb dessen eigenem Kontrollfluss aufgerufen.

Frameworks: Probleme

Frameworks haben allerdings in der Praxis mit ein paar Problemen zu kämpfen. Das Hauptproblem ist dabei, die vorzusehenden Erweiterungspunkte zu bestimmen, ohne schon alle konkreten Anwendungsfälle zu

kennen. Werden zu wenige oder die falschen Erweiterungspunkte eingebaut, ist das Framework zu unflexibel. Werden zu viele Erweiterungspunkte eingebaut, wird das Framework zu komplex und schwierig zu warten.

Was soll erweiterbar sein?

In der Praxis ist es nicht einfach, die Punkte zu bestimmen, die für Erweiterungen zugänglich sein sollen. Oft wird zu viel von den Eigenschaften der Framework-Klassen öffentlich gemacht. Ist diese öffentlich gemachte Information dann einmal von Framework-Anwendern genutzt worden, kann sie praktisch nicht mehr geändert werden und verhindert zu einem gewissen Grad notwendige Umbauarbeiten innerhalb des Frameworks. Nach unserer Erfahrung hat ein Framework diesen Zustand erreicht, wenn häufiger der Satz fällt: »Aber ihr verwendet das Framework doch ganz falsch! Dafür war diese Methode nie gedacht!« In diesem Fall wurde zu viel von den Interna des Frameworks offengelegt, oder es ist einfach insgesamt zu komplex geworden.

JUnit

Frameworks haben sich deshalb meist dort bewährt, wo sie für einen klar definierten und überschaubaren Einsatzbereich konzipiert worden sind. Ein gutes Beispiel ist das JUnit-Framework, das erfolgreich für den Test von Java-Modulen eingesetzt wird. Zunächst einmal weist JUnit ein sauberes Design auf.

Ein weiterer sehr wichtiger Punkt ist aber, dass der von JUnit abgedeckte Bereich so eingegrenzt ist, dass die notwendigen Erweiterungspunkte des Frameworks überschaubar geblieben sind. Deshalb ist JUnit für den gewählten Einsatzbereich auch so nützlich und stabil. Erst die Erweiterung der Sprache Java um die Annotations führte zu größeren Änderungen an JUnit.

Frameworks für technische Abläufe

In den meisten Anwendungsfällen versuchen Frameworks, die technischen Abläufe zu kapseln, sodass die Umsetzung der Fachlichkeit erfolgen kann, ohne sich mit allen technischen Details beschäftigen zu müssen. JUnit und die bereits erwähnten MVC-Frameworks sind Beispiele dafür. Komplexer wird die Aufgabenstellung, wenn Gemeinsamkeiten von fachlichen Abläufen und Objekten über ein Framework strukturiert werden sollen. Sofern das innerhalb eines Unternehmens mit einer weitgehend homogenen Sicht auf Objekte und Prozesse geschieht und außerdem das entstehende Framework auf beobachteten Gemeinsamkeiten verschiedener Bereiche basiert, ist eine Framework-Umsetzung auch noch aussichtsreich.

Fachliche Frameworks

Die Ausdehnung von Frameworks auf die Fachlichkeit von Geschäftsanwendungen über verschiedene Unternehmen und Branchen hinweg hat sich dagegen als sehr komplexe Aufgabe erwiesen. IBM hat mit den San

Francisco Framework Classes gegen Ende der 90er-Jahre einen Versuch unternommen, Frameworks auf den fachlichen Bereich von Unternehmensanwendungen auszudehnen. Ziel war es dabei, die Gemeinsamkeiten von Geschäftsobjekten und den zugeordneten Prozessen über ein fachliches Framework abzubilden. Die spezifischen Ausprägungen sollten dann über die Mittel der zur Verfügung stehenden Erweiterungspunkte an die fachlichen Anforderungen erfolgen. Die San Francisco Classes waren allerdings nie wirklich erfolgreich.

Nach unserer Einschätzung sind Frameworks, die relevante Teile von Fachlichkeit bereits abbilden, nur dann realistisch einsetzbar, wenn nicht nur das Framework, sondern auch die Geschäftsprozesse selbst anpassbar sind. Der klassische Ansatz von SAP illustriert das: Dort gibt die Software einen relevanten Teil der Prozesse vor, ein nutzendes Unternehmen muss sich in bestimmtem Umfang daran anpassen.

Container

In der Praxis hat sich eine besondere Form von Frameworks etabliert, deren Fokus darauf liegt, den Lebenszyklus von Objekten selbst zu verwalten. Diese Frameworks werden als *Container* bezeichnet.

Container

Container sind eine spezielle Form von Frameworks, die sich um den Lebenszyklus von Objekten kümmern. Für die so verwalteten Objekte bieten Container dann Basisdienste an und machen sie für Anwender nutzbar.

Der Begriff *Container* rührt daher, dass die Objekte einer Applikation im Container enthalten sind. Sie werden also komplett unter dessen Kontrolle gestellt. Dafür, dass sie die Dienste eines Containers in Anspruch nehmen können, müssen Objekte, die von einem Container verwaltet werden, im Gegenzug ebenfalls eine Leistung erbringen. Der mit dem Container geschlossene Kontrakt erfordert dabei häufig, dass sich die vom Nutzer des Containers eingebrachten Komponenten eine Reihe von Schnittstellen zur Verfügung stellen bzw. implementieren und sich an bestimmte Konventionen halten.

