

Bernhard Lahres, Gregor Rayman

Objektorientierte Programmierung

Das umfassende Handbuch



Auf einen Blick

1	Einleitung	13
2	Die Basis der Objektorientierung	27
3	Die Prinzipien des objektorientierten Entwurfs	39
4	Die Struktur objektorientierter Software	65
5	Vererbung und Polymorphie	155
6	Persistenz	299
7	Abläufe in einem objektorientierten System	337
8	Module und Architektur	503
9	Aspekte und Objektorientierung	527
10	Objektorientierung am Beispiel: Eine Web-Applikation mit PHP 5 und Ajax	573
A	Verwendete Programmiersprachen	623
B	Literaturverzeichnis	641

Inhalt

1	Einleitung	13
1.1	Was ist Objektorientierung?	13
1.2	Hallo liebe Zielgruppe	14
1.3	Was bietet dieses Buch (und was nicht)?	15
1.3.1	Bausteine des Buchs	16
1.3.2	Crosscutting Concerns: übergreifende Anliegen	19
1.3.3	Die Rolle von Programmiersprachen	20
1.4	Warum überhaupt Objektorientierung?	22
1.4.1	Gute Software: Was ist das eigentlich?	23
1.4.2	Die Rolle von Prinzipien	24
1.4.3	Viele mögliche Lösungen für ein Problem	25
2	Die Basis der Objektorientierung	27
2.1	Die strukturierte Programmierung als Vorläufer der Objektorientierung	28
2.2	Die Kapselung von Daten	31
2.3	Polymorphie	32
2.4	Die Vererbung	34
2.4.1	Vererbung der Spezifikation	34
2.4.2	Erben von Umsetzungen (Implementierungen)	35
3	Die Prinzipien des objektorientierten Entwurfs	39
3.1	Prinzip 1: Prinzip einer einzigen Verantwortung	40
3.2	Prinzip 2: Trennung der Anliegen	45
3.3	Prinzip 3: Wiederholungen vermeiden	47
3.4	Prinzip 4: Offen für Erweiterung, geschlossen für Änderung	50
3.5	Prinzip 5: Trennung der Schnittstelle von der Implementierung	53
3.6	Prinzip 6: Umkehr der Abhängigkeiten	56
3.6.1	Umkehrung des Kontrollflusses	60
3.7	Prinzip 7: Mach es testbar	62
4	Die Struktur objektorientierter Software	65
4.1	Die Basis von allem: das Objekt	65
4.1.1	Eigenschaften von Objekten: Objekte als Datenkapseln	67

4.1.2	Operationen und Methoden von Objekten	74
4.1.3	Kontrakte: Ein Objekt trägt Verantwortung	79
4.1.4	Die Identität von Objekten	81
4.1.5	Objekte haben Beziehungen	83
4.2	Klassen: Objekte haben Gemeinsamkeiten	84
4.2.1	Klassen sind Modellierungsmittel	84
4.2.2	Kontrakte: die Spezifikation einer Klasse	88
4.2.3	Klassen sind Datentypen	92
4.2.4	Klassen sind Module	102
4.2.5	Sichtbarkeit von Daten und Methoden	105
4.2.6	Klassenbezogene Methoden und Attribute	112
4.2.7	Singleton-Methoden: Methoden für einzelne Objekte.....	116
4.3	Beziehungen zwischen Objekten	117
4.3.1	Rollen und Richtung einer Assoziation	119
4.3.2	Navigierbarkeit	120
4.3.3	Multiplizität	120
4.3.4	Qualifikatoren	125
4.3.5	Beziehungsklassen, Attribute einer Beziehung	126
4.3.6	Implementierung von Beziehungen	128
4.3.7	Komposition und Aggregation	129
4.3.8	Attribute	132
4.3.9	Beziehungen zwischen Objekten in der Übersicht	133
4.4	Klassen von Werten und Klassen von Objekten	133
4.4.1	Werte in den objektorientierten Programmiersprachen ...	134
4.4.2	Entwurfsmuster »Fliegengewicht«	137
4.4.3	Aufzählungen (Enumerations)	140
4.4.4	Identität von Objekten	147
5	Vererbung und Polymorphie	155
5.1	Die Vererbung der Spezifikation	155
5.1.1	Hierarchien von Klassen und Unterklassen	155
5.1.2	Unterklassen erben die Spezifikation von Oberklassen.....	157
5.1.3	Das Prinzip der Ersetzbarkeit	161
5.1.4	Abstrakte Klassen, konkrete Klassen und Schnittstellen-Klassen	167
5.1.5	Vererbung der Spezifikation und das Typsystem	176
5.1.6	Sichtbarkeit im Rahmen der Vererbung	183
5.2	Polymorphie und ihre Anwendungen	193
5.2.1	Dynamische Polymorphie am Beispiel	195
5.2.2	Methoden als Implementierung von Operationen	200

5.2.3	Anonyme Klassen	208
5.2.4	Single und Multiple Dispatch	210
5.2.5	Die Tabelle für virtuelle Methoden	228
5.3	Die Vererbung der Implementierung	239
5.3.1	Überschreiben von Methoden	241
5.3.2	Das Problem der instabilen Basisklassen	249
5.3.3	Problem der Gleichheitsprüfung bei geerbter Implementierung	254
5.4	Mehrfachvererbung	261
5.4.1	Mehrfachvererbung: Möglichkeiten und Probleme	261
5.4.2	Delegation statt Mehrfachvererbung	268
5.4.3	Mixin-Module statt Mehrfachvererbung	271
5.4.4	Die Problemstellungen der Mehrfachvererbung	273
5.5	Statische und dynamische Klassifizierung	289
5.5.1	Dynamische Änderung der Klassenzugehörigkeit	290
5.5.2	Entwurfsmuster »Strategie« statt dynamischer Klassifizierung	294

6 Persistenz 299

6.1	Serialisierung von Objekten	299
6.2	Speicherung in Datenbanken	300
6.2.1	Relationale Datenbanken	300
6.2.2	Struktur der relationalen Datenbanken	301
6.2.3	Begriffsdefinitionen	302
6.3	Abbildung auf relationale Datenbanken	307
6.3.1	Abbildung von Objekten in relationalen Datenbanken	307
6.3.2	Abbildung von Beziehungen in relationalen Datenbanken	311
6.3.3	Abbildung von Vererbungsbeziehungen auf eine relationale Datenbank	315
6.4	Normalisierung und Denormalisierung	320
6.4.1	Die erste Normalform: Es werden einzelne Fakten gespeichert	322
6.4.2	Die zweite Normalform: Alles hängt vom ganzen Schlüssel ab	323
6.4.3	Die dritte Normalform: Keine Abhängigkeiten unter den Nichtschlüssel-Spalten	326
6.4.4	Die vierte Normalform: Trennen unabhängiger Relationen	330
6.4.5	Die fünfte Normalform: Einfacher geht's nicht.....	332

7 Abläufe in einem objektorientierten System..... 337

7.1	Erzeugung von Objekten mit Konstruktoren und Prototypen	338
7.1.1	Konstruktoren: Klassen als Vorlagen für ihre Exemplare	338
7.1.2	Prototypen als Vorlagen für Objekte	342
7.1.3	Entwurfsmuster »Prototyp«	348
7.2	Fabriken als Abstraktionsebene für die Objekterzeugung	349
7.2.1	Statische Fabriken	352
7.2.2	Abstrakte Fabriken	355
7.2.3	Konfigurierbare Fabriken	360
7.2.4	Registraturen für Objekte	364
7.2.5	Fabrikmethoden	368
7.2.6	Erzeugung von Objekten als Singletons	377
7.2.7	Dependency Injection	386
7.3	Objekte löschen	397
7.3.1	Speicherbereiche für Objekte	397
7.3.2	Was ist eine Garbage Collection?	399
7.3.3	Umsetzung einer Garbage Collection	400
7.4	Objekte in Aktion und in Interaktion	412
7.4.1	UML: Diagramme zur Beschreibung von Abläufen	412
7.4.2	Nachrichten an Objekte	421
7.4.3	Iteratoren und Generatoren	421
7.4.4	Funktionsobjekte und ihr Einsatz als Eventhandler	433
7.4.5	Kopien von Objekten	442
7.4.6	Sortierung von Objekten	452
7.5	Kontrakte: Objekte als Vertragspartner	455
7.5.1	Überprüfung von Kontrakten	455
7.5.2	Übernahme von Verantwortung: Unterklassen in der Pflicht	457
7.5.3	Prüfungen von Kontrakten bei Entwicklung und Betrieb	470
7.6	Exceptions: Wenn der Kontrakt nicht eingehalten werden kann	471
7.6.1	Exceptions in der Übersicht	472
7.6.2	Exceptions und der Kontrollfluss eines Programms	478
7.6.3	Exceptions im Einsatz bei Kontraktverletzungen	484
7.6.4	Exceptions als Teil eines Kontraktes	488
7.6.5	Der Umgang mit Checked Exceptions	493
7.6.6	Exceptions in der Zusammenfassung	501