Ein Beispiel dafür sind die Container für *Java Enterprise Edition*-Anwendungen (*JEE*). Dort dient der Anwendungsserver als Container für verschiedene Anwendungen und Komponenten, der für eine Reihe von übergreifenden Aufgaben zuständig ist. Er koordiniert, welche Anwendung die Dienste welcher Komponenten verwendet und wann welche Dienste gestartet und gestoppt werden.

Container im Allgemeinen sind aber Laufzeitumgebungen, in denen Objekte verwaltet werden und dort bestimmte Services nutzen können. Es ist also nicht grundsätzlich notwendig, dass ein solcher Container Teil eines Applikationsservers ist.

Leichtgewichtige Container

Diese Bindung an Applikationsserver aufzuheben, ist eine Zielsetzung der sogenannten leichtgewichtigen Container. Ein Beispiel für solche Container ist der Ansatz des Spring-Frameworks, bei dem die fachlichen Klassen alle als einfache Java-Klassen, sogenannte *Pojos*, umgesetzt werden.[3]

Spring Framework

Ein gutes Beispiel für leichtgewichtige Container ist das *Spring Framework* (*https://spring.io/projects/spring-framework*).

Das Framework bietet neben vielen anderen Funktionalitäten auch einen Container, der das Anlegen und das Zusammenspiel von Objekten steuert. Die zugehörige Konfiguration kann wahlweise in einer XML-Konfigurationsdatei, direkt im Java-Code oder über Java-Annotations erfolgen.

Dependency Injection

Den Mechanismus der *Dependency Injection* haben wir bereits in Abschnitt 7.2.7 beschrieben. Dependency Injection ist ein Mechanismus, der als Bestandteil einer Architektur verwendet werden kann. So ist zum Beispiel bei Verwendung des Spring-Frameworks Dependency Injection ein zentrales Prinzip. Über Dependency Injection werden die Abhängigkeiten zwischen genutzten Modulen und ihren Nutzern aus dem Code ausgelagert. Der Container, der unsere Objekte verwaltet, ist dafür zuständig, die genutzten Objekte bereitzustellen. Die Entscheidung, ob dabei ein einziges Exemplar ausreicht oder ob für jeden Nutzungsfall ein eigenes Exemplar notwendig ist, liegt in der Konfiguration bzw. beim Container. Bei Nutzung dieses Mechanismus ist es zum Beispiel auch nicht mehr notwendig, mit Singletons zu arbeiten. Ein Singleton wäre in diesem Fall durch Konfiguration über den Container zu erreichen, wir haben aber bei

3 Der Begriff *Pojo* für *Plain Old Java Objects* wurde von Martin Fowler eingeführt (*http://www.martinfowler.com/bliki/POJO.html*). Die Motivation war, den guten alten Java-Klassen durch einen hippen Namen wieder zu mehr Präsenz zu verhelfen. Überhaupt gelingt es Martin Fowler häufig, Begriffe für Vorgehensweisen und Sachverhalte zu prägen. So stammt auch der Begriff *Dependency Injection* für eine bestimmte Form der Umkehrung des Kontrollflusses von ihm und Rod Johnson. Value Objects in der JEE-Spezifikation wurden zu *Data Transfer Objects*, nachdem Martin Fowler diese Benennung kritisiert hatte.

keiner Klasse mehr die unbedingte Voraussetzung, dass nur ein Exemplar davon existieren kann.

Plug-ins

Am Beispiel der freien Entwicklungsplattform Eclipse ist zu sehen, dass sich komplexe Applikationen auch auf der Grundlage von sogenannten *Plug-ins* aufbauen lassen.

Plug-in ist wieder ein recht schillernder Begriff. Wir bauen unsere Definition auf der Beschreibung von Martin Fowler auf, der eine große Erfahrung in der Definition von schwammigen Begriffen mitbringt.[4]

Plug-in

Ein Plug-in ist ein Modul, das an einem Erweiterungspunkt eines Programms eingesetzt werden kann. Dabei wird über eine zentrale Konfiguration gesteuert, welches Plug-in verwendet werden soll und wie das Plug-in selbst konfiguriert wird.

Ein Plug-in löst zwei Probleme durch eine zentrale, zur Laufzeit ausgewertete Konfiguration: Konfigurationsinformation, verteilt in Fabriken über die Applikation, ist schwer zu pflegen. Außerdem soll eine Änderung der Konfiguration nicht erfordern, dass ein Neubau oder eine erneute Auslieferung notwendig wird.

Plug-ins, wie sie von Eclipse verwendet werden, basieren auf der Definition von Erweiterungspunkten. Hier ist es zum einen möglich, eigene Erweiterungen einzubringen. Auf der anderen Seite können sie aber bereits wieder selbst Erweiterungsmöglichkeiten definieren, sodass eine gestaffelte Erweiterung der Plattform möglich wird. Diese Möglichkeit wird von Eclipse ganz klassisch als Konzept der Erweiterungspunkte (*Extension Points*) bezeichnet.

Es deutet einiges darauf hin, dass Eclipse mit diesem Konzept zumindest für den zunächst gewählten Anwendungsbereich (nämlich eine Entwicklungsplattform) auf einem guten Weg ist. Der Ansatz vermeidet viele Fallen der klassischen Frameworks und hält die Komplexität durch seine klare Aufgabentrennung über Erweiterungspunkte in überschaubarem Rahmen.

4 Siehe auch Martin Fowler: *Patterns für Enterprise Application-Architekturen.* MITP 2003. Kurzfassung unter *http://www.martinfowler.com/eaaCatalog/plugin.html*.