8 Module und Architektur 503

8.1	Module als konfigurierbare und änderbare Komponenten	503
8.1.1	Relevanz der Objektorientierung für Softwarearchitektur	503
8.1.2	Erweiterung von Modulen	505
8.2	Die Präsentationsschicht: Model, View, Controller (MVC)	511
8.2.1	Das Beobachter-Muster als Basis von MVC	512
8.2.2	MVC in Smalltalk: Wie es ursprünglich mal war.....	513
8.2.3	MVC: Klärung der Begriffe	514
8.2.4	MVC in Webapplikationen: genannt »Model 2«	518
8.2.5	MVC mit Fokus auf Testbarkeit: Model-View-Presenter	523

9 Aspekte und Objektorientierung 527

9.1	Trennung der Anliegen	527
9.1.1	Kapselung von Daten	531
9.1.2	Lösungsansätze zur Trennung von Anliegen	532
9.2	Aspektorientiertes Programmieren	539
9.2.1	Integration von aspektorientierten Verfahren in Frameworks	539
9.2.2	Bestandteile der Aspekte	541
9.2.3	Dynamisches Crosscutting	541
9.2.4	Statisches Crosscutting	548
9.3	Anwendungen der Aspektorientierung	550
9.3.1	Zusätzliche Überprüfungen während der Übersetzung.....	551
9.3.2	Logging	552
9.3.3	Transaktionen und Profiling	553
9.3.4	Design by Contract	556
9.3.5	Introductions	559
9.3.6	Aspektorientierter Observer	560
9.4	Annotations	562
9.4.1	Zusatzinformation zur Struktur eines Programms	563
9.4.2	Annotations im Einsatz in Java und C#	565
9.4.3	Beispiele für den Einsatz von Annotations	566

10 Objektorientierung am Beispiel: Eine Web-Applikation mit PHP 5 und Ajax 573

10.1	OOP in PHP	574
10.1.1	Klassen in PHP	574
10.1.2	Dynamische Natur von PHP	578
10.2	Das entwickelte Framework – Trennung der Anliegen – Model View Controller	578
10.2.1	Trennung der Daten von der Darstellung	579
10.3	Ein Dienst in PHP	580
10.3.1	Datenmodell	581
10.3.2	Dienste – Version 1	583
10.4	Ein Klient in Ajax	586
10.4.1	Bereitstellung der Daten	587
10.4.2	Darstellung der Daten	589
10.5	Ein Container für Dienste in PHP	598
10.5.1	Dispatcher	601
10.5.2	Fabrik	603
10.5.3	Dependency Injection	604
10.5.4	Sicherheit	610
10.6	Ein Klient ohne JavaScript	615
10.7	Was noch übrigbleibt	619

Anhang 621

A	Verwendete Programmiersprachen	623
A.1	C++	623
A.2	Java	626
A.3	C#	629
A.4	JavaScript	629
A.5	CLOS	632
A.6	Python	635
A.7	Ruby	637
B	Literaturverzeichnis	641
B.1	Allgemeine Bücher zur Softwareentwicklung	641
B.2	Bücher über die UML und die verwendeten Programmier Sprachen	643
	Index	645

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

1 Einleitung

1.1 Was ist Objektorientierung?

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

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

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

Da meldete sich Alan Kay zu Wort und hakte nach:

»Diese Sprache unterstützt also keine Vererbung?«

»Das ist korrekt.«

»Und sie unterstützt keine Polymorphie?«

»Das ist korrekt.«

»Und sie unterstützt auch keine Datenkapselung?«

»Das ist ebenfalls korrekt.«

»Dann scheint mir das keine objektorientierte Sprache zu sein.«

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

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

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

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

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

1.2 Hallo liebe Zielgruppe

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

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

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

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

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

Wer sollte unser Buch lesen? Wir wollen Studenten der Informatik eine praktische, interessante und hoffentlich auch unterhaltsame Einführung und Vertiefung zum Thema

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

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

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

Was wir
voraussetzen

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

1.3 Was bietet dieses Buch (und was nicht)?

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

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

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

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

1.3.1 Bausteine des Buchs

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

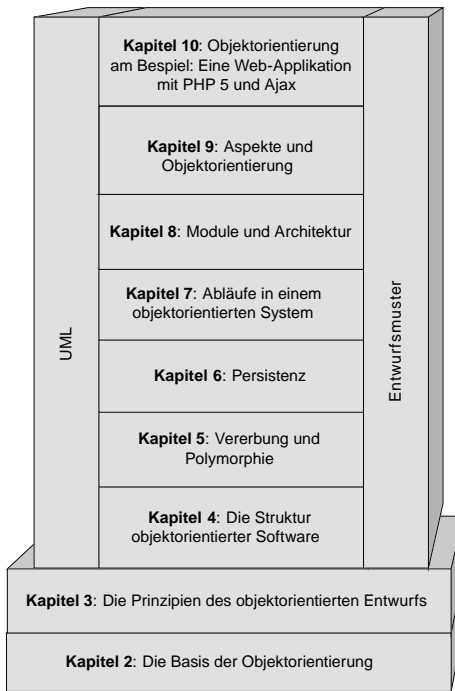


Abbildung 1.1 Modularer Aufbau der Kapitel

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

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

Kapitel 3:
Prinzipien

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

Kapitel 4:
Struktur

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

Kapitel 5:
Vererbung und
Polymorphie

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

Kapitel 6:
Persistenz

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

Kapitel 7:
Abläufe

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

**Kapitel 8:
Architektur**

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

**Kapitel 9:
Aspekte**

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

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

**Kapitel 10:
Objektorientierung am
Beispiel einer
Web-Applikation**

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

**Anhang:
Programmiersprachen**

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

1.3.2 Crosscutting Concerns: übergreifende Anliegen

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

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

Unified Modeling Language

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

Objektorientierte Analyse

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

Entwurfsmuster

Entwurfsmuster (engl. *Design Patterns*) für den objektorientierten Entwurf wurden mit der Publikation *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software* als Begriff geprägt. Die Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides führen dabei eine standardisierte Beschreibungsform für wiederkehrende Vorgehensweisen beim Entwurf guter objektorientierter Software ein¹. Aufgrund von Umständen, die mit der Sortierung unseres Alphabets zusammenhängen, wird die Publikation meist mit Erich Gamma assoziiert. Da das aber nicht korrekt ist, werden wir uns mit dem Namen [Entwurfsmuster 2004] darauf beziehen, da die aktuelle deutsche Ausgabe aus dem

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

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

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

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

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

Deutsche und englische Begriffe

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

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

1.3.3 Die Rolle von Programmiersprachen

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

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

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

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

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

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

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

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

► **Java**

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

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

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

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

1.4 Warum überhaupt Objektorientierung?

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

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

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

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

1.4.1 Gute Software: Was ist das eigentlich?

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

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

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

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

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

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

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

Trotz der unterschiedlichen Sichtweisen kristallisieren sich hier einige zentrale Kriterien für gute Software heraus³:

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

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

1.4.2 Die Rolle von Prinzipien

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

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

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

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

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

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

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

1.4.3 Viele mögliche Lösungen für ein Problem

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

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

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

Diskussionen können hilfreich sein.