Einfacher Einbau der Erweiterungspunkte

Was hier ebenfalls wichtig ist und was nicht bei allen Komponentenmodellen bisher beachtet wurde: Es muss einfach sein, einen Erweiterungspunkt einzubauen, sobald klar wird, dass dieser benötigt wird. Damit sind wir nicht gezwungen, von vornherein alle möglichen Erweiterungspunkte in unserem Modul vorwegzunehmen.

8.2 Die Präsentationsschicht: Model, View, Controller (MVC)

Ein sehr erheblicher Teil der Funktionalität auch von objektorientierten Systemen spielt sich bei der Interaktion mit den Anwenderinnen und Anwendern von Software ab. Für die Modellierung dieser Interaktion in der Präsentationsschicht gibt es verschiedene Ansätze. Am weitesten verbreitet ist dabei der sogenannte *MVC-Ansatz* (*Model-View-Controller*).

Mit Model-View-Controller (MVC) wird ein Interaktionsmuster in der Präsentationsschicht von Software beschrieben. MVC ist wohl einer der schillerndsten Begriffe im Bereich der objektorientierten Programmierung. Viele Varianten haben sich herausgebildet, teilweise einfach aufgrund eines falschen Verständnisses des ursprünglichen MVC-Musters, teilweise als Weiterentwicklung oder Anpassung an neue Anwendungsfälle.

Da es sich bei MVC nach wie vor um das wichtigste und verbreitetste Muster für die Präsentationsschicht von objektorientierten Anwendungen handelt, gehen wir in diesem Kapitel ausführlich darauf ein.

8.2.1 Das Beobachter-Muster als Basis von MVC

Eine ganz zentrale Art von Information in objektorientierten Systemen ist die Information darüber, dass ein Objekt seinen Zustand geändert hat.

Interaktionen in der Präsentationsschicht

Die Interaktionen, die hierbei entstehen, können komplex sein. Nach dem *Prinzip einer Verantwortung* sollten Sie aber vermeiden, dass die betroffenen Objekte sich gegenseitig kennen müssen. Für derartige Fälle bietet es sich an, das Entwurfsmuster »Beobachtetes-Beobachter« anzuwenden. Das Muster wird auch kurz Beobachter-Muster genannt. In Abschnitt 5.4, »Mehrfachvererbung«, hatten wir es bereits an einem Beispiel vorgestellt. Abbildung 8.1 zeigt die bei der Anwendung des Musters beteiligten Klassen noch einmal in der Übersicht für den allgemeinen Fall.

In der Abbildung ist zu sehen, dass sich die Beobachter über die Operation `anmelden()` registrieren können. Wenn sie das getan haben, werden sie

über Änderungen des Zustands im beobachteten Objekt informiert. Das geschieht darüber, dass dieses Objekt nach Änderungen seine eigene Operation `benachrichtigen()` aufruft. Diese wird alle registrierten Beobachter durchgehen und deren Operation `aktualisieren()` aufrufen. Damit haben alle Beobachter die Möglichkeit, auf die Zustandsänderung zu reagieren.

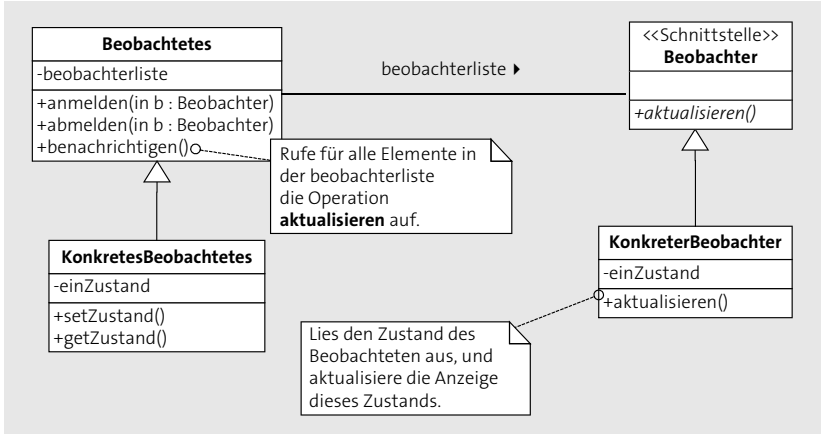


Abbildung 8.1 Beobachter und beobachtete Objekte

Innerhalb von MVC wird das Muster verwendet, um Änderungen an einem Modell an die Objekte zu kommunizieren, die das Modell darstellen, die sogenannten Views. Dabei sollen die Modelle nichts darüber wissen müssen, von welchen Objekten sie denn nun dargestellt werden.

8.2.2 MVC in Smalltalk: Wie es ursprünglich mal war

Für einen Überblick beginnen wir direkt am Anfang: MVC wurde zusammen mit der objektorientierten Programmiersprache Smalltalk eingeführt. Bei MVC handelte es sich ursprünglich um ein Konzept, das die Interaktionen eines Benutzers (vor allem über Mausaktionen) sauber von den dadurch veränderten Daten und deren Darstellung trennen sollte.

Controller: Verarbeitung von Eingaben

Dabei ist der *Controller* für die Verarbeitung der Eingaben (zum Beispiel Mausklicks) zuständig und für deren Kommunikation an das Modell. Das *Model* ist passiv, es wird vom Controller befragt und modifiziert. Der *View*[5] befragt das Modell, um auf dieser Grundlage seine Darstellung anzupassen.

5 Es ist gängig, den englischen Begriff *View* für eine Darstellungskomponente zu verwenden. Allerdings herrscht Uneinigkeit darüber, ob der Artikel nun als »der View« oder »die View« zu wählen ist. Wir verwenden im Folgenden die Version »der View«.

Wie aus Abbildung 8.2 hervorgeht, ist die Beziehung zwischen View und Controller klar definiert: Sie treten immer als Paar auf, und jeder kennt den jeweils anderen. Ein Modell kann aber von beliebig vielen View/Controller-Paaren betreut werden, was die Möglichkeit von verschiedenen Sichten auf dasselbe Modell eröffnet.

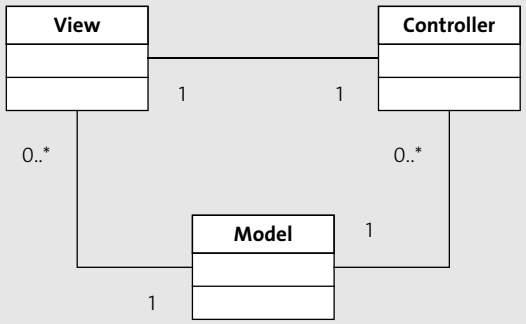


Abbildung 8.2 Beziehung zwischen Model, View und Controller in Smalltalk

Die Interaktion über das Beobachter-Muster findet zwischen Model und View statt. In Abbildung 8.3 ist dargestellt, wie das Muster für die Zusammenarbeit dieser beiden Bestandteile eingesetzt wird.

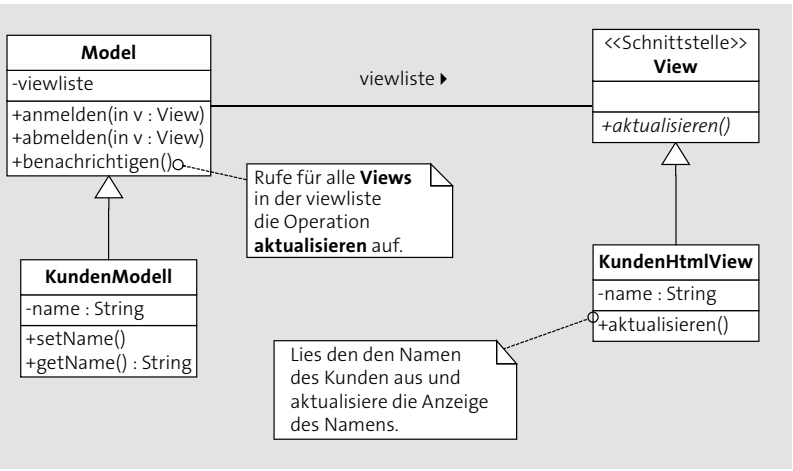


Abbildung 8.3 Beobachter-Muster in MVC

Im dargestellten Beispiel werden Änderungen an einem `KundenModell` an alle beteiligten Views kommuniziert. In diesem Fall hat ein Kunde lediglich einen Namen zugeordnet. Bei Änderungen an diesem Namen ruft das Modell die Operation `aktualisieren` auf. Wenn sich `KundenHtmlView` beim

Modell durch Aufruf der Operation `anmelden()` als Beobachter registriert hat, wird dieser mit benachrichtigt und kann in der Methode `aktualisieren()` die Anzeige des Namens anpassen.

8.2.3 MVC: Klärung der Begriffe

Obwohl die ursprünglich in Smalltalk eingeführte Trennung in Model, View und Controller recht eindeutig ist, werden wir in Diskussionen sehr unterschiedliche Verständnisse dessen antreffen, was denn nun ein Modell und was ein Controller ist.

Das rührt unter anderem daher, dass uns bestimmte Aufgaben, für die zum Beispiel ein Controller in Smalltalk zuständig war, mittlerweile ganz selbstverständlich abgenommen werden. Deshalb klären wir hier zunächst einmal, was denn mit Model, View und Controller gemeint ist.

Was ist denn nun eigentlich ein Controller?

Ein Controller hat in MVC eine recht generische Aufgabe: Er muss die Interaktionen eines Benutzers bzw. einer Benutzerin über die verschiedenen Eingabemöglichkeiten wie Tastatur oder Maus entgegennehmen und in konsolidierter Form weiterleiten.

Controller in MVC (Eingabe-Controller)

Ein Controller hat in der MVC-Variante von Smalltalk die Aufgabe, die Anwendung von den Komplexitäten der Eingabemechanismen abzuschotten. Zum Beispiel ist er dafür zuständig, Mausereignisse einem konkreten Bildschirmbereich zuzuordnen.

Die Funktionalität des Controllers ist mittlerweile meist recht selbstverständlich in grundlegende Bibliotheken integriert, die Anwendungsprogrammierer und -programmiererinnen in der Regel gar nicht mehr zu Gesicht bekommen.

Sie benötigen eben in der Regel keinen explizit angegebenen Controller mehr, um einen Mausklick auch einem konkreten Eingabeelement eines Views zuzuordnen. Diese Dinge werden Ihnen meist durch Module abgenommen, die in Betriebssystemen oder Basisbibliotheken integriert sind.

Wir verwenden für diese Art von Controller den Begriff *Eingabe-Controller*. Beim Entwickeln von Anwendungen muss man sich aber nur noch selten direkt mit Eingabe-Controllern beschäftigen.

Sie sollten den Eingabe-Controller auch möglichst klar unterscheiden von einem Applikations-Controller, der Abläufe innerhalb einer Applikation

steuert. Zum Beispiel ist dieser dafür zuständig, einen Folgedialog aufgrund einer Benutzereingabe zu ermitteln. Das hat aber mit der ursprünglichen Verwendung in MVC nur wenig zu tun. MVC ist nämlich auch in Smalltalk nie ein komplettes Architekturmodell gewesen, sondern eine Lösung für eine ganz bestimmte Teilaufgabe: die Interaktionen in der Präsentationsschicht sauber zu trennen.