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

DISKUSSION:
Was gibt's denn hier zu diskutieren?

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

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

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

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

[zB] Konkrete Beispiele erleichtern Ihnen das Verstehen.

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

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

3 Die Prinzipien des objektorientierten Entwurfs

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

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.

Einfache Aufgaben
– komplexe
Programme

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.

Vorbereitung
auf Änderungen

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

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

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

3.1 Prinzip 1: Prinzip einer einzigen Verantwortung

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

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

[»]

Module

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

[»]

Verantwortung (Responsibility) eines Moduls

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

Module und Untermodule

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

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

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

Prinzip einer einzigen Verantwortung (Single Responsibility Principle)

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

[«]

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

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

Code mit unklaren
Aufgaben

Vielleicht kennen Sie eine verdächtig aussehende Verpackung im Kühlschrank der Kaffeeküche? Eigentlich kann der Inhalt nicht mehr genießbar sein. Aber warum sollten Sie es wegwerfen? Sie sind doch nicht der Kühlschrankbeauftragte, und außerdem haben Sie gehört, Ihr Boss solle den schwedischen Surströmming¹ mögen – warum sollten Sie das Risiko eingehen, den Boss zu verärgern?

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

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



Abbildung 3.1 Ein Kühlschrank mit unklaren Verantwortlichkeiten

**Vorteile des
Prinzips**

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

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

**Anforderungen
ändern sich.**

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

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

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

Erhöhung der Wartbarkeit

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).

Erhöhung der Chance auf Mehrfachverwendung

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

DISKUSSION:
Mehr Verantwortung, mehr Verwendungen?

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

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

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

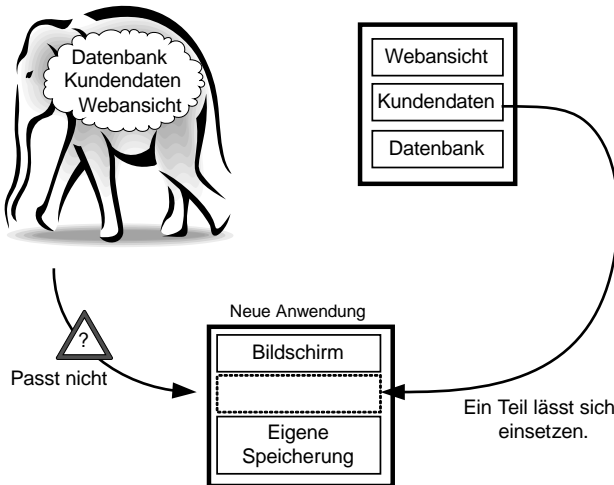


Abbildung 3.2 Mehrfachverwendung einzelner Module von Software

Regeln zur
Realisierung
des Prinzips

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

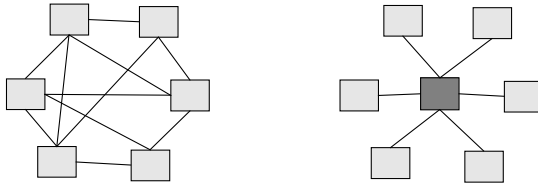
► **Regel 1: Kohäsion maximieren**

Ein Modul soll zusammenhängend (kohäsiv) sein. Alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein. Haben Teile eines Moduls keinen Bezug zu anderen Teilen, können Sie davon ausgehen, dass Sie diese Teile als eigenständige Module implementieren können. Eine Zerlegung in Teilmodule bietet sich damit an.

► **Regel 2: Kopplung minimieren**

Wenn für die Umsetzung einer Aufgabe viele Module zusammenarbeiten müssen, bestehen Abhängigkeiten zwischen diesen Modulen. Man sagt auch, dass diese Module gekoppelt sind. Sie sollten die Kopplung zwischen Modulen möglichst gering halten. Dies können Sie oft erreichen, indem Sie die Verantwortung für die Koordination der Abhängigkeiten einem neuen Modul zuweisen.

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



hoher Grad an Kopplung reduziert durch ein neues Modul

Abbildung 3.3 Module mit unterschiedlichem Grad der Kopplung

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

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.

Vorsicht: Verschleierung von Abhängigkeiten

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

3.2 Prinzip 2: Trennung der Anliegen

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

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

Trennung der Anliegen (Separation of Concerns)

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

[«]

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

Trennen der Sprache und der Darstellung von der Anwendungslogik

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

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

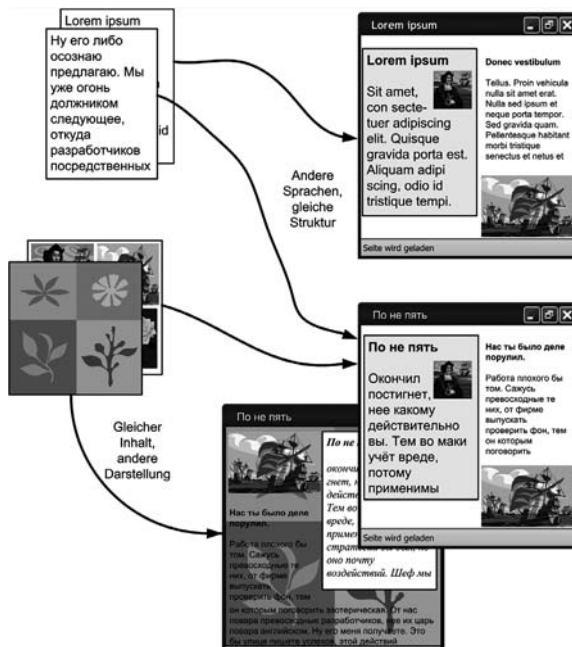


Abbildung 3.4 Trennung der Anliegen bei der Darstellung von HTML-Seiten

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

Anliegen bei Online-Banking

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

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

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

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

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

Anliegen in unterschiedlichen Modulen

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

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

DISKUSSION:
Objektorientierung und Trennung der Anliegen

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

3.3 Prinzip 3: Wiederholungen vermeiden

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

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

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

Implizite Abhängigkeiten

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

[»]

Wiederholungen vermeiden (Don't repeat yourself)

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

Prinzip in der Praxis

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

Regeln zur Umsetzung des Prinzips

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

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

Aufruf einer Funktion `fullName()`. Hier kann man keine generell gültige Antwort geben. Gibt es eine Anforderung, dass der volle Name in der Form `firstName + ' ' + lastName` dargestellt werden soll? Wenn ja, sollte diese Anforderung explizit an einer Stelle, in der Funktion `fullName()` umgesetzt werden².

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

Austausch
von Daten

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

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

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

Kopieren von
Quelltext

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

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

DISKUSSION:
Redundanz und
Performanz

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

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

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

3.4 Prinzip 4: Offen für Erweiterung, geschlossen für Änderung

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

Module in
verschiedenen
Umgebungen

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

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

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

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

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

Änderungen vermeiden

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

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

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



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

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

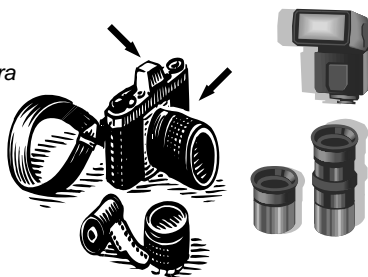


Abbildung 3.5 Eine kompakte und eine erweiterbare Fotokamera

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

Erweiterbarkeit eines Moduls

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

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

[»]

Offen für Erweiterung, geschlossen für Änderung (Open-Closed-Principle)

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

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

Definierte Erweiterungspunkte

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

Indirektion

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

Funktionalität erkennen

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

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

Modul anpassen und es um die benötigten Erweiterungspunkte bereichern.

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

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.