Was ist denn nun eigentlich ein Modell?

Auch in Bezug auf das Modell innerhalb von MVC gibt es hin und wieder Unklarheiten. Möglicherweise kommt das auch daher, dass gerade in Smalltalk praktisch jedes Objekt als Modell agieren kann.

Das heißt aber lediglich, dass zumindest in der Theorie jedes Smalltalk-Objekt darstellbar ist und auf Nachrichten reagieren kann, die möglicherweise eine Zustandsänderung bewirken. Entscheidend dabei ist jedoch, dass das Modell den Zustand des gesamten Konstrukts verwaltet und als Referenz für die Darstellung dient.

In der Regel wird das Modell zwar weitere Komponenten in Anspruch nehmen, die dann die fachliche Funktionalität umsetzen, das Modell selbst ist aber nur durch die oben genannten Eigenschaften definiert.

Modell in MVC

Ein Modell in MVC muss also folgende Eigenschaften aufweisen:

- Es muss einen Zustand verwalten können.
- Es muss darstellbar sein.
- Es muss auf Aufforderungen zu Änderungen reagieren können.

Damit hat ein Modell in MVC zunächst nicht notwendig mit der Logik in der Domäne zu tun, auch nicht unbedingt mit Persistenz.

Und schließlich noch der View

Views im MVC-Modell sind die Komponenten, die für die Darstellung des Modells verantwortlich sind. Die Darstellungsarten sind dabei nicht eingeschränkt. Grundsätzlich könnte ein View die Daten des Modells auch durch die Nutzung eines Sprachgenerierungsmoduls vorlesen. Allerdings erscheint für diesen Fall die Bezeichnung View nicht mehr völlig adäquat.

Views sollten sich ausschließlich mit der Darstellung beschäftigen und möglichst keine fachliche Logik enthalten. Allerdings werden in einigen Fällen Funktionen des Controllers mit im View integriert.

View und Controller zusammengefasst

Controller erledigen in der Regel sehr gleichförmige Aufgaben. Schon bei Smalltalk gibt es einen kleinen Satz von Standardcontrollern, deren Exemplare dann jeweils einem View und einem Modell zugeordnet werden.

Sowohl bei den Microsoft Foundation Classes (MFC) als auch bei der Java-Oberflächenbibliothek Swing ist denn auch der Controller keine eigenständige Entität mehr. Bei Microsoft nennt sich das Ganze dann *Document-View*, bei Swing wird das Verfahren hin und wieder auch *Model-Delegate* genannt.

Auch wenn die Begriffe hier schon wieder etwas verwirrend sind: Es handelt sich doch in beiden Fällen um eine Modellierung, bei der View und Controller zusammenfallen, bei Microsoft eben im View, bei Swing im Delegate – der Oberflächenkomponente, die für die Anzeige und für die Behandlung von Benutzereingaben zuständig ist.

Vorsicht, Falle Nummer 1: MVC ist nicht gleich Schichtenmodell

MVC ist ein Muster für Interaktionen in der Präsentationsschicht. Es setzt damit bereits voraus, dass wir uns grundsätzlich auf eine Architektur eingelassen haben, die Schichten vorsieht und die Präsentation vom Rest der Anwendung trennt.

MVC ist keine komplette Architektur

In der Praxis und in der Literatur haben wir aber häufiger eine Interpretation von MVC gesehen, die dieses Muster generell zur kompletten Architektur für eine Applikation erhoben hat. Außerdem wird oft auch die Rolle des Modells überdehnt, indem diesem die Verantwortung für Geschäftslogik, Persistenz oder andere zentrale Aspekte zugeschoben wird.

Sie sollten aber MVC nicht mit einer kompletten Schichtenarchitektur verwechseln. Das Modell ist nämlich nicht per se für die Umsetzung der Schicht der Anwendungslogik zuständig. Natürlich können beide Aufgaben (Bereitstellung der Anwendungslogik und Darstellbarkeit in der Präsentationsschicht) in der Praxis von denselben Objekten übernommen werden. So können Sie natürlich eine Enterprise Java Bean als Modell verwenden, um sie direkt über die Präsentationsschicht darzustellen. Die Modell-Eigenschaft ist dabei aber nach wie vor völlig unabhängig von den anderen Eigenschaften der EJB, wie z. B. der Fähigkeit, Daten persistent zu machen.

Allerdings gibt es mittlerweile Erweiterungen des Musters, die auch die Geschäftslogik mit in den Fokus nehmen. Das gilt vor allem für den Ansatz MVVM (Model View ViewModel).

Model View ViewModel (MVVM)

MVVM ist eine Erweiterung des MVC-Ansatzes, bei dem eine direkte Kopplung eines Views (der UI-Darstellung) an ein ViewModel mit den anzuzeigenden Daten erfolgt. Der Datenaustausch erfolgt mittels einer direkten Bindung von Datenelementen an Oberflächenelemente.

Das ViewModel ist damit praktisch ein Vermittler zwischen der reinen Darstellung (View) und der über das Model repräsentierten Geschäftslogik auf der anderen Seite. Das ViewModel greift auf die Logik im Model zu, soll aber selbst nichts über den View wissen. Damit ist die Darstellung über den View austauschbar ohne dass ViewModel oder Model angepasst werden müssten.

Stefan: *Hmm, so ganz klar ist die Beschränkung auf die Präsentationsschicht damit ja in der MVVM-Variante von MVC nicht mehr.*

Bernhard: *Wo geht das darüber hinaus? Immerhin ist ja der zentrale Teil von MVVM die Trennung zwischen den darzustellenden Daten und der Darstellung.*

Stefan: *Naja, im MVVM-Ansatz hält zwar das ViewModel die Daten für die Anzeige. Das Model selbst ist aber für Datenspeicherung und Geschäftslogik zuständig, trotzdem ist es Bestandteil des Ansatzes. Damit ist das Model doch eher Teil der Geschäftslogik als der reinen Präsentationsschicht.*

Bernhard: *Da hast du recht. MVVM ist damit eine Erweiterung gegenüber dem ursprünglichen MVC. Indem das spezifische ViewModel in Interaktion mit dem View tritt, lässt sich allerdings doch wieder die Präsentation von der Geschäftslogik trennen, die dann im Model landet.*

Stefan: *Ja, so können wir das sehen. Allerdings finde ich die Namensgebung in diesem Ansatz etwas unglücklich, weil das Model hier seinen Namen behält, aber gegenüber dem ursprünglichen MVC-Ansatz eine ganz andere Rolle einnimmt.*

Vorsicht, Falle Nummer 2: Nicht jeder View braucht eine eigene Controllerklasse

Durch die bestehende 1:1-Verbindung zwischen Views und Controllern in MVC könnte man leicht auf die Idee kommen, dass diese Beziehung nicht nur für die Exemplare, sondern auch für die Klassen gelten könnte.

Das ist aber schon bei der ursprünglichen Version von MVC in Smalltalk nie so geplant gewesen. Dort bekommt zwar jeder View sein spezielles

und eigenes Exemplar eines Controllers zugewiesen, da auch ein Controller einen Zustand besitzt, der direkt mit dem View zusammenhängt. Aber in Smalltalk gab es eine kleine Anzahl von Controller-Klassen, und deren Exemplare wurden zusammen mit den jeweiligen spezifischen Views erzeugt.

Auch waren in Smalltalk die Controller für Aufgaben zuständig, die Sie heute in der Regel bei der Entwicklung einer Applikation gar nicht mehr als Aufgabe wahrnehmen, weil sie von Bibliotheken übernommen werden, die wir mittlerweile ganz selbstverständlich voraussetzen. Sie müssen sich eben in der Regel nicht mehr damit beschäftigen, wie Sie aus den Koordinaten eines Mausklicks darauf schließen könnten, welches Element der Oberfläche davon nun wie betroffen ist. Das war aber eine der Aufgaben von Controllern im ursprünglichen MVC.

8.2.4 MVC in Webapplikationen: genannt »Model 2«

Bereits in frühen Versionen der JEE-Spezifikation gab es zwei Varianten des empfohlenen MVC-Musters, die *Model 1* und *Model 2* genannt wurden.[6] Diese Begriffe haben sich länger gehalten, obwohl sie in der Spezifikation mittlerweile nicht mehr enthalten sind.

Model 1

Model 1 ist dabei eine sehr einfache Variante, in der Zugriffe auf Modelle (meist Java Beans) direkt aus JSP-Seiten[7] heraus erfolgen. Abbildung 8.4 zeigt eine Interaktion auf Basis dieses Modells.

Model 1 ist sehr einfach und überschaubar, allerdings nur für relativ kleine Anwendungen zu empfehlen. Änderungen an der Navigationsstruktur durch die verschiedenen Seiten sind nur mit großem Aufwand durchzuführen, da dafür direkt in die JSP-Seiten eingegriffen werden muss.

Model 1 für Webapplikationen

Model 1 ist eine einfache Architektur für die Interaktion mit der Präsentationsschicht von Webapplikationen. Es existiert keinerlei Controller, sondern der View (eine JSP-Seite) kommuniziert direkt mit dem Modell. Das Modell wird dabei in der Regel durch eine sogenannte Java Bean repräs-

6 Auszusprechen als Model One und Model Two. Die JEE-Spezifikation (Java Platform, Enterprise Edition) haben wir in Abschnitt 4.4.4, »Identität von Objekten«, mit dem Fokus auf Enterprise Java Beans kennengelernt.

7 JSP (Java Server Pages) sind ebenfalls ein standardisiertes Verfahren aus der JEE-Spezifikation, wie Java-Code in HTML-Seiten integriert werden kann. Eine JSP-Seite ist ein HTML-Dokument, das bei seiner Anzeige dynamisch modifiziert und mit Daten angereichert wird.

sentiert. Java Beans sind einfache Java-Klassen, die sich an Konventionen für die Namen von Operationen halten, die auf ihre Daten zugreifen.

Bei Verwendung von Model 1 liegt die Entscheidung darüber, welche Folgeseite angezeigt wird, allein bei der aktuellen JSP-Seite. Als Reaktion auf eine bestimmte Benutzereingabe wird direkt dort über die in der Folge zu präsentierende JSP-Seite entschieden.