Aufwand durch
Erweiterungs-
punkte

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

3.5 Prinzip 5: Trennung der Schnittstelle von der Implementierung

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

Trennung der Schnittstelle von der Implementierung (Program to interfaces)

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

[«]

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.

Schnittstelle ist
Spezifikation

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

[zB]
Protokoll-
ausgaben

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.

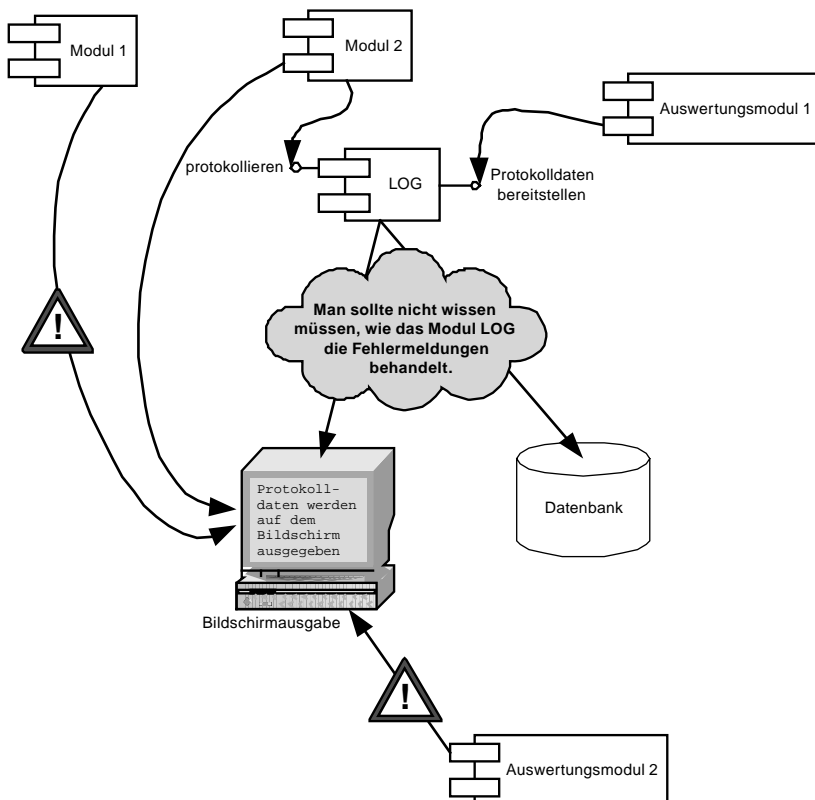


Abbildung 3.6 Trennung der Schnittstelle von der Implementierung

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

Problem 1:
Funktionalität
nicht genutzt

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

Problem 2:
Sich auf die
Implementierung
verlassen

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.

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

[zB]
Verwendung von
Schriftgrößen

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.

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

Vorsicht:
Java- und
C#-Interfaces

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

3.6 Prinzip 6: Umkehr der Abhängigkeiten

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

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

[»]

Abstraktion

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

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

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

[»]

Umkehr der Abhängigkeiten (Dependency Inversion Principle)

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

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

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

Definition:
Abstraktion

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.

Weg vom Top-down-Entwurf

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 MFC⁴ einsetzen.

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

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

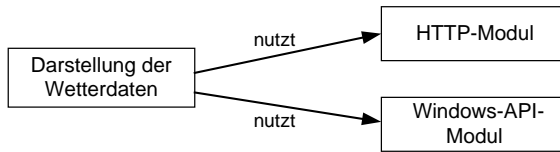


Abbildung 3.7 Abhängigkeiten in unserer Beispielanwendung

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

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

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

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

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

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

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

Portables Design

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

Abstraktion,
Abhängigkeiten
und die
Änderungen

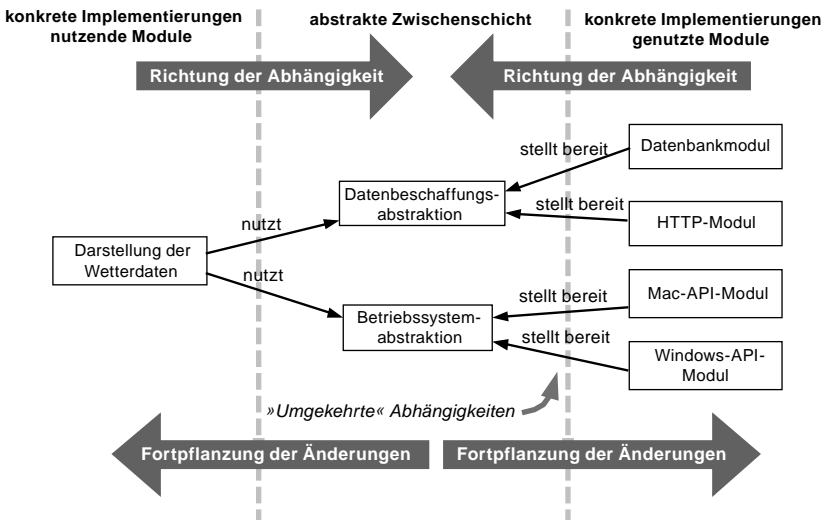


Abbildung 3.8 Beispielanwendung mit »umgekehrten« Abhängigkeiten

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

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

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

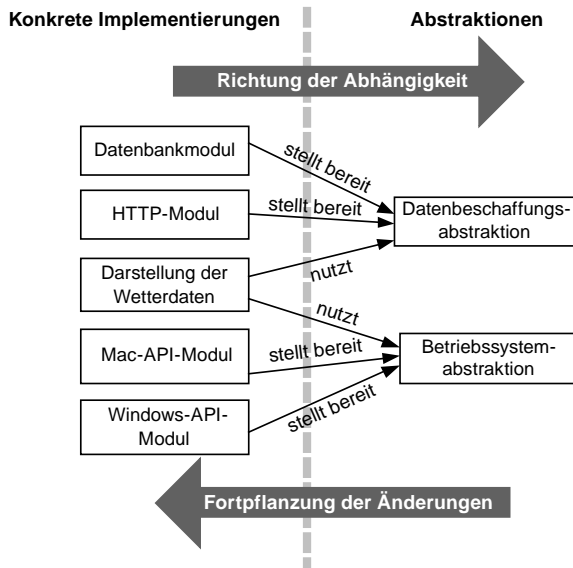


Abbildung 3.9 Konkrete Module sollen von abstrakten Modulen abhängig sein.

3.6.1 Umkehrung des Kontrollflusses

Achtung: Dies ist nicht Hollywood!

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

[»]

Umkehrung des Kontrollflusses (engl. Inversion of Control)

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

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

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

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

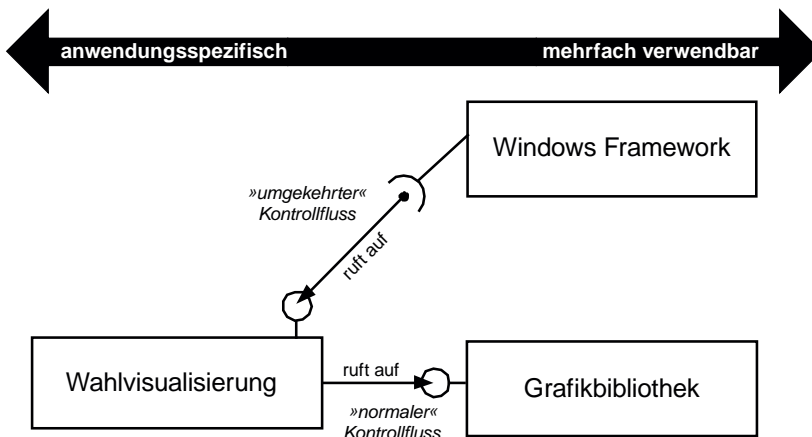


Abbildung 3.10 Umkehrung des Kontrollflusses

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

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

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

3.7 Prinzip 7: Mach es testbar

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

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

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

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

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

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

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

[»]

Unit-Tests

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

Schwierige Tests

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

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

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

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

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.

Mock-Objekte

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.