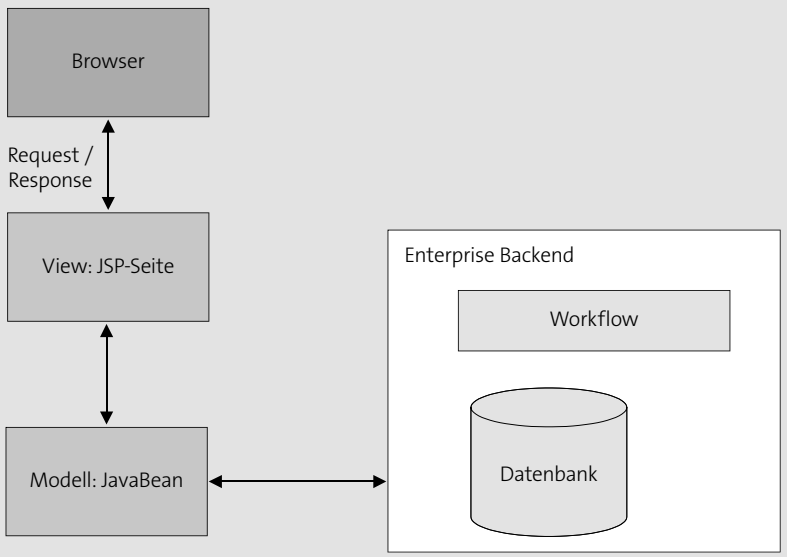


Abbildung 8.4 Model 1 für Webapplikationen

Model 2 Model 2 für Webapplikationen lagert die Entscheidung darüber, welche Folgeseiten aufgerufen werden, in eine eigene Komponente aus und erlaubt damit mehr Flexibilität.

Model 2 für Webapplikationen

Bei Model 2 für Webapplikationen erfolgt die Steuerung für die Dialogabfolge durch eine eigene Komponente, den sogenannten Controller. Der Controller übernimmt dabei Aufgaben, die gegenüber der ursprünglichen Variante in MVC angepasst sind. Die Aufgabenverteilung ist bei Verwendung von Model 2 damit folgende:

- **Der View:** Eine JSP-Seite, die Daten darstellt und Dateneingaben vom Benutzer annimmt. Diese Seite kann Logik enthalten, allerdings entscheidet sie nicht darüber, welche JSP als Folgeseite angezeigt wird.

- **Der Controller:** Ein Servlet, das grundsätzlich von JSP aus aufgerufen wird und aufgrund der vorgenommenen Eingaben entscheidet, welche Aktion ausgeführt werden soll und welche Folgeseite aufgerufen wird.
- **Das Modell:** Ein Objekt, das die darzustellenden Daten hält, meist eine Java Bean.

Eine Umsetzung der Model-2-Variante von MVC bietet das Framework *Jakarta Struts*. In Abbildung 8.5 ist dargestellt, wie eine Interaktion auf Basis dieses Modells erfolgt.

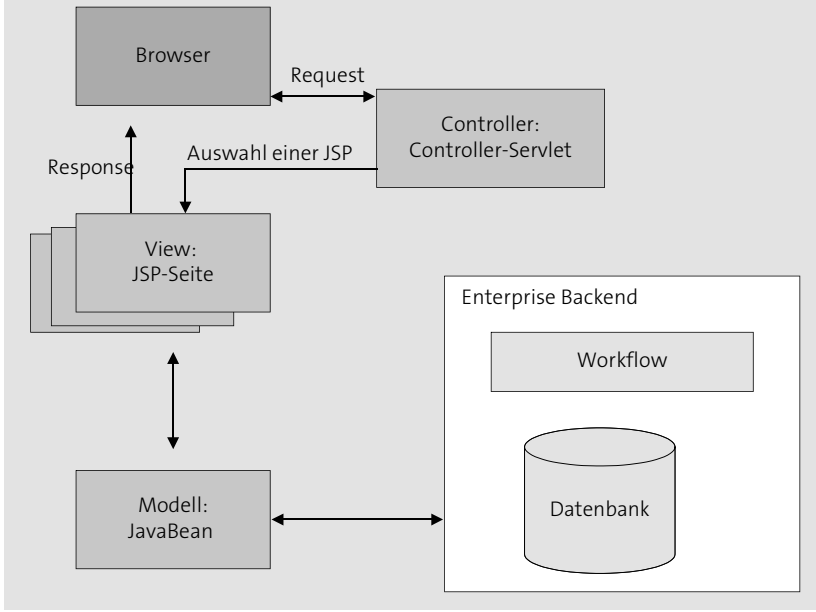


Abbildung 8.5 Model 2 für Webapplikationen

Hier wird ersichtlich, dass der Controller gegenüber Smalltalk weitere Aufgaben bekommen hat: Er steuert einen Teil der Applikationslogik, ist also nicht mehr nur für die Kommunikation der Benutzerinteraktion an Modell und View zuständig. Sie haben es hier mit einer Erweiterung des MVC-Musters zu tun.

8.2.5 MVC mit Fokus auf die Testbarkeit: Model-View-Presenter

Ein Aspekt, der in der Regel bei der Modellierung der Präsentationsschicht nur unzureichend betrachtet wird, ist die Testbarkeit der resultierenden Applikation.

Tests von Oberflächen sind schwierig

Es ist ein Erfahrungswert, dass Tests unter Beteiligung von grafischen Benutzeroberflächen wesentlich schwieriger zu automatisieren sind als Tests von Softwarekomponenten, die ohne eine grafische Darstellung auskommen.

Es gibt eine ganze Reihe von Produkten, die versuchen, dieses Problem durch automatisierte Tests unter Beteiligung von grafischen Oberflächen zu lösen. Besser ist es aber, bereits beim Design unserer Software sicherzustellen, dass der überwiegende Teil ohne Beteiligung von grafischen Benutzeroberflächen getestet werden kann. Ein Muster in der Präsentationsschicht, das das Problem angeht, wurde von Martin Fowler kategorisiert und nennt sich *Model-View-Presenter*. Unter *http://martinfowler.com/eaaDev/ModelViewPresenter.html* hat Martin Fowler das Pattern allerdings mittlerweile weiter differenziert, sodass es in die zwei Varianten *Passive View* und *Supervising Controller* zerfällt.[8] Beide unterscheiden sich aber im Kern nur durch den Anteil von Logik, der im View verbleibt. Das Muster ist auch nahe am bereits diskutierten MVVM-Ansatz, in dem das ViewModel die Aufgaben des Presenters wahrnimmt. Grundgedanke bei allen diesen Mustern ist, die eigentliche Darstellung (den View) weitgehend von technischer und fachlicher Logik frei zu halten.

Je direkter eine Applikation mit den Darstellungskomponenten gekoppelt ist, desto schwieriger wird es, sie komplett zu testen.

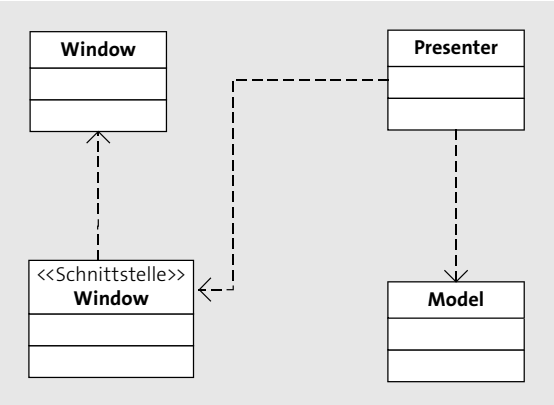


Abbildung 8.6 Beziehung Model – View – Presenter

Beim Ansatz des Model-View-Presenters werden Ereignisse direkt vom View verarbeitet. Aber der View delegiert sie gleich weiter an die Presenter-Klasse. Diese wiederum gibt möglicherweise Rückmeldungen an den View. Warum ist es nun aber sinnvoll, in der Kommunikation zwischen View und Presenter eine Abstraktionsschicht über eine Schnittstellenklasse einzuziehen?

8 Die Beschreibungen dazu finden sich ebenfalls bei Martin Fowler: für Passive View unter *http://martinfowler.com/eaaDev/PassiveScreen.html*, für Supervising Controller unter *http://martinfowler.com/eaaDev/SupervisingPresenter.html*.

Abstraktion über eine Schnittstellenklasse

Nun, eine Möglichkeit ist die Wiederverwendung von Presenter-Klassen, wenn ein Presenter mit mehreren verschiedenen Views zusammenarbeiten soll, die lediglich dasselbe Interface implementieren. Dieser Fall ist in der Praxis aber eher selten.

Viel wichtiger ist die daraus entstehende Möglichkeit, den View für Testzwecke durch ein Ersatzobjekt (ein sogenanntes Mock Object) zu ersetzen. Damit haben Sie die Möglichkeit, sämtliche Logik Ihres Programms weitgehend automatisiert zu testen. Lediglich die konkrete Darstellung bleibt außen vor, aber diese ist in der Regel weniger fehleranfällig als andere Teile. Sie können durch einen Test der Presenter-Klasse zum Beispiel auch feststellen, ob Querabhängigkeiten zwischen Oberflächenelementen korrekt ausgewertet werden.

Diskussion: Ist MVP zu viel Aufwand?

Gregor: *Testbarkeit schön und gut. Aber ist das jetzt nicht etwas übertrieben? Überleg mal: Für jeden Dialog, jedes Fenster in unserer Anwendung müssen wir nun noch einmal ein Interface definieren und auch noch eine zusätzliche Presenter-Klasse, die wir uns eigentlich auch schenken könnten.*

Bernhard: *Okay, ich gebe zu, dass das nach unnötiger Komplexität aussieht. Ich denke auch nicht, dass es für alle Softwaresysteme Sinn ergibt. Aber meine Erfahrung ist einfach, dass oft eine ganze Menge Aufwand in GUI-spezifischen Anwendungsteilen versenkt wird, weil diese nicht vernünftig testbar sind.*

Gregor: *Aber ist denn nicht der Einsatz von GUI-spezifischen Testtools genau für solche Fälle gedacht? Wir können da doch zum Beispiel einfach Selenium anwerfen, und dann das ganze System mit den GUI-Komponenten zusammen durchtesten.*

Bernhard: *Das ist nicht das Gleiche. Regressionstests sind nach meiner Erfahrung viel einfacher zu erstellen und auch zu pflegen, wenn sie sich nur mit Sourcecode beschäftigen. Sobald ich Makros dazu schreiben, mich auf Beschriftungen in Dialogen verlassen und ein externes Tool anwerfen muss, wird das ganze Verfahren selbst schon sehr kompliziert und fehleranfällig.*

Gregor: *Na gut, zugegeben, das Testen wird einfacher. Aber ob damit der zusätzliche Aufwand in der Implementierung wettgemacht wird, davon bin ich noch nicht komplett überzeugt. Und warum überhaupt eine zusätzliche*

Klasse, die View-Klasse selbst würde doch völlig ausreichen, warum teste ich nicht diese direkt?

Bernhard: *Ja, das ist ein guter Punkt. Allerdings sind in der Praxis die View-Klassen oft stark mit der ganz konkreten Darstellung verbunden und lassen sich eben nicht davon unabhängig verwenden. Genau diese Trennung versuchen wir mit unserer neu eingezogenen Schnittstelle erst zu erreichen.*