Auswirkungen auf das Design

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

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

Index

A

- Abarbeitungsreihenfolge, für Sammlung von Objekten festlegen 452
- Abgeleitete Klassen arbeiten nicht mehr? 229
- Abhängigkeit
 - implizit* 48, 49
 - sichtbar machen* 45
 - zwischen Modulen und Klassen aufheben* 387
- Ablauf 337
 - Beschreibung* 412
- Ablaufsteuerung 506
- Abstract Windowing Toolkit 566
- Abstrakte Fabrik 355
 - Definition* 357
 - Verwendung* 357
- Abstrakte Klassen 167, 170
 - in C++* 174
 - in Java und C#* 174
 - Umsetzung* 173
- Abstrakte Methoden 170
- Abstraktion 56
 - Definition* 56
- Accept Event Action 415
- action-mappings 521
- Advice, Definition 543
- Aggregate 129
- Aggregation 119, 129
 - Definition* 129
 - UML* 130
- Ajax 586
- Aktion, Sichtweisen 412
- Aktivitätsbereich 414
- Aktivitätsdiagramm 412, 413
- Alternativschlüssel 305
- Analysemodell 86
- Änderungen an Basisklassen und Schnittstellen 229
- Anforderung, Änderung 42
- Anliegen 45
 - durch ein Modul repräsentieren* 45
 - in einem Modul* 530
 - in unterschiedlichen Modulen* 47
 - Trennung* 46
 - übergreifend* 530
- Anmerkung → Annotation
- Annotation 562
 - @Override* 565
 - @SuppressWarnings* 565
 - Definition* 565
 - vordefiniert* 565
- Anonyme Klassen 208
 - Java* 438
- Anwendungscontainer 539
- Anwendungsfalldiagramm 412
- Anwendungsrahmen 506
- AspectJ 22, 547
- Aspekt, Definition 543
- Aspekte 527
- Aspektorientierte Frameworks 540
- Aspektorientierter Observer 560
- Aspektorientiertes Programmieren 539
- Aspektorientierung, Anwendung 550
- Assoziation 117
 - Formen* 118
 - in UML* 119
 - Richtung* 119
 - Rollen* 119
- Assoziationsklasse 127
- Assoziationsklassen, UML 127
- Asynchrone Nachricht 419
- Attribute 68, 132
- Aufzählung
 - Elemente mit Methoden* 141
 - typischer und serialisierbar* 145
- Aufzählungen 140
 - als abgegrenzte Mengen von Objekten* 140
- Ausnahme → Exception
- Automatische Speicherbereinigung → Garbage Collection
- AWT 566

B

- Basisklassen
 - instabil* 249
 - Kopplung mit abgeleiteter Klasse* 250

Benutzerfreundlichkeit 24
 Beobachter-Muster
 mit Aspektorientierung umsetzen 560
 MVC 512
 Beziehung
 als Assoziation, Komposition, Aggregation? 132
 Attribute 126
 beidseitig navigierbar 128
 Einschränkungen in UML 124
 einwertig 128
 Implementierung 128
 in relationaler Datenbank abbilden 311
 Klassen und Objekte 117
 mehrwertig 123, 124, 128
 Navigierbarkeit 120
 Richtung 119
 Umsetzung 128
 zwischen Objekt und Teilen 119
 Beziehung → Assoziation
 Beziehungsklasse 126, 127
 UML 126
 Boyce-Codd-Normalform 329

C

C# 629
 Methoden überschreiben 249
 partielle Klasse 377
 Sichtbarkeitsstufen 112
 Typisierung 100
 C++ 623
 Compiler 626
 Klassen als Module 104
 Methoden überschreiben 248
 späte Bindung 228
 Struktur 623
 Syntax 624
 Typisierung 100, 101
 C++-Konstruktoren, Polymorphie 238
 Cascading Stylesheets 591
 Chain of Responsibility 367
 Checked Exception 491
 als Unchecked Exception weiterreichen 498
 Umgang 493
 Class Table Inheritance 319
 Clone-Operation 446
 Java 447

CLOS 632
 Struktur 633
 Syntax 633
 Code Scattering, Definition 530
 Code Smell 488
 Code Tangling, Definition 530
 Code-Durcheinander → Code Tangling
 Code-Redundanz
 vermeiden durch Vererbung 244
 vermeiden mittels Fabrik 355
 Code-Streuung → Code Scattering
 Collection → Sammlung
 Common Lisp Object System 632
 Composite Pattern → Entwurfsmuster, Kompositum
 Concrete Table Inheritance 318
 Constructor Call 545
 Constructor Execution 545
 Constructor Injection 395
 Container 539
 Definition 509
 Komplexitäten abbauen 392
 Controller in MVC, Definition 514
 Copy-Konstruktor 341
 Java 444
 Crosscutting
 dynamisch 541
 statisch 541
 Crosscutting Concern 541
 Definition 530
 implementieren 541
 in Klassen einarbeiten 541
 CSS 591

D

Daten 28
 Typen 29
 Datenbankidentität 150
 Datenelemente, Zugriff 546
 Datenkapselung 31, 67, 68
 Bedeutung 71
 Nachteile 73
 Datenmodell
 in dritte Normalform bringen 326
 in zweite Normalform überführen 324
 Datensatz 303
 Datenstruktur, definieren 49
 Datentypen 92

Delegaten-Klassen 440
 Definition 439
 Delegation
 als Alternative zu Vererbung 249, 268
 Definition 269
 Delphi, Typisierung 101
 Demeter-Prinzip
 Nutzen 208
 Verletzung 208
 Denormalisierung 321
 Dependency Injection 386, 387, 510
 Beispiel in PHP5 604
 Einsatz 396
 Übergabe 394
 Varianten 394
 Dependency Inversion Principle → Prinzip
 Umkehr der Abhängigkeiten
 Design by Contract 556
 Designmodell 86
 Design-Patterns → Entwurfsmuster
 Diamantenregel 282
 Diensterverzeichnis, Definition 396
 Dispatcher 199
 Don't repeat yourself 47
 Double-checked Locking 381
 Downcast 180
 Dritte Normalform 326
 Ducktyping 95
 Dynamisch typisierte
 Programmiersprachen, rein
 spezifizierende 170
 Dynamische Klassifizierung 289
 Dynamische Pointcuts 554
 Dynamische Typisierung 95
 Dynamischer Speicher 397
 Dynamisches Crosscutting 541
 Dynamisches Typsystem 94

E

Eigenschaft eines Objekts 67
 Einfache Klassifizierung 86
 Eingabe-Controller 515
 Einweben 541
 Einwertige Beziehung 128
 Enterprise Java Beans 151
 Entity Beans 152, 154
 Entity-Relationship-Darstellung 312

Entwurfsmuster 19
 Abstrakte Fabrik 349
 Beobachtetes-Beobachter 263, 512
 Besucher 223
 Fliegengewicht 134, 137
 Kompositum 227
 Prototyp 348
 Schablonenmethode 245
 Strategie 294, 506
 Enumerations 140
 Ereignis 421
 Ersatzkomponente 63
 Ersetzbarkeit 276
 Unterklassen 457
 Erweiterung, Module 505
 Erweiterungsmodul 52
 Erweiterungspunkt 52, 505
 bestimmen 507
 hinzufügen 53
 Eventhandler 421, 433
 Definition 435
 Exception 471, 472
 Ausführungspfade 480
 Einsatz 475
 Kontrakte formulieren 489
 Kontrollfluss eines Programms 478
 Teil eines Kontrakts 488
 vs. Fehlercode 477
 werfen 472
 Exception Handling 480
 Exception Safety 479
 Exception-Sicherheit 479
 Exemplar 84
 Exemplare einer Klasse
 erzeugen 113
 verwalten 113
 Exposed Joinpoints 544

F

Fabrik 349
 abstrakt 355
 Beispiel in PHP5 603
 Definition 349
 für Module unsichtbar machen 387
 Konfigurierbar 360
 Schnittstelle 357
 über Datei konfigurieren 361

Fabrikmethode 368
Anwendung 372
Bedingung 372
Definition 349
Eigenschaften 371
Unterschied abstrakte Fabrik 371
Factory Pattern 350
Fehlercode 477
Fehlersituation, Bekannt 489
field access 546
Finale Klassen 249
Flache Kopie 449
Fliegengewicht-Entwurfsmuster 138
Flyweight 138
Fragile Base Class Problem 250
Fragile Binary Interface Problem 228
Framework 53, 506
Fremdschlüssel 306
Function Objects → Funktionsobjekte
Fünfte Normalform 332
Funktion 302
Definition 29
Funktionale Abhängigkeit 304
Funktionalität
Abweichungen festhalten 52
umsetzen 48
zwangsweise nutzen 244
Funktionsobjekte 421, 433
Definition 436

G

Garbage Collection 399
Arten 400
Lebensdauer von Objekten 409
Umsetzung 400
Varianten 399
Geerbte Methode, überschreiben 565
Generalisierung 156
Generat 534
Generator 421
C# 431
Java 430
Problem 430
Zweck 422
Generator als Methode 430
Generics 98
Generierter Code 534

Geschachtelte Klassen 191
in C# und C++ 192
Java 191
Geschützt innerhalb des Packages 111
Geschützte Datenelemente, Zugriff 184
Gleichheit 254
Eigenschaften 255
prüfen 254
prüfen in Java 255
Gleichheitsprüfung
bei Vererbungsbeziehung 255
Formale Kriterien 255
Globale Variable und Singleton 385

H

Heap 398
Hibernate 537
Hierarchie von Klassen
virtuelle-Methoden-Tabelle 230

I

Identität 81, 133, 147, 151
Implementierung 53
Beziehungen 128
vererben 239
Implementierungen, Aufrufen 243
Implizite Abhängigkeit 48, 49
Indirektion 52
inner classes 191
Instabile Basisklassen 249
Instance → Exemplar
Interaktionsübersichtsdiagramm 413
Interface Injection 395
Nachteil 395
Interface → Schnittstellen-Klasse
Interzeptor
Definition 542
Implementierung 542
Introduction 549, 559
Warnungen 550
Introspektion 538
Invariante 90
Inversion of Control 60
Iterator 424
Definition 187, 425
dynamisch 425

J

Jakarta Struts 519

Java 626

finale statische Variable 146

Generator 430

geschachtelte Klassen 191

Identität von Objekten 152

Klassen als Module 102

Methoden überschreiben 248

Protokolle 351

Sichtbarkeitsstufen 111

Struktur 626

Syntax 627

Typisierung 99

virtuelle Machine (Garbage Collection)
409

JavaScript 77, 629

Erweiterung von Objekten 345

Funktionen 343

Hierarchie von Prototypen 345

Klassen 342

Objekt erzeugen 342

Struktur 630

Syntax 631

Typisierung 100

Vererbung 345

Vererbungskette 347

JavaScript Object Notation → JSON

JDBC 369

Joinpoint

Arten 544

Aufruf einer Operation 545

Ausführung einer Methode 545

Ausführung eines Konstruktors 545

Definition 542

offen gelegt 544

Zugriff auf die Datenelemente 546

jQuery 592

JSON 588

JUnit 508

K

Kapselung von Daten 31, 68

Kardinalität 123

Klassen 84, 92, 176

Abstrakt 167

als Vorlagen 338

Anonym 208

Beziehungen untereinander 117

Definition 84

Elemente 102

erweitern mit Aspektorientierung 559

Erzeugung von Exemplaren 545

Exemplare erzeugen 113

final 249

Konformität 160

konkret 167

Kontrakt 88

Kopplung mit Typ 94

Methoden und Daten zuordnen 112

Modul 102

Multiplizität 121

Parametrisiert 96

Schnittstelle 89

Schnittstelle trennen 55

Spezifikation 87

Spezifikation durch Kontrakt 88

spezifizierend 169

Klassen als Module, Java 102

Klassen und Typen koppeln 94

Klassen von Objekten 133

Klassen von Werten 133

Klassenbasierte Elemente, Verwendung
113

Klassenbasierte Sichtbarkeit 107

Klassenbezogene Attribute 112

Klassenbezogene Konstanten 115

Klassenbezogene Methoden 112

Hilfsfunktionen 114

Klassenhierarchie auf relationale

Datenbank abbilden 315

Klassenhierarchien anpassen bei

Typsystemen 176

Klassenmanipulation durch Aspekte 541

Klassenzugehörigkeit dynamisch ändern
290

Klassifizierung 289

Definition 85

dynamisch 290

einfach 86

mehrfach 86

Kohäsion maximieren 44

Kommunikationsmodul 45

Komparator 452

Komplexität 22

beherrschen durch Strukturierung 29

reduzieren 45, 48

Komplexität beherrschen, Prinzipien 39
 Komposition 119, 129
 bei Hierarchie 131
 Definition 129
 Einsatz 131
 UML 130
 Konfigurationsdateien 564
 Konfigurierbare Fabrik 360
 in Sprachen ohne Reflexion 363
 Umsetzung 361
 Umsetzung in Java 361
 Konformität 160
 Konstante
 benannte 48
 klassenbezogen 115
 Konstruktor 338
 Aufruf 545
 Ausführung 545
 Gruppen 339
 mit Initialisierung 340
 Konstruktoraufruf 113
 Konstruktoren 113
 Kontrakt 455
 Klassen 88, 157
 Operation 79
 prüfen 464
 Prüfung durch Methode 468
 überprüfen 455
 Überprüfung mit Aspektorientierung 556
 von Objekten 79
 Kontraktverletzung
 durch Programmierfehler 489
 Exception 484
 Kontrollfluss eines Programms
 unterbrochen 472
 verlassen 472
 Kontrollfluss, Umkehrung 387
 Kopie 442
 als Prototyp 443
 Eigenschaften 447
 flach 449
 Sammlung 443
 Tiefe 448
 und Zyklische Referenz 449
 Kopieren aller referenzierten Objekte 406
 Kopiervorgang, Endlosschleife 449
 Kopplung minimieren 44
 Korrektheit 24
 Kovariante Typen 276

L

Laufzeitpolymorphie 193
 Laufzeitumgebungen 509
 Law of Demeter 206
 Law of leaky Abstractions 228
 Lazy Initialization 380, 382
 Leichtgewichtiger Container 392, 510
 lightweight container 392
 Link (UML) 119
 Logging
 Lösung mit Aspektorientierung 552

M

Manipulation von Klassen 541
 Mark and Sweep 400
 Markieren und Löschen
 Problem 406
 Markieren von referenzierten Objekten 400
 Mehrfache Klassifizierung 86
 Mehrfachvererbung 261
 Datenstrukturen in C++ 283
 Datenstrukturen in Python 283
 Ersetzung durch Komposition 287
 Operationen und Methoden in Python 280
 Problemstellung 273
 von Operationen und Methoden C++ 279
 Mehrfachvererbung der Implementierung 262
 ersetzen in Java und C# 268
 Problem 265
 Mehrfachvererbung der Spezifikation 261
 Java und C# 275
 Mehrfachverwendbarkeit, Modul 43
 Mehrfachverwendung 138
 Vorteil 139
 Mehrwertige Beziehungen 123
 Message Driven Beans 152
 Metainformation 536, 562
 Definition 536
 Metaobjekt, Definition 536
 method_missing 205
 Methode
 abstrakt 170
 als Implementierung von Operationen 200

- anhand von Typ der Objekte bestimmen* 210
 - Aufruf* 206
 - Ausführung* 545
 - Definition* 75
 - für Überschreiben sperren* 248
 - Implementierung* 75
 - paarweise aufrufen* 245
 - Scheitern anzeigen* 477
 - überschreiben* 241
 - überschrieben* 246
 - Ursache für Nichterfüllung der Aufgabe* 475
 - Methodenaufruf, verkettet* 206
 - Methodenimplementierung, Interface* 549
 - Mixin*
 - C++* 273
 - Definition* 271
 - mit Klassen verwenden* 559
 - Ruby* 271
 - Mock-Objekte* 63
 - Model 1 für Webapplikationen* 519
 - Model 2 für Webapplikationen* 519
 - Modell in MVC, Definition* 515
 - Modellierung, der Schnittstelle* 182
 - Modellierungsplacebo* 131
 - Model-View-Controller → MVC*
 - Model-View-Presenter* 523
 - Modul* 40, 503
 - Abhängigkeiten* 40, 43, 44, 59
 - Änderung* 59
 - Änderung vermeiden* 51
 - Anforderungen umsetzen* 41
 - Anpassungsfähigkeit* 50
 - Erweiterbarkeit* 51
 - Erweiterung* 505
 - Formulierung der Abhängigkeiten* 53
 - Identifikation* 42
 - Kopplung* 44
 - Mehrfachverwendbarkeit* 43
 - Schnittstelle* 53
 - Testbarkeit* 62
 - Verantwortung* 40, 41
 - Verwendungsvarianten* 52
 - zusammenführen* 48
 - Zweck* 41
 - Multiple Dispatch*
 - Entwurfsmuster Besucher* 215, 216
 - Java* 212
 - mit Unterstützung durch die Programmiersprache* 215
 - ohne Unterstützung durch die Programmiersprache* 212
 - Praxisbeispiel* 215, 216
 - Multiplizität* 120
 - Darstellung in UML* 121
 - Definition* 121
 - MVC* 511, 512
 - Ansatz* 511
 - Begriffsdefinitionen* 514
 - in Webapplikationen* 518
 - Testbarkeit* 523
 - Ursprung* 513
- ## N
-
- Nachbedingung* 90
 - Nachricht an Objekte* 421
 - Namenskonvention* 563
 - Nassi-Shneiderman-Diagramm* 31
 - Natürlicher Schlüssel* 309
 - Navigierbarkeit* 120
 - UML* 120
 - Nebenläufigkeit*
 - Java* 380
 - Neustart bei Kontraktverletzung* 485
 - NULL-Werte* 306
- ## O
-
- Oberklassen* 156
 - ändern* 176
 - Vererbung* 160
 - ObjectPascal, Typisierung* 101
 - Objekt* 31, 65
 - Aktion und Interaktion* 412
 - Assoziation* 117
 - Attribute* 132
 - Beziehungen* 117
 - Darstellung in UML* 69
 - Datenkapselung* 67
 - Definition* 65
 - Eigenschaften* 68
 - erzeugen* 338
 - Funktionalität* 74
 - gleichartig* 84
 - Gleichheit* 254

- identisch* 148
- Identität* 81, 133, 147
- in relationaler Datenbank abbilden* 307
- Klassenzugehörigkeit bestimmen* 360
- Konstruktion* 236
- Kontrakt* 455
- kopieren* 341, 442
- Laufzeit überdauern* 299
- Lebenszyklus* 509
- Methode* 75
- Nachrichtenaustausch* 419
- Operation* 455
- prüft Kontrakt* 464
- Rolle* 83
- Sammlung* 421
- Serialisierung* 299
- sortieren* 452
- Spezifikation von mehreren Klassen*
 - erfüllen* 261
 - und Daten* 76
 - und Operation* 74
 - und Routine* 423
 - Verhalten modifizieren* 344
 - Zustand merken* 442
- Objekt erzeugen
 - JavaScript* 342
 - Konstruktor* 338
 - Prototyp* 342
 - Singleton* 377
 - Verfahren* 338
- Objekt und Exemplar 85
- Objekt und Routine 423
- Objektbasierte Sichtbarkeit 108
- Objekte und Werte 81
- Objekteigenschaften 67
- Objekterstellung 338
- Objektfluss 415
- Objektidentität 151
- Objektinitialisierungs-Joinpoint 546
- Objektknoten 415
- Objektkopie
 - erstellen* 444
 - flach* 341
 - tief* 341
 - Zweck* 442
- Objektorientierte Analyse 19
- Objektorientierte Architektur 503
- Objektorientierte Programmiersprachen
 - Grundelemente* 14
- Objektorientierte Software
 - Struktur* 65
- Objektorientiertes System
 - Komplexität reduzieren* 45
 - Vorteil* 421
 - Zustände verwalten* 415
- Objektorientiertes Systemdesign 504
- Objektorientierung
 - Basis* 27
 - Definition* 13
 - Grundelemente* 27
 - Prinzipien* 24, 39
 - Zweck* 22
- Objekt-relationale Abbildungsregeln
 - Dritte Normalform* 327
 - Vierte Normalform* 332
 - Zweite Normalform* 325
- Objekt-Relationale Mapper 301
- Observable-Observer → Entwurfsmuster, Beobachtetes-Beobachter
- Offen für Erweiterung, geschlossen für Änderung 50, 226
- Open-Closed-Principle 50
- Operation 74
 - auf Objekten ohne Klassenbeziehung aufrufen* 202
 - auf primitiven Datentypen* 114
 - Aufruf* 74, 545
 - Definition* 74
 - Deklaration* 76, 170
 - implementieren* 200
 - Kontrakt* 79
 - mehreren Objekten zuordnen* 211
 - Semantik* 455
 - Syntax* 455
- Operationen mit gleicher Signatur
 - C#* 277
 - Java* 275

P

- Parametrisierte Klassen 96
 - C++* 192
 - in UML* 96
 - Java* 98
- Parametrisierte statische Fabrik 355
- Partielle Klasse 376
- Persistenz 299

PHP 574
 Dynamische Typisierung 578
 Klassen 574
 Klassen- und Objektvariablen 575
 Klassenkonstanten 576
 Klassenmethoden 576
 Konstruktor 577
 Plugin 511
 Pointcut
 Arten 547
 Definition 543
 Polymorphe Methoden 229
 Konstruktion und Destruktion von
 Objekten 236
 Polymorphie 32
 Definition 193
 statisch 194, 229
 Vorteile 34, 198
 Polymorphie im Konstruktor
 Java 237
 Postconditions 90
 Präsentationsschicht 511
 Preconditions 89
 Primitive Datentypen 114
 Prinzip der Datenkapselung
 Vorteile 32
 Prinzip der Ersetzbarkeit 162, 178, 276
 Gründe für Verletzung 166
 Vererbung der Spezifikation 161
 Prinzip einer einzigen Verantwortung 40,
 73
 Entwurfsmuster Besucher 226
 Regeln 44
 Vorteil 42
 Prinzip Trennung der Schnittstelle von der
 Implementierung 53
 Prinzip Trennung von Anliegen 45
 Prinzip Umkehr der Abhängigkeiten 56
 Prinzip Wiederholungen vermeiden
 Umsetzung 48
 Prinzipien 39
 Private Methoden, Ruby 108
 Private Vererbung 185
 in Delegationsbeziehung umwandeln 186
 Profiling 553
 program to interfaces 53
 Programmabbruch 485
 Programm direkt beenden 487
 Programm in definierter Weise beenden
 486

Programmierfehler, Ursachen 489
 Programmiersprachen, strukturierte 28
 Programmverhalten ändern, mit
 Annotations 566
 Protected 183
 protected internal 112
 Protokollierung der Abläufe 552
 Prototyp 342
 Definition 342
 Entwurfsmuster 348
 Prozedur, Definition 29
 Prozess 423
 Prüfung des Kontrakts
 an Aufrufstelle 469
 bei Entwicklung 470
 beim Aufruf von Operationen 468
 gegenüber Implementierung 468
 mit Aspektorientierung 470
 Python 95, 635
 Syntax 635
 Typisierung 100

Q

Qualifikatoren 125
 UML 126
 Quelle 534
 Quelltext 534
 ändern 50
 in verschiedene Quelltextmodule verteilen
 377
 kopieren 49
 mit generierten Anteilen 535
 Redundanz 533
 Wiederholungen 47
 Quelltextgenerierung 533
 Probleme 534

R

RAI 482
 Redundanz 47
 bewusst 321
 vermeiden 244
 Reference Counting 400
 Referenz 148
 Referenzielle Integrität 312
 Referenzzähler 400
 Reflection → Reflexion

Reflexion 363
Definition 538
 Registratur 360, 364, 371
 Relation 303
 Relationale Datenbanken 300
Begriffsdefinition 303
Partitionierung 309
Struktur 301
 Resource Acquisition is Initialisation 482
 Responsibility 40
 Richtung einer Assoziation 119
 Rollen
UML 119
 Routinen 423
als Objekte 436
Definition 29
 Ruby 374, 637
private Methoden 108
Sichtbarkeitsstufe Geschützt 110
Syntax 638
Typisierung 100

S

Sammlung 424
Kopie 443
über generischen Mechanismus kopieren 447
 Sammlung von Objekten, stellt Iterator bereit 187
 Sammlungsbibliothek 187
 San Francisco Framework Classes 508
 Schablonenmethode 245, 507
 Schlüssel 304
 Schlüsselkandidaten 305
 Schnittstelle
Definition 75
Implementierung 53
implizit und explizit 275
minimal vs. benutzerorientiert 549
von Implementierung trennen 75
von Klasse trennen 55
 Schnittstelle einer Klasse, Definition 89
 Schnittstellen-Klassen 55, 168
Definition 168
in C++ 174
in Java und C# 174
Umsetzung 173
 Schwach typisierte Programmiersprachen 99
 Separate Fabrik 373
 Separation of Concerns 45
 Sequenzdiagramm 413, 419
 Serialisierung von Objekten 299
 Service Locator 396
 Session Beans 151
 set 124
 Sichtbarkeit
auf aktuelles Objekt einschränken 107
Vererbung 183
 Sichtbarkeitskonzept
klassenbasierte 107
objektbasiert 108
 Sichtbarkeitskonzept, klassenbasiert 184
 Sichtbarkeitsstufe
Geschützt 183
Öffentlich 106
Privat 106
Zweck 106
 Sichtbarkeitsstufe XE 112
 Signal Send Action 415
 Single Responsibility Principle 40
 Single Table Inheritance 317
 Singleton
Einsatz 385
oder globale Variable 385
statische Initialisierung 382
Umsetzung in Java 378
 Singleton-Klasse 377
 Singleton-Methoden 116
Definition 116
 Smalltalk, Typisierung 100
 Smart Pointer 401
 soft references 400
 Software
Anforderungen 23
Design verbessern 64
Testbarkeit 62
Umsetzung der Ziele 24
 Softwarearchitektur 503
 Sortierkriterien 453
 Späte Bindung 194
realisieren 229
 Speicherbereich 397
 Speicherersparnis 139
 Sperre mit zweifacher Prüfung 381
 Spezifikation einer Klasse 91

Spolsky, Joel 228
 Stack 397
 Standardkonstruktor 340
 Stateful Session Beans 151, 153
 Stateless Session Beans 151, 152
 Statisch typisiert 99
 Statisch typisierte Programmiersprachen
 rein spezifizierende Klassen 169
 Statische Fabrik 352
 parametrisiert 355
 Umsetzung 354
 Statische Klassifizierung 289
 Statische Polymorphie 194, 229
 Statischer Speicher 397
 Statisches Crosscutting 548
 Definition 541
 Statisches Typsystem 93
 Strategieklassen 296
 Struktur
 Darstellung 31
 von objektorientierter Software 65
 von Programmen und Daten 28
 Struktur des Programms, bei Laufzeit lesen 537
 Strukturierte Programmierung 28
 Mechanismen 28
 Subclass → Unterklasse
 Swing 566
 Synchroner Nachricht 419

T

Tabelle für virtuelle Methoden 421
 Template Method 247
 Test 62
 automatisiert 62
 Vorteile 63
 Testprogramm 62
 throws-Klausel erweitern 494
 Tiefe Kopie 449
 Timingdiagramm 413
 Top-down-Entwurf 56
 To-Space 406
 Transaktion 553
 über dynamische Pointcuts 554
 Trennung der Anliegen 45, 527
 Trennung der Schnittstelle von der Implementierung 53
 Trennung, Daten und Code 71

try-catch-Block 475
 Tupel 303
 Typ eines Objekts
 zum Typ einer Unterklasse konvertieren 180
 Typbestimmung zur Laufzeit 181
 Typisierte Sprachen
 redundanter Code 96
 Typisierung
 schwach 99
 stark 99
 Vor- und Nachteile 101
 Typkonflikte 93
 Typsystem 94
 Definition 92
 dynamisch 94, 95
 statisch 93
 Vererbung der Spezifikation 176
 Typumwandlung in Java 99

U

Übergabe an abhängige Module →
 Dependency Injection
 Übergreifende Anliegen → Crosscutting
 Concern
 Überprüfte Exception → Checked
 Exception
 Überprüfung während der Übersetzung
 mit Aspektorientierung 551
 Überschreiben von Methoden 242
 Überschriebene Methoden 246
 Umkehr der Abhängigkeiten 56
 UML 412
 Aggregation 130
 Assoziation 119
 Assoziationsklassen 127
 Beziehungsklasse 126
 Darstellung eines Objekts 69
 Diagrammtypen 412
 Einschränkungen von Beziehungen 124
 Komposition 130
 Navigierbarkeit 120
 Qualifikatoren 126
 Rollen 119
 UML-Diagramme, Verwendung 69
 Unified Modelling Language 19
 Unit-Test 62
 Unterklasse 157

Unterklassen 156
 erben Funktionalität 239
 Exemplare 161
 Nachbedingungen ändern 164
 und Invariante 165
 Vorbedingung der Operationen 459
 Vorbedingungen verändern 164
 Untermodul 40
 Unterprogramm aufrufen 30
 update anomaly 324

V

value object → Wertobjekt
 Variable 29
 Vererbung 34
 Erweiterung von Modulen 505
 öffentliche Sichtbarkeit 184
 privat 185
 Sichtbarkeit 183, 184
 Varianten 241
 Vererbung der Implementierung 35, 239
 Problem 250
 Programmiersprachen 240
 Verbot der Modifikation 244
 Vererbung der Spezifikation 34
 Definition 157
 Typsistem 96, 176
 Vererbung von Implementierungen 244
 Vererbungsbeziehungen in relationaler
 Datenbank abbilden 315
 Vererbungsmöglichkeit, fehlend 271
 Vergleichicher 452
 Vergleichsoperation der Basisklasse
 umsetzen 257
 Verhalten von Programmen betrachten
 413
 Verletzung eines Kontrakts
 Programmabbruch? 485
 Vertrag → Kontrakt
 Vielgestaltigkeit 193
 Vierte Normalform 330
 virtual 249
 Virtuelle Methoden
 Fehler bei Anpassungen 232
 hinzufügen 234
 Reihenfolge 231
 überschreiben 232

Virtuelle Vererbung 287
 Virtuelle-Methoden-Tabelle 228, 229
 Umsetzung 232
 Visitor 217
 VMT → Virtuelle-Methoden-Tabelle
 Vor- und Nachbedingung, Anforderungen
 457
 Vorbedingung 89
 prüfen 468
 Überprüfung 556

W

Wartbarkeit 24
 erhöhen 43
 verbessern 48
 Weavring 541
 Werte 81, 133
 als Objekte implementiert 134
 Identität 134
 objektorientierte Programmiersprachen
 134
 Wertobjekt 134, 135
 Änderung 137
 Definition 135
 Identität 151
 Wiederholung
 automatisch generiert 50
 Entstehung 49
 in Quelltexten 47
 Wiederholung vermeiden 47

Z

Zählen von Referenzen 400
 Problem 403
 Zusatzinformation in Programmstruktur
 einbinden 564
 Zusicherung 456
 Zustand, Modellierung 417
 Zuständigkeitskette 367
 Zustandsautomaten 412
 Zustandsdiagramm 412, 415
 Zyklische Referenz 403
 bei Kopien 